Structural Methods for the Synthesis of Well-Formed Concurrent Specifications

Josep Carmona Vargas

Draft submitted in Partial Fulfillment of the Requirements for the Degree of Doctor in Computer Science

Universitat Politècnica de Catalunya
Software Department

A todos aquellos que siempre estarán dentro de mí.

Acknowledgments

I would like to thank my advisor Jordi Cortadella, who showed me what real research is. It is difficult to summarize in a paragraph all the help and encouragement that Jordi has been giving to me in the last four years. I am really grateful to him and hope that we will keep on working together in the future.

I would also like to thank my colleagues in the Escola Politècnica Superior de Castelldefels, and my former colleagues in the Software department of the UPC. I am specially grateful to Jetty Kleijn and Grzegorz Rozenberg, who hosted me nine months at the Leiden Institute of Advanced Computer Science.

In the last four years, I had several discussions about the topics of my thesis with people directly or indirectly related. I think it is fair to mention them here: Enric Pastor, Javier Esparza, Alex Yakovlev, Victor Khomenko, Josep Diaz, Luciano Lavagno, Peter Notebaert, Ivan Blunno, Jetty Kleijn, Radu Negulescu. Michael Yoeli and some others. I also thank Sanjeevan Kanapathipillai who decoded some of the cryptic English sentences found in early stages of this manuscript.

This work was supported in part by a doctoral grant of the *Generalitat de Catalunya*, grant 2000FI-00472, by a grant in the SEGRAVIS project (Syntactic and Semantic Integration of Visual Modeling Techniques) under contract HPRN-CT-2002-00275, the *Ministerio de Educación y Ciencia* of Spain under contract CICYT TIC 2001-2476 and by the Asynchronous Circuit Design Work Group (ACiD-WG), contract IST-1999-29119.

Finally, I would like to thank all my family and friends for obvious reasons, and thanks to music, for always being there whenever I needed it.

Preface

A specification of a concurrent system describes a set of components that operate in a parallel environment and eventually interact. When modeling such concurrent behavior, the set of states that the system can reach are typically very large or even infinite. This phenomenon is known as the *state space explosion problem*. Consequently, algorithms that work on the state space of such type of systems suffer from the state space explosion problem, thus having high complexity.

Formal methods is a convenient tool for the analysis, verification and synthesis of concurrent systems. They are mathematically-based languages, methods and tools that can be used in some cases despite the complexity of the system under consideration. Structural methods are formal methods that use the structure of the model in order to reason about its underlying behavior. Those methods are specially suited when the system is large and highly concurrent. In this work we present structural methods for the synthesis of concurrent systems, with direct application to asynchronous circuits.

Asynchronous circuits are the simplest class of concurrent systems. Despite of their simplicity, problems like the state space explosion problem already exist. Therefore, state-based algorithms for the automatic synthesis of asynchronous circuits can only synthesize small size specifications. This work provides structural methods for the synthesis of large specifications. The methods developed perform as a state-based method would do if there was enough memory or machine-power. By using graph algorithms or linear algebra, the design flow presented avoids the computation of the whole state space. Experimental results show the significant improvement with respect to existing approaches.

Contents

1 Introduction **11**

2 Basic Definitions **19**
2.1 Preliminaries 19
2.1.1 Sets 19
2.1.2 Vectors and Matrices 20
2.1.3 Sequences 21
2.2 Reactive Systems 21
2.3 Transition Systems 22
2.4 Petri Nets 23
2.5 Linear Algebra and Petri Nets 29
2.5.1 Linear Programming 29
2.5.2 Approximation of the Reachability Set of a PN 30
2.6 Asynchronous Circuits 32
2.6.1 Classes of Asynchronous Circuits 33
2.6.2 Control Circuits 33
2.6.3 State Graphs 34
2.6.4 Signal Transition Graphs 36
2.6.5 Synthesis of Speed-Independent Circuits 36

3 Compatibility of Reactive Systems **43**
3.1 Introduction 44
3.2 Properties of Reactive Transition Systems 46
3.3 I/O Compatibility. 46
3.4 A Polynomial-time Decision Procedure for I/O Compatibility 49
3.5 I/O Compatibility and Observational Equivalence. 50
3.5.1 Observational Equivalence 50
3.5.2 A Sufficient Condition for I/O Compatibility. 51
3.6 Conclusion 54

4 Transformations for Synthesis **57**
4.1 Introduction 57
4.2 I/O Compatible Transformations 62

4.2.1 Kit of PN Transformations 62
4.2.2 I/O Compatible Transformations over RPN 64
4.3 Encoding Technique 68
4.3.1 Encoding Transformation 69
4.3.2 I/O Compatible Encoding Technique over STG 74
4.4 Conclusion 80

5 ILP for Verification and Synthesis 83
5.1 Introduction 83
5.2 ILP for Verifying State Encoding 84
5.2.1 ILP for USC Checking 85
5.2.2 ILP for CSC Checking 87
5.3 ILP for Synthesis 89
5.3.1 Computing a Support for Synthesis 89
5.3.2 Projection into the CSC Support 92
5.4 Experimental Results for USC/CSC Checking 94
5.5 Conclusion 97

6 Synthesis of Asynchronous Circuits 101
6.1 Introduction 101
6.2 The Design Flow 102
6.3 Synthesis Examples 104
6.3.1 Synthesis of the VME Bus Controller 104
6.3.2 Synthesis of the PpArbCsc(2,3) Example 110
6.4 Experimental Results 110
6.5 Conclusion 111

7 Conclusions 117

Chapter 1

Introduction

Concurrent systems have been studied for the last forty years by the Computer Science community. From the beginning, it was clear that the use of mathematically-based languages, methods and tools represented the best way for specifying and analyzing such systems. This is what *formal methods* are: theories that allow one to construct or verify complex systems. In some cases, formal methods can be used despite the complexity of the concurrent system.

It is widely accepted that there is not a unique formal model for the specification and analysis of a concurrent system. In the last two decades several new models appeared, which contributed to a deeper understanding of how concurrent systems work, and to realize what were the difficulties in modeling them. Among other formal models, we can mention Petri nets [67], CCS [55], CSP [39], Temporal Logic [48] and I/O automata [50].

Once the specification model is chosen, the designer must be provided with a kit of methods that allow the design and verification of the concurrent system under consideration. The whole process of design and verification is called *system engineering*, and the development of efficient methods and tools for each one of those two steps represents one of the biggest challenges for today's technology. Many researchers in Computer Science in the last decades have been addressing this issue, and still a lot needs to be done in the future [21].

The need for more research on developing new methods and tools for the synthesis and verification of concurrent systems is because most of the existing methods are very complex. The hardness in the complexity of the algorithms is due to the fact that most of the methods known up-to-date can only be applied when the underlying state space is known. In general, the state space corresponding to the specification of a concurrent system is exponential on its size. This phenomenon is known as *the state space explosion problem*. Even when using implicit data structures ([10]) or partial order approaches ([54]) it is not enough to guarantee the avoidance of exponential

state spaces.

The state space explosion problem appears even if the concurrent system in consideration describes a very restricted behavior. For instance, the behavior can be restricted to only allow binary actions to take place in the system, thus describing an *asynchronous circuit*. This class of systems represents the simplest concurrent systems that one can deal with: an asynchronous circuit is simply defined as an arbitrary interconnection of gates that compute some function. Despite its simplicity, the development of theories and methods for asynchronous circuit design and verification is important because:

1. Successful theories and methods working in the simplest case (i.e. asynchronous circuits) can be generalized to bigger classes of concurrent systems.

2. Asynchronous circuits describe a more natural way of computing when compared to clocked (synchronous) circuits, offering a different approach to digital systems design.

Despite the second reason noted above, nowadays digital systems are synchronous: in the underlying circuit of every digital system, a global signal (the clock) makes every component to be aware of when it is supposed to finish its computation. This process is repeatedly done even if the component itself has no computations to perform. Therefore, it is natural that *power consumption* and *performance* problems can appear in circuits designed according to the synchronous paradigm. Moreover, the fact that a global signal is distributed along the circuit makes almost impossible to design fully *modular* synchronous circuits. Some other drawbacks, like the problem of the *clock skew* and the *Electromagnetic Compatibility (EMC)* are also present in the synchronous paradigm. Several papers and books in the literature highlight the limitations of synchronous circuits [3, 73, 22].

The asynchronous paradigm is free from all the problems existing in the synchronous paradigm. It is, by definition, modular and the power consumption in every asynchronous circuit is clearly lower compared to an equivalent synchronous circuit, because computation is only performed when needed. All the other problems of synchronous circuits are also naturally avoided by the asynchronous paradigm [5].

Despite the clear advantages of asynchronous circuits, they are seldom used. The main reason is because, in general, asynchronous circuits are very difficult to design. For designing a synchronous circuit, one must simply define the combinational logic necessary to compute the function, and surround it with latches [38]. Then by adapting the clock rate long enough to allow the combinatorial circuit to produce its expected output, the designer avoids the existence of errors on the circuit's function (called *hazards*). Note that in this way, even if the circuit performs very rarely the most time consuming

computation π, the clock cycle in an synchronous circuit is determined by π.

On the contrary, to prevent hazards from appearing in an asynchronous circuit can be sometimes an art, especially if the system under consideration is a complex one of medium/big size. This is because in an asynchronous circuit, the fact that no global synchronization exists implies that for performing a given function, the system can progress into several intermediate states that result from the inherent parallelism of the asynchronous model. Given that every intermediate state is valid for the system, the designer must ensure that the state itself is free from hazards. For large and complex systems, this task can be almost impossible to perform manually. The development of methods and tools for the automated design (synthesis) and verification of asynchronous circuits is one of the hot topics addressed by several researchers in the last two decades [27, 62, 49, 4, 59, 68]. The work presented here describes a novel approach for the synthesis of asynchronous circuits.

In the past few years, several researchers have provided reasons about why it is needed to start thinking of introducing asynchronous ideas in actual synchronous systems. Following this advice, some of the important microelectronics companies are incrementally introducing asynchronous design principles in their designs. Philips can be considered the main company on exploiting asynchronous. Other companies such as Intel, Sun, IBM and Infineon are also trying to get rid of the clock in some parts of their designs. This mixture between clocked and unclocked circuits made part of the asynchronous community to start focussing their research on less radical architectures, like the *Globally Asynchronous Locally Synchronous* (GALS). In a GALS system, modules are globally managed by means of handshakes (i.e. asynchronously), but the modules are considered to be locally synchronous. This architecture allows the designer to decouple timing constraints on different modules and therefore some of the typical problems of pure synchronous circuits like the clock skew are alleviated. Moreover, the GALS approach allows one to substitute synchronous modules by asynchronous ones, while preserving the functionality of the circuit [58]. This fact also motivates the need to continue working on the development of methods and tools for the automatic design of asynchronous circuits.

Although several academic tools exist for the synthesis of asynchronous circuits, it is assumed that there is a big difference these days between CAD synthesis tools for synchronous and asynchronous circuits. There is still a lot to be done within the asynchronous community in order to develop CAD tools with similar degree of maturity of the synchronous approach. Some of the leading researchers on asynchronous are still pointing out the lack of mature CAD tools for asynchronous design: in the International Conference on Application and Theory of Petri Nets (ICATPN'02), the author of the invited paper [79] claimed the need of tools for the synthesis of asyn-

chronous controllers from an interpreted Petri net representing a circuit. In the suggested approach, the Petri net is considered to be obtained from a syntax-directed translation of a *Hardware Description Language* (HDL) program specifying the behavior to synthesize. Two main features characterize the Petri nets obtained by this approach: (1) the size is linear on the size of the HDL program, but the state space is usually very large, and (2) the net *inherits* the good structure of the program.

Moreover, the paper stresses the importance of the application of *global optimization*, that can result in a significant improvement of the quality of the circuit synthesized. However, the existing approaches for performing global optimization techniques can only be applied when the state space of the underlying system is known, and therefore all those techniques suffer from the well known state space explosion problem. Usually, global optimization techniques can handle specifications with at most 20-30 binary variables. Even when applying partial order techniques ([71]) or using implicit compact representations of the state space ([10]) does not help in general to overcome the potential overhead of the state-based global optimization methods.

In this thesis we provide the theory, methods and a CAD tool for the synthesis of asynchronous circuits from interpreted Petri nets. Although being able to synthesize any specification in the class of Free-choice Petri nets ([26]), we believe that the approach is specially well suited to be applied to those specifications coming from a syntax-directed translation from a HDL. The reason for this comes from the fact that the approach strongly relies on using structural methods in almost all the stages of the synthesis process, which can benefit from the well-structuredness property of this type of specifications.

The proposed design flow can overcome the state space explosion problem by both using *structural methods* and by decomposing the system into smaller subparts and performing the synthesis on each subpart afterwards. The structural methods used are:

- *Graph Theory.* Mainly Petri net transformations, either to ensure implementability of the specification or for projecting the initial Petri net into some set of variables. The transformations can also be applied when trying to improve the final implementation according to some objective function.

- *Linear Programming.* By using the marking equation of the interpreted Petri net to synthesize, it can be codified in a integer linear programming problem the check of some important implementability conditions. It is also possible to use integer linear programming techniques to support the process of decomposition of a specification.

The experimental results obtained by our tool show that it can handle

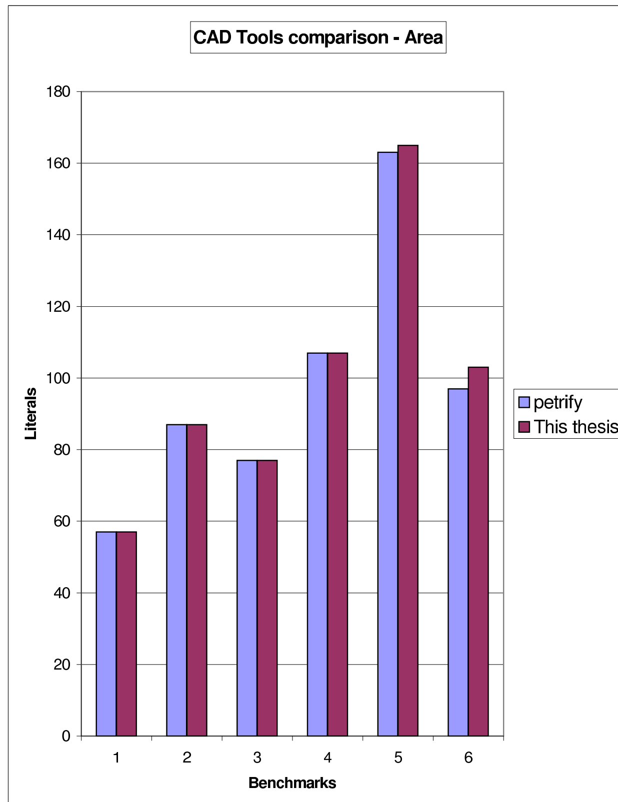


Figure 1.1: Comparison for area of the synthesized circuits.

specifications with hundreds of binary variables, and the quality of the circuits obtained is comparable to the one obtained by state-based approaches. As a motivating example for this work, Figures 1.1 and 1.2 present a comparison of the results obtained by our CAD tool and petrify [22], a well known state-based CAD tool for the synthesis of asynchronous controllers.

Figure 1.1 presents a comparison on the number of literals synthesized by each tool. It is significant the similarity of the quality of both approaches, despite of the fact that in the first approach global optimization techniques have been applied to the complete state space of each specification, while in our approach the specification has been decomposed and synthesized separately. This similarity in the solutions obtained happens frequently. It means that, very often it is not necessary to compute the complete state space for the synthesis of a given signal of the circuit.

A comparative analysis on CPU time is shown in Figure 1.2. The y-axis depicts the logarithm of the time needed by each tool to perform the synthesis. In the case of benchmarks 3,4, 8 and 10, the state-based tool was aborted after ten hours of computation. The figure shows a clear improvement on orders of magnitude of our approach.

In conclusion, we present a theory for the synthesis of asynchronous controllers. The resulting CAD tool can synthesize specifications with the same quality and significantly less time than present state-based methods. It can handle bigger specifications, provided that structural methods are the core of this work.

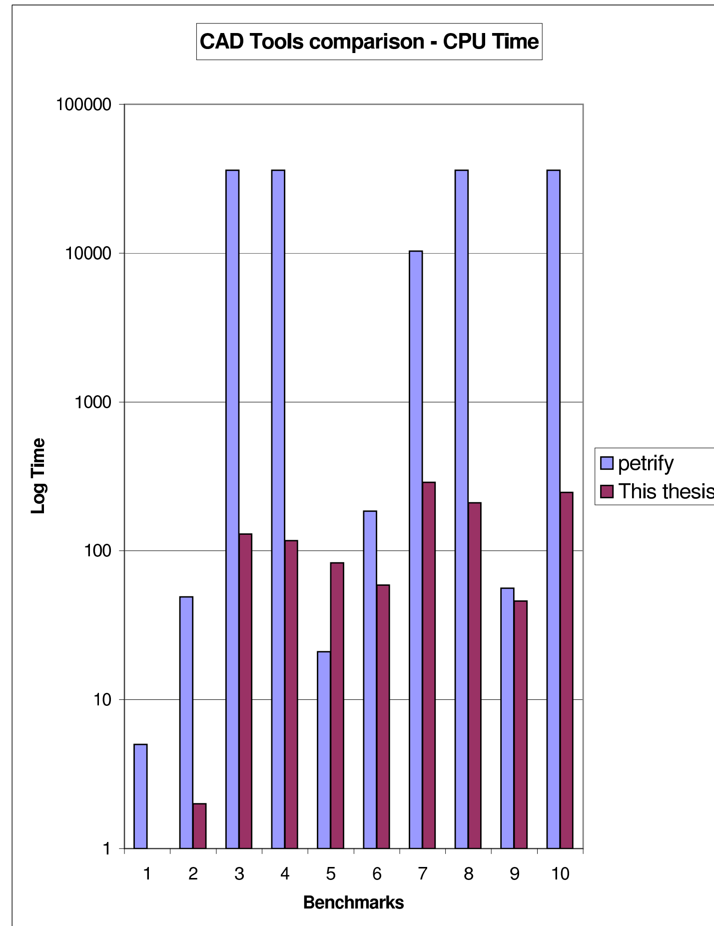


Figure 1.2: Comparison for CPU time for synthesizing every benchmark.

Organization of the Thesis

The necessary background and formal definitions of the basic notions appear in Chapter 2. Once the mathematical preliminaries are introduced, four main topics are addressed: Transitions Systems, Petri Nets, Linear Algebra and Asynchronous Circuits.

Chapter 3 presents the conditions that two reactive systems must fulfill in order to interact without having errors or deadlocks. The relation described,

called *I/O Compatibility*, is then used in Chapter 4 to introduce a kit of synthesis rules that can be applied to a Petri net specifying a reactive system. It is shown how to apply those rules while preserving the I/O Compatibility between the system and its environment. A structural encoding technique for the synthesis of asynchronous circuits is also described in Chapter 4, which can also be modified to preserve the I/O Compatibility.

Integer Linear Programming (ILP) techniques are used in Chapter 5 to present methods for the verification and synthesis of asynchronous circuits. The models introduced focuses on the problem of the encoding: in the verification part, ILP models are introduced to semi-decide the problem of the correct encoding for asynchronous circuits. On the synthesis part, a novel method is introduced that allows one to split a specification into several subparts while preserving the implementability conditions.

The whole theory introduced in Chapters 3, 4 and 5 is merged in Chapter 6, where a complete design flow for the synthesis of asynchronous circuits is presented, and finally two examples of complete synthesis are described.

Chapter 2

Basic Definitions

Las cosas que yo sé
las sabe un tonto cualquiera
– Kiko Veneno, *Salta la Rana*

The necessary background for the theory presented in the following chapters is introduced here. Section 2.1 describes the notation used throughout the book. Section 2.2 presents reactive systems, the general class of systems that are the object of study. Sections 2.4 and 2.3 describe two mathematical models, Petri nets and Transition systems, used to specify a reactive system. Section 2.5 presents the relation between the model of Petri nets with the Linear Algebra theory. Finally, Section 2.6 defines formally what is an asynchronous circuit, and explains how the synthesis of an asynchronous circuit is performed, when some delay model is assumed.

2.1 Preliminaries

2.1.1 Sets

Definition 2.1.1 (Sets) *A* set X *is a collection of distinct objects, called* elements *or* members*. We use $x \in X$ to express that x is a member of the set X.*

Two ways of describing a set X are introduced:

- Exhaustive list of its elements: $X = \{a, b, c\}$, $X = \{x_1, ..., x_n\}$.
- Members of the set must satisfy some condition. The format will be $X = \{x \mid x \text{ satisfies condition } \Pi\}$ where the condition can be expressed either in natural language or as some predicate in first-order logic .

Given two sets X and Y, $X = Y$ and $X \neq Y$ denote equality and inequality of the sets X and Y, respectively. We write $X \subseteq Y$ to denote

that X is a subset of Y, and $X \subset Y$ to denote that X is a proper subset of Y, i.e., $X \subseteq Y$ and $X \neq Y$. $X \setminus Y$ denotes the set of members of X that are not members of Y. The *Cartesian product* of X and Y, denoted by $X \times Y$ is defined by

$$X \times Y = \{(x, y) \,|\, x \in X \wedge y \in Y\}$$

and the operator can be defined for any given (finite) arity. The Cartesian product of $X_1, ..., X_n$ is defined by

$$X_1 \times ... \times X_n = \{(x_1, ..., x_n) \,|\, x_1 \in X_1 \wedge ... \wedge x_n \in X_n\}$$

When all the sets are the same, the Cartesian product can be abbreviated as X^n. For instance, $X \times X \times X$ is X^3. Moreover, in this situation more than one dimension can be defined in the Cartesian product:

$$X^{n \times m} = \{((x_1, ..., x_m), ..., (x_n, ..., x_m)) \,|\, x_i \in X\}$$

Finally, some symbols are used to denote universal sets:

- the set of rational numbers, $\mathbb{Q}$
- the set of integers, $\mathbb{Z}$
- the set of nonnegative integers, $\mathbb{N}$
- the set of binary numbers, $\mathbb{B} = \{0, 1\}$

2.1.2 Vectors and Matrices

A vector v of dimension n over a set X is an element from X^n, i.e., $v \in X^n$. It will be denoted by $(v_1, v_2, ..., v_n)$. For instance, $v = (3, 1/2, 5.2)$ is a vector of dimension three over $\mathbb{Q}$, i.e. $v \in \mathbb{Q}^3$. Given two vectors x and y of dimension n, $x \cdot y$ denotes the product of the two vectors, defined by

$$x \cdot y = (x_1 y_1, ..., x_n y_n)$$

Vector $v|_P$ denotes a new vector formed only by the components appearing in the index set P.

A matrix $\mathbf{C}$ of dimension $n \times m$ over a set X is an element from $X^{n \times m}$. The operations $v\mathbf{C}$ and $\mathbf{C}v$ denote the left and right products of matrix $\mathbf{C}$ and the vector v, only defined if the dimensions agree. Along the book, a matrix will be denoted with bold capitals and a vector with italics. The symbol $\mathbf{0}$ denotes a vector such that every component is 0.

2.1.3 Sequences

Let Σ be a set, called *alphabet*. A finite sequence (of length n) on Σ is a mapping $\{1, ..., n\} \rightarrow \Sigma$. We represent a finite sequence $\sigma : \{1, ..., n\} \rightarrow \Sigma$ as the word $x_1\, x_2\, ...x_n$, where $x_i = \sigma(i)$, for $1 \leq i \leq n$. The empty sequence is defined as $\epsilon : \emptyset \rightarrow \Sigma$, with length 0. For instance, if $\Sigma = \{a, b, c, d\}$ a possible sequence of length 5 on Σ is $\sigma = aaacd$.

If $\sigma = x_1\, x_2\, ...x_n$ and $\gamma = y_1\, y_2\, ...y_m$ are finite sequences then the concatenation of σ and γ, denoted by $\sigma\, \gamma$, is the sequence $x_1\, x_2\, ...x_n\, y_1\, y_2\, ...y_m$ of length $n + m$. For any sequence σ, $\sigma\, \epsilon = \sigma$.

The operator $\#(\sigma, x)$ denotes the number of occurrences of x in σ. For instance, if $\sigma = aaacd$, $\#(\sigma, a) = 3$, $\#(\sigma, c) = 1$ and $\#(\sigma, b) = 0$.

Finally, the projection of sequence σ into set X, denoted by $\sigma|_X$, is a new sequence obtained by removing from σ those elements not belonging to X.

2.2 Reactive Systems

Reactive systems [36] are systems that operate in a distributed environment. The events observed in a reactive system can be either input events, output events or internal events. An input event represents a change in the environment for which the system must react. In contrast, an output event is generated by the system and can force other systems in the environment to react to. Finally, an internal event represents system's local progress, not observable by the environment. Typical examples of reactive system are a computer, a television set and a vending machine.

In this work, we deal with the synthesis problem of reactive systems. Given a system and its environment, a synchronization protocol is committed in such a way that, at any state, the environment is guaranteed to produce only those input actions acceptable by the system. This assumption about the environment is opposite to other models in the literature like I/O automata [51], where the system must always be able to accept environment's actions.

More specifically, the problem faced in this work is: given the specification of a reactive system, generate an implementation realizable with design primitives that commits the protocol established with the environment. Here we focus on the synthesis of asynchronous circuits (see Section 2.6 for a general description of this type of systems), where the events of the reactive system are rising and falling signal transitions and the design primitives are logic gates.

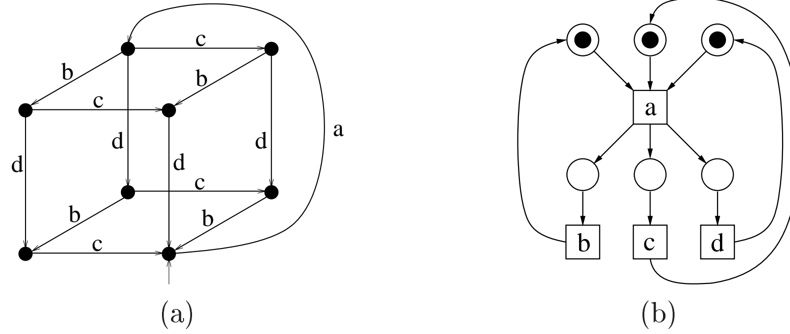


Figure 2.1: (a) Transition System. (b) Petri net.

2.3 Transition Systems

A Transition System is an automaton which describes the behavior of a system of processes. It contains a set of possible states and a set of transitions representing potential changes of the system's state. A general description of the model and extensions can be found in [2].

Definition 2.3.1 (Transition System) *A Transition System (TS) is a* 4-tuple $A = (S, \Sigma, T, s_{in})$ *where*

- S *is the set of states*
- Σ *is the alphabet of events*
- $T \subseteq S \times \Sigma \times S$ *is the set of transitions*
- $s_{in} \in S$ *is the initial state*

Figure 2.1(a) depicts an example of TS. The initial state is denoted by an incident arc without source state.

Definition 2.3.2 (Reachability in a TS) *The transitions are denoted by* (s, e, s') *or* $s \xrightarrow{e} s'$. *An event is said to be* enabled *in the state* s, *denoted by the predicate* $\mathsf{En}(s, e)$, *if* $(s, e, s') \in T$, *for some* s'. *The* reachability relation *between states is the transitive closure of the transition relation* T. *The predicate* $s \xrightarrow{\sigma} s'$ *denotes a sequence of events* σ *that leads from* s *to* s' *by firing transitions in* T. *A state* s *is* terminal *if no event is enabled in* s. *A TS is* finite *if* S *and* T *are finite sets.*

Definition 2.3.3 (Language of a TS) *A TS can be viewed as an automaton with alphabet* Σ, *where every state is an accepting state. For a TS* A, *let* $L(A)$ *be the corresponding language, i.e. its set of sequences starting from the initial state.*

Definition 2.3.4 (Deterministic TS) *A TS is deterministic if for each state s and each event e there can be at most one state s' such that $s \xrightarrow{e} s'$.*

The synchronous product of two transition systems is a new transition system which models the *interaction* between both systems [2]. This operator will be used in Chapter 3 to present methods for checking the correct interaction between two reactive systems.

Definition 2.3.5 (Synchronous Product) *Let $A = (S^A, \Sigma^A, T^A, s^A_{in})$, $B = (S^B, \Sigma^B, T^B, s^B_{in})$ be two TSs. The synchronous product of A and B, denoted by $A \times B$ is another TS (S, Σ, T, s_{in}) defined by*

- $s_{in} = \langle s^A_{in}, s^B_{in} \rangle \in S$
- $\Sigma = \Sigma^A \cup \Sigma^B$
- *$S \subseteq S^A \times S^B$ is the set of states reachable from s_{in} according to the following definition of T.*
- *Let $\langle s_1, s'_1 \rangle \in S$.*
  - *If $e \in \Sigma^A \cap \Sigma^B$, $s_1 \xrightarrow{e} s_2 \in T^A$ and $s'_1 \xrightarrow{e} s'_2 \in T^B$, then $\langle s_1, s'_1 \rangle \xrightarrow{e} \langle s_2, s'_2 \rangle \in T$*
  - *If $e \in \Sigma^A \setminus \Sigma^B$ and $s_1 \xrightarrow{e} s_2 \in T^A$, then $\langle s_1, s'_1 \rangle \xrightarrow{e} \langle s_2, s'_1 \rangle \in T$*
  - *If $e \in \Sigma^B \setminus \Sigma^A$ and $s'_1 \xrightarrow{e} s'_2 \in T^B$, then $\langle s_1, s'_1 \rangle \xrightarrow{e} \langle s_1, s'_2 \rangle \in T$*
  - *No other transitions belong to T*

The evens in a TS can be interpreted as the actions taking place in a reactive system. This will allow to model a reactive system with a TS. An interpreted TS is called *reactive transition system*:

Definition 2.3.6 (Reactive Transition System) *A Reactive Transition System (RTS) is a TS (S, Σ, T, s_{in}) where Σ is partitioned into three pairwise disjoint subsets of input (Σ_I), output (Σ_O) and internal (Σ_{INT}) events. $\Sigma_{OBS} = \Sigma_I \cup \Sigma_O$ is called the set of observable events*

2.4 Petri Nets

A Petri net is a mathematical representation of a concurrent system. The theory of Petri nets was introduced by Carl Adam Petri in his dissertation *Kommunikation mit Automaten* [67]. A good summary on Petri net theory can be found in [57], whereas [33] presents an up-to-date survey both in theory and applications of Petri nets nowadays.

The model is composed of two parts: a net and a marking. The net represents the static structure of the system, while the marking denotes a

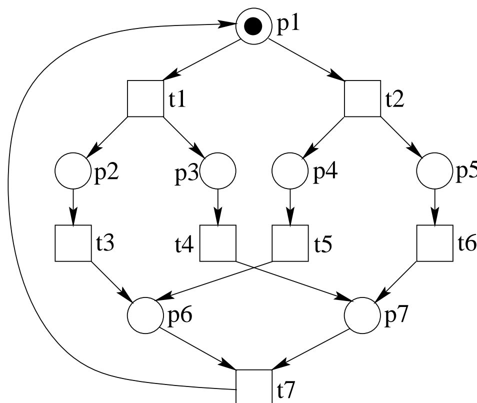


Figure 2.2: Petri net.

distributed global state. The description of the net is done by specifying two types of nodes and a flow relation that makes the underlying graph to be bipartite. Nodes called *places* are used to denote atomic states of the system, while nodes called *transitions* denote changes of the states. Finally, a marking is a distribution of tokens over the places.

Definition 2.4.1 (Petri Net) *A Petri Net (PN) is a 4-tuple* $N = \langle P, T, F, m_0 \rangle$*, where:*

- P *is the set of places,*
- T *is the set of transitions,*
- $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ *is the flow relation, and*
- $m_0 \in \mathbb{N}^{|P|}$ *is the initial marking.*

For any two nodes x and y, if $F(x, y) > 0$ then there is an arc from x to y, with weight $F(x, y)$. *Ordinary* nets are those where $F : (P \times T) \cup (T \times P) \to \{0, 1\}$. The net of Figure 2.2 is ordinary. Along this work we will assume always ordinary nets.

A marking of a PN is a $|P|$-vector m, where the component p of the vector is a natural number. If k is assigned to place p by marking m (denoted $m(p) = k$), we will say that p is marked with k tokens at m. Figure 2.2 shows a graphical view of a Petri net. Typically, transitions are denoted by boxes (or black bars), places are denoted by circles and the flow relation as directed arcs between the two sets, forming the bipartite structure. The marking m of a place p is graphically indicated by placing $m(p)$ tokens (black dots) on each place p. In the PN of Figure 2.2, $m_0(p_1) = 1$ and $m_0(p_i) = 0$, for $2 \leq i \leq 7$, or on its vector notation $m_0 = (1, 0, 0, 0, 0, 0, 0,)$.

The following definitions assume a given PN $N = (P, T, F, m_0)$.

Definition 2.4.2 (Paths and Cycles) *A* path *is a sequence $u_1 \dots u_r$ of nodes such that $\forall i, 1 \leq i < r : F(u_i, u_{i+1}) > 0$. A path is called* simple *if no node appears more than once on it. A simple path is called a* cycle *if all nodes along path are different except the initial and the final node.*

Definition 2.4.3 (Pre-sets, Post-sets) *Given a node $x \in P \cup T$, the set ${}^\bullet x = \{y \mid F(y, x) > 0\}$ is the pre-set of x and the set $x^\bullet = \{y \mid F(x, y) > 0\}$ is the post-set of x.*

For instance, in Figure 2.2, ${}^\bullet t_1 = \{p_1\}$, $t_1^\bullet = \{p_2, p_3\}$, ${}^\bullet p_1 = \{t_7\}$ and $p_1^\bullet = \{t_1, t_2\}$.

Definition 2.4.4 (Enabledness and Firing Rule) *A transition t is* enabled *at marking m if each place $p \in {}^\bullet t$ is marked with at least $F(p, t)$ tokens. When a transition t is enabled, it can* fire *by removing $F(p, t)$ tokens from place $p \in {}^\bullet t$ and adding $F(t, q)$ tokens to each place $q \in t^\bullet$.*

In the PN of Figure 2.2, the only transitions enabled at the initial marking are t_1 and t_2.

Definition 2.4.5 (Reachability and Feasible Sequences) *A marking m' is* reachable *from m if there is a sequence of firings $t_1 t_2 \dots t_n$ that transforms m into m', denoted by $m[t_1 t_2 \dots t_n\rangle m'$. A sequence of transitions $t_1 t_2 \dots t_n$ is a* feasible sequence *if it is firable from m_0. The set of reachable markings from m_0 is denoted by $[m_0\rangle$.*

For instance, in the PN of Figure 2.2, the marking $m = (0110000)$ is reachable through the feasible sequence t_1, while the marking $m' = (0011000)$ is not reachable.

It is widespread the use of a transition system to describe the behavior of a system of processes [2]. The model of Petri nets does not represent the behavior explicitly, but the causality relations among the set of actions of the system. However, by considering the set of reachable markings as the set of states of the system, and the transitions among this markings as the transitions between the states, a transition system can be obtained representing the underlying behavior of the PN. This transition system is called *reachability graph*.

Definition 2.4.6 (Reachability Graph) *Given a PN $N = (P, T, F, m_0)$, its* reachability graph *is a transition system, denoted by $RG(N)$ and defined by*

- *$[m_0\rangle$ is the set of states*
- *T is the alphabet of events*

- $\{\langle m, e, m'\rangle \mid m, m' \in [m_0\rangle \wedge m \stackrel{e}{\to} m'\}$ *is the set of transitions*
- m_0 *is the initial state*

Figure 2.1 depicts an example of reachability graph (left), associated to the PN on the right.

Definition 2.4.7 (Deterministic PN) *A PN N is deterministic if $RG(N)$ is deterministic.*

Definition 2.4.8 (Liveness) *A PN is live iff every transition can be infinitely enabled through some feasible sequence of firings from any marking in $[m_0\rangle$.*

Definition 2.4.9 (Boundedness and Safeness) *A PN is k-bounded if no marking in $[m_0\rangle$ assigns more than k tokens to any place of the net. A net is bounded if it is k-bounded for some k. A PN is safe if it is 1-bounded.*

The PN of Figure 2.2 is an example of live and safe PN.

Definition 2.4.10 (Home Marking and Reversibility) *A marking is a* home marking *if it is reachable from every marking of $[m_0\rangle$. A net is* reversible *if the initial marking m_0 is a home marking.*

Following the sequence of firings in the PN of Figure 2.2, it can be seen that the initial marking of the PN is a home marking. Therefore the net is reversible. However, if we assign $m_0' = (0110000)$ as initial marking, the net is no longer reversible because m_0' is not a home marking. Note that reversibility is neither necessary nor sufficient condition for liveness/boundedness: the net of Figure 2.2 with m_0' as initial marking is still live and safe.

Definition 2.4.11 (Triggering and Disabling) *Let $R \subseteq [m_0\rangle$ be the set of markings where transition t_i is enabled. Transition t_j* triggers *transition t_i if there exists a reachable marking m such that $m[t_j\rangle m'$, $m \notin R$ and $m' \in R$. Transition t_j* disables *transition t_i if there exists a reachable marking m enabling both t_i and t_j, but in the marking m' such that $m[t_j\rangle m'$, t_i is not enabled.*

The triggering and disabling relations are illustrated using the PN of Figure 2.2. The only transition enabled at marking $m = (0000011)$ is t_7. After firing t_7 the initial marking is reached, where both transitions t_1 and t_2 are enabled. Therefore t_7 triggers both t_1 and t_2. Moreover, at m_0 t_1 disables t_2 and vice versa.

By imposing restrictions on the underlying structure of a Petri net, several subclasses can be defined. In this work three subclasses are of interest: *State Machines*, *Marked Graphs* and *Free choice PNs*.

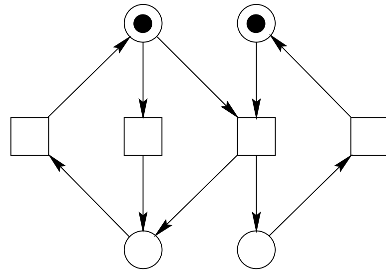


Figure 2.3: Non-free choice net.

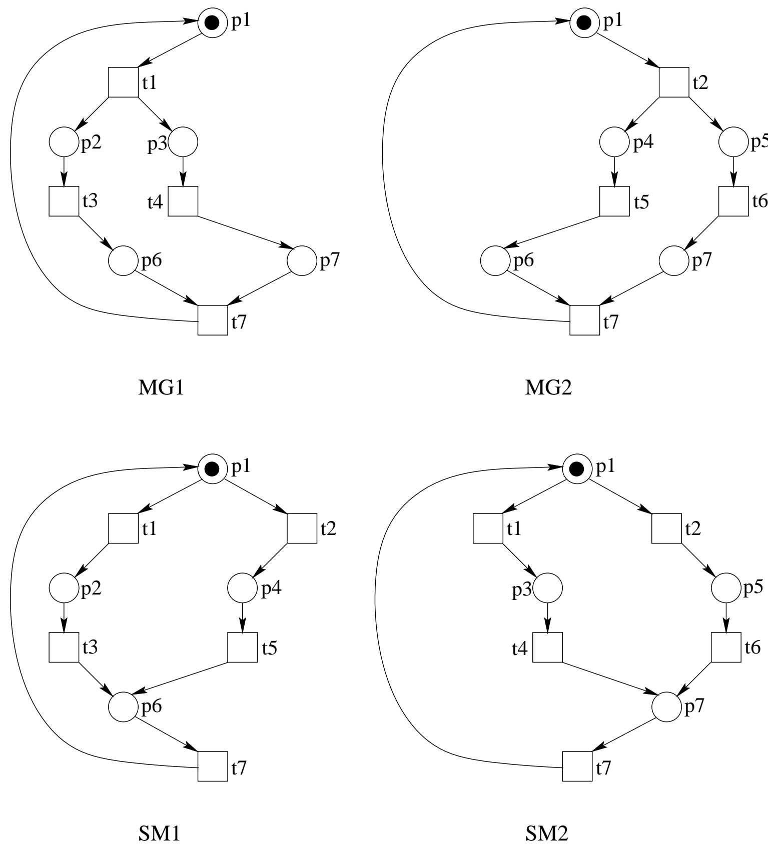


Figure 2.4: An MG-cover (MG1,MG2) and an SM-cover (SM1,SM2) of PN from Figure 2.2.

A State Machine models a sequential system, where conflicts between events can be expressed but concurrency is only possible if more than one token is distributed over the net. On the other side, a Marked graph represents a concurrent-synchronizing system where conflicts between events do not exist.

A Free choice PN can be seen as a State Machine enriched with Marked

Graph-like concurrency and synchronization. Informally, in a Free choice net whenever some output transition of a place p is enabled, all the transitions in the post-set of p are enabled, and therefore it is always possible "to freely choose" which one of them fires. This class of nets plays a central rôle in the theory of net systems because although being able to express non-trivial behaviors, there still exist powerful methods for its analysis and synthesis. A typical example is the *reachability problem*, EXPSPACE-hard for arbitrary nets ([53]) but polynomial for the class of free choice live safe and reversible Petri nets ([25]). Figures 2.2 and 2.3 depict examples of a Free Choice and a non Free choice Petri net, respectively.

Definition 2.4.12 (Petri Net subclasses)

- *A* State Machine *(SM) is a* PN *such that each transition has exactly one input place and one output place.*
- *A* Marked Graph *(MG) is a* PN *such that each place has exactly one input transition and one output transition.*
- *A* Free choice Petri net *(FC) is a* PN *such that for every place p and transition t, if $F(p,t) > 0$ then for every place q such that $F(q,t) > 0$, the equality $F(p,t) = F(q,t)$ holds.*

From the previous definition it can be seen that a PN which is an SM it is also FC. Analogously, if a PN is a MG it is also FC. The converse to the two previous inclusions does not hold: Figure 2.2 shows a FC PN which is neither a SM nor a MG. However, a live and safe free-choice PN can be decomposed into a set of SMs or MGs:

Theorem 2.4.1 (Free-choice decomposition [35]) *A free-choice live and safe Petri net (FCLSPN) can be decomposed into a set of strongly-connected state-machines (marked graphs). An* SM-cover *(*MG-cover*) of a FCLSPN is a subset of state machines (marked graphs) such that every place (transition) is included at least in one state machine (marked graph). Moreover, a FCLSPN can be also decomposed into a set of strongly-connected* one-token *state-machines, i.e. state-machines that at most contain one token at any reachable marking.*

Figure 2.4 shows MG and SM covers of the FCLSPN depicted in Figure 2.2.

We use PNs to model reactive systems (see Section 2.2), in order to derive algorithms that work at the net level and can implement the behavior of the system. For that purpose a notion of PN modeling a reactive systems is next introduced, leading to the definition of *reactive Petri net*. The new model can be seen as a *labeled Petri net* [66], where labels on transitions represent events occurring in a reactive system.

Definition 2.4.13 (Reactive Petri Net) *A Reactive Petri Net (RPN) is a* 3-tuple $((P, T, F, m_0), \Sigma, \Lambda)$ *where*

- (P, T, F, m_0) *is a PN*
- Σ *is partitioned as in Definition 2.3.6*
- $\Lambda : T \to \Sigma$

Section 2.6.4 contains examples of RPNs modeling digital circuits.

Finally, the concept of *redundant place* is defined:

Definition 2.4.14 (Redundant place) *Let* $N = (P \cup \{p\}, T, F, m_0)$ *be a Petri net. Place* p *is called redundant if the net* $N' = (P, T, F', m_0')$ *derived from removing* p *in* N *has the same set of feasible sequences, i.e* $L(RG((N)) = L(RG(N'))$.

2.5 Linear Algebra and Petri Nets

Linear algebra theory [61] has proven to be useful for facing many problems in very different areas [1]. Chapter 5 contains methods for the verification and synthesis of reactive systems using linear algebraic techniques. The basic theory supporting the methods of Chapter 5 is presented in this section.

2.5.1 Linear Programming

A *linear inequality* or *constraint* is given by an integral vector $a \in \mathbb{Z}^n$ and an integer b. It is represented by

$$a \cdot x \leq b$$

and it is *feasible* over a set A if there exists some assignment $k \in A^n$ to x satisfying $a \cdot k \leq b$.

A *system of linear inequalities* is a set of linear inequalities. It is *feasible* if there exists a vector that satisfies all inequalities of the set. If it is finite then it has a matrix based representation

$$\mathbf{A} \cdot x \leq v_b$$

where the vectors a of the linear inequalities are the rows of the matrix $\mathbf{A}$ and the numbers b are the components of the vector v_b.

Definition 2.5.1 (Linear Programming Problem) *A* linear programming problem *(LP) is a system $A \cdot x \leq b$ of linear inequalities, and optionally a linear function $c^T \cdot x$ called the* objective function*. A solution of the problem is a vector of rational numbers that satisfies the constraints. A solution is* optimal *if it maximizes the value of the objective function (over the set of all solutions). An LP is* feasible *if it has a solution.*

Definition 2.5.2 (Integer Linear Programming Problem) *An* integer linear programming problem *(ILP) is a LP where additionally, the integrality on the set of solutions is required.*

The complexity of solving a linear problem depends on the domain under consideration:

Proposition 2.5.1 (Complexity of Linear Programming [70])

1. *Each system of linear equalities over $\mathbb{Q}$ can be solved in polynomial time (LP).*
2. *The solubility of systems of linear inequalities over $\mathbb{Z}$ or $\mathbb{N}$ is NP-complete (ILP).*

2.5.2 Approximation of the Reachability Set of a PN

Computing the reachability graph from a given PN is a very hard problem, because the size of the reachability graph may grow exponentially with respect to the size of the PN, or it even can be infinite. The main reason is that the concurrency in the PN must be implicitly expressed in the reachability graph. The interested reader can find in [74] a discussion on the rôle of concurrency in relation with the size of the reachability graph.

Therefore it is interesting to approach the problem of reachability using other models or techniques. In this section we describe how to use ILP techniques to compute approximations of reachable markings of a PN.

Given a firing sequence $m_0 \xrightarrow{\sigma} m$ of a PN N, the number of tokens for each place p in m is equal to the number of tokens of p in m_0 plus the number of tokens added by the input transitions of p appearing in σ minus the tokens removed by the output transitions of p appearing in σ:

$$m(p) = m_0(p) + \sum_{t \in {}^\bullet p} \#(\sigma, t) F(t, p) - \sum_{t \in p^\bullet} \#(\sigma, t) F(p, t)$$

The concepts of *incidence matrix* and *Parikh vector* are next introduced:

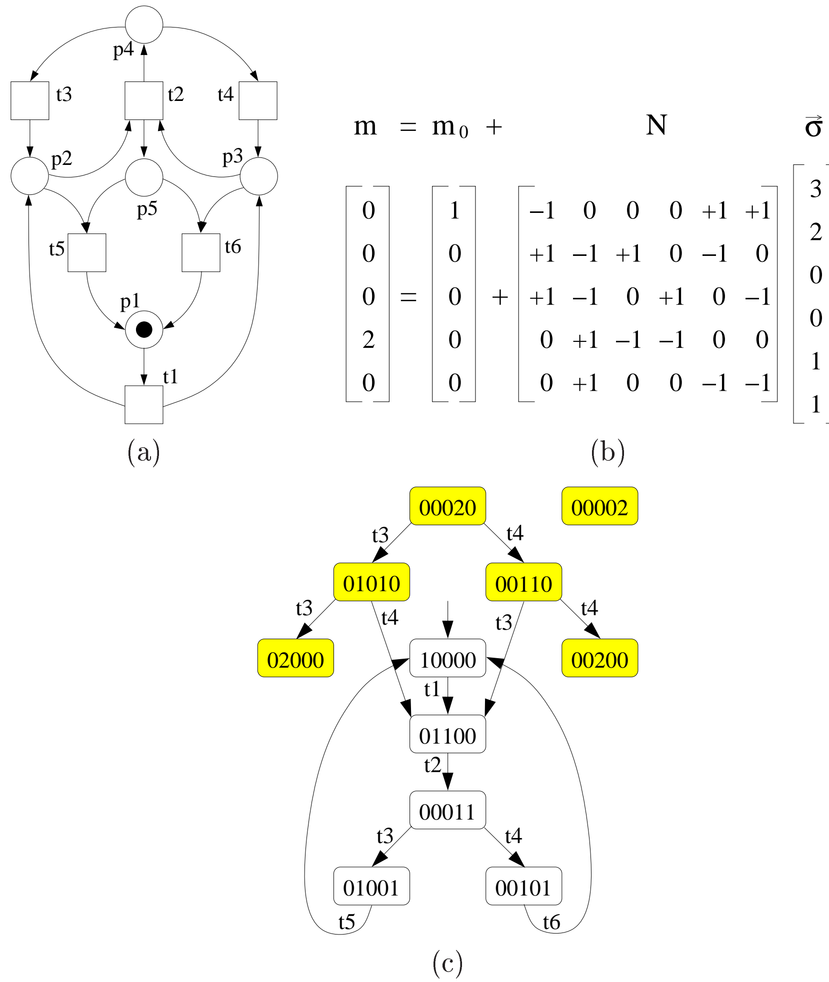


Figure 2.5: (a) Petri net, (b) Spurious solution $m = (00020)^T$, (c) Potential reachability graph.

Definition 2.5.3 (Incidence matrix of a PN) *The matrix* $\mathbf{N} \in \mathbb{Z}^{|P|\times|T|}$ *defined by*

$$\mathbf{N}(p,t) = F(p,t) - F(t,p)$$

is called the incidence matrix *of* N.

Definition 2.5.4 (Parikh vector) *Let* σ *be a feasible sequence of* N. *The vector* $\vec{\sigma}$ *defined by*

$$\vec{\sigma} = (\#(\sigma, t_1), ..., \#(\sigma, t_n)) \tag{2.1}$$

is called the Parikh vector *of* σ.

Using the previous definitions, the token conservation equations for all the places in the net can be written in the following matrix form:

$$m = m_0 + \mathbf{N} \cdot \vec{\sigma}$$

Which leads to the definition of the linear description of the reachability set by means of an ILP:

Definition 2.5.5 (Marking Equation) *If a marking m is reachable from m_0, then there exists a sequence σ such that $m_0 \xrightarrow{\sigma} m$, and the following problem has at least the solution $X = \vec{\sigma}$*

$$m = m_0 + \mathbf{N} \cdot X \qquad (2.2)$$

The equation $m = m_0 + \mathbf{N} \cdot X$ is called the marking equation.

Special attention must be paid in previous definition: the marking equation only provides a necessary condition for reachability. If the marking equation is infeasible, then m is not reachable from m_0, but the inverse does not hold in general: there are markings satisfying the marking equation which are not reachable. Those markings are said to be *spurious* [72]. Figure 2.5(a)-(c) presents an example of spurious marking: the Parikh vector $\vec{\sigma} = (320011)$ and the marking $m = (00020)$ are a solution to the marking equation of the Petri net of Figure 2.5(a), as shown in Figure 2.5(b)[1]. However, m can not be reachable by any feasible sequence: only sequences visiting *negative* markings can lead to m. Figure 2.5(c) depicts the graph containing the reachable markings and the spurious markings (shadowed). This graph is called the *potential reachability graph*. The initial marking is represented by the state (10000).

2.6 Asynchronous Circuits

Asynchronous circuits are digital circuits that react to the changes of their input signals according to the functionality of the gates of the circuit [11]. Synchronous circuits can be considered as a particular case of asynchronous circuits in which some specific design rules and operation mode are imposed.

In general, any arbitrary interconnection of gates is considered an asynchronous circuit. The synthesis problem consists in generating a proper interconnection of gates that commits a correct interaction with the environment according to some specified protocol.

This section presents some discussion about the different types of asynchronous circuits nowadays. Afterwards the models used in this work for the specification and synthesis of asynchronous circuits are introduced.

[1] Both in the figure and the explanation, we abuse the notation and skip the commas in the definition of Parikh vectors and markings.

2.6.1 Classes of Asynchronous Circuits

Asynchronous circuits can be classified as being *speed-independent*, *delay-insensitive* or *bounded-delay* circuits, depending on the delay assumptions that are made [11].

- *Speed-independent circuits* [56] (SI): the behavior of a circuit in this class is insensitive to the delay of its components (gates), although it can be sensitive to variations in the delays of the wires.
- *Delay-insensitive circuits* (DI): the behavior is independent of both gate and wire delays. This class of circuits although being very robust, has proven to be very small, and therefore the class of possible behaviors implemented is limited [52].
- *Asynchronous bounded-delay circuits*: the behavior is correct under some delay assumptions.

In this work we focus on the synthesis of speed-independent circuits. In modern VLSI technology, it may appear unrealistic the SI assumption because communication across a large chip can actually take much longer than any gate switching. However several reasons induce to consider SI circuits useful nowadays [46]:

- *The optimization capabilities of the more robust class of DI circuits are almost non-existent, and synthesis techniques amount to a little more than a syntax-driven translation from a specification language and peephole optimizations,*
- *any point-to-point communication can be modeled as a computation delay without loss of generality,*
- *it is possible to enforce communication protocols between subcircuits that ensure no dependency on communication delays, and*
- *it enables the use of Boolean optimization techniques to efficiently implement the circuit.*

2.6.2 Control Circuits

The most common way to design hardware is by separating the design of *control logic* from that of *datapath logic*. Designing the control logic means to implement the control flow of the system modeled, while designing datapath logic means to deal with the operational part required in the system [78]. The design of the datapath can be carried on by using standard library components.

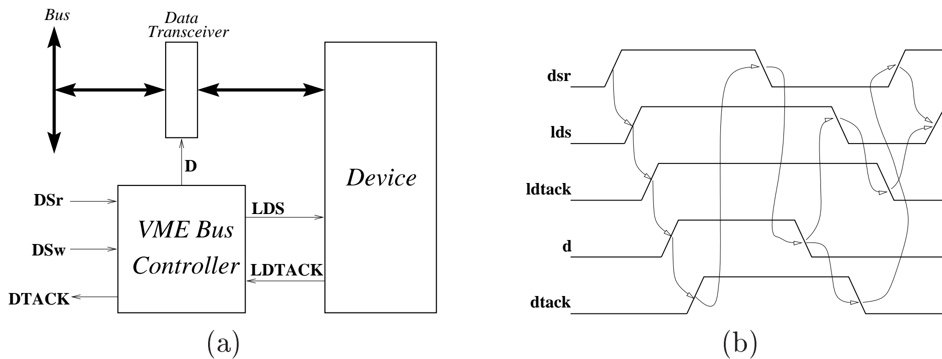


Figure 2.6: (a) Interface, (b) Timing Diagram.

In this work we focus on the synthesis of control logic circuits. We are interested in the synthesis of systems exhibiting a complex control logic structure, where actual methods for synthesis can not succeed.

2.6.3 State Graphs

Asynchronous circuits can be modeled with a RTS, where the events represent changes in the value of the system signals. The VME Bus Controller example in Fig. 2.6 will be used for illustrating the concepts. The interface is depicted in Fig. 2.6(a), where the circuit controls data transfers between the bus and the device. Figure 2.6(b) shows the timing diagram corresponding to the read cycle.

A transition labeled as x_i+ (x_i-) denotes a rising (falling) of signal x_i: it switches from 0 to 1 (1 to 0). Figure 2.7 shows the RTS specifying the behavior of the bus controller for the read cycle. Each state of an asynchronous circuit can be encoded with a *binary vector*, representing the signal values on that state. The set of encoded states are *consistently encoded* if no state can have an enabled rising (falling) transition $a+$ ($a-$) when the value of the signal in that state is 1 (0) (see Section 2.6.5 for a formal definition of consistency). Correspondingly, for each signal of a RTS representing an asynchronous circuit, a partition of the states of the RTS can be done by separating the states where the signal has value one, from those where the signal has value zero. This partition can only be done when the underlying asynchronous circuit is consistently encoded. Figure 2.8(a) shows the partition induced by considering signal `lds` in the RTS of Fig. 2.7. Each transition from `LDS=0` to `LDS=1` is labeled with `lds+` and each transition from `LDS=1` to `LDS=0` is labeled with `lds-`. A binary vector can be assigned to each state if such partition is done for each signal of the system. The encoded transition system is called *State Graph*.

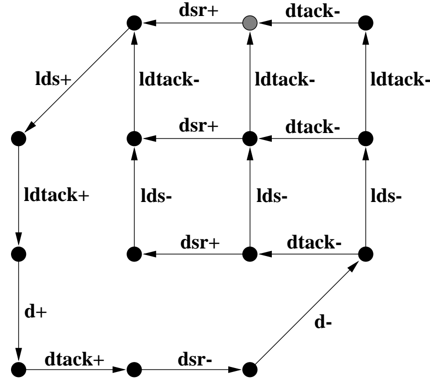


Figure 2.7: Transition System specifying the bus controller.

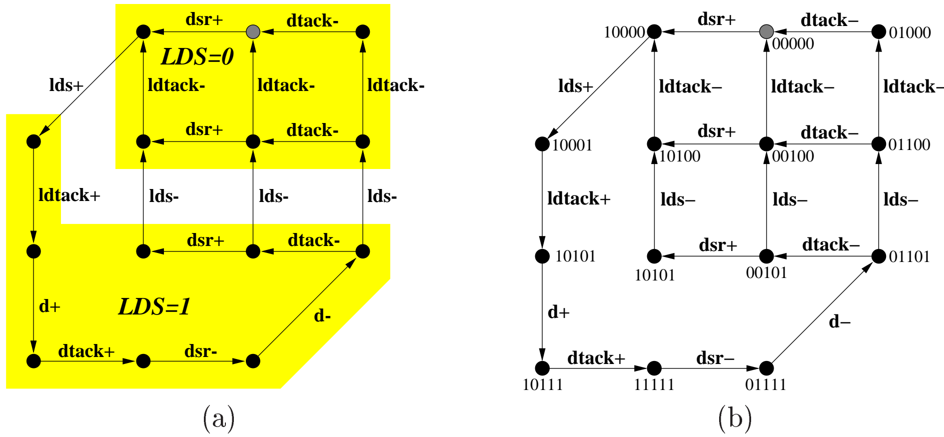


Figure 2.8: (a) Partition induced by signal `lds`, (b) State graph of the read cycle. States are encoded with the vector (`dsr`, `dtack`, `ldtack`, `d`, `lds`).

Definition 2.6.1 (State Graph) *A State Graph (SG) is a* 3-tuple $A = (A', \mathcal{X}, \lambda)$ *where*

- $A' = (S, \Sigma, T, s_{in})$ *is a RTS*
- $\mathcal{X}$ *is the set of signals partitioned into inputs ($\mathcal{I}$), observable outputs ($\mathcal{O}bs$) and internal outputs ($\mathcal{I}nt$), and $\Sigma = \mathcal{X} \times \{+, -\} \cup \{\varepsilon\}$, where all transitions not labeled with the silent event (ε) are interpreted as signal changes*
- $\lambda : S \to \mathbb{B}^{|\mathcal{X}|}$ *is the state encoding function*

Figure 2.8(b) shows the SG of the bus controller. We will denote by $\lambda_x(s)$ the value of signal x in state s. The following definitions relate signal

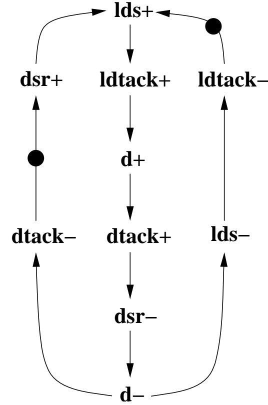


Figure 2.9: Signal Transition Graph specifying the bus controller.

transitions with states. For the sake of simplicity, the definitions assume systems without silent events. Chapter 4 presents the conditions under which a silent event can be removed while preserving the implementability conditions.

2.6.4 Signal Transition Graphs

As in the case of the RTS model, events of a RPN can represent signal changes of an asynchronous circuit. The model is called *Signal Transition Graph* [69].

Definition 2.6.2 *A Signal Transition Graph (STG) is a 3-tuple* $(N, \mathcal{X}, \Lambda)$, *where*

- $N = ((P, T, F, m_0), \Sigma, \Lambda)$ *is a RPN.*
- $\mathcal{X}$ *and* Σ *are defined as in Definition 2.6.1.*

An example of STG specifying the bus controller is shown in Figure 2.9. Places of the STG with only one predecessor and one successor transition, are not shown graphically as convention. The RTS associated to an STG is an SG. The SG associated to the STG of Figure 2.9 is shown in 2.8(b).

2.6.5 Synthesis of Speed-Independent Circuits

Speed-independent (SI) circuits is the class of asynchronous circuits that work correctly regardless of the delay of their components (gates). Currently, there is a robust theory, design flow and some tools [22] that support the automatic synthesis of SI circuits.

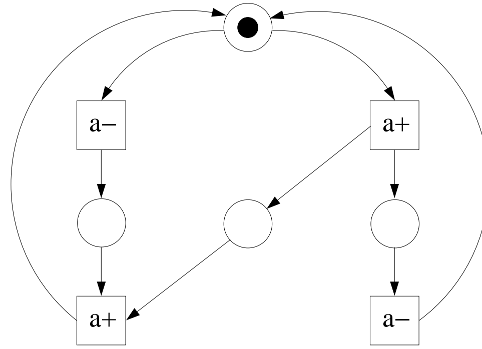


Figure 2.10: Unbounded and unconsistent STG.

However, one of the major problems of the methods used for synthesis is that they require an explicit knowledge of the state graph. Highly concurrent systems often suffer the state explosion problem and, for this reason, the size of the state graph can be a limiting factor for the practical application of synthesis methods.

In this section, some basic concepts on the logic synthesis of SI circuits are presented. We refer the reader to [22] for a deeper theory on how to implement SI circuits. Here, we will focus on explaining the main step in synthesis: the derivation of the Boolean equations that model the behavior of the digital circuit.

Implementability as a Logic Circuit

A set of properties that guarantee the existence of a SI circuit is introduced below. They are defined at the level of SG, but can be easily extended to STGs. Instead of giving new definitions for STGs, we will simply consider that a property holds in an STG if it holds in its underlying SG.

The properties are the following: boundedness, consistency, complete state coding and output persistency.

Boundedness.

A necessary condition for the implementability of a logic circuit is that the set of states is finite. Although this seems to be an obvious assumption at the level of SG, it is not so obvious at the level of STG, since an STG with a finite structure may have a infinite number of reachable markings.

Figure 2.10 shows an example of unbounded STG: the infinite sequence $a+\, a-\, a+\, a- \ldots$ never reaches a marking twice, and therefore the underlying SG is infinite.

Consistency.

As shown in Figure 2.8, each signal x_i defines a partition of the set of states. The consistency of an SG refers to the fact that the events x_i+ and x_i- are the only ones that cross these two parts according to their meaning: switching from 0 to 1 and from 1 to 0, respectively. This is captured by the definition of consistent SG.

Definition 2.6.3 (Consistent SG) *An SG is* consistent *if for each transition $s \stackrel{e}{\to} s'$ the following conditions hold:*

- *if $e = x_i+$, then $\lambda_i(s) = 0$ and $\lambda_i(s') = 1$;*
- *if $e = x_i-$, then $\lambda_i(s) = 1$ and $\lambda_i(s') = 0$;*
- *in all other cases, $\lambda_i(s) = \lambda_i(s')$.*

where λ_i denotes the component of the encoding vector corresponding to signal x_i.

The STG of Figure 2.10 is not consistent: in the initial marking, both transitions $a+$ and $a-$ are enabled, and therefore the initial marking belongs to both the set of states where a is 0 and the set of states where as is 1. This implies that no partition can be made for signal a.

Complete State Coding

This property can be illustrated with the example of Figure 2.8(b), in which there are two states with the same binary encoding: 10101. Moreover, the states with the same binary code are behaviorally different. This fact implies that the system does not have enough information to determine how to react by only looking at the value of its signals.

The distinguishability of behavior by state encoding is captured by the following two definitions.

Definition 2.6.4 (Unique State Coding) *[20] An SG satisfies the* Unique State Coding (*USC*) *condition if every state in S is assigned a unique binary code. Formally, USC means that the state encoding function, λ, is injective.*

Definition 2.6.5 (Complete State Coding) *[20] An SG satisfies the* Complete State Coding (*CSC*) *condition if for every pair of states $s, s' \in S$ having the same binary code the sets of enabled non-input signals are the same.*

Both properties are sufficient to derive the Boolean equations for the synthesized circuit. However, given that only the behavior of the non-input signals must be implemented, encoding ambiguities for input signals are acceptable.

Output persistency

This property is required to ensure that the discrete behavior modeled with SG has a robust correspondence with the real analog behavior of electronic circuits.

Definition 2.6.6 (Disabling) *An event x is said to* disable *another event y if there is a transition $s \xrightarrow{x} s'$ such that y is enabled in s but not in s'.*

Definition 2.6.7 (Output persistency) *An SG is said to be output persistent if for any pair of events x and y such that x disables y, both x and y are input events.*

In logic circuits, disabling an event may result in non-deterministic behavior. Imagine, for example, that an AND gate has both inputs at 1 and the output at 0. In this situation, the gate starts the process to switch the signal towards 1 in a continuous way. If one of the inputs would fall to 0 during this process, the output would interrupt this process and start moving the signal to 0, thus producing an observable glitch. To avoid these situations, that may produce unexpected events, the property of output persistency is required.

Deriving Boolean equations

The procedure to derive Boolean next-state functions for output signals from an SG is introduced. The procedure defines an incompletely specified function from which a gate implementation can be obtained after Boolean minimization.

The next definition will be used later to explain how to derive Boolean equations from an SG under the SI assumptions.

Definition 2.6.8 (Excitation and quiescent regions) *The positive and negative* excitation regions (ER) *of signal $x \in \mathcal{X}$, denoted by $\mathsf{ER}(x+)$ and $\mathsf{ER}(x-)$, are the sets of states in which $x+$ and $x-$ are enabled, respectively, i.e.*

$$\begin{aligned} \mathsf{ER}(x+) &= \{s \in S \mid \exists s \xrightarrow{x+} s' \in T\} \\ \mathsf{ER}(x-) &= \{s \in S \mid \exists s \xrightarrow{x-} s' \in T\} \end{aligned}$$

The positive and negative quiescent regions (QR) *of signal $x \in \mathcal{X}$, denoted by $\mathsf{QR}(x+)$ and $\mathsf{QR}(x-)$ are the sets of states in which x has the same value, 1 or 0, and is stable, i.e.*

$$\begin{aligned} \mathsf{QR}(x+) &= \{s \in S \mid \lambda_x(s) = 1 \ \wedge \ s \notin \mathsf{ER}(x-)\} \\ \mathsf{QR}(x-) &= \{s \in S \mid \lambda_x(s) = 0 \ \wedge \ s \notin \mathsf{ER}(x+)\} \end{aligned}$$

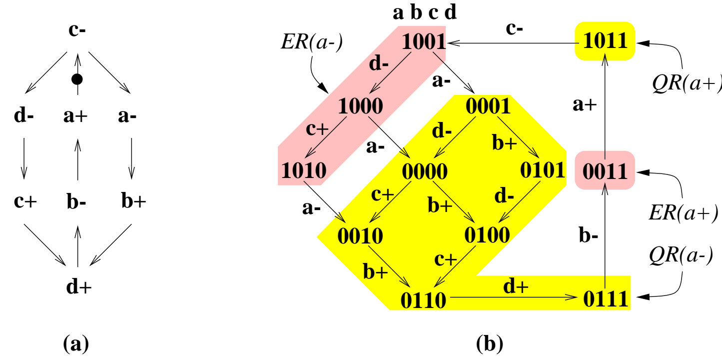


Figure 2.11: Example *abcd*: (a) Signal Transition Graph, (b) State Graph

An incompletely specified n-variable *logic function* is a mapping $F : \mathbb{B}^n \rightarrow \{0, 1, -\}$. Each element $\mathbb{B}^n$ is called a *vertex* or binary code. A *literal* is either a variable x_i or its complement $\overline{x_i}$. A *cube* c is a set of literals, such that if $x_i \in c$ then $\overline{x_i} \notin c$ and vice versa. Cubes are also represented as an element $\{0, 1, -\}^n$, in which value 0 denotes a complemented variable $\overline{x_i}$, value 1 denotes a variable x_i, and $-$ indicates the fact that the variable is not in the cube. A *cover* is a set of implicants which contains the on-set and does not intersect with the off-set.

Given a specification with n signals, the derivation of an incompletely specified function F^x for each output signal x and for each $v \in \mathbb{B}^n$ can be formalized as follows:

$$F^x(v) = \begin{cases} 1 & \text{if } \exists\ s \in \mathsf{ER}(x+) \cup \mathsf{QR}(x+) : \lambda(s) = v \\ 0 & \text{if } \exists\ s \in \mathsf{ER}(x-) \cup \mathsf{QR}(x-) : \lambda(s) = v \\ - & \text{if } \nexists\ s \in S : \lambda(s) = v \end{cases}$$

The set of vertices in which $F^x(v) = 1$ is called the on-set of signal x ($\mathsf{ON}(x)$), whereas the codes in which $F^x(v) = 0$ is called the off-set of x (($\mathsf{OFF}(x)$).

The previous definition is ambiguous when there are two states, s_1 and s_2, for which $\lambda(s_1) = \lambda(s_2) = v$, $s_1 \in \mathsf{ER}(x+) \cup \mathsf{QR}(x+)$ and $s_2 \in \mathsf{ER}(x-) \cup \mathsf{QR}(x-)$. This ambiguity is precisely what the CSC property avoids, and this is why CSC is a necessary condition for implementability.

Figure 2.11 depicts an STG and the corresponding SG. Figure 2.12 shows the Karnaugh maps of the incompletely specified functions for signals a and d, and its corresponding implementation with logic gates.

Notice that in the implementation of signal a, a three-input AND gate is used. When no restriction is imposed in the fan-in (number of inputs) of the gates used for the implementation of a signal, it is called *complex gate implementation*. It assumes a universal library of gates where the designer can find logic gates of arbitrary fan-in. When the library does not contains some of the gates used in the implementation, a process of *technology map-*

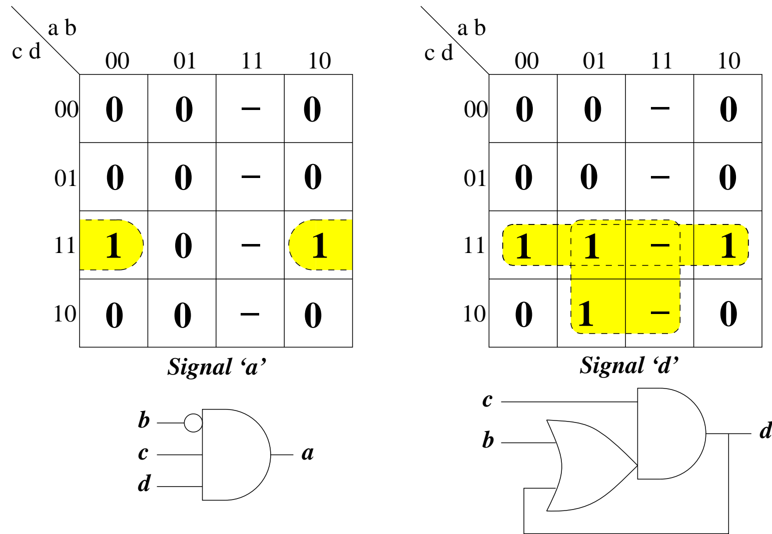


Figure 2.12: Complex gate implementation for the *abcd* example

ping must be done to accomplish the implementation with the existing gates in the library [22].

The complex gate implementation is not the only one that can be obtained: from the incompletely specified functions, other circuit architectures can also be derived [22].

Chapter 3

Compatibility of Reactive Systems

Jugadoras, jugadores
esclavas y patrones
enciende la luz, si quieres ver algo
te ensucias fácil jugando en el barro.
– Mala Rodriguez, *Jugadoras, jugadores*

The synthesis of a reactive system is a complex task, specially when the system is large and/or highly concurrent [36]. One of the possibilities to overcome this difficulty relies on decomposing the system into different subsystems that interact according to some specified protocol. The functionality of the overall system is then obtained by the *composition* of the functionalities of the subsystems [37, 40]. From a software engineering point of view, this allows to distribute the task of implementing the complete system into different designers, with the only restriction that each subsystem must fulfill the protocol.

Therefore one of the crucial points in the synthesis of a reactive system is to be able to decide whether a set of subsystems can be composed and *interact* according to some specified protocol. For that purpose, it is necessary first to define what does interaction mean, when is the interaction correct and what properties do we want to have in this dialogue.

The following questions are answered in this chapter:

- *When can two reactive systems be connected and interact ?*
- *When is this interaction correct ?*
- *Is this problem decidable ?*

This chapter is based on the results presented in [12, 13].

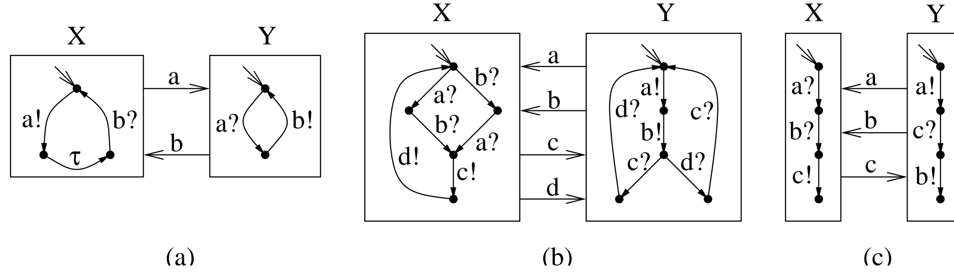


Figure 3.1: Connection between different reactive systems (the suffixes ? and ! are used to denote input and output events, respectively).

3.1 Introduction

The notion we want to model is *Input/Output compatibility*. It is inspired in the work by Dill [27]. We now illustrate this notion with some examples and show why other equivalences for concurrent systems are not appropriate.

Figure 3.1(a) depicts two reactive systems, X and Y, synchronized by a pair of events, a and b. Event a is an output for X and an input for Y, whereas b is an input for X and an output for Y. Moreover, X has an internal event τ. When enabled, internal and output events may take an unbounded, but finite, delay to fire. At each state, a system has only a (possibly empty) subset of input events enabled. If a non-enabled input is produced by the other partner, a communication failure is produced.

The transition systems in Fig. 3.1(a) are observational equivalent. However, they are not I/O compatible, according to the notion presented here. In the initial state, only event a (produced by X) is enabled. After firing a synchronously in both systems, a new state is reached. In this state, Y is ready to produce b. However, X is not ready to accept b before τ is produced and, thus, a communication failure occurs when Y fires b and X has not fired τ yet. Therefore, observational equivalence does not imply I/O compatibility.

Figure 3.1(b) shows that I/O compatibility does not imply observational equivalence. The synchronization of X and Y through the input and output events produces the following language: $(abcd)^*$. In the initial state, X is ready to accept a and b in any order, i.e. they can fire concurrently. However, Y produces a and b sequentially. This situation is reversed for events c and d, accepted concurrently by Y but produced sequentially by X. In either case, the synchronization between X and Y is correct and both systems can interact without any failure. However, it is easy to see that X and Y are not observationally equivalent.

Figure 3.1(c) depicts another undesired situation. After having produced event a, both systems block waiting for each other to fire some event. Thus, a deadlock is produced. This interaction would be considered "fair" in I/O

automata theory [51].

There is another situation not acceptable for I/O compatible systems: livelock. This situation occurs when one of the systems can manifest an infinite internal behavior without any interaction with the other partner. Livelock-freeness can be checked in polynomial time for finite RTSs.

I/O compatibility is inspired in the notion of *conformation*, introduced by Dill [27]. Conformation models the fact that a specification is correctly realized by a given implementation. In Dill's work, the systems specified are digital circuits, and the mathematical model used is called *trace structure*. A *complete trace structure* is a four-tuple containing the set of input signals (I), the set of output signals (O), the set of traces leading to a success (S) and the set of traces leading to a failure (F), with $S, F \subseteq (I \cup O)^{\infty}$. A complete trace structure models complete executions of a circuit, allowing to express liveness properties. The formal model for specifying a system considered here, RTS, is more restricted that the one presented by Dill for complete trace structures. However, the properties covered by our model, including some notion of liveness, can be checked in polynomial time, whereas it is PSPACE-complete for complete trace structures.

I/O automata [51] is a model similar to RTS. In fact, any RTS can be expressed as an I/O automata by including a failure state that is the sink of transitions labeled with the input events not enabled at each state. In [51], a notion of *automata satisfaction* is presented, expressing when an I/O automata specification is correctly implemented by another I/O automata. The main difference between their satisfaction notion and the notion presented in this chapter is that we guarantee the absence of deadlock situations in the dialogue between two I/O compatible systems. Moreover, the fact that systems are assumed to be livelock-free allows a local definition of the I/O compatibility, in contrast to the trace-based definition in I/O automata. I/O compatibility has also relations with other equivalences like *testing equivalence* [24], built-in at CIRCAL [34].

In the area of asynchronous systems, several authors have defined different relations to model the concepts of refinement and realization [11, 82, 60, 77, 41]. Among them, we emphasize the one proposed by Brzozowski and Seger [11]. They introduced formally the concept of *input-properness* and defined a realization notion stronger than I/O compatibility, that requires language equivalence.

Finally, Verhoeff proposed the XDI refinement for delay-insensitive systems. This type of refinement assumes that the dialogue between two systems is produced by introducing any arbitrary delay in the communication, i.e. an event is received some time later than it is produced. Analogously to [27], the expressive power of the XDI model allows to include progress concerns in the model. Differently to the RTS model, the XDI model can not express internal progress (only input/output events are allowed in the model).

3.2 Properties of Reactive Transition Systems

Depending on the interpretation of the events in an RTS, different properties can be defined.

Definition 3.2.1 (Livelock) *A livelock is an infinite trace of only internal events. An RTS is* livelock-free *if it has no livelocks.*

Livelocks can be detected in polynomial time in finite RTSs. The problem is reduced to the detection of cycles in a graph in which only the edges labeled with internal events are taken into account.

Definition 3.2.2 (Input-properness) *An RTS is* input-proper *when for every internal transition $s \stackrel{e}{\rightarrow} s'$, with $e \in \Sigma_{INT}$ and for every input event $i \in \Sigma_I$, $\mathsf{En}(s', i) \implies \mathsf{En}(s, i)$.*

In other words, input-properness is a property that indicates that the enabledness of an input event in a given state depends only on the observable trace leading to that state. Input-properness was introduced in [11] and is a crucial concept to preserve I/O compatibility, as shown later in Sect. 3.5. It avoids the situations in which the system is doing some "pending" internal work when the environment is producing an input event.

The underlying idea of input-properness was previously presented by Dill [27] when, as a result of hiding an output signal, the same trace could be considered both as success and failure.

Definition 3.2.3 (Mirror) *The* mirror *of A, denoted by $\overline{A}$, is another RTS identical to A, but in which the input and output alphabets of A have been interchanged.*

3.3 I/O Compatibility.

A formal description of the conditions needed for having a correct dialogue between two RTSs is given in this section. We call this set of conditions *I/O compatibility*. The properties derived from the I/O compatibility can be stated in natural language:

(a) Safeness: if system A can produce an output event, then B must be prepared to accept the event.

(b) Liveness: if system A is blocked waiting for a synchronization with B, then B must produce an output event in a finite period of time.

Theorems 3.3.1, 3.3.2 and 3.3.3 presented below define formally this properties.

Two RTSs are *structurally I/O-compatible* if they share the observational set of events, in a way that they can be connected.

Definition 3.3.1 (Structural I/O Compatibility) *Let* $A = (S^A, \Sigma^A, T^A, s^A_{in})$ *and* $B = (S^B, \Sigma^B, T^B, s^B_{in})$ *be two* RTSs*. A and B are* structurally I/O compatible *if* $\Sigma^A_I = \Sigma^B_O$, $\Sigma^A_O = \Sigma^B_I$, $\Sigma^A \cap \Sigma^B_{INT} = \emptyset$ *and* $\Sigma^B \cap \Sigma^A_{INT} = \emptyset$.

The following definition gives a concise formalization of the conditions needed for characterizing the correct interaction of two RTSs:

Definition 3.3.2 (I/O Compatibility) *Let* $A = (S^A, \Sigma^A, T^A, s^A_{in})$ *and* $B = (S^B, \Sigma^B, T^B, s^B_{in})$ *be two structurally I/O compatible* RTSs*. A and B are* I/O compatible*, denoted by* $A \rightleftharpoons B$*, if A and B are livelock-free and there exists a relation* $R \subseteq S^A \times S^B$ *such that:*

1. $s^A_{in} R s^B_{in}$.
2. Receptiveness (output events of one party are expected by the other party):
   (a) *If* $s_1 R s'_1$, $e \in \Sigma^A_O$ *and* $s_1 \xrightarrow{e} s_2$ *then* $\mathsf{En}(s'_1, e)$ *and* $\forall s'_1 \xrightarrow{e} s'_2 : s_2 R s'_2$.
   (b) *If* $s_1 R s'_1$, $e \in \Sigma^B_O$ *and* $s'_1 \xrightarrow{e} s'_2$ *then* $\mathsf{En}(s_1, e)$ *and* $\forall s_1 \xrightarrow{e} s_2 : s_2 R s'_2$.
3. Internal Progress (internal process preserves the interaction):
   (a) *If* $s_1 R s'_1$, $e \in \Sigma^A_{INT}$ *and* $s_1 \xrightarrow{e} s_2$ *then* $s_2 R s'_1$.
   (b) *If* $s_1 R s'_1$, $e \in \Sigma^B_{INT}$ *and* $s'_1 \xrightarrow{e} s'_2$ *then* $s_1 R s'_2$.
4. Deadlock-freeness (both parties can not be blocked at the same time):
   (a) *If* $s_1 R s'_1$ *and* $\{e \mid \mathsf{En}(s_1, e)\} \subseteq \Sigma^A_I$ *then* $\{e \mid \mathsf{En}(s'_1, e)\} \not\subseteq \Sigma^B_I$.
   (b) *If* $s_1 R s'_1$ *and* $\{e \mid \mathsf{En}(s'_1, e)\} \subseteq \Sigma^B_I$ *then* $\{e \mid \mathsf{En}(s_1, e)\} \not\subseteq \Sigma^A_I$.

Let us consider the examples of Fig. 3.1. In Fig. 3.1(a), the receptiveness condition fails and therefore X and Y are not I/O compatible. However, the RTSs of Fig. 3.1(b) are I/O compatible. Finally, Fig. 3.1(c) presents an example of violation of the deadlock-freeness condition.

Condition 4 has a strong impact on the behavior of the system. It guarantees that the communication between A and B has no deadlocks (see theorem 3.3.3).

Lemma 3.3.1 *Let A and B be two* RTSs *such that* $A \rightleftharpoons B$*, let R be an I/O compatible relation between A and B and let* $A \times B = (S, \Sigma, T, s_{in})$ *be the synchronous product of A and B. Then,* $\langle s, s' \rangle \in S \Rightarrow sRs'$

Proof: If $\langle s, s' \rangle \in S$, then there is a trace σ that leads from s_{in} to $\langle s, s' \rangle$. We prove the lemma by induction on the length of σ.

- Case $|\sigma| = 0$. The initial states are related in Condition 1 of Definition 3.3.2.

- Case $|\sigma| > 0$. Let $\sigma = \sigma' e$, with $|\sigma'| = n$, and assume that it holds for any trace up to length n. Let $\langle s_1, s_1' \rangle$ be the state where the event e is enabled. The induction hypothesis ensures that s_1 is I/O compatible to s_1'. Two situations can happen in s_1 depending on the last event e of σ: either 1) $e \in \Sigma_O \cup \Sigma_{INT}$ is enabled in s_1, or 2) only input events are enabled in s_1. In situation 1), Conditions 2-3 of Definition 3.3.2 guarantee that s is I/O compatible to s'. In situation 2), applying Condition 4 of Definition 3.3.2 ensure that some non-input event is enabled in state s_1' of B. Definition 2.3.5 and Conditions 2-3 on s_1' and the enabled non-input event e guarantees s to be I/O compatible to s'.

□

Theorem 3.3.1 (Safeness) *Let A and B be two RTSs such that $A \rightleftharpoons B$, and a trace $\sigma \in L(A \times B)$ of their synchronous product such that $s_{in} \xrightarrow{\sigma} \langle s, s' \rangle$. If A can fire an output event in s, then the same event is enabled in state s' of B.*

Proof: It immediately follows from Lemma 3.3.1 and the condition of receptiveness in the definition of I/O compatibility. □

Theorem 3.3.2 (Absence of Livelocks) *Let A and B be two RTSs such that $A \rightleftharpoons B$, and let $A \times B$ be the synchronous product of A and B. Then, $A \times B$ is livelock-free.*

Proof: The definition of synchronous product implies that only livelocks appear in $A \times B$ if either A or B has a livelock. But A and B are livelock-free because $A \rightleftharpoons B$. □

The following theorem is the one that proves the absence of deadlocks produced by the interaction between two I/O compatible RTSs.

Theorem 3.3.3 (Liveness) *Let A, B be two RTSs such that $A \rightleftharpoons B$, and a trace $\sigma \in L(A \times B)$ of their synchronous product such that $s_{in} \xrightarrow{\sigma} \langle s, s' \rangle$. If only input events of A are enabled in s, then there exists some trace $\langle s, s' \rangle \xrightarrow{\sigma'} \langle s, s'' \rangle$ such that some of the input events of A enabled in s are also enabled in s'' as output events of B.*

Proof: By Lemma 3.3.1 we have that sRs'. We also have that $\{e \mid \mathsf{En}(s, e)\} \subseteq \Sigma_I^A$. By Condition 4 of Definition 3.3.2 we know that $\{e \mid \mathsf{En}(s_1', e)\} \nsubseteq \Sigma_I^B$. Theorem 3.3.2 guarantees the livelock-freeness of $A \times B$, and therefore from $\langle s, s' \rangle$ there exists a trace of internal events reaching a state $\langle s, s'' \rangle$ where no

internal event is enabled. We know by Lemma 3.3.1 that sRs''. Condition 4 of Definition 3.3.2, together with the fact that no internal event is enabled in s'' implies that there exists an output event enabled in s'', which is enabled as input in s. □

3.4 A Polynomial-time Decision Procedure for I/O Compatibility

A procedure for deciding if two finite RTS are I/O compatible is presented in this section. It is based on the synchronous product of transition systems.

Theorem 3.4.1 *Let $A = (S^A, \Sigma^A, T^A, s^A_{in})$, $B = (S^B, \Sigma^B, T^B, s^B_{in})$ be two livelock-free RTSs. $A \rightleftharpoons B$ iff $A \times B = (S, \Sigma, T, s_{in})$ fulfills the following properties:*

1. *(a) For each state $s \in S^A$, for each event $e \in \Sigma^A_O$:*
   if $\mathsf{En}(s, e)$ holds and $\langle s, s' \rangle \in S$ then $\mathsf{En}(\langle s, s' \rangle, e)$ holds.

   (b) For each state $s' \in S^B$, for each event $e \in \Sigma^B_O$:
   if $\mathsf{En}(s, e)$ holds and $\langle s, s' \rangle \in S$ then $\mathsf{En}(\langle s, s' \rangle, e)$ holds.

2. *For every $\langle s, s' \rangle \in S$, if $\langle s, s' \rangle \in S$ is a terminal state, then s and s' are terminal states in A and B, respectively.*

Proof: The proof is divided into two parts:

Sufficiency.
Let R be an I/O compatibility relation between A and B and $\langle s, s' \rangle \in S$. Lemma 3.3.1 guarantees that sRs'.

1. Since sRs', then $\mathsf{En}(s', e)$ holds in B. By the definition of synchronous product, $\mathsf{En}(\langle s, s' \rangle, e)$ holds. (Similarly for 1(b)).

2. Every non-input event e enabled in s or s' induces e to be enabled in $\langle s, s' \rangle$. If only input events are enabled in one of the states, condition 4 of Definition 3.3.2 guarantees the enabling in the other state of a non-input event, and the definition of synchronous product ensures the existence of a transition leaving from $\langle s, s' \rangle$.

Necessity.
We will proof that S is an I/O compatible relation between A and B. State $\langle s^A_{in}, s^B_{in} \rangle$ belongs to S by definition of synchronous product. Let $\langle s, s' \rangle \in S$. Property 1, together with the definition of synchronous product implies the receptiveness condition of Definition 3.3.2. Condition 3 (internal progress)

of Definition 3.3.2 holds by the definition of synchronous product: every internal event e enabled in s (s') is also enabled in $\langle s, s' \rangle$, and the state(s) of S reached by the firing of e in $\langle s, s' \rangle$ are exactly the pairs of I/O compatible states induced by Condition 3 with s and s'. Condition 4 (deadlock-freeness) of Definition 3.3.2 also holds: if the events enabled in s are input events, then given that $\langle s, s' \rangle$ is not terminal (due to Property 2), the only possibility for having an event enabled in $\langle s, s' \rangle$ in Definition 2.3.5 is when a non-input event is enabled in s'. □

Theorem 3.4.1 enables the use of the synchronous product for deciding the I/O compatibility of two finite RTSs in polynomial-time[1]. It consists in computing the synchronous product in the first step, and then checking the conditions 1 and 2 of the theorem.

3.5 I/O Compatibility and Observational Equivalence.

In the first part of this section, the *observational equivalence* relation [55] is defined. Section 3.5.2 presents the relationship between I/O compatibility and observational equivalence.

3.5.1 Observational Equivalence

The *observational equivalence* relation between two reactive systems was first introduced by Milner in [55]. The relation identifies those systems whose observable behavior is indistinguishable.

Definition 3.5.1 *Let $A = (S^A, \Sigma^A, T^A, s^A_{in})$ and $B = (S^B, \Sigma^B, T^B, s^B_{in})$ be two RTSs. A and B are* observational equivalent *($A \approx B$) iff $\Sigma^A_{OBS} = \Sigma^B_{OBS}$ and there exists a relation $R \subseteq S \times S'$ satisfying*

1. $s^A_{in} R s^B_{in}$.
2. (a) $\forall s \in S^A$, $\exists s' \in S^B$ *s.t.* sRs'.
   (b) $\forall s' \in S^B$, $\exists s \in S^A$ *s.t.* sRs'.
3. (a) $\forall s_1 \in S^A$, $s'_1 \in S^B$: *if* $s_1 R s'_1$, $e \in (\Sigma^A_{OBS})$ *and* $s_1 \xrightarrow{e} s_2$ *then* $\exists \sigma_1, \sigma_2 \in (\Sigma^B_{INT})*$ *such that* $s'_1 \xrightarrow{\sigma_1 e \sigma_2} s'_2$, *and* $s_2 R s'_2$.
   (b) $\forall s_1 \in S^A$, $s'_1 \in S^B$: *if* $s_1 R s'_1$, $e \in (\Sigma^A_{OBS})$ *and* $s'_1 \xrightarrow{e} s'_2$ *then* $\exists \sigma_1, \sigma_2 \in (\Sigma^A_{INT})*$ *such that* $s_1 \xrightarrow{\sigma_1 e \sigma_2} s_2$, *and* $s_2 R s'_2$.

[1]Figure 3.5 shows why it is necessary to consider only livelock-free RTSs in Theorem 3.4.1. Systems 1 and 2 are I/O compatible, but System 1 could have a livelock in the state reached after the sequence $b\tau_1 a$.

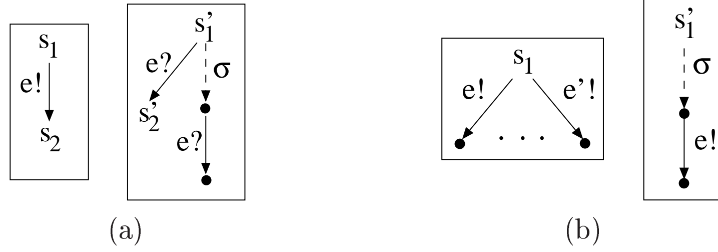


Figure 3.2: Conditions 2(a) and 4(a) from the proof of Theorem 3.5.1.

The two RTSs of Fig. 3.1(a) are observational equivalent, because every observable sequence of one of them can be executed in the other. Figures 3.1(b)-(c) depict examples of non-observationally equivalent systems.

3.5.2 A Sufficient Condition for I/O Compatibility.

A sufficient condition for having I/O compatibility between two reactive systems can be obtained when combining the notions of observational equivalence and input-properness:

Theorem 3.5.1 *Let $A = (S^A, \Sigma^A, T^A, s^A_{in})$, $B = (S^B, \Sigma^B, T^B, s^B_{in})$ be two livelock-free RTSs with $\Sigma^A_I = \Sigma^B_O$ and $\Sigma^A_O = \Sigma^B_I$. If A and B are input proper and $A \approx B$, then $A \rightleftharpoons B$.*

Proof: Let R be the relation induced by the observational equivalence between A and B. We will prove that R is also an I/O compatibility relation between A and B. R must fulfill the conditions of the I/O compatibility relation:

- **Condition 1:** $s^A_{in} R s^B_{in}$ by Definition 3.5.1.
- **Condition 2(a):** let $s_1 R s'_1$, and assume $s_1 \stackrel{e}{\to} s_2$, with $e \in \Sigma^A_O$. Figure 3.2(a) depicts the situation. The observational equivalence of s_1 and s'_1 implies that a trace σ of internal events exists in s'_1 enabling e. The event e is an input event in B, and therefore the input-properness of B ensures that in every state s' of σ, $\mathsf{En}(s', e)$ holds. In particular, it also holds in the first state and, thus, $\mathsf{En}(s'_1, e)$. The definition of R ensures that every s'_2 such that $s'_1 \stackrel{e}{\to} s'_2$ is related with s_2 by R.
- **Condition 3(a):** let $s_1 R s'_1$ and assume $s_1 \stackrel{e}{\to} s_2$, with $e \in \Sigma^A_{INT}$. The definition of R implies that $s_2 R s'_1$.
- **Condition 4(a):** let $s_1 R s'_1$, and suppose $\{e \,|\, \mathsf{En}(s_1, e)\} \subseteq \Sigma^A_I$. Figure 3.2(b) depicts the situation. Let e be one of the input events

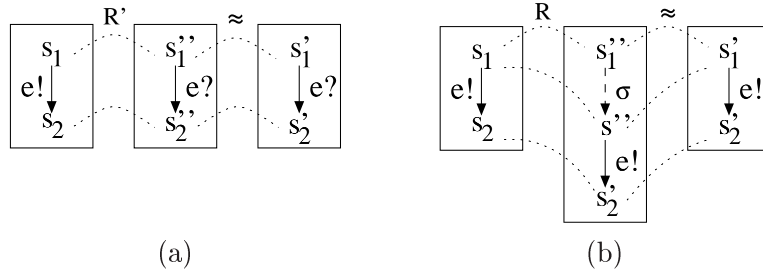


Figure 3.3: Conditions 2(a) and 2(b) from the proof of Theorem 3.5.2.

enabled in s_1. The observational equivalence between s_1 and s'_1 requires that a sequence σ of internal events exists enabling e starting in s'_1, and given that e in not input in B implies $\{e \,|\, \mathsf{En}(s'_1, e)\} \nsubseteq \Sigma_I^B$.

An identical reasoning can be applied in the symmetric cases (conditions 2(b), 3(b) and 4(b)). □

When considering a system A and some I/O compatible system B, any transformation of B preserving both input-properness and observational equivalence will lead to another I/O compatible system:

Theorem 3.5.2 *Let* $A = (S^A, \Sigma^A, T^A, s_{in}^A)$, $B = (S^B, \Sigma^B, T^B, s_{in}^B)$ *and* $C = (S^C, \Sigma^C, T^C, s_{in}^C)$ *be three RTSs. If* $A \rightleftharpoons B$, $B \approx C$, *and* C *is input-proper then* $A \rightleftharpoons C$.

Proof: Let R' be the relation between A and B, and $\approx$ the observational equivalent relation between states from B and C. Define the relation R as:

$$\forall s \in S^A, s'' \in S^B, s' \in S^C : sR's'' \wedge s'' \approx s' \;\Leftrightarrow (s, s') \in R$$

The conditions that R must satisfy are the ones of Definition 3.3.2. Remember that $A \rightleftharpoons B$ implies that $\Sigma_O^B = \Sigma_I^A$ and $\Sigma_I^B = \Sigma_O^A$. Moreover, relation $B \approx C$ implies that $\Sigma_{OBS}^B = \Sigma_{OBS}^C$.

- **Condition 1:** the initial states are related in R by definition.
- **Condition 2(a):** let $s_1 R s'_1$, and suppose $s_1 \stackrel{e}{\to} s_2$ with $e \in \Sigma_O^A$. Figure 3.3(a) depicts the situation. Given that $s_1 R' s''_1$, e is enabled in s''_1 and for each s''_2 such that $s''_1 \stackrel{e}{\to} s''_2$, $s_2 R' s''_2$. The observational equivalence of s''_1 and s'_1, together with the fact that C is input-proper implies that e is also enabled in s'_1 (identical reasoning of condition 2(a) in Theorem 3.5.1), and the definition of $\approx$ implies that each s'_2 such that $s'_1 \stackrel{e}{\to} s'_2$ must be related in $\approx$ with s''_2. Then each s'_2 such that $s'_1 \stackrel{e}{\to} s'_2$ is related by R with s_2.

- **Condition 2(b):** let $s_1 R s_1'$, and suppose $s_1' \xrightarrow{e} s_2'$ with $e \in \Sigma_O^B$. Figure 3.3(b) depicts the situation. The observational equivalence of s_1'' and s_1' implies that there is a sequence σ of internal events starting in s_1'' and enabling e, and every state of σ is observational equivalent to s_1'. Moreover, every state of σ is also related to s_1 by the condition 3(b) of R'. In particular, s_1 is related by R' with the state s'' of σ s.t. $s'' \xrightarrow{e} s_2''$; applying Condition 2(b) of R', $\mathsf{En}(s_1, e)$ holds and for each e s.t. $s_1 \xrightarrow{e} s_2$, $s_2 R' s_2''$. The definition of R and $\approx$ induces that each such s_2 is related with s_2' by R.

- **Condition 3(a):** let $s_1 R s_1'$, and suppose $s_1 \xrightarrow{e} s_2$ with $e \in \Sigma_{INT}^A$. Then Condition 3(a) of R' ensures $s_2 R' s_1''$ and then applying the definition of R implies $s_2 R s_1'$.

- **Condition 3(b):** let $s_1 R s_1'$, and suppose $s_1' \xrightarrow{e} s_2'$ with $e \in \Sigma_{INT}^C$. Then $s_1'' \approx s_2'$, and then $s_1 R s_2'$.

- **Condition 4(a):** let $s_1 R s_1'$, and suppose $\{e|\mathsf{En}(s_1, e)\} \subseteq \Sigma_I^A$. Condition 4(a) of R' ensures that $\{e|\mathsf{En}(s_1'', e)\} \nsubseteq \Sigma_I^B$: let a be an event such that $s_1'' \xrightarrow{a} s_2''$, with $a \notin \Sigma_I^B$. If $a \in \Sigma_O^B$, the related pair $s_1'' \approx s_1'$ ensures that in s_1' there is a feasible sequence of internal events (which can be empty) enabling a, and therefore $\{e|\mathsf{En}(s_1', e)\} \nsubseteq \Sigma_I^C$. If $a \in \Sigma_{INT}^B$, applying Condition 3(b) of R' and the definition of $\approx$, $s_1 R' s_2''$ and $s_1'' \approx s_1'$ is obtained, respectively. The same reasoning applied to s_1, s_1'' and s_1' can now be applied to s_1, s_2'' and s_1'. Given that B is livelock-free, the sequence of internal events starting in s_1'' and passing through s_2'' must end in a state s'' where a observable event a' is enabled. State s'' is also related by R' with s_1, and by $\approx$ with s_1' (applying inductively the same reasoning applied to s_2''). Event a' belongs to Σ_O^B because otherwise a violation of Condition 2(b) in R' arise. The previous case ($a \in \Sigma_O^B$, enabled in s_1'') can be applied to s''.

- **Condition 4(b):** let $s_1 R s_1'$, and suppose $\{e|\mathsf{En}(s_1', e)\} \subseteq \Sigma_I^C$. Let a such that $s_1'' \xrightarrow{a} s_2''$. If $a \in \Sigma_O^B$, then a contradiction arises because $s_1'' \approx s_1'$ and $\{e|\mathsf{En}(s_1', e)\} \subseteq \Sigma_I^C$. If $a \in \Sigma_I^B$, then identical conditions make $\mathsf{En}(s_1', a)$ to hold. If $a \in \Sigma_{INT}^B$, then Conditions 3(a) of R' and $\approx$ ensure that $s_1 R' s_2''$ and $s_2'' \approx s_1'$, and the same reasoning of s_1, s_1' and s_1'' can be applied to s_1, s_1' and s_2'' (but not infinite times, because B is livelock-free). Therefore a feasible sequence of internal events (which can be empty) exist from s_1'' reaching a state s'' such that $\{e|\mathsf{En}(s'', e)\} \subseteq \Sigma_I^C$, with $s_1 R' s''$ and $s'' \approx s_1'$. Condition 4(b) of R' ensures that $\{e|\mathsf{En}(s_1, e)\} \nsubseteq \Sigma_I^A$.

$\square$

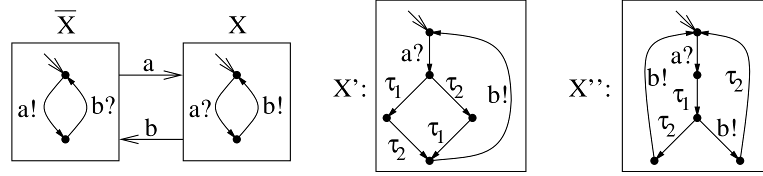


Figure 3.4: Relation between observational equivalence, input-properness and I/O compatibility.

Figure 3.4 shows an example of application of Theorem 3.5.2. The transformation of X which leads to X' preserves both observational equivalence and input-properness, and then, $\overline{X}$ and X' can safely interact.

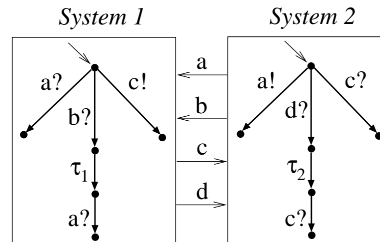


Figure 3.5: Two I/O compatible systems that are not input-proper.

Finally, it must be noted that I/O compatibility does not require input-properness, as shown in Fig. 3.5. This occurs when the non-input-proper situations are not reachable by the interaction of the two systems.

3.6 Conclusion

A characterization of the class of systems that can be connected and interact within a correct dialogue has been presented in this chapter. Afterwards, an algorithm based on the synchronous products has been introduced, providing a polynomial procedure for deciding whether two systems are I/O compatible. Finally the relation between one of the most well-known notions of equivalence over reactive systems, observational equivalence, and I/O compatibility is showed, with the concept of input-properness being the key link between them.

The formal setting of this chapter makes the I/O compatibility notion valid for any type of reactive system that can be described with an automaton. As future work we are interested in adapting the I/O compatibility to more abstract scenarios, like software systems or groupware systems [30]. In groupware systems, the notion of synchronization in sending or receiving a message can be more flexible, and therefore several new problems may

appear in the interaction. Moreover, it is also interesting to investigate the way several systems *collaborate* in sending or receiving a message.

In Chapter 4 we will use the I/O compatibility to derive synthesis rules that allow to transform a system while preserving the correctness of the dialogue with the environment.

Chapter 4

Petri Net Transformations for Synthesis

Coisas pequenas são
coisas pequenas
são tudo o que eu te quero dar
– Madredeus, *Coisas Pequenas*

Very often a reactive system needs to be modified to accomplish some architectural requirement or even simply for its improvement. Such modifications can have a strong impact on the final implementation, leading in the worst case to failures of the whole system.

In this chapter we tackle the problem of transforming the specification of a reactive system while preserving the correct (i.e. I/O compatible) interaction with the environment. Next we present an application of the theory for the case of asynchronous circuits and the problem of the encoding.

This chapter is based on the results presented in [8, 15, 16, 17].

4.1 Introduction

Once the specification of a reactive system is done, the next step is to synthesize it, which amounts to implementing the underlying behavior by using the primitives that exist in the target architecture. Several problems can appear in this process: the initial specification can not be straightforward implementable with the existing primitives, and therefore the designer must transform the initial specification in order to fit to the actual requirements. However, if the system is transformed then some other systems in the environment may need to be changed in order to be able to interact with the new system. This is an undesirable situation because it can lead to a circular chain of transformations, or simply because it is expensive to do when

a small modification was needed and big part of the environment is already functioning.

Petri net transformations is the main topic of discussion in this chapter. The rules presented here have different goals: in Section 4.2, the focus is on defining those transformations of a reactive Petri net that preserve I/O compatibility with its environment. With the help of these transformations, a designer can modify an initial specification for the sake of arbitrary purposes (correctness, improvement, etc ...) while ensuring that there will be no problem in the dialogue between the transformed system and its environment. A particular case, sketched in the first part of this introduction, is when the reactive Petri net is modeling an asynchronous circuit and the purpose is to derive Boolean equations implementing its behavior.

In Section 4.3, we focus in finding a transformation that can be applied to a signal transition graph (a reactive Petri net modeling an asynchronous circuit) that makes the transformed net to be free from encoding problems. As said in Chapter 2, the problem of finding a correct encoding is one of the hard problems when facing the synthesis of a speed-independent circuit.

I/O Compatible Transformations

Here we want to sketch how to use the transformations presented in Section 4.2 for the particular case of the synthesis of an asynchronous circuit. In the example of Figures 4.1(b-d) the goal is to synthesize a circuit that can have a correct dialogue with the environment. We will assume that the components of the circuit have arbitrary delays. Likewise, the environment may take any arbitrary delay to produce any enabled output event.

Let us first have a look at Figure 4.1(b). The marked graph in the environment can be considered as a specification of a concurrent system. The underlined transitions denote input events. Thus, an input event of the environment must have a correspondence with an output event of the system, and vice versa. The behavior denoted by this specification can be informally described as follows:

> *In the initial state, the environment will produce the event $x+$. After that, the environment will be able to accept the events $y+$ and $z+$ concurrently from the system. After the arrival of $z+$, the environment will produce $x-$, that can occur concurrently with $y+$. Next, it will wait for the system to sequentially produce $z-$ and $y-$, thus leading the environment back to the initial state.*

The circuit shown in Figure 4.1(b) behaves as specified by the adjacent marked graph. In this case, the behavior of the system is merely a mirror of the behavior of the environment and then both are observational equivalent. Moreover, given that both the environment and the system are input-proper, Theorem 3.5.1 guarantees that the dialogue between them is compatible.

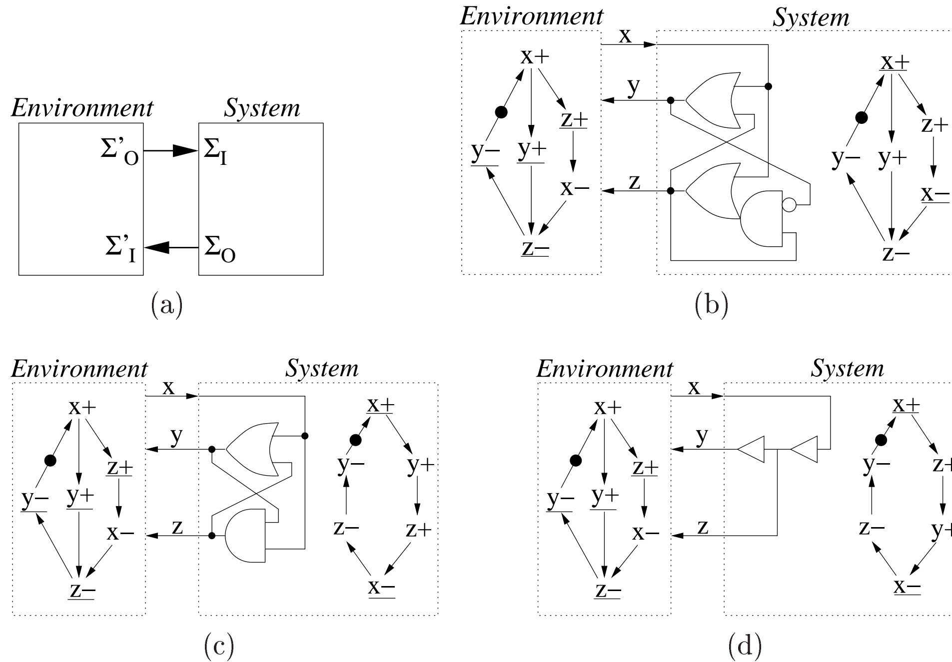


Figure 4.1: (a) Connection between system and environment, (b) mirrored implementation of a concurrent system, (c) valid implementation with concurrency reduction, (d) invalid implementation.

Let us analyze now the system in Figure 4.1(c). In this case, the circuit implements a behavior in which $y+$ and $z+$ are produced sequentially (the transformation adds a place connecting $y+$ to $z+$)[1]. Still, the system can maintain a correct dialogue, since the environment is able to accept more behaviors than the ones produced by the system. We can observe that, even though the behavior is less concurrent, the implementation is simpler.

Let us finally look at Figure 4.1(d), in which the events $z+$, $y+$ and $x-$ are produced sequentially in this order (the transformation adds a place connecting $z+$ to $y+$ and then adds a place connecting $y+$ to $x-$). Due to this reduction in concurrency, two buffers are sufficient to implement such behavior. Even though the set of traces produced by the system is included in the set of traces produced by the environment, the dialogue between both is not correct. To illustrate that, let us assume the events $x+$ and $z+$ have been produced from the initial state. We are now in a state in which $x-$ is enabled in the environment (output event) but not enabled in the system. This violates the receptiveness condition for a correct dialogue: if an output event is enabled in one component, the corresponding event must also be enabled in the other component. In practice, if the environment

[1]For the sake of readability, redundant places are not shown in this example.

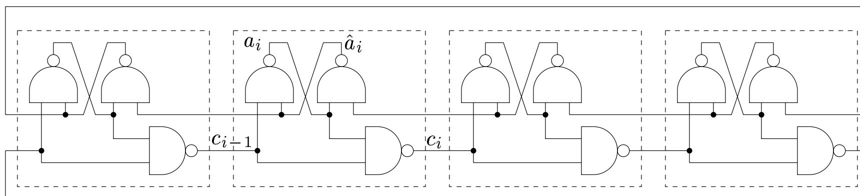


Figure 4.2: Distributor built from David cells [45].

would produce $x-$, the circuit could react with the event $z-$ before $y+$ is produced.

In this example, two Petri net transformations have been applied. It has been explained why one of them is acceptable and one of them is not. In Section 4.2 a kit of Petri net rules is presented, to be applied over the class of free-choice live and safe Petri nets (FCLSPN). Afterwards, the conditions under which the rules are acceptable are introduced, when the specification is a deterministic reactive Petri net.

Encoding Technique

In the speed-independent delay model, the Complete State Coding is a necessary condition for a specification to be implementable as a set logic equations (see Section 2.6.5). Unfortunately, till now no method has been able to effectively tackle the problem of finding an encoding of the specification that guarantees an implementation. Even the known structural methods working for some subclasses of STGs rely on the fact that heuristics with affordable computational cost will find a solution with high probability [81, 64]. In this chapter we present a method that guarantees a correct encoding and works at the level of the Petri net.

The method presented in Section 4.3 has been inspired by previous work for the direct synthesis of circuits from Petri nets. One of the relevant techniques was proposed in [76], where a set of cells that mimic the token flow in Petri nets was designed. The circuit was built by abutting the cells and producing a structure isomorphic to the Petri net. This type of cells, called David cells, were initially proposed in [23].

Figure 4.2 depicts a very simple example on how these cells can be abutted to build a distributor that controls the propagation of activities along a ring. The behavior of one of the cells in the distributor can be summarized by the following sequence of events:

$$\cdots \rightarrow \underbrace{c_{i-1}-}_{\substack{i\text{-th cell}\\ \text{excitation}}} \rightarrow \underbrace{a_i+ \rightarrow \hat{a}_i-}_{i\text{-th cell setting}} \rightarrow$$

$$\rightarrow \underbrace{\hat{a}_{i-1}+ \ \rightarrow \ a_{i-1}- \ \rightarrow \ c_{i-1}+}_{(i-1)\text{-th cell resetting}} \ \rightarrow \ \underbrace{c_i -}_{\substack{(i+1)\text{-th cell} \\ \text{excitation}}} \ \rightarrow \ \cdots$$

In [76], each cell was used to represent the behavior of one of the transitions of the Petri net.

The example of the VME Bus Controller from Section 2.6.3 is used to introduce the methods presented in Section 4.3. From Section 2.6.3, we know that the STG has encoding problems. Figure 4.3 illustrates the method presented in Section 4.3: for each place of the net, a new signal is introduced mimicking the token flow on that place. At the expense of incrementing its size, the resulting net is free from encoding conflicts. The crucial point is that, given that the technique works at the level of the net, it avoids to suffer from the state explosion problem.

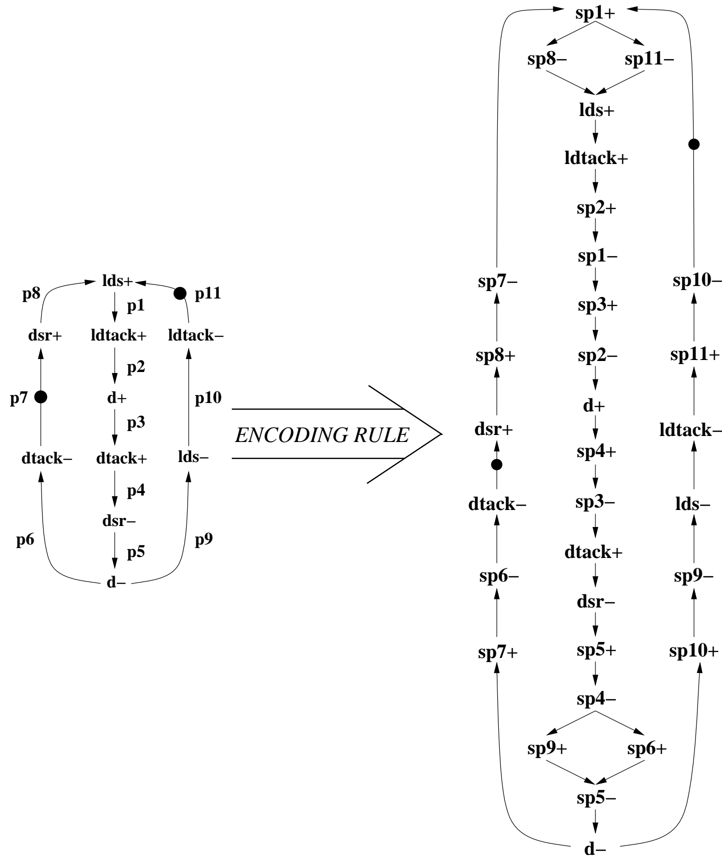


Figure 4.3: Encoding rule applied to the VME Bus Controller example.

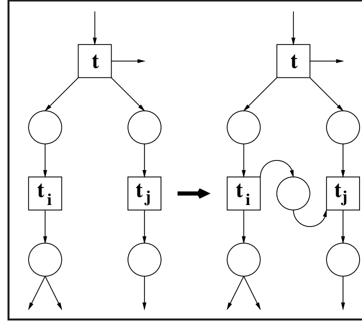
4.2 I/O Compatible Transformations

4.2.1 Kit of PN Transformations

Three rules are presented for modifying the structure of a FCLSPN. The rule ϕ_r is used for sequencing two concurrent transitions. It was first defined in [7]. Here a reduced version is presented. Rule ϕ_i does the opposite: it increases the concurrency between two ordered transitions. ϕ_i can be obtained as a combination of the ones appearing in [57]. Finally, rule ϕ_e removes a given transition. It was first presented in [47]. All three rules preserve the liveness, safeness and free-choiceness.

Rule ϕ_r

The purpose of the rule ϕ_r is to eliminate the concurrency between two transitions of the PN. This is done by inserting a place that connects the two transitions, ordering their firing. The following figure presents an example of concurrency reduction between transitions t_i and t_j[2].



The formal definition of the rule is:

Let $N = (P, T, F, m_0)$, $N' = (P', T, F', m'_0)$ be two FCLSPNs, and transitions $t_i, t_j \in T$. Then, $\phi_r(N, t_i, t_j) = N'$ if:

Conditions on N:

1. $\{t\} = {}^\bullet({}^\bullet t_i) = {}^\bullet({}^\bullet t_j)$
2. ${}^\bullet t_i = \{p_i\} \wedge |p_i^\bullet| = 1$
3. ${}^\bullet t_j = \{p_j\} \wedge |p_j^\bullet| = 1$
4. $m_0(p_i) = m_0(p_j)$

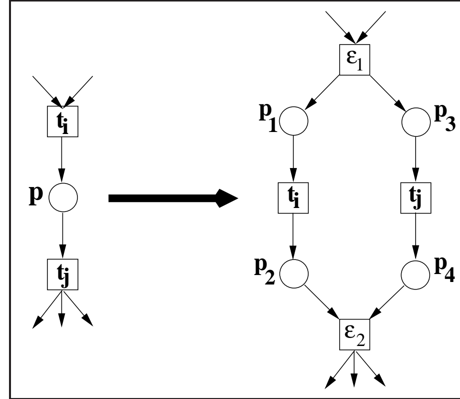
Conditions on N':

1. $P' = P \cup \{p\}$
2. $F' = F \cup \{(t_i, p), (p, t_j)\}$
3. $m'_0|_P = m_0|_P \wedge m'_0(p) = 0$

[2]For the sake of simplicity in the definition of the rules, we will abuse of the notation and use $(x, y) \in F$ and $(x, y) \notin F$ for $F(x, y) = 1$ and $F(x, y) = 0$, respectively.

Rule ϕ_i

Inversely to rule ϕ_r, rule ϕ_i removes the causality relation between two ordered transitions, making them concurrent.The following figure presents an example of increase of concurrency between transitions t_i and t_j.



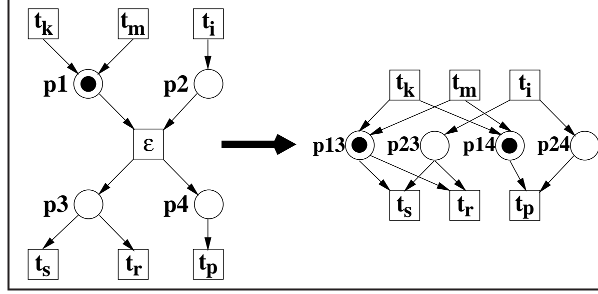
The formal definition of the rule is:

Let $N = (P, T, F, m_0)$, $N' = (P', T', F', m'_0)$ be two FCLSPNs, and transitions $t_i, t_j \in T$. In the following definition, places p'_k represent new places originated from places either in ${}^\bullet t_i$ $(k = i)$ or in $t_j^\bullet$ $(k = j)$. Then, $\phi_i(N, t_i, t_j) = N'$ if:

Conditions on N:	Conditions on N':
1. $\{(t_i, p), (p, t_j)\} \subseteq F$ 2. $\lvert {}^\bullet p \rvert = \lvert p^\bullet \rvert = 1$ 3. $\forall q \in {}^\bullet t_i : \lvert q^\bullet \rvert = 1$ 4. $t_i \notin (t_j^\bullet)^\bullet$	1. $P' = (P \setminus \{p\}) \cup \{p'_i \mid p_i \in {}^\bullet t_i\} \cup \{p'_j \mid p_j \in t_j^\bullet\}$ 2. $F' = (F \setminus \{(t_i, p), (p, t_j)\}) \cup \{(y, p'_i) \mid (y, p_i) \in F\} \cup \{(p'_i, t_j) \mid (p_i, t_i) \in F\} \cup \{(p'_j, y) \mid (p_j, y) \in F\} \cup \{(t_i, p'_j) \mid (t_j, p_j) \in F\} \cup \{(y, p'_j) \mid (y, p_j) \in F \wedge y \neq t_j\}$ 3. $m'_0\lvert_{(P \setminus \{p\})} = m_0\lvert_{(P \setminus \{p\})} \wedge \forall k : m'_0(p'_k) = m_0(p_k) + m_0(p)$

Rule ϕ_e

The rule ϕ_e eliminates a transition from the PN. The following figure presents an example of elimination of transition ε.



The formal definition of the rule is:

Let $N = (P, T, F, m_0)$, $N' = (P', T', F', m'_0)$ be two FCLSPNs, transition $\varepsilon \in T$ and let $P_\varepsilon = ({}^\bullet\varepsilon) \times (\varepsilon^\bullet)$. Then, $\phi_e(N, \varepsilon) = N'$ if:

Conditions on N:	Conditions on N':
1. $\forall p : p \in {}^\bullet\varepsilon : p^\bullet = \{\varepsilon\}$	1. $P' = (P \setminus ({}^\bullet\varepsilon \cup \varepsilon^\bullet)) \cup P_\varepsilon$ 2. $T' = T \setminus \{\varepsilon\}$ 3. $F' = (F \setminus \{ (a,b) \mid (a,b) \in F \wedge (a = \varepsilon \vee b = \varepsilon)\}) \cup \{(y, \langle p_1, p_2\rangle) \mid (y, p_1) \in F\} \cup \{(\langle p_1, p_2\rangle, y) \mid (p_2, y) \in F\}$ 4. $m'_0\|_{P \setminus ({}^\bullet\varepsilon \cup \varepsilon^\bullet)} = m_0\|_{P \setminus ({}^\bullet\varepsilon \cup \varepsilon^\bullet)} \wedge \forall \langle p_1, p_2\rangle \in P_\varepsilon : m'_0(\langle p_1, p_2\rangle) = m_0(p_1) + m_0(p_2)$

4.2.2 I/O Compatible Transformations over RPN

The I/O compatible relation operator ($\rightleftharpoons$) can be lifted to RPNs:

Definition 4.2.1 (I/O compatible relation over RPN) *Let A and B be two RPNs with corresponding RTSs RTS(A) and RTS(B). $A \rightleftharpoons B$ if RTS$(A) \rightleftharpoons$ RTS(B).*

For each transformation of the kit presented in Section 4.2.1, the following sections enumerate those situations where the transformation can be applied to the underlying FCLSPN of a deterministic RPN while preserving the I/O compatible relation.

I/O compatible application of ϕ_r

The application of $\phi_r(A, e_1, e_2)$ preserves $\rightleftharpoons$ when neither e_1 nor e_2 is an input transition. In fact, it is sufficient to require only e_2 to be non-

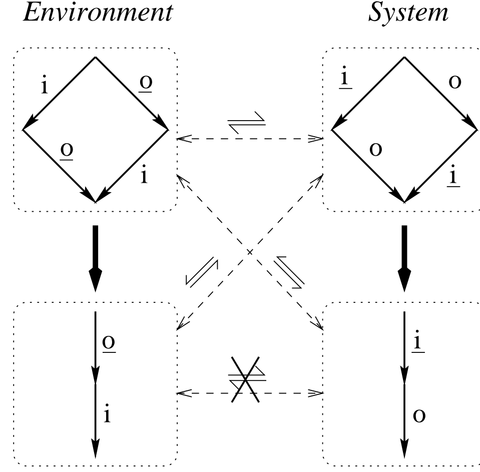


Figure 4.4: Different possibilities for reducing concurrency.

input for the preservation of $\rightleftharpoons$, but then deadlock situations may arise. Figure 4.4 exemplifies this: initially, both environment and system can safely interact. Moreover, if either the environment or the system are transformed by reducing concurrency between an input and an output, the interaction can still be safe. However, the two transformed systems can not interact. The formalization of the transformation is:

Theorem 4.2.1 *Let the RPNs A, B and C with underlying FCLSPN and corresponding deterministic RTSs $(S^A, \Sigma^A, T^A, s^A_{in})$, $(S^B, \Sigma^B, T^B, s^B_{in})$ and $(S^C, \Sigma^C, T^C, s^C_{in})$, respectively. Assume $\Sigma^C = \Sigma^B$. If*

1. $A \rightleftharpoons B$
2. $\phi_r(B, e_1, e_2) = C$, *with* $e_1, e_2 \notin \Sigma^B_I$

then $A \rightleftharpoons C$.

Proof: *Case $\{e_1, e_2\} = \{o_1, o_2\} \subseteq \Sigma^B_O$. The other cases are similar.*
Let R' be the relation between A and B. Define R as:

$$\forall s \in S^A, s'' \in S^B, s' \in S^C : sR's'' \wedge s^B_{in} \xrightarrow{\sigma} s'' \wedge s^C_{in} \xrightarrow{\sigma} s' \Leftrightarrow sRs'$$

The following items treat individually conditions 1-4 of Definition 3.3.2:

- **Condition 1:** taking $\sigma = \lambda$ implies $s^A_{in} R s^C_{in}$.

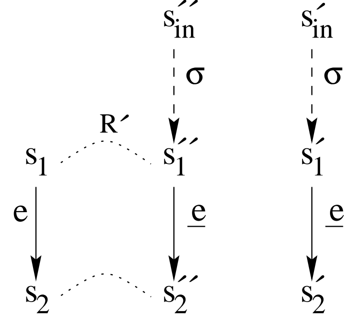


Figure 4.5: Conditions 2(a) from the proof of Theorem 4.2.1.

- **Condition 2(a):** let $s_1 R s'_1$, and suppose $s_1 \xrightarrow{e} s_2$ with $e \in \Sigma_O^A$. Figure 4.5 depicts the situation. Condition 2(a) of R' ensures that there exists $s''_2 \in S^B$ s.t. $s''_1 \xrightarrow{e} s''_2$ and $s_2 R' s''_2$. By definition of R, σ is enabled both in s_{in}^B and s_{in}^C. Then, each place marked by the sequence σ in B is also marked in C, because the flow relation of B is included in the flow relation of C. Given that the initial marking of B is preserved in C and the set of predecessor places for each input event is also preserved, implies that e is also enabled in s'_1. The definition of R makes each s'_2 s.t. $s'_1 \xrightarrow{e} s'_2$ to be related with s_2.

- **Condition 2(b):** let $s_1 R s'_1$, and suppose $s'_1 \xrightarrow{e} s'_2$ with $e \in \Sigma_O^C$. The set of predecessor places of e in B is a subset or is equal to the one in C. Moreover, given that both the initial marking of B is identical to the one in C, and each place marked by the sequence σ in B is also marked in C, implies that e is also enabled in s''_1, i.e. $s''_1 \xrightarrow{e} s''_2$. Condition 2(b) of R' ensures that $\mathsf{En}(s_1, e)$, and each s_2 such that $s_1 \xrightarrow{e} s_2$ is related by R' with s''_2. The definition of R induces that each such s_2 is related with s'_2.

- **Condition 3(a):** let $s_1 R s'_1$, and suppose $s_1 \xrightarrow{e} s_2$ with $e \in \Sigma_{INT}^A$, then a similar reasoning of Condition 2(a) can be applied.

- **Condition 3(b):** let $s_1 R s'_1$, and suppose $s'_1 \xrightarrow{e} s'_2$ with $e \in \Sigma_{INT}^C$, then a similar reasoning of Condition 2(b) can be applied.

- **Condition 4(a):** let $s_1 R s'_1$, and suppose $\{e | \mathsf{En}(s_1, e)\} \subseteq \Sigma_I^A$. Condition 4(b) of R' ensures that $\{e | \mathsf{En}(s''_1, e)\} \nsubseteq \Sigma_I^B$. If the non-input event enabled in s''_1 is different from o_2, then similar reasoning of previous cases guarantees that the event is also enabled in s'_1. If the event enabled in s''_1 is o_2 and no non-input event is enabled in s'_1, we will proof that o_2 is also enabled in s'_1. Assume the contrary: o_2 is enabled in s''_1 but no non-input event is enabled in s'_1. Applying the same reasoning

of case 2(a) we can conclude that the place p such that $\{p\} = {}^\bullet o_2$ in B has a token in the marking m corresponding to state s'_1. Moreover, the liveness of C ensures that from m there is a feasible sequence δ (let δ be minimal) reaching a marking m' where o_1 is enabled. The minimality of δ, together with the fact that the new place p' added by ϕ_r between o_1 and o_2 is unmarked in m (otherwise o_2 is enabled in s'_1, because $\{p, p'\} = {}^\bullet o_2$ in C) imply that $o_2 \notin \delta$, and therefore $m'(p) = 2$, which contradicts the safeness of C.

- **Condition 4(b):** let $s_1 R s'_1$, and suppose $\{e|\mathsf{En}(s'_1, e)\} \subseteq \Sigma_I^C$, then similar reasons of the previous cases ensure that $\{e|\mathsf{En}(s''_1, e)\} \subseteq \Sigma_I^B$ and Condition 4(b) of R' ensures that $\{e|\mathsf{En}(s_1, e)\} \nsubseteq \Sigma_O^A$.

Finally, it can be proven that the language of $\mathsf{RTS}(C)$ is a subset of the language of $\mathsf{RTS}(B)$. Therefore, no infinite trace of internal events can exist in C implying that C is livelock-free. □

I/O compatible application of ϕ_i

The application of ϕ_i preserves $\rightleftharpoons$ when:

1. at least one of the transitions involved is internal, and
2. no internal transition is inserted as trigger of an input transition

The purpose is to avoid the increase of concurrency between two observable transitions, in order to forbid the generation of unexpected traces either on the environment or on the system. More formally:

Theorem 4.2.2 *Let the RPNs A, B and C with underlying FCLSPN and corresponding deterministic RTSs* $(S^A, \Sigma^A, T^A, s^A_{in})$, $(S^B, \Sigma^B, T^B, s^B_{in})$ *and* $(S^C, \Sigma^C, T^C, s^C_{in})$. *Assume* $\Sigma^C = \Sigma^B$. *If*

1. $A \rightleftharpoons B$
2. $\phi_i(B, e_1, e_2) = C$, *with either* $e_1 \in \Sigma^B_{INT}$ *or* $e_2 \in \Sigma^B_{INT}$
3. B *is input-proper*
4. $\forall e \in (e_2^\bullet)^\bullet : e \notin \Sigma_I^B$

then $A \rightleftharpoons C$.

Proof: Conditions 1-4 of transformation ϕ_i ensure to preserve both the observational equivalence and the input-properness of B. Theorem 3.5.2 induces $A \rightleftharpoons C$. □

I/O compatible application of ϕ_e

Rule ϕ_e only preserves $\rightleftharpoons$ when applied to internal transitions.

Theorem 4.2.3 *Let the RPNs A, B and C with underlying FCLSPN and corresponding deterministic RTSs $(S^A, \Sigma^A, T^A, s^A_{in})$, $(S^B, \Sigma^B, T^B, s^B_{in})$ and $(S^C, \Sigma^C, T^C, s^C_{in})$. Assume $\Sigma^C_{OBS} = \Sigma^B_{OBS}$. If*

1. $A \rightleftharpoons B$
2. $\phi_e(B, e) = C$, *with* $e \in \Sigma^B_{INT}$
3. *B is input-proper*

then $A \rightleftharpoons C$.

Proof: If the observational languages of two deterministic systems coincide, then they are observational equivalent [31]. It can be proven that the observational language of C is the same to the one of B. Moreover, provided that B is input proper and the causality relations regarding internal events on B are preserved in C, C is also input proper and therefore, applying the determinism of B and Theorem 3.5.2 implies $A \rightleftharpoons C$. □

The transformations presented above can introduce redundant places in the target net. For dealing only with place-irredundant nets, the kit is augmented with a rule for eliminating redundant places. Linear programming techniques exist that decide the redundancy of a place efficiently [72]. Moreover, each time a transformation is performed, it can be locally determined the potential redundant places, and therefore the redundancy checking is only applied to a few places.

4.3 Encoding Technique

This section presents a transformation applied to STGs. The features of this transformation are the following:

- It guarantees the USC property.
- It preserves free-choiceness.
- It preserves consistency, liveness, safeness and observational equivalence with respect to the input and output signals.
- It has linear complexity on the size of the STG.

This is the first method that *guarantees* a solution for the encoding problem and tackles the problem in linear complexity for the class of FCLSPNs. The transformation is based on the insertion of a signal for each place of the STG that mimics the token flow on that place.

1. Create the *silent* transitions ε_1 and ε_2.

2. For each place $p \in {}^\bullet t$, create a new transition with label $sp-$ and insert new arcs and places for creating a simple path from ε_1 to ε_2, passing through $sp-$.

3. For each place $q \in t^\bullet$, substitute the arc (t, q) by the arc (ε_2, q), create a new transition labeled as $sq+$ and insert new arcs and places for creating a simple path from t to ε_1, passing through $sq+$.

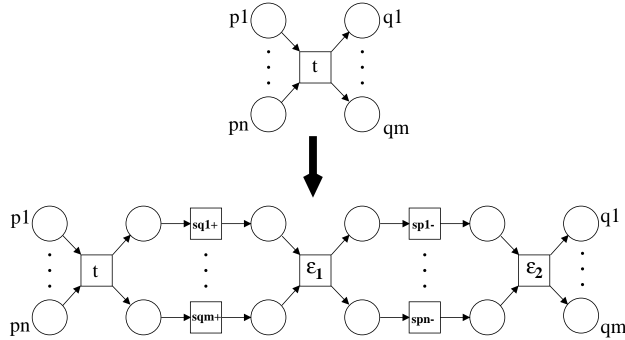


Figure 4.6: Transformation rule for each transition $t \in \mathcal{T}$.

The transformations will be presented as a rule to be applied to the transitions of the STG. Before the application of the Structural Encoding, the set of signals of the STG has been augmented with one signal sp for each place p of the STG. In order to simplify the presentation of the rules and the corresponding proofs, we will use silent transitions on the definition of the rules.

4.3.1 Encoding Transformation

Let $S = \langle\langle\mathcal{P}, \mathcal{T}, \mathcal{F}, m_0\rangle, \mathcal{X}, \Lambda\rangle$ be an STG with underlying FCLSPN. The Structural Encoding of S derives the STG $Enc(S)$ in which a new internal signal sp has been created for each place $p \in \mathcal{P}$, and the transformation rule described in Figure 4.6 has been applied to each transition $t \in \mathcal{T}$. The new transitions appearing in $Enc(S)$, labelled with $sp*$, will be called *E-transitions*.

Figure 4.7 shows how each place is encoded by at least two transitions in a way that depending on wheter the transitions are predecessors or successors, the encoding is negative or positive, respectively. In the Figure this is presented for the places q and q_i.

Let us now prove properties on $Enc(S)$.

Proposition 4.3.1 *$Enc(S)$ is free-choice.*

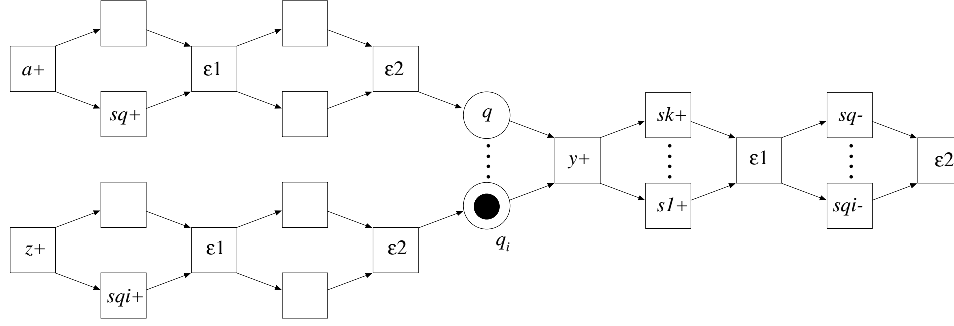


Figure 4.7: Encoding for places q and q_i.

Proof: Every new place p appearing in $Enc(S)$ has $|^\bullet p| = |p^\bullet| = 1$ by construction. For each place p (transition t) of S, the set $p^\bullet$ ($^\bullet t$) is identical both in S and $Enc(S)$. Given that S is free-choice, $Enc(S)$ is also free-choice.

□

Proposition 4.3.2 *$Enc(S)$ is live, safe and is observationally equivalent to S with respect to the input and output signals.*

Proof: The transformation for structural encoding is a trivial combination of a set of transformations proposed by Berthelot that preserve liveness, safeness and home marking [7]. These transformations also preserve the behavior condition: each conflict resolution in $Enc(S)$ is performed by some observable transition, i.e. for every transition $x*$ and place p such that $p \in {}^\bullet x*$ and $p^\bullet > 1$ then $x \in \mathcal{O}bs$.

From the behavior condition, it immediately follows that observational equivalence is also preserved. □

Proposition 4.3.3 *$Enc(S)$ is consistent.*

Proof: Given that the observational equivalence is preserved, consistency directly holds for the signals already in S. It only remains to prove that it also holds for the E-transitions of the new inserted signals.

By construction, the new *sp* and *sq* signals mimic the token flow in places. Given that the dynamic behavior corresponds to a *safe* PN, no more than two consecutive rising or falling transitions can occur for these signals. □

Lemma 4.3.1 *Let R be the new set of places inserted in S for constructing $Enc(S)$. Every feasible complementary set between two reachable markings m and m' of $Enc(S)$ satisfies the equality $m|_R = m'|_R$.*

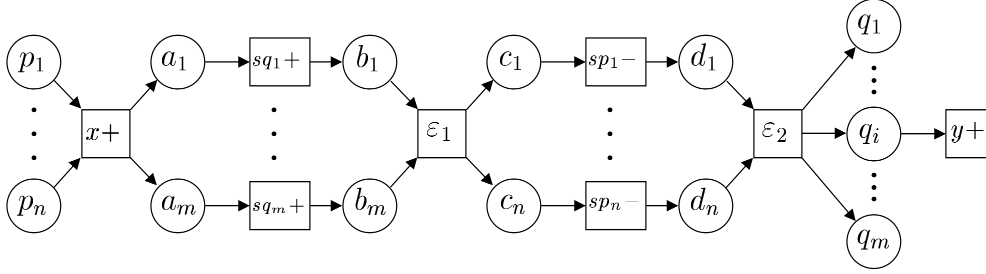


Figure 4.8: Place encoding to guarantee USC.

Proof: To prove that $m|_R = m'|_R$ means to prove that every new place inserted in S has the same amount of tokens both in m and m'. Figure 4.8 depicts a fragment of $Enc(S)$ that results from applying the transformation rule to a transition with $p_1 \ldots p_n$, and $q_1 \ldots q_m$ as predecessor and successor places, respectively. Without loss of generality, we will assume that the label of the transition is $x+$, and that place q_i has one successor transition with label $y+$. By definition, if a feasible complementary set exists between m and m' then $\lambda(m) = \lambda(m')$. With two exceptions that will be discussed later, the marking of the new places inserted (places a, b, c and d in Figure 4.8) can be uniquely determined as follows:

$$
\begin{aligned}
m(a_i) = 1 &\Leftrightarrow sq_i = 0 \wedge sp_1 = \cdots = sp_n = 1 \wedge x = 1 \\
m(b_i) = 1 &\Leftrightarrow sq_i = 1 \wedge sp_1 = \cdots = sp_n = 1 \wedge x = 1 \\
m(c_i) = 1 &\Leftrightarrow sp_i = 1 \wedge sq_1 = \cdots = sq_m = 1 \wedge x = 1 \\
m(d_i) = 1 &\Leftrightarrow sp_i = 0 \wedge sq_1 = \cdots = sq_m = 1 \wedge x = 1
\end{aligned}
$$

When defining the previous equations, it is important to use the fact that the STG is safe and consistent. We will only prove the equality for $m(a_i)$. The other equalities can be proved in a similar way.

$\Rightarrow$

$m(a_i) = 1$ implies $sq_i = 0$, since sq_i+ is enabled. Otherwise the STG would not be consistent. $m(a_i) = 1$ also implies $sp_1 = \cdots = sp_n = 1$, since the liveness and safeness of the STG imply that ε_1 has not fired after $x+$ has fired. Therefore, none of the sp_i- transitions has fired yet, while all sp_i+ transitions already fired before $x+$. Finally, $m(a_i) = 1$ clearly implies $x = 1$.

$\Leftarrow$

By the consistency of signal x, the only markings in which $sp_1 = \cdots = sp_n = 1$ and $x = 1$ correspond to markings in which some tokens are held in the places after $x+$ but before $sp_1 - \ldots sp_n -$. The fact that $sq_i = 0$ implies that place a_i has a token.

As mentioned before, there are two exceptions in which the binary code does not uniquely identify the marking in the new places inserted. One exception corresponds to the submarkings in which $m(b_1) = \cdots = m(b_m) = 1$ and $m(c_1) = \cdots = m(c_n) = 1$, respectively. These submarkings are only separated by a silent transition, ε_1, that makes them observationally equivalent. The other exception corresponds to the submarkings separated by ε_2.

Finally, given that $\lambda(m) = \lambda(m')$ we can conclude that the previous equations also hold for m', and therefore the marking in R is identical both in m and m'. □

Lemma 4.3.2 *Let q_i be a place of S, and T_c be a feasible complementary set between two reachable markings m and m' of $Enc(S)$. Then, $m(q_i) = 1 \Rightarrow m'(q_i) = 1$.*

Proof: Without loss of generality, let q_i be the one of Figure 4.8. Assume that $m(q_i) = 1$. If no transition in $q_i^\bullet$ belongs to T_c then the claim trivially holds.

The initial situation is depicted in Figure 4.9.

Assume, without loss of generality (due to the free-choiceness of $Enc(S)$), that $y+ \in T_c$, and let m'' be the marking reached after firing $y+$. From m'' it is possible to fire the set of E-transitions which result from the encoding of $y+$. One transition of this set is sq_i-, but note that $\lambda(m)_{sq_i} = 1$. Two situations can happen:

1. $sq_i- \in T_c$: then at least a transition sq_i+ belongs to T_c. But every copy of a sq_i+-transition is either in $(x+^\bullet)^\bullet$ or in $(z+^\bullet)^\bullet$, where $z+$ is another transition such that ${}^\bullet z+ = {}^\bullet x+$[3]. Assume that the transition belonging to T_c is the one in $(z+^\bullet)^\bullet$. The safeness of $Enc(S)$ ensures that the set of places from the encoding of $z+$ is unmarked in m, because otherwise there is a marking reachable from m having two tokens on q_i. But Lemma 4.3.1 ensures that the marking in the new places inserted for encoding $z+$ is the same both in m and m', and therefore every E-transition from the encoding of $z+$ belongs to T_c, adding again a token to q_i.

2. $sq_i- \notin T_c$: no transition $y-$ can appear in T_c after the firing of $y+$ because transition ε_2 from the encoding of $y+$ does not belongs to T_c, and then given that $Enc(S)$ is consistent no sequence of transitions in T_c can enable $y-$ after the firing of $y+$. Therefore $y-$ appears before of $y+$ in T_c. Again, the consistency of $Enc(S)$ implies that there exists a place q such that $q \in {}^\bullet y+$, $m(q) = 0$ and some $sq+$ transition in the path between $y-$ and $y+$ must be fired in order to put a token in q.

[3] In the simplest case, $z+ = x+$.

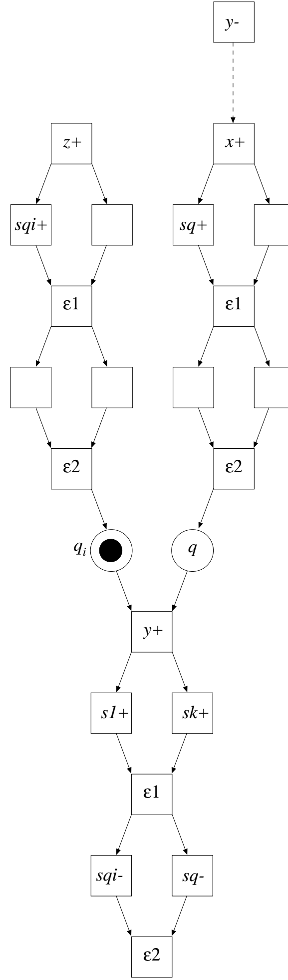


Figure 4.9: Initial situation for proof of Lemma 4.3.2.

Then there must be in T_c an $sq-$ transition, but the actual situation is not possible in $Enc(S)$, because:

- The positive E-transitions from the encoding of $y+$ (let s_1+, ..., s_k+) must be in T_c in order to enable $sq-$,
- Lemma 4.3.1 ensures that the same marking exists both in m and m' with respect to the set of places from the encoding of $y+$. Given that $sq_i- \notin T_c$ implies that transition ε_2 from the encoding of $y+$ is not in T_c, and then in m (m') some of this places are marked.
- And then the consistency of the E-signals inserted s_1, ..., s_k in $Enc(S)$ implies that the set of negative transitions s_1-, ..., s_k- can not be in T_c, because otherwise them can be autoconcurrent.

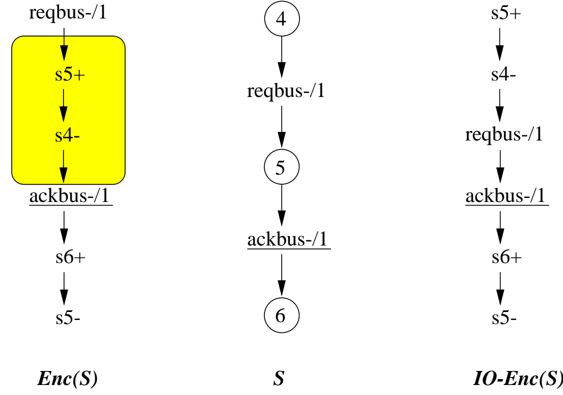


Figure 4.10: In the center the initial STG fragment. $Enc(S)$: technique of the previous section. *IO-Enc*(S): technique presented in this section to preserve I/O compatibility.

But then T_c is not a complementary set.

□

Theorem 4.3.1 *Enc(S) has the USC property.*

Proof: This follows from Lemmas 4.3.1 and 4.3.2, together with the boundedness of $Enc(S)$.

□

4.3.2 I/O Compatible Encoding Technique over STG

The encoding technique presented in Section 4.3.1 does not preserve the I/O compatibility, because when the transformation rule is applied to an output signal transition being a trigger of an input signal transition, internal events (induced by the intermediate places of the initial STG) are inserted delaying the input signal transition, and therefore there is a violation of Condition 1 (receptiveness) of the I/O compatibility definition. The situation is depicted in Figure 4.10 (left): the E-transitions in the shaded box ($s5+$ and $s4-$) delay the input signal transition $ackbus-/1$.

Figure 4.10 (right) shows how the encoding technique presented in this section will be applied to the STG fragment of the figure. The only difference between the new technique presented in this section and the one presented in the previous section is in the way non-input signal transitions are encoded (transition $reqbus-/1$ in the figure): for every transition of this type, the E-transitions associated are inserted *before* of the transition itself. Input signal transitions are encoded in the same way as in the technique presented in the previous section. Figure 4.11 describes formally the new encoding technique for non-input signal transitions.

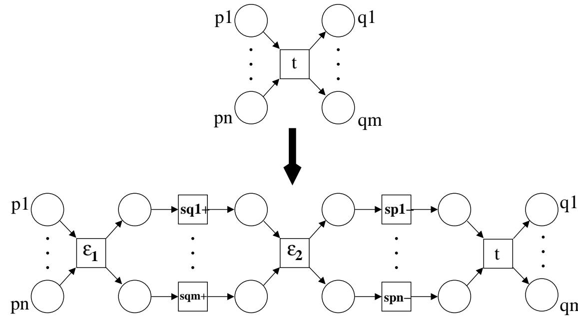


Figure 4.11: Transformation rule for non-input signals to preserve the I/O Compatibility.

However, the new encoding technique presented here will also violate the I/O compatibility for those STGs having two input signal transitions $i1$ and $i2$ in sequence, because the encoding of $i1$ will delay $i2$ anyway. The last part of this section will define formally the class of STGs not having two transitions in sequence. For that class of STGs (called IO-STG), I/O compatibility can be ensured.

Let us first concentrate on reasoning about whether the new encoding technique ensures a correct encoding in the transformed net. All the properties except USC can be proved in the same way for this new technique.

Unfortunately, the new technique derived can only ensure USC if the underlying Petri net is a marked graph. However, if the underlying Petri net is Free choice and some additional structural condition is fulfilled, CSC can be ensured. Informally, the structural condition is on places that have more than one transition on their pre-set. This type of places are called *join* places. The structural condition, called *simple join condition*, described graphically in Figure 4.12, is the following:

> *Every transition in the pre-set of a join place p must have p as its only successor place.*

The formal definition of the class of FC Petri nets satisfying the simple join condition is defined now:

Definition 4.3.1 (Simple Join Petri net) *A simple join Petri net (SJ) is a FCLSPN such that every join place fulfills the simple join condition, i.e.* $\forall p : |^\bullet p| > 1 \implies \forall t \in {}^\bullet p : t^\bullet = \{p\}$.

As was mention before, the encoding technique presented in this section can not guarantee unique state encoding for general FCLSPN. This can be seen in the Figure 4.13: in the initial marking, the complementary sequence $\{busctl+, ..., reqbus+, nakbus+, ..., s17-\}$ (in boldface) connects two different markings having the same binary code assigned, thus violating the USC

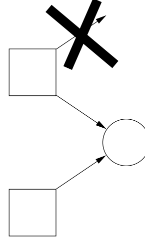


Figure 4.12: simple join condition to ensure a correct encoding in the modified encoding technique.

property (the underlined transitions denote input events). However, we are going to see that if the underlying Petri net is SJ, the set of output signals enabled in both markings (*busctl+* in the example) will be the same.

The insertion of the E-transtions in the initial STG by the new encoding technique makes the new STG to almost characterize the markings by looking at the value of the signals nearby, like in the technique of previous section. For input signal transitions or E-transitions resulting from the encoding of an input signal, Lemmas 4.3.1 and 4.3.2 also hold for the new technique, provided that input transitions are encoded in the same way. However, for output signal transitions and E-transitions associated, Lemmas 4.3.1 and 4.3.2 only hold for SJ nets.

In the encoding of an output signal transition, the reasoning used in Lemma 4.3.1 for characterizing places a_i, b_i and c_i (Figure 4.8) holds if the simple join condition is satisfied in the net. For those places, it must be considered now that $x = 0$ in the description of the marking. Using the consistency of the signals and the safeness of the transformed net, an identical reasoning of Lemma 4.3.1 can be applied, ensuring the characterization of any a_i, b_i and c_i place by the value of a set of signals.

However, Lemma 4.3.1 does not hold for places d_i resulting from the encoding of an output signal transition. For such d_i-places, we can not characterize the markings that put a token on them just by looking at the value of the signals. This can only happen when two output transitions share some join place on their post-set, as can be seen in the situation of Figure 4.14: when two output transitions share some place on their post-set, (place i in the figure), the markings m_1 and m_2 enabling each transition can have the same code assigned, because both output transitions create a copy of E-transition *spi+*.

Let us first define formally the two markings m_1 m_2 that have the same code:

$$m_1(d) = 1, m_1(d') = 0, m_1(j) = 0, m_1(i) = 0, m_1(k) = 1, C$$

$$m_2(d) = 0, m_2(d') = 1, m_2(j) = 1, m_2(i) = 0, m_2(k) = 0, C$$

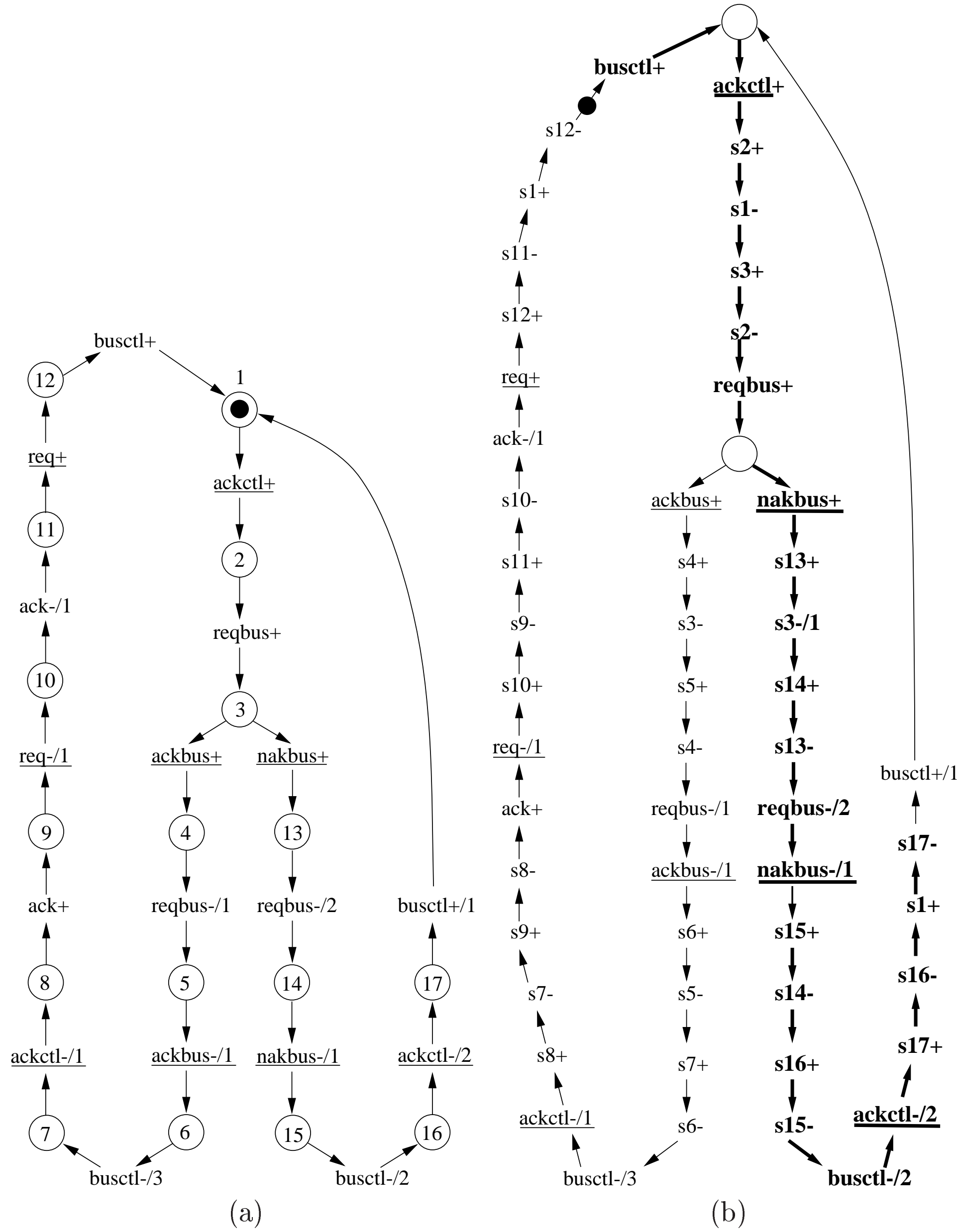


Figure 4.13: (a) *Alloc-Outbound* example, (b) *IO-Enc(Alloc-Outbound)* has CSC. It has not USC due to the complementary sequence (in boldface) between two different markings.

where C represents the marking for the rest of places of the net in m_1 and m_2. Note that both markings have the same token assignment in C, provided that a slight difference in the marking of the rest of places means that the transitions that added or removed the token will make the codes of the

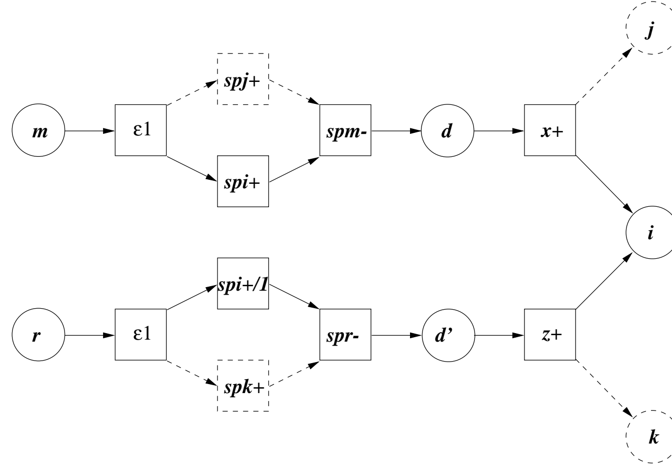


Figure 4.14: Situation where two markings can have the same code.

conflicting markings different, and therefore no conflict will exist. Now let us reason why the distribution of tokens in places d, d', i, j and k in m_1 and m_2 is the previous one: if m_1 (m_2) enables $x+$ ($z+$) implies that $m_1(d) = 1$ ($m_2(d') = 1$). $m_1(d') = 0$ ($m_2(d) = 0$) because if $m_1(d') = 1$ ($m_2(d) = 1$) then after firing $z+$ and $x+$ at m_1 (m_2) there will be two tokens on place i, contradicting the safeness of the transformed net. Also the safeness of the transformed net implies $m_1(j) = m_1(i) = 0$ and $m_2(i) = m_2(k) = 0$.

The values of the signals in m_1 and m_2 are:

$$spj = 1, spi = 1, spk = 1, spm = 0, spr = 0, x = 0, z = 0$$

And then, the only possibility for $spj = 1$ at m_2 is when the token added to j by the firing of $x+$ (if $x+$ does not fires in the complementary sequence between m_1 and m_2 we are done) is still in place j at m_2, i.e. $m_2(j) = 1$. This reasoning can be symmetrically done for m_1 and spk, leading to $m_1(k) = 1$. Again, this is only true for SJ nets. Figure 4.15 shows a situation where, provided that the simple join condition does not hold on the join place p_1, a CSC conflict exists, described by the complementary sequence in boldface.

Now consider the marking m_3 reached after firing $x+$ at m_1. It can be formally defined as:

$$m_3(d) = 0, m_3(d') = 0, m_3(j) = 1, m_3(i) = 1, m_3(k) = 1, C$$

which is the same marking reached after firing $z+$ at m_2. Then the consistency of the transformed net imply that x and z are indeed the same signal, i.e. $z+ = x_i+$. This ensures that, according to what happens in Figure 4.13 with $busctl+$, no possible conflict exists for the transition preceding the join place.

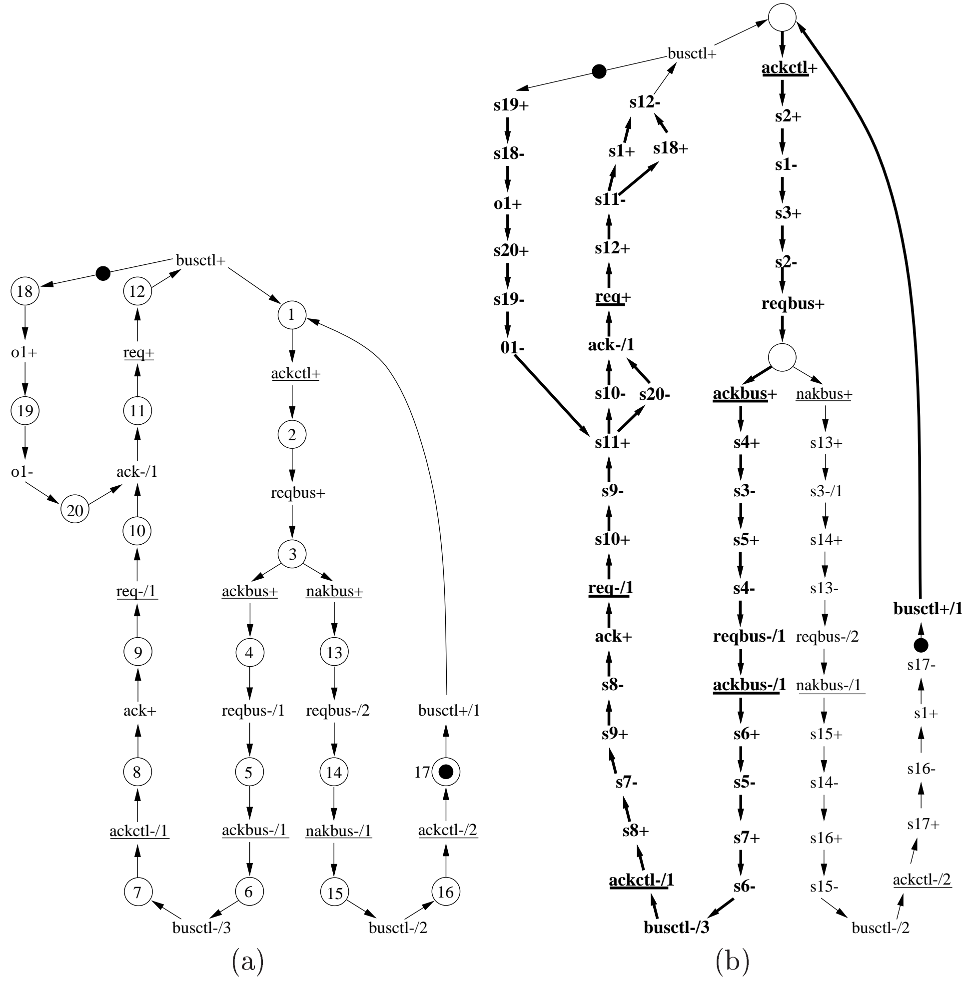


Figure 4.15: (a) Modified *Alloc-Outbound* example, (b) Applying the encoding technique to preserve I/O compatibility does not solves the CSC conflicts

However, depending on whether the simple join condition holds or not, we can have a CSC conflict in the transformed net, because the set of transitions enabled at m_1 (enabled by the token at place k and the token at places in C) can be different from the set of transitions enabled at m_2 (enabled by the token at place j and the token at places in C). If the simple join condition holds, places j and k do not exist (in Figure 4.14, only solid lines will represent the net) and therefore the same set of transitions will be enabled by C both in m_1 and m_2. If the simple join condition does not hold, we can have situations like the one described in Figure 4.15(b), where in the initial marking transitions $busctl+/1$ and $sp19+$ are enabled and after firing the complementary sequence shown in boldface, only transition $busctl+$ is enabled.

In conclusion, we can only ensure a correct encoding with the technique presented in this section for SJ nets. For general FCLSPNs, some easy transformations can be done to guarantee a correct encoding, but then the new Petri net can violate the Free choice condition. As future work, we plan to study the situation where this happens and whether the Free choice condition can be reestablished.

The remainder of the section will define the class of STGs where the new technique can be applied and I/O compatibility ensured. For the preservation of the I/O compatibility with respect to the environment, the encoding technique presented in this section can only be applied to a restricted class of specifications: the IO-STG class contains those STGs fulfilling both that no input signal transition triggers another, and that the transitions in the post-set of a *choice* place are all input signal transitions. More formally:

Definition 4.3.2 *An IO-STG is a Free-Choice STG where the following conditions hold:*

1. $\forall a_i*$ *with* $a \in \mathcal{I} : \forall b_j* : (b_j* \in (a_i*^\bullet)^\bullet \Rightarrow b \notin \mathcal{I})$.
2. $\forall p \in \mathcal{P} : (|p^\bullet| \geq 2 \Rightarrow \forall b_j* \in p^\bullet : b \in \mathcal{I})$.

And now we can state that when the encoding technique presented in this section (called *IO-Enc*) is applied to an IO-STG, the new system can work correctly in the environment where the initial system is assumed to work correctly:

Theorem 4.3.2 *Let S be an input-proper IO-STG with environment $\bar{S}$ satisfying that $S \rightleftharpoons \bar{S}$. Then IO-Enc$(S) \rightleftharpoons \bar{S}$*

Proof: Direct application of Theorem 3.5.2, taking $B = S$, $A = \bar{S}$ and $C =$ *IO-Enc*(S). The input-properness of *IO-Enc*(S) is derived from the input-properness of S and the fact that no new internal transition is delaying an input in *IO-Enc*(S). □

4.4 Conclusion

Several Petri net transformations have been introduced in this chapter. The rules presented in the first section of the chapter, aim to support the synthesis of reactive systems modeled as a Petri net. The theory of I/O Compatibility is used for showing that, when the rules are applied according to some structural conditions, the correct interaction with the environment can be guaranteed. As a future work it is interesting both to extend the kit of rules and to weaken the restriction of application of the kit presented.

The second part introduces an encoding technique for the synthesis of asynchronous circuits. To the best of our knowledge, it is the first structural

technique that works in the class of STGs with underlying FCLSPN. It is also shown how to modify the technique to preserve the I/O Compatibility with the environment, which also requires to restrict the class of behaviors where the technique can be applied.

Chapter 5

ILP Models for Synthesis and Verification of Asynchronous Circuits

Everything
You think you know
– Massive Attack, *Everywhen*

In this chapter we present methods that use integer programming techniques to verify the CSC/USC properties, necessary conditions for the existence of a speed-independent implementation. Moreover a method for computing the set of signals needed for the implementation of a given signal is presented.

The methods consist in adding constraints to the marking equation in order to prove that the reachable states of the system are correctly encoded. In the case of the verification of CSC/USC properties, the experimental results show a speed-up of several orders of magnitude with respect to the existing approaches.

This chapter is based in the results presented in [14].

5.1 Introduction

The synthesis of asynchronous circuits from a given formalism (i.e. an automaton or a Petri net) can be separated into two steps [22]: (i) checking and (possibly) forcing implementability conditions and (ii) deriving the next-state function for each signal generated by the system. Most of the existing CAD tools for synthesis perform steps (i) and (ii) at the underlying state graph level, thus suffering from the well known *state explosion* problem. These tools, although using symbolic techniques for alleviating the

cost of representing the state space, can only synthesize specifications with moderate size.

In order to avoid the state explosion problem, structural methods for steps (i) and (ii) have been proposed in the literature [75, 63, 44]. The work proposed in [75, 63] uses graph theoretic-based algorithms while [44] use both graph theoretic (by using causal order partial semantics of the Petri net, called unfoldings [54]) and linear algebraic techniques. A new and promising direction is presented in [43], where the encoding problem is faced by adopting the Boolean Satisfiability (SAT) approach.

To the best of our knowledge, the work in [44] is the first one that uses linear algebraic techniques to approach the encoding problem. Although completely characterizing the encoding problem, the techniques presented in [44] (and also in [43]) need to compute the unfolding of the net, whose size can be exponential on the size of the net. In addition, the checking of the Complete State Coding (CSC) in [44] needs to solve non-linear integer programming problems, which is $\mathcal{NP}$-*hard* ([61]). The work presented in this chapter proposes linear algebraic methods for deriving sufficient conditions for the encoding problem and novel methods for performing the synthesis in a modular fashion. In our approach, the computation of the unfolding is not performed, at the expense of checking only sufficient conditions for synthesis. However, the experimental results indicate that this approach is highly accurate and provides a speed-up of several orders of magnitude with regard to [44, 43].

Moreover, a novel algorithm for computing the set of signals needed to synthesize a given signal is presented, which also uses integer programming techniques. This allows to project the behavior into that set of signals and perform the synthesis on the projection.

In summary, the work presented aims at facing the two important steps (i) and (ii) in the synthesis of asynchronous circuits: it proposes powerful methods for checking CSC/USC and a novel method for decomposing the specification into smaller ones *while preserving the implementability conditions.* The methods presented here in combination with the ones presented in Chapter 4 provide a complete design flow for the synthesis of controllers, described in the following chapter.

5.2 ILP for Verifying State Encoding

In this section it is shown how to formulate an ILP problem in order to verify if a given specification is correctly encoded.

The concept of *complementary sequence* is relevant in this chapter:

Definition 5.2.1 (Complementary sequence and Balanced Signals) *Given a set of signals $\Sigma = \{a_1, \ldots, a_n\}$, and a transition sequence σ, we de-*

fine **code_change**(Σ,σ) *as a* $|\Sigma|$*-vector where component* i *corresponds to the result of the equation* $\sum_{a_i+} \#(\sigma, a_i+) - \sum_{a_i-} \#(\sigma, a_i-)$. *When the i-th component of the vector is 0, we say that the signal* a_i *is* balanced. *A* complementary sequence σ *is a feasible transition sequence such that* **code_change**(Σ,σ) $= \mathbf{0}$.

A balanced signal in a sequence means that the number of positive and negative transitions of the signal in the sequence is the same. In a complementary sequence all the signals are balanced. For instance, the sequence

$$d+, dtack+, dsr-, d-, dtack - anddsr+$$

is a complementary sequence because the signals appearing on it, i.e. d, $dtack$ and dsr, are balanced.

5.2.1 ILP for USC Checking

A USC conflict appears in the SG of a system when there are two reachable markings m_1, m_2 such that m_2 is reachable from m_1 by firing a complementary sequence z, i.e. $m_0 \overset{x}{\to} m_1 \overset{z}{\to} m_2$. Using the marking equation (see Section 2.5), a sufficient condition for USC can be obtained. Before of defining the model, we present an introductory example.

The example is the VME Bus Controller, presented in Section 2.6.3. As said in the previous paragraph, the theory presented in the following sections is based in the marking equation of a Petri net.

In the STG of Figure 5.1(a), places names is shown explicitly. From this STG, the incidence matrix associated to the STG is the following:

	$lds+$	$dsr+$	$ldtack+$	$ldtack-$	$d+$	$dtack-$	$dtack+$	$lds-$	$drs-$	$d-$
p_1	+1	0	-1	0	0	0	0	0	0	0
p_2	0	0	+1	0	-1	0	0	0	0	0
p_3	0	0	0	0	+1	0	-1	0	0	0
p_4	0	0	0	0	0	0	+1	0	-1	0
p_5	0	0	0	0	0	0	0	0	+1	-1
p_6	0	0	0	0	0	-1	0	0	0	+1
p_7	0	-1	0	0	0	+1	0	0	0	0
p_8	-1	+1	0	0	0	0	0	0	0	0
p_9	0	0	0	0	0	0	0	-1	0	+1
p_{10}	0	0	0	-1	0	0	0	+1	0	0
p_{11}	-1	0	0	+1	0	0	0	0	0	0

The initial marking of the underlying Petri net is

$$m_0 = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1)$$

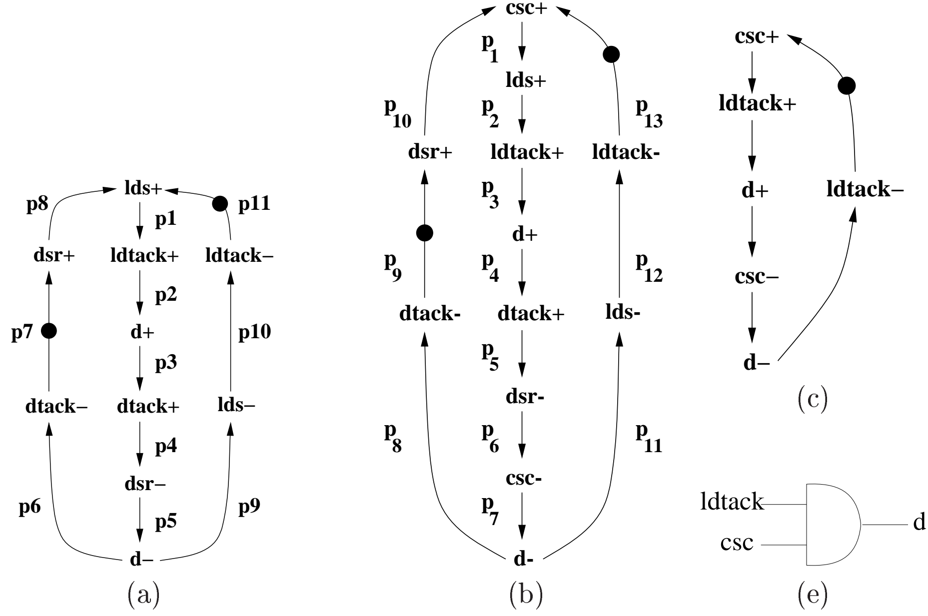


Figure 5.1: (a) STG, (b) STG with CSC, (c) Projection for signal d, (d) Circuit implementing d.

Assuming m_0 as initial marking, the assignment

$$x = (1, 1, 1, 0, 0, 0, 0, 0, 0, 0)$$

to the vector x is a solution to the marking equation ($m_1 = m_0 + \mathbf{N}x$). It means that the sequence of transitions corresponding to the Parikh vector x is firable at m_0, and it leads to m_1, where m_1 is

$$m_1 = (0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$$

from the marking m_1 above, the assignment

$$z = (0, 1, 0, 0, 1, 1, 1, 0, 1, 1)$$

is again a solution to the marking equation, ($m_2 = m_1 + \mathbf{N}z$), where the following marking m_2 results

$$m_2 = (0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0)$$

which is clearly a marking different from m_1. The non-zero positions of vector z correspond to transitions $d+$, $dtack+$, $dsr-$, $d-$, $dtack-$ and $dsr+$, and then the sequence of transitions corresponding to z is a complementary sequence: the number of positive and negative transitions of each signal appearing in z is the same. So, according to the marking equation, there are two different markings m_1 and m_2 such that m_2 is reachable from m_1 by firing a complementary sequence. It is clear that, the code $\lambda(m_1)$ is equal to $\lambda(m_2)$. We found a USC conflict.

Theorem 5.2.1 *Let $S = ((P, T, F, m_0), \Sigma, \Lambda)$ be a consistent STG. S has USC if the following ILP problem is infeasible:*

ILP model for USC checking:

Reachability conditions:

$$m_1 = m_0 + \mathbf{N}x$$
$$m_2 = m_1 + \mathbf{N}z$$
$$m_1, m_2, x, z \geq 0, x, z \in \mathbb{Z}^{|T|}$$

$$\begin{aligned} &\mathit{code_change}(\Sigma, z) = \mathbf{0} \\ &m_1 \neq m_2 \end{aligned} \tag{5.1}$$

Proof: If no solution exists for (5.1) then no possible complementary sequence exists between any pair of reachable markings $m_1, m_2 \in [m_0\rangle$ such that $m_1 \neq m_2$. □

In fact the constraint $m_1 \neq m_2$ is not linear, but it can be replaced by testing instead if at least one place has different amount of tokens in m_1 and m_2. Therefore the initial non-linear problem can be transformed to $|P|$ linear problems. However, if the system is k-bounded, any reachable marking can be encoded with a $|P|$ k-ary vector. This allows us to express the inequality between m_1 and m_2 as the inequality of two k-ary numbers [44].

Note that the marking equation provides only necessary conditions for a marking to be reachable. This means that either m_1 or m_2 or both can be spurious markings. In the example, both m_1 and m_2 are real markings because the underlying Petri net is a marked graph, and in this class of nets the marking equation characterizes reachability ([72]). Moreover, if the Petri net was FCLSPN and reversible, the existence of spurious markings can also be avoided by using the set of traps of the system [25].

In conclusion the USC conflict detected is in fact a real one. As said before, provided that the method presented here uses the marking equation as the main basis for reachability, implies that the method can only *semidecide* the problem of USC: only when the model is infeasible we are sure that the specification is free from USC conflicts.

5.2.2 ILP for CSC Checking

A CSC conflict exists when there exist two reachable markings m_1, m_2 such that m_2 is reachable from m_1 though a complementary sequence z and the set of non-input signals enabled in m_1 is different from the one in m_2. Note that the definition of CSC allows to check individually for each non-input signal a whether a has a CSC violation. When every non-input signal fulfills the CSC conditions, the entire system has CSC. The check of CSC

for each non-input signal can be performed in the following way: let a_i* be a transition of signal a. Then, a CSC conflict exists if: (i) m_2 is reachable from m_1 by firing a complementary sequence, (ii) m_1 and m_2 have the same code, (iii) a_i* is enabled in m_1 and (iv) for every transition a_j* of signal a, a_j* is not enabled in m_2. The enabledness of a transition x at a marking m can be characterized by the sum of tokens of the places in ${}^\bullet x$ at m: x is enabled at m if and only if the sum of tokens of the places in ${}^\bullet x$ is equal to the number of places in ${}^\bullet x$.

Now we can present a sufficient condition for CSC for each non-input signal a:

Theorem 5.2.2 *Let $S = ((P, T, F, m_0), \Sigma, \Lambda)$ be a consistent STG and non-input signal $a \in \Sigma$. S has CSC for a if the following problem is infeasible for each transition a_i*:*

ILP model for CSC checking:

$$\begin{array}{ll} (i) & \boxed{\textit{Reachability conditions (same as in (5.1))}} \\ (ii) & \texttt{code_change}(\Sigma, z) = \mathbf{0} \\ (iii) & \sum_{p \in {}^\bullet a_i *} m_1(p) = |{}^\bullet a_i *| \\ (iv) & \forall a_j * : \sum_{p \in {}^\bullet a_j *} m_2(p) < |{}^\bullet a_j *| \end{array} \quad (5.2)$$

Proof: If (5.2) has no solutions, no complementary sequence exists between any pair of reachable markings m_1 and m_2, with only m_1 enabling signal a. □

Note that the constraint $m_1 \neq m_2$ is not needed in (5.2). If we continue with the example of the VME Bus Controller, it can be shown that the USC conflict described in the previous section is also a CSC conflict for signal d. Given that it has been shown in the previous section that the assignments $x = (1, 1, 1, 0, 0, 0, 0, 0, 0, 0)$ and $z = (0, 1, 0, 0, 1, 1, 1, 0, 1, 1)$ satisfy the first two constraints, now we describe here the fulfillment of constraints (iii) and (iv) by x and z. The former constraint is satisfied because

$$\sum_{p \in {}^\bullet d+} m_1(p) = m_1(p_2) = 1 = |\{p_2\}| = |{}^\bullet d+|$$

and constraint (iv) is also satisfied provided that

$$\sum_{p \in {}^\bullet d+} m_2(p) = m_2(p_2) = 0 < 1 = |\{p_2\}| = |{}^\bullet d+|$$

note that constraint (iv) is not verified for transition $d-$, because the consistency of S is assumed.

In conclusion, a CSC conflict has been detected in the VME Bus Controller example. Given that the the conflict is a real one, it is mandatory to solve it in order to been able to synthesize the specification in the speed-independent delay model. Chapter 4 has presented a structural method to encode specifications in order to have a correct encoding.

5.3 ILP for Synthesis

In this section we propose a novel method to calculate the subset of signals onto which the STG must be projected to implement each signal. The support of the next-state function of each signal will be a subset of this support.

5.3.1 Computing a Support for Synthesis

The problem faced in this section is the following: given an STG $S = ((P, T, F, m_0), \Sigma, \Lambda)$ and a non-input signal $a \in \Sigma$, can we compute a subset Σ' of Σ such that it is enough for implementing f_a? Two conditions must be satisfied by Σ' [20]:

1. $Trig(a) \subseteq \Sigma'$.
2. $S_{\Sigma'}$ must have CSC for signal a.

Let such Σ' be called *a CSC support of signal a in S*:

Definition 5.3.1 (CSC Support) *Let S be an STG with set of events Σ, and a non-input signal $a \in \Sigma$. A set $\Sigma' \subseteq \Sigma$ is a CSC support of a in S if $S_{\Sigma'}$ has no CSC conflicts for signal a and $Trig(a) \subseteq \Sigma'$.*

For example, a possible CSC support for signal d from the STG shown in Figure 5.1(c) is $\{ldtack, csc\}$. Figure 5.1(d) shows the projection induced by this CSC support, and in Figure 5.1(e) it is shown the final implementation. The rest of this section is devoted to explain how to compute efficiently a CSC support for a given signal a.

The computation of a CSC support can be performed iteratively: starting from an initial assignment, ILP techniques can be used to guide the search. Imagine we have an initial set of signals $\Sigma' \subseteq \Sigma$, candidate to be the CSC support of a given signal a. A way of determining whether Σ' is a CSC support for signal a is by solving the following ILP problem:

ILP model for checking CSC support:

$$\begin{array}{l} \boxed{(i),\ (iii) \text{ and } (iv) \text{ from } (5.2)} \\ \texttt{code_change}(\Sigma', z) = \mathbf{0} \end{array} \qquad (5.3)$$

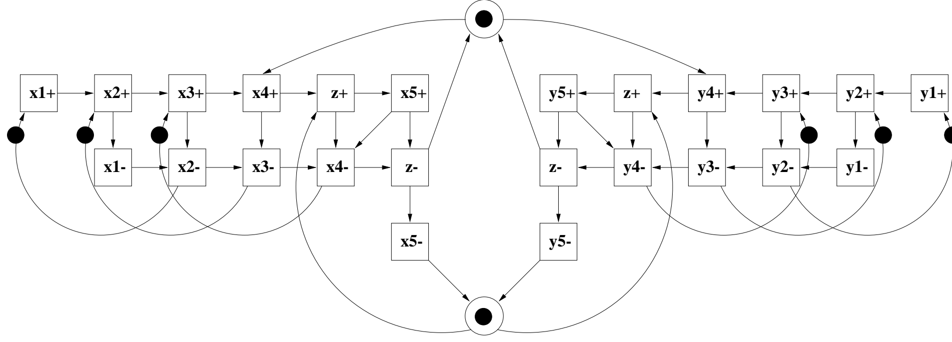


Figure 5.2: PpArbCsc(2,3)

If (5.3) is infeasible, then Σ' is enough for implementing a. Otherwise the set Σ' must be augmented (from signals in $\Sigma \setminus \Sigma'$) with more signals until (5.3) is infeasible. Moreover if (5.3) is feasible, adding a balanced signal b from $\Sigma \setminus \Sigma'$ will not turn the problem infeasible because z is still balanced for $\Sigma' \cup \{b\}$. On the contrary, adding an unbalanced signal will assign a different code to markings m_1 and m_2 of (5.3). Therefore, the unbalanced signals in z will be the candidates to be added to Σ'. The algorithm for finding a CSC support set for a non-input signal a is the following:

Algorithm for the calculation of CSC support:

CSC_Support (STG S, Signal a) **returns** CSC support of a

$\Sigma' := \text{Trig}(a) \cup \{a\}$

while (5.3) is infeasible **do**

Let b be an unbalanced signal in z

$\Sigma' := \Sigma' \cup \{b\}$

endwhile

return Σ'

Let us show how the algorithm performs the computation of the CSC support for signal d from the STG of Figure 5.1(b), where a new signal (csc) has been inserted in the original STG to solve the encoding problem. The new incidence matrix **N'** is the following, where transitions (columns) follow the order ($lds+, dsr+, ldtack+, ldtack-, d+, dtack-, dtack+, lds-, drs-, d-, csc-$ $, csc+$):

For computing the CSC support of signal d, the initial candidate assigned by the algorithm is

$$\Sigma' = Trigg(d) \cup \{d\} = \{ldtack, csc, d\}$$

p_1	-1	0	0	0	0	0	0	0	0	0	0	+1
p_2	+1	0	-1	0	0	0	0	0	0	0	0	0
p_3	0	0	+1	0	-1	0	0	0	0	0	0	0
p_4	0	0	0	0	+1	0	-1	0	0	0	0	0
p_5	0	0	0	0	0	0	+1	0	-1	0	0	0
p_6	0	0	0	0	0	0	0	0	+1	0	-1	0
p_7	0	0	0	0	0	0	0	0	0	-1	+1	0
p_8	0	0	0	0	0	-1	0	0	0	+1	0	0
p_9	0	-1	0	0	0	+1	0	0	0	0	0	0
p_{10}	0	+1	0	0	0	0	0	0	0	0	0	-1
p_{11}	0	0	0	0	0	0	0	-1	0	+1	0	0
p_{12}	0	0	0	-1	0	0	0	+1	0	0	0	0
p_{13}	0	0	0	+1	0	0	0	0	0	0	0	-1

However for such **N'** and Σ' the problem (5.3) turns out to be infeasible, and therefore $\{ldtack, csc, d\}$ is a valid CSC support for signal d.

In general it can happen that the set of trigger signals is not enough to guarantee a valid CSC support. This makes algorithm CSC_Support to iterate, adding a new signal at each iteration in order to guarantee a correct encoding for the signal. We illustrate this phenomenon using the STG of Figure 5.2, from [42]. Provided that the STG has forty places and twenty four transitions, the incidence matrix is not shown here.

Imagine that we want to compute the CSC support for signal x_4. The initial assignment by the algorithm is

$$\Sigma' = Trigg(x_4) \cup \{x_4\} = \{z, x_3, x_5, x_4\}$$

However, such Σ' does not makes problem (5.3) to be infeasible: the model is found solvable for the solution $x = \vec{\sigma_1}$ and $z = \vec{\sigma_2}$, where

$$\sigma_1 = x_1 + x_2 + x_3 + \quad and \quad \sigma_2 = y_1 + y_2 + y_3 + y_4 +$$

and therefore, another signal must be added. The algorithm adds x_1

$$\Sigma' = \Sigma' \cup \{x_1\} = \{z, x_3, x_5, x_4, x_1\}$$

and still the new signal added does not induce the new problem to be infeasible. Note that in algorithm CSC_Support the order chosen for selecting a new signal to add among the set of unbalanced ones is arbitrary, and therefore it is imprecise whether the signal added is in fact necessary to make the new problem infeasible. Choosing for instance lexicographical order, the algorithm will make five more iterations. For instance, in the third iteration the algorithm uses as a candidate for CSC support the following set of signals

$$\Sigma' = \{z, x_3, x_5, x_4, x_1, x_2, y_1, y_2\}$$

and the model is again found solvable for the solution $x = \vec{\sigma_1}$ and $z = \vec{\sigma_2}$, where

$$\sigma_1 = x_1 + x_2 + x_3 + \quad and \quad \sigma_2 = y_1 + y_2 + y_3 + y_1 - y_4 + y_2 -$$

The final set of signals that makes the model (5.3) to be infeasible is

$$\Sigma' = \{z, x_1, x_2, x_3, x_4, x_5, y_1, y_2, y_2, y_4\}$$

In fact most of the new signals added are not really necessary for obtaining a CSC support for signal x_4. Next section shows how to modify problem (5.3) in order to avoid unnecessary inclusions of signals in the computation of the CSC support for a given signal.

Implementation note

In order to avoid, as much as possible, unnecessary inclusions of signals in the CSC support of a signal, we add an objective function to the problem (5.3). Assume E_0 and E_1 are the set of signals in Σ with initial value `0` and `1`, respectively. We want to search for solutions of (5.3) such that the number of unbalanced signals is minimal. The minimization of the objective function

$$min\, [\texttt{code_change}(E_0, z) - \texttt{code_change}(E_1, z)]$$

avoids any vector z as a solution if there is another vector z' such that a signal with initial value 0 (1) is not balanced in $\texttt{code_change}(E_0, z)$ ($\texttt{code_change}$ (E_1, z)) but is balanced in $\texttt{code_change}(E_0, z')$ ($\texttt{code_change}(E_1, z')$). Therefore, this function reduces the number of unbalanced signals and, thus, less choices are possible when the model is feasible and a new unbalanced signal must be added to Σ'.

Using the objective function on the previous example, it can be realized that the set

$$\Sigma' = \{z, x_3, x_4, x_5, y_4\}$$

makes the model (5.3) to be infeasible, and therefore it is a CSC support valid for signal x_4. Note that it is avoided the inclusion of half of the signals computed in the previous section.

5.3.2 Projection into the CSC Support

Assume that for a non-input signal a its CSC support set CSC(a) has been computed by Algorithm CSC_Support. The next step is to derive the projection of the STG S into CSC(a) ($S_{\mathsf{CSC}(a)}$). The projection is computed by the transformations described in [16]. It is assumed that the projections

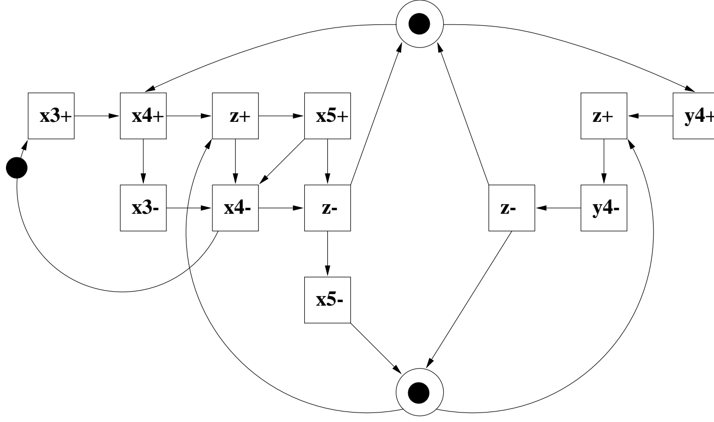


Figure 5.3: Projection of example PpArbCsc(2,3) onto signal x_4.

preserve trace equivalence on the set of traces with respect to the signals in $\mathsf{CSC}(a)$ (i.e.$L(S)|_{CSC(a)} = L(S_{CSC(a)})$).

Although $S_{\mathsf{CSC}(a)}$ and S are different STGs, next theorem shows that projection preserves CSC:

Theorem 5.3.1 *Let S be an STG with CSC for the non-input signal a, and $S_{\mathsf{CSC}(a)}$ be the projection preserving trace equivalence with S on the set CSC(a). Then $S_{CSC(a)}$ has CSC.*

Proof: By contradiction. Let us assume there are two reachable markings s'_1, s'_2 in $S_{CSC(a)}$ such that $s'_0 \stackrel{\sigma'_1}{\to} s'_1$, $s'_0 \stackrel{\sigma'_2}{\to} s'_2$, $\lambda(s'_1) = \lambda(s'_2)$ and only s'_1 enables some transition a_i* of signal a. Let $s_1, s_2, \sigma_1, \sigma_2$ such that $s_0 \stackrel{\sigma_1}{\to} s_1$, $s_0 \stackrel{\sigma_2}{\to} s_2$, $\sigma_1|_{\mathsf{CSC}(a)} = \sigma'_1$ and $\sigma_2|_{\mathsf{CSC}(a)} = \sigma'_2$. Finally, let $Trig(a_i*)$ denote the set of triggering transitions of a_i*. Two cases arise:

$\lambda(s_1) = \lambda(s_2)$: then given that we have CSC for signal a in S either both s_1 and s_2 enable a or none of them enables a. If both enable a then given that the set of trigger transitions of a_i* are in the CSC support of a and $s_2 \stackrel{a_i*}{\to}$ implies that $Trig(a_i*) \subseteq \sigma'_2$. But then if every trigger transition is fired in σ'_2, there is a state s reachable by firing some prefix of σ'_2 at s'_0 which enables a_i*. If $s \neq s'_2$ we have that $a_i* \in \sigma'_2$, and therefore $a_i* \in \sigma_2$ because $\sigma'_2 \subseteq \sigma_2$. At this point, the fact that $s_2 \stackrel{a_i*}{\to}$ implies that $\sigma_2 = \delta a_i * \gamma$, with $Trig(a_i*) \subseteq \gamma$. We can iterate this process again with γ, and using the finiteness of σ_2 we can conclude that $s'_2 \stackrel{a_i*}{\to}$, contradicting the assumption that a_i* is only enabled in s'_1. If neither s_1 nor s_2 enables a then the trace $\sigma'_1 a_i*$ belongs to $L(S_{\mathsf{CSC}(a)})$ but does not belong to $L(S)|_{\mathsf{CSC}(a)}$, contradicting the assumption $L(S)|_{CSC(a)} = L(S_{CSC(a)})$.

$\lambda(s_1) \neq \lambda(s_2)$: if both s_1 and s_2 enable a or none of them enables a, then the same reasoning of the previous case can be applied. If only one of them

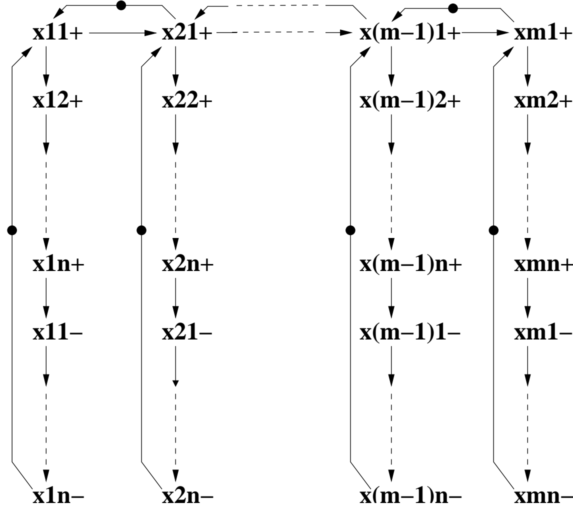


Figure 5.4: ART(m, n).

enables a, then $\mathsf{CSC}(a)$ is not a valid CSC support because some signal from $\Sigma \setminus \mathsf{CSC}(a)$ which makes $\lambda(s_1) \neq \lambda(s_2)$ hold is not added to $\mathsf{CSC}(a)$ in order to make $\lambda(s'_1)$ and $\lambda(s'_2)$ different. □

Figure 5.3 shows the projection for the signal x_4 of the STG depicted in Figure 5.2. The CSC support used is $\Sigma' = \{z, x_3, x_4, x_5, y_4\}$.

5.4 Experimental Results for USC/CSC Checking

The methods presented in Section 5.2 have been implemented in moebius, a tool for the synthesis of speed-independent circuits. The experiments have been performed on a *Pentium*^{TM} 4/2.53 Ghz and 512M RAM.

Several parameterizable examples have been used to compare with other existing approaches and to evaluate the impact of the size of the specification on the efficiency of the method. The following examples have been used:

- PPWK(m, n) and PPARB(m, n): examples modeling m pipelines weakly synchronized. In addition PPARB(m, n) also includes arbitration. Every benchmark in this set has CSC conflicts. These examples were obtained from [44].

- PPWKCSC(m, n) and PPARBCSC(m, n): a modification of the previous benchmarks to fulfill the CSC property.

- TANGRAMCSC(m, n): examples obtained by translating a synthetic Tangram program into a netlist of handshake components. The generic netlist is shown in Fig. 5.5, whereas a Tangram program from which this structure can be obtained is shown in Figure 5.6, where the whole

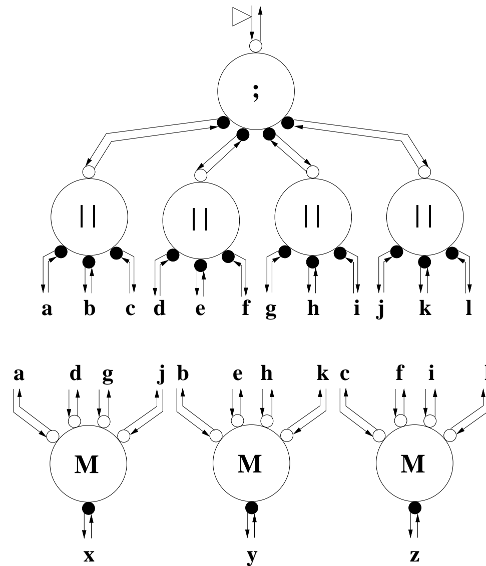


Figure 5.5: Netlist of handshake components from a Tangram program.

```
P1 (a:! int & b:! int & ... & l:! int).
begin
  forever do
    [a!1 || b!2 || c!3];
    [d!4 || e!5 || f!6];
    [g!7 || h!8 || i!9];
    [j!10 || k!11 || l!12];
  od
end

P2 (a:? int & b:? int & ... & l:? int &
    x:! int & y:! int & z:! int).
begin
  xr,yr,zr: var int
  forever do
    [[a?xr | d?xr | g?xr | j?xr]; x!xr] ||
    [[b?yr | e?yr | h?yr | k?yr]; y!yr] ||
    [[c?zr | f?zr | i?zr | l?zr]; z!zr]
  od
end
```

Figure 5.6: Tangram program from which the structure of Figure 5.5 can be obtained, as the parallel composition of P1 and P2.

system is build up as the parallel composition of the two processes defined P1 and P2. Each handshake component is specified as a Petri net and the final controller is obtained as the composition of all Petri nets.

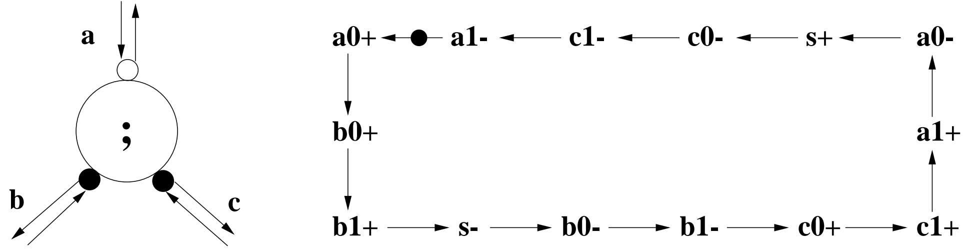


Figure 5.7: STG for a sequencer in a Tangram program.

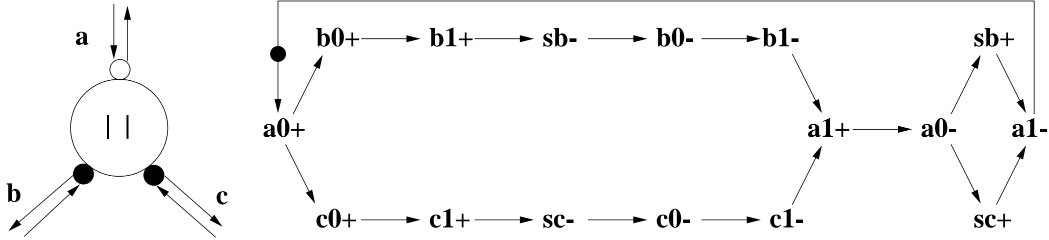


Figure 5.8: STG for a parallelizer in a Tangram program.

Figures 5.7, 5.8 and 5.9 show the STGs for the sequencer, parallelizer and mixer, respectively. The symbols ";", "||" and "M" represent sequencers, parallelizers and mixers, respectively. Each n-way component is implemented as a tree of 2-way components. This is a parameterizable benchmark that represents a typical controller obtained from the direct translation of languages like Tangram [6] or Balsa [28].

- Art(m, n): examples modeling a different way of synchronizing m pipelines. The STG is depicted in Figure 5.4. Every benchmark in this set has CSC conflicts.

- ArtCsc(m, n): transformation of the corresponding benchmark by means of the insertion of a new set of signals in order to fulfill the CSC property [16]. The nets in this class of benchmarks are extremely large compared to the corresponding benchmarks (for instance, Art$(30, 9)$ has 636 places while ArtCsc$(30, 9)$ has 2312) implying an exponential growth of the underlying state space. Therefore the check of CSC/USC for this benchmarks is a hard task.

The experiments for CSC/USC detection are presented in Tables 5.1 and 5.2. Each table reports the CPU time of each approach in seconds. We use 'time' and 'mem' to indicate that the algorithm did not complete in less than 10 hours or produced memory overflow, respectively. The tools for comparing the experimental results are:

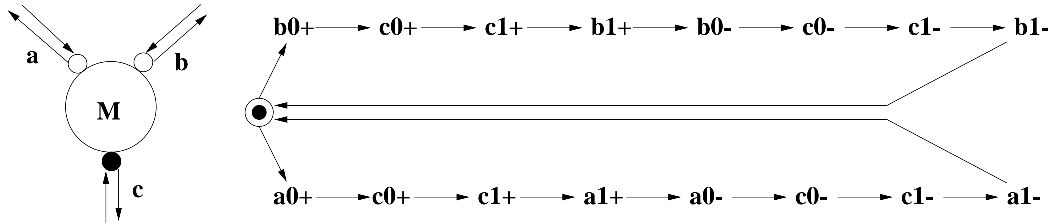


Figure 5.9: STG for a mixer in a Tangram program.

- CLP: the approach presented in [44] for the verification of USC/CSC. It uses non-linear integer programming methods and the unfolding of the net.
- SAT: the approach presented in [43] for the verification of CSC[1]. It uses a satisfiability solver and the unfolding of the net.
- ILP: the approach presented here.

From the results one can conclude, as it was expected, that checking USC is simpler than checking CSC, given the different nature of the two problems. Moreover, when some encoding conflict exists, the ILP solver can find it in short time. This is explained by the fact that proving the absence of encoding conflicts requires an exhaustive exploration of the *branch-and-bound* tree visited by ILP solvers. The superiority of ILP with respect to CLP and SAT is evident.

5.5 Conclusion

Although ILP is NP-complete, in practice there are real applications where the algorithms for mathematical programming perform very well. We believe that if the methods presented in this chapter are applied to well-structured STGs, the encoding problem can be considered as one of those good performing applications of ILP. The experimental results show this fact, over an extensive set of examples.

These well-structured STGs can be obtained by syntax-directed translation of HDL specifications in order to perform the synthesis [65, 9, 18]. Although the majority of the benchmarks used are not of this type, we have seen experimentally that the structures of the nets obtained from HDL specifications are always very regular, and therefore the results should be as good as the ones presented here. However, as future work we plan to do a deeper study on how good the methods perform on this type of specifications.

[1] Checking for USC is not implemented in SAT

benchmark	$\|P\|$	$\|T\|$	$\|\Sigma\|$	CLP	SAT	ILP
PpWk(2,9)	71	38	19	0.09	0.04	0.15
PpWk(2,12)	95	50	25	0.42	0.37	0.20
PpWk(3,6)	70	38	19	0.12	0.04	0.11
PpWk(3,9)	106	56	28	10.89	0.10	0.25
PpWk(3,12)	142	74	37	933.37	0.04	0.35
PpWkCsc(2,9)	72	38	19	3.14	0.18	0.16
PpWkCsc(2,12)	96	50	25	246.26	1.02	0.31
PpWkCsc(3,6)	72	38	19	2.97	0.04	0.19
PpWkCsc(3,9)	108	56	28	2075.48	0.31	0.41
PpWkCsc(3,12)	144	74	37	time	1.41	0.80
PpArb(2,9)	86	48	23	0.01	0.02	0.05
PpArb(2,12)	110	60	29	0.00	0.05	0.11
PpArb(3,6)	92	54	25	0.00	0.06	0.12
PpArb(3,9)	128	72	34	0.01	0.08	0.08
PpArb(3,12)	164	90	43	0.00	0.33	0.19
PpArbCsc(2,9)	88	48	23	40.89	0.61	0.28
PpArbCsc(2,12)	112	60	29	1021.51	16.24	0.45
PpArbCsc(3,6)	95	54	25	61.30	0.56	0.34
PpArbCsc(3,9)	131	72	34	time	1.52	0.69
PpArbCsc(3,12)	167	90	43	time	16.39	1.30
TangramCsc(3,2)	142	92	38	0.01	0.01	1.08
TangramCsc(4,3)	321	202	83	0.06	0.04	9.00
Art(10,9)	216	198	99	0.00	0.42	0.06
Art(20,9)	436	398	199	5.00	10.35	0.24
Art(30,9)	656	598	299	38.02	81.82	0.56
Art(40,9)	876	798	399	138.04	264.57	0.92
Art(50,9)	1096	998	499	377.00	630.41	1.46
ArtCsc(10,9)	752	630	315	time	14 m	3 m
ArtCsc(20,9)	1532	1270	635	time	mem	27 m
ArtCsc(30,9)	2312	1910	955	time	mem	1.5 h
ArtCsc(40,9)	3092	2550	1275	time	mem	3.5 h
ArtCsc(50,9)	3872	3190	1595	time	mem	7 h

Table 5.1: CSC detection for well-structured STGs.

benchmark	$\|P\|$	$\|T\|$	$\|\Sigma\|$	CLP	ILP
PpWk(3,9)	106	56	28	10.53	0.03
PpWk(3,12)	142	74	37	876.63	0.05
PpWkCsc(3,9)	108	56	28	2002.29	0.67
PpWkCsc(3,12)	144	74	37	time	1.17
PpArb(3,9)	128	72	34	0.01	0.06
PpArb(3,12)	164	90	43	0.00	0.08
PpArbCsc(3,9)	131	72	34	time	1.05
PpArbCsc(3,12)	167	90	43	time	1.69
TangramCsc(3,2)	142	92	38	0.01	1.07
TangramCsc(4,3)	321	202	83	0.06	6.52
Art(40,9)	876	798	399	146.02	1.26
Art(50,9)	1096	998	499	328.04	1.95
ArtCsc(40,9)	3092	2550	1275	time	14 m
ArtCsc(50,9)	3872	3190	1575	time	23 m

Table 5.2: USC detection for well-structured STGs.

Chapter 6

A Design Flow for the Synthesis of Asynchronous Circuits

This chapter presents a complete design flow for the synthesis of asynchronous circuits. It is built from the theory presented in the previous chapters. The design flow is capable of checking implementability conditions, such as, complete state coding, and deriving a gate netlist to implement the specified behavior. It can synthesize specifications with few thousands of transitions in the Petri net, providing a speed-up of several orders of magnitude with regard to other existing approaches. Moreover, the quality of the circuits derived is comparable to the optimal ones that can be obtained by using state-based logic minimization techniques. The complete design flow has been implemented in the tool moebius.

This chapter is based in the results presented in [14].

6.1 Introduction

Several methods have been presented in the literature for the synthesis of speed-independent circuits. One can classify them by the way the synthesis is performed:

- *State-based methods* [32, 83, 22] perform the synthesis from the state space of the specification. They can derive optimal implementations, but suffer from the state explosion problem and therefore can only synthesize small/medium size specifications.

- *Structural methods* [75, 80, 63], working at the level of the Petri net, can synthesize big size specifications but the behavior accepted in the design flow is restricted and the quality of the circuits is in general

not comparable to the state-based approach. The fact that the underlying state space of the specification is approximated can make these methods to overestimate or underestimate the fulfillment of the implementability conditions.

- *Hardware Description Language methods* [6, 28] where a syntax-directed translation into communicating handshaking components, that are later implemented as handshake circuits. This approach guarantees an implementation by construction, and moreover the resulting specification to synthesize is well-structured, provided the well-structuredness of a HDL. However, this approach does not exploit the potential optimizations that can be performed at the logic level. Typically, the size of the circuits obtained is linear on the size of the specification.

The design flow presented in the next section combines the three type of methods presented above, aiming at keeping the advantages of all three while avoiding their disadvantages.

State-based methods are used in the final stage of the flow, when the specification has been decomposed into smaller ones that can be easily handled by this type of methods. This allows to use the optimization capabilities of state-based approaches.

Structural methods, namely graph-based algorithms and linear algebra, are used along the design flow in order to support the steps needed for the speed-independent synthesis. The methods range from checking implementability conditions by solving a linear programming problem to transform the specification via Petri net transformations to improve the quality of the resulting implementation.

Finally, it is assumed that the specifications are well-structured. The flow can accept any behavior represented in a free-choice Petri net. We believe that specifications derived from HDL can be mapped into this class of nets [9, 8]. Moreover, the fact that HDL are well-structured by construction helps in alleviating the complexity of the structural methods used.

6.2 The Design Flow

In this section a new design flow for the synthesis of asynchronous circuits is presented. The major gains with respect to previous methods are:

- The synthesis of a speed-independent circuit implementing non-input signals can be done in most of the cases, despite of the size of the whole specification.

- The use of structural methods allows to deal with very large specifications.

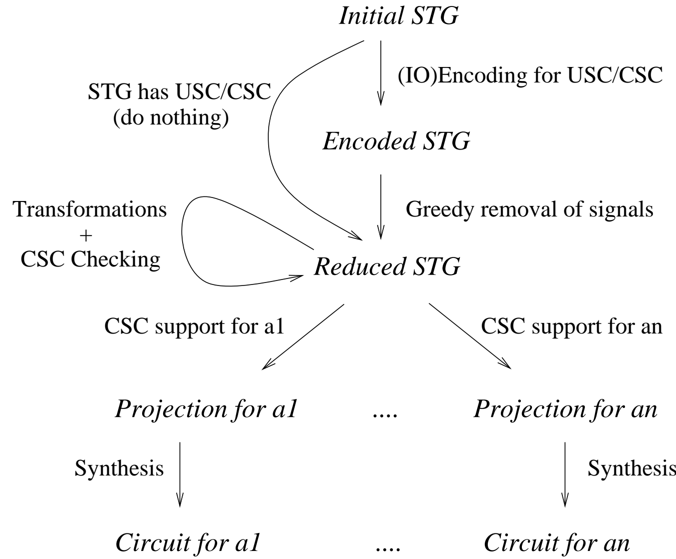


Figure 6.1: Synthesis of asynchronous circuits.

- Optimization techniques can be applied in some stages of the process.
- The correct interaction with the environment can be preserved.

The synthesis flow is depicted in Figure 6.1. Given a consistent STG with underlying FCLSPN, the encoding technique presented in Chapter 4 is applied when the ILP check for correct encoding described in Chapter 5 gives a positive answer. The resulting STG contains a new set of signals that ensure the USC or CSC property, depending on the encoding technique applied. Since many of these signals may be unnecessary to guarantee either USC or even CSC, they are iteratively removed using greedy heuristics until no more signals can be removed without violating the USC/CSC property. This greedy process makes use of the ILP methods presented in Chapter 5. The reduced STG is next projected onto different sets of signals to implement each individual output signal.

The kit of transformations presented in Chapter 4 can also be applied once the STG is known to be correctly encoded. This is represented in Figure 6.1 by a self-loop in the *Reduced STG* node. Although its application is not automatic in the tool, the designer can apply them and each time verify if the transformed STG is correctly encoded. For the automation of the application of the transformations, several heuristics can be used. For instance, transformation ϕ_r (concurrency reduction) can be used to improve the performance of the circuit by means of reducing the length of the *critical cycle* of the net. Some heuristics for area estimation can also be applied: for instance, using structural methods [63] it can be estimated the number of literals in each excitation region, and try to guide the application of the

rules to reduce it. Some preliminary work on this can be found in [8].

One weak point of the design flow presented is that, when the projections are computed, several redundant places can appear, because transformation ϕ_e can increase the number of places in the net each time it is applied. However, some heuristics have been implemented in the tool that, using sufficient conditions for a place to be redundant (which can be evaluated in constant time provided that the transformations are local), find the majority of such redundant places. The redundant places found are removed immediately. This solution alleviates considerably the problem.

It is important to note that the design flow presented here is only restricted to free choice STGs when the specification has encoding problems. If the initial STG has a correct encoding and the projections applied preserve trace equivalence, then we can guarantee correctness on the resulting implementation. This will be illustrated in the next section, where a non-free choice specification is synthesized by our tool.

6.3 Synthesis Examples

In this section we present the complete synthesis of two tiny specifications. Despite of their magnitude, the examples used illustrate how the synthesis is performed in the design flow proposed.

The first example used, the VME Bus Controller, allows us to describe each step of the design flow: encoding, greedy removal of signals, CSC support computation, projection and synthesis.

The second example presented is PpArbCsc(2,3), a non-free choice specification. It is shown that, provided that the STG is initially correctly encoded, the design flow can synthesize it despite of not being a free choice net.

6.3.1 Synthesis of the VME Bus Controller

Let us explain step by step how to perform the synthesis of the VME Bus Controller example, depicted in Figure 4.3 (left). In Section 5.2, it is detected a USC conflict, which is also a CSC conflict. Then, in order to be able to synthesize the STG we have to transform it first. We apply the encoding technique of Section 4.3, which guarantees USC on the resulting STG. The STG obtained is shown in Figure 4.3(right).

From the STG of Figure 4.3(right), a greedy process for the removal of the signals inserted by the encoding technique is done. The process is shown in Figures 6.2, 6.3 and 6.4. At each step, one signal is eliminated if, when removed from the STG, the new specification does not has encoding conflicts.

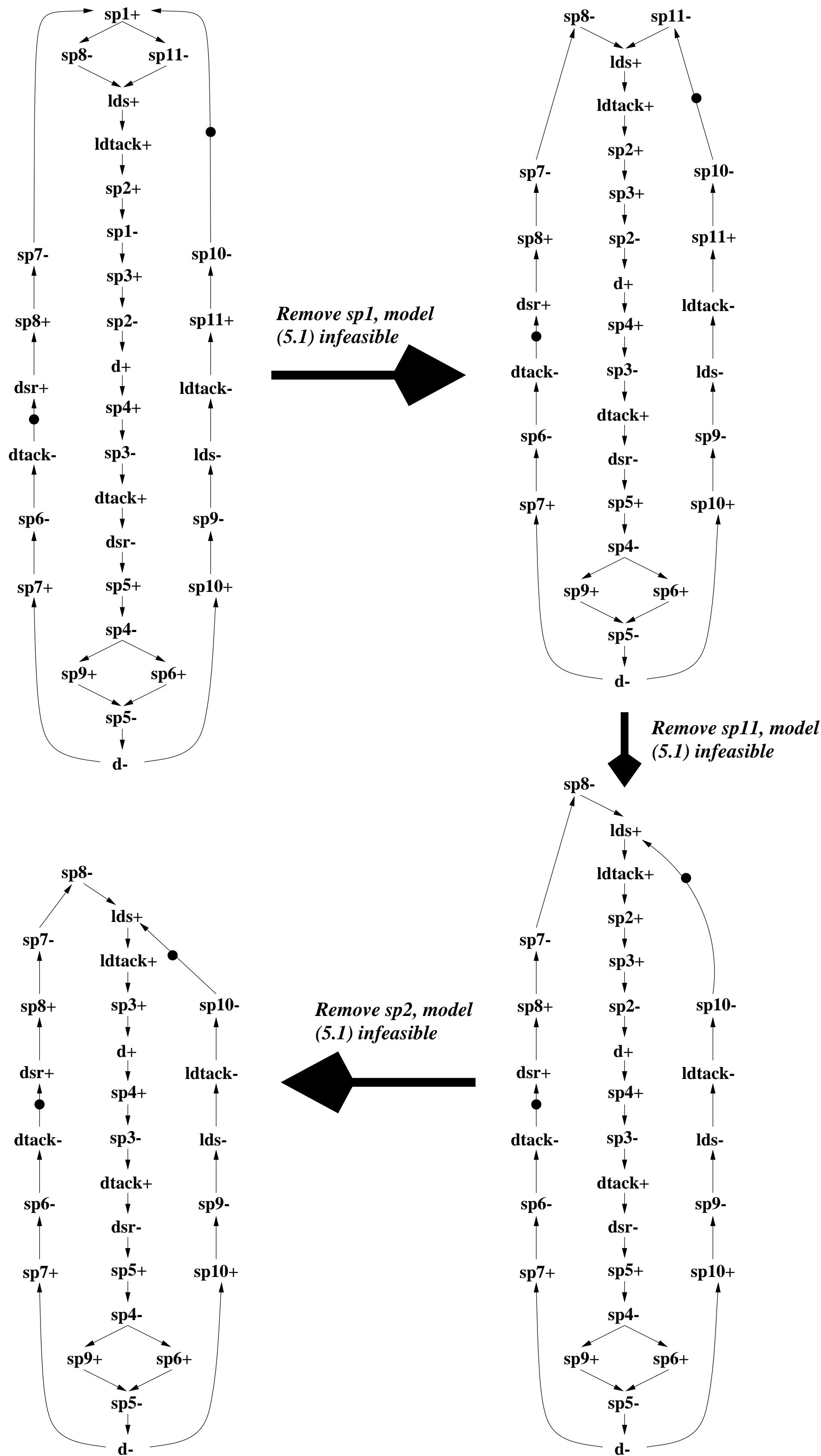


Figure 6.2: Greedy removal of signals $sp1$, $sp11$ and $sp2$.

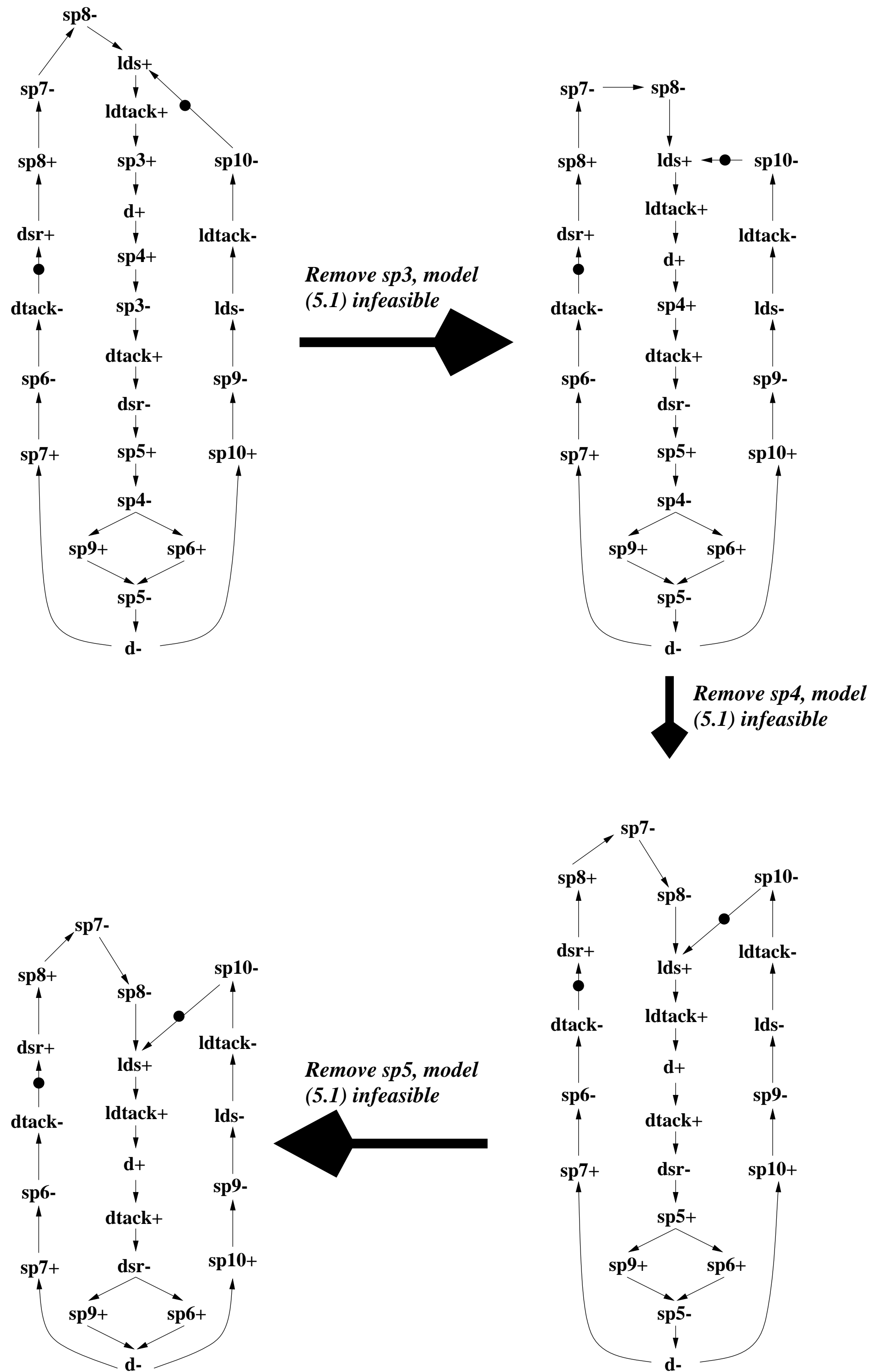


Figure 6.3: Greedy removal of signals $sp3$, $sp4$ and $sp5$.

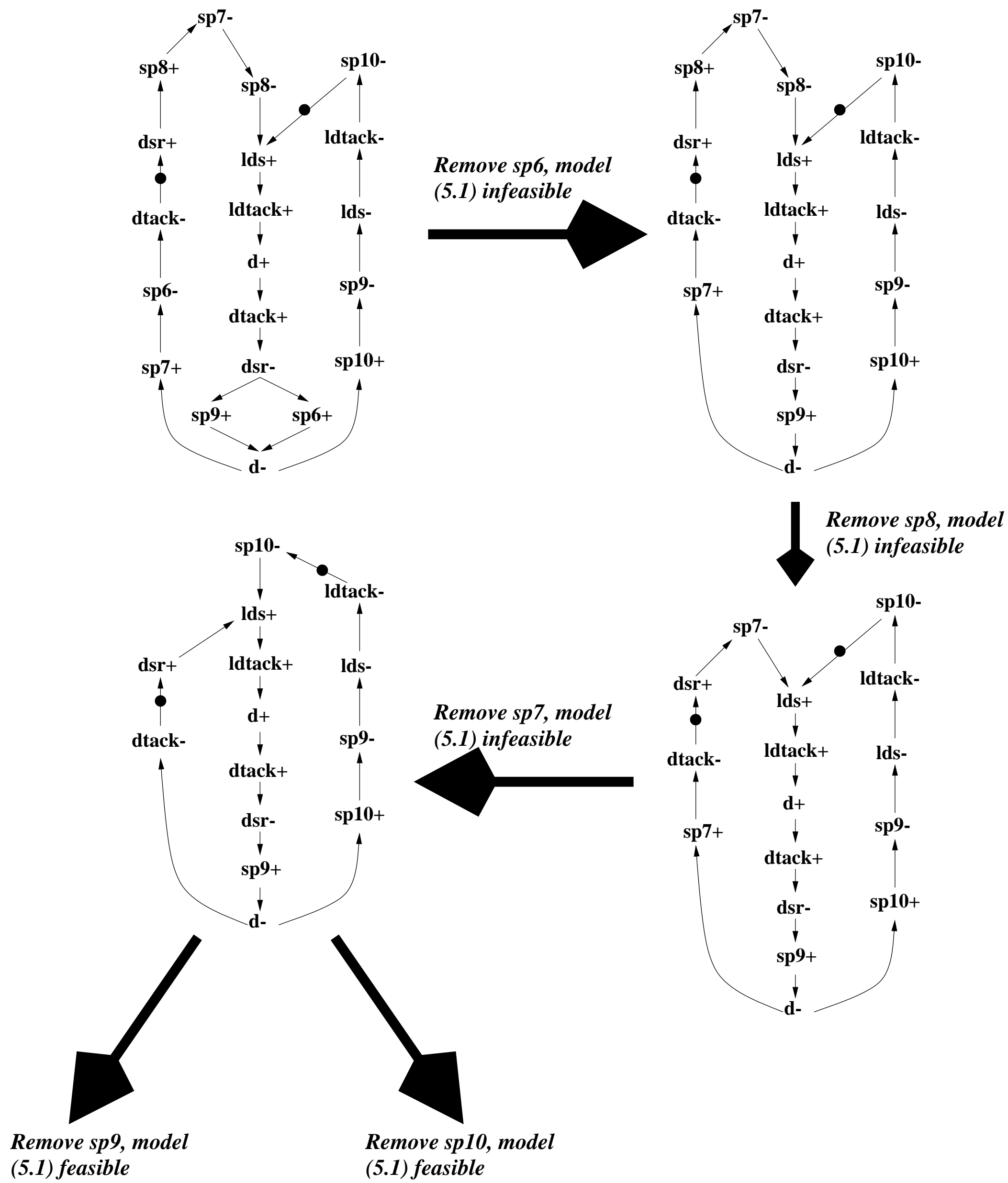


Figure 6.4: Greedy removal of signals $sp6$, $sp8$ and $sp7$.

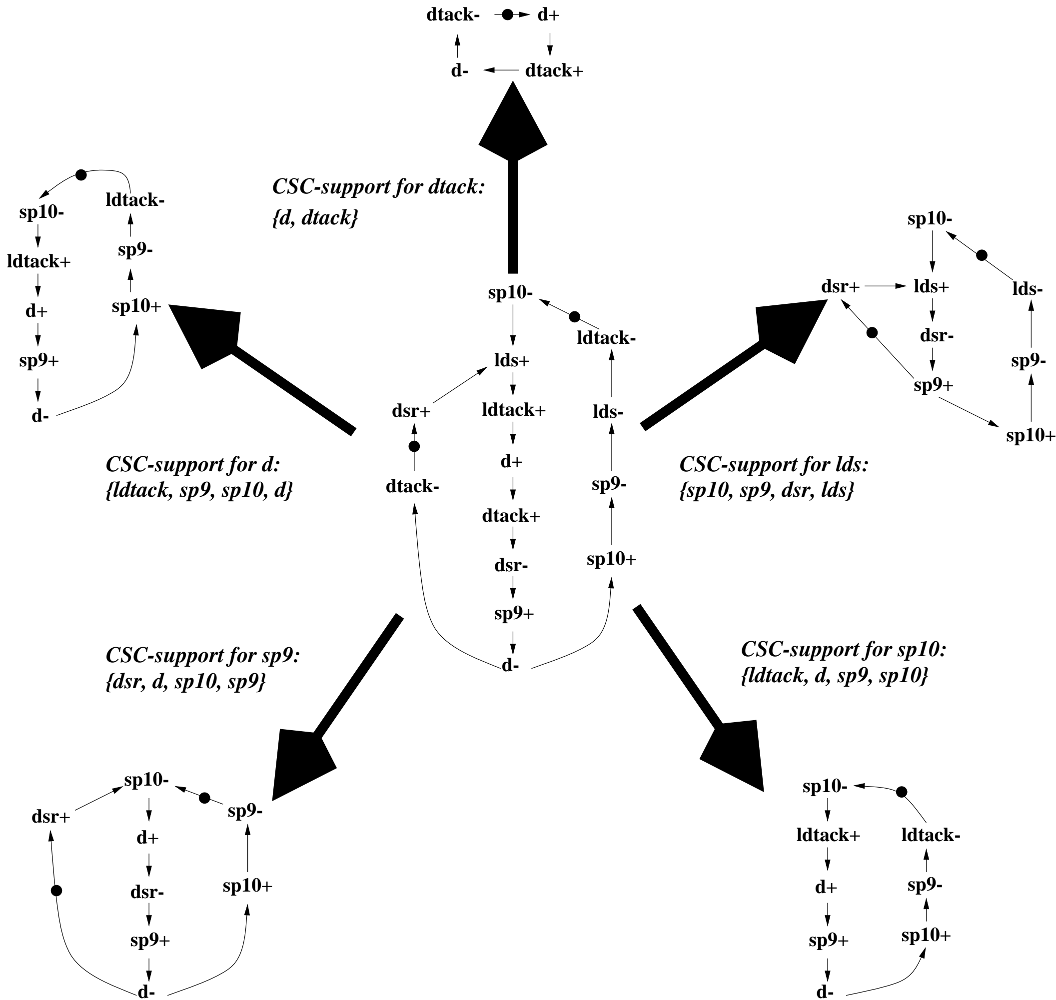
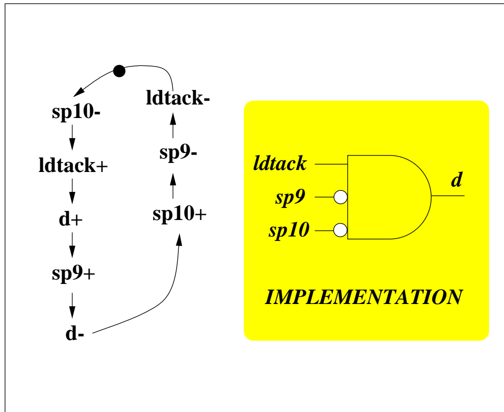


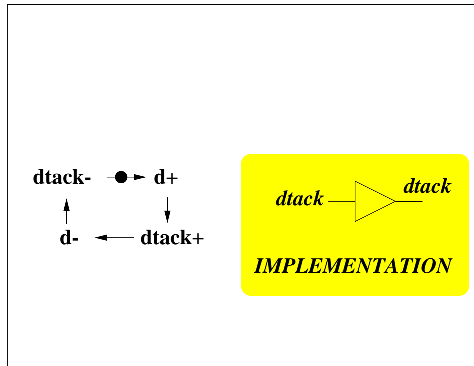
Figure 6.5: CSC support computation and projection for the VME Bus Controller example.

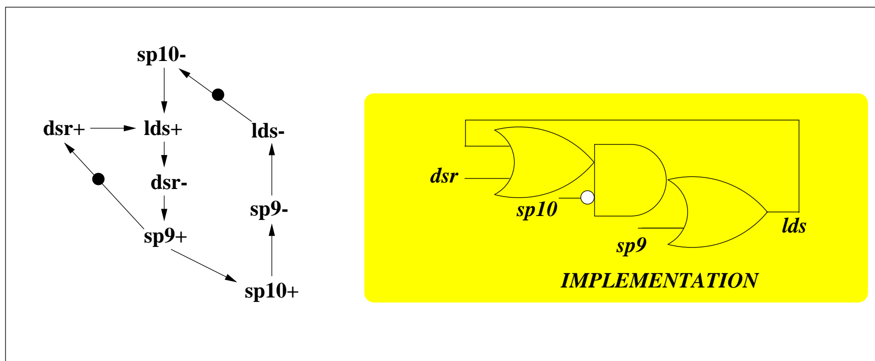
From the final STG shown in Figure 6.4, neither signal sp_9 nor signal sp_{10} can be removed without introducing encoding conflicts. This finishes the process of greedy removal of signals.

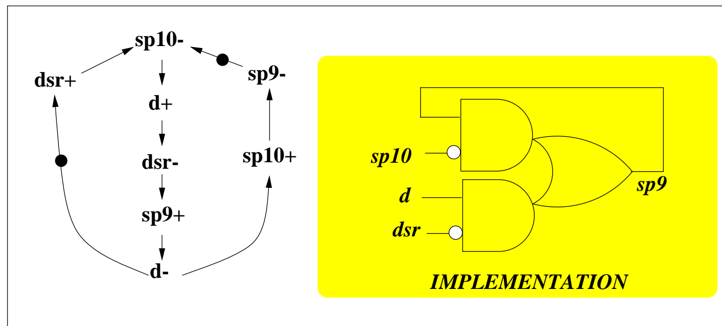
Once the reduced STG is reached, it must be computed the CSC support for each non-input signal, applying the algorithm CSC_Support, which is presented in Section 5.3.1. Afterwards the projection of the STG into the CSC support of each non-input signal is performed. This is illustrated in Figure 6.5.

And finally, a speed-independent synthesis of each projection is performed. Figure 6.6 presents the synthesis for the five non-input signals remaining in the STG. Provided the small size of the projections, state-based techniques can be applied for performing the synthesis.









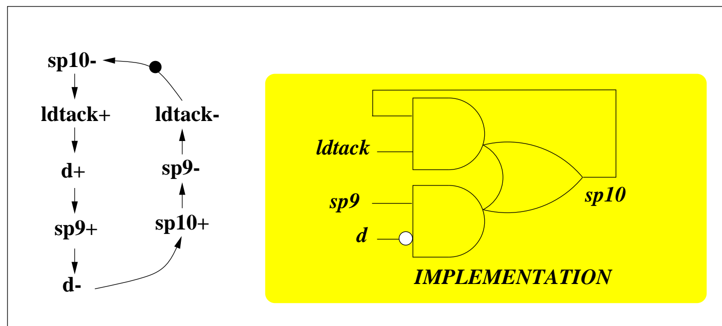


Figure 6.6: Speed-independent synthesis of the VME Bus Controller.

In the example the tool petrify has been used for the synthesis, using the 'complex gates' architecture.

When synthesizing the projection for each non-input signal a, every signal in the projection but a is considered as an input signal. This prevents the logic synthesis to try to solve possible conflicts for the rest of signals.

6.3.2 Synthesis of the PpArbCsc(2,3) Example

If each step of the design flow presented in this chapter must be performed, its application is limited to STGs with underlying FCLSPNs. However, if the initial specification has a correct encoding, the design flow can be applied independently of the class of the underlying Petri net, provided that the projections preserve trace equivalence.

In this section we present the synthesis of the PpArbCsc(2,3) example, shown in Figure 5.2. It is a non-free choice specification, which has a correct encoding. The example models a 2-pipeline synchronized, with arbitration between some of the output signals. We focus on the synthesis of one of the pipelines, represented by the signals $x1$, $x2$, $x3$, $x4$, $x5$ and z. The synthesis of the remaining pipeline is symmetrical.

Provided that the initial STG has a correct encoding, the first two steps of the design flow (i.e., encoding and greedy removal of signals) are not applied. The CSC support computation and projection for signals $x1$, $x2$, $x3$, and for signals $x4$, $x5$ and z is shown in Figures 6.7 and 6.8, respectively. The CPU time for computing the support and performing the projection is negligible for such a tiny example.

From each projection, the speed-independent synthesis is performed using the tool petrify. Figures 6.9 and 6.10 depict the final implementation for signals $x1$, $x2$, $x3$, and for signals $x4$, $x5$ and z, respectively. The 'technology mapping' option of petrify has been chosen, in order to realize that C-elements can be used for the implementation of some signals in the pipeline. Apart for the synthesis of signal $x4$, where a 3-way OR gate and a 3-way AND gate are needed for its implementation, the rest of signals can be synthesized with 2-way gates.

6.4 Experimental Results

Experiments have been performed on some of the benchmarks described in the previous chapter. Table 6.1 shows experiments on synthesis to check the quality of the generated circuits. The column 'Lit' reports the number of literals, in factored form, of the netlist. The results are compared with the circuits obtained by `petrify` [22], a state-based synthesis tool, on the same controllers. From the reported CPU time, the time needed for computing a support and for projection was negligible when compared to the time needed for deriving logic equations.

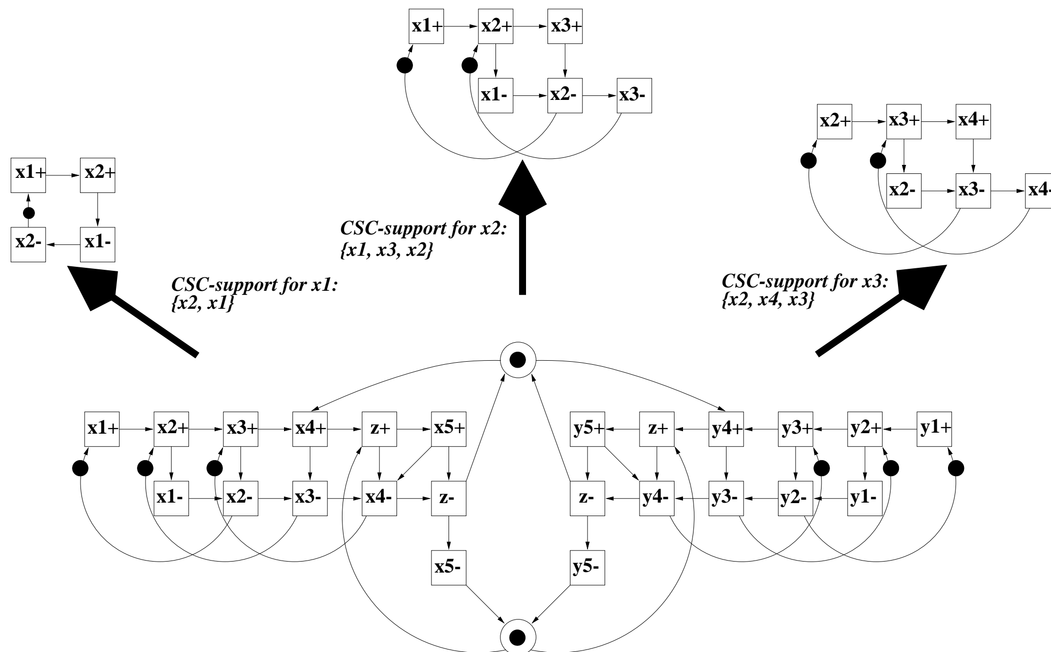


Figure 6.7: CSC support computation and projection for PpArbCsc(2,3) example: signals $x1$, $x2$, $x3$.

Table 6.1 shows that the quality of the circuits obtained by the ILP-based technique is comparable to that of the circuits obtained by `petrify`. Moreover it is clear that our approach can deal with larger specifications.

TheTangramCsc$(4,3)$ example, shown in Figure 5.5, illustrates the suitability of our approach for the synthesis of specifications generated from a HDL. According to [6], the cost of implementing the handshake components is the following[1]:

<table>
<tr><th>Component</th><th>C-elements</th><th>2-input gates</th><th>literals</th></tr>
<tr><td>2-way sequencer (;)</td><td>1</td><td>2</td><td>9</td></tr>
<tr><td>2-way parallelizer (||)</td><td>3</td><td>4</td><td>23</td></tr>
<tr><td>2-way mixer (M)</td><td>2</td><td>1</td><td>12</td></tr>
</table>

The circuit in Fig. 5.5 has 3 sequencers, 8 parallelizers and 9 mixers: 319 literals. This would be the cost obtained by a *syntax-directed translation*. The cost obtained by logic synthesis methods is significantly smaller.

6.5 Conclusion

In this chapter we have presented a new design flow for the synthesis of asynchronous circuits. In the approach proposed, all but the last step are

[1] A C-element is assumed to cost 5 literals: $c = ab + c(a + b)$.

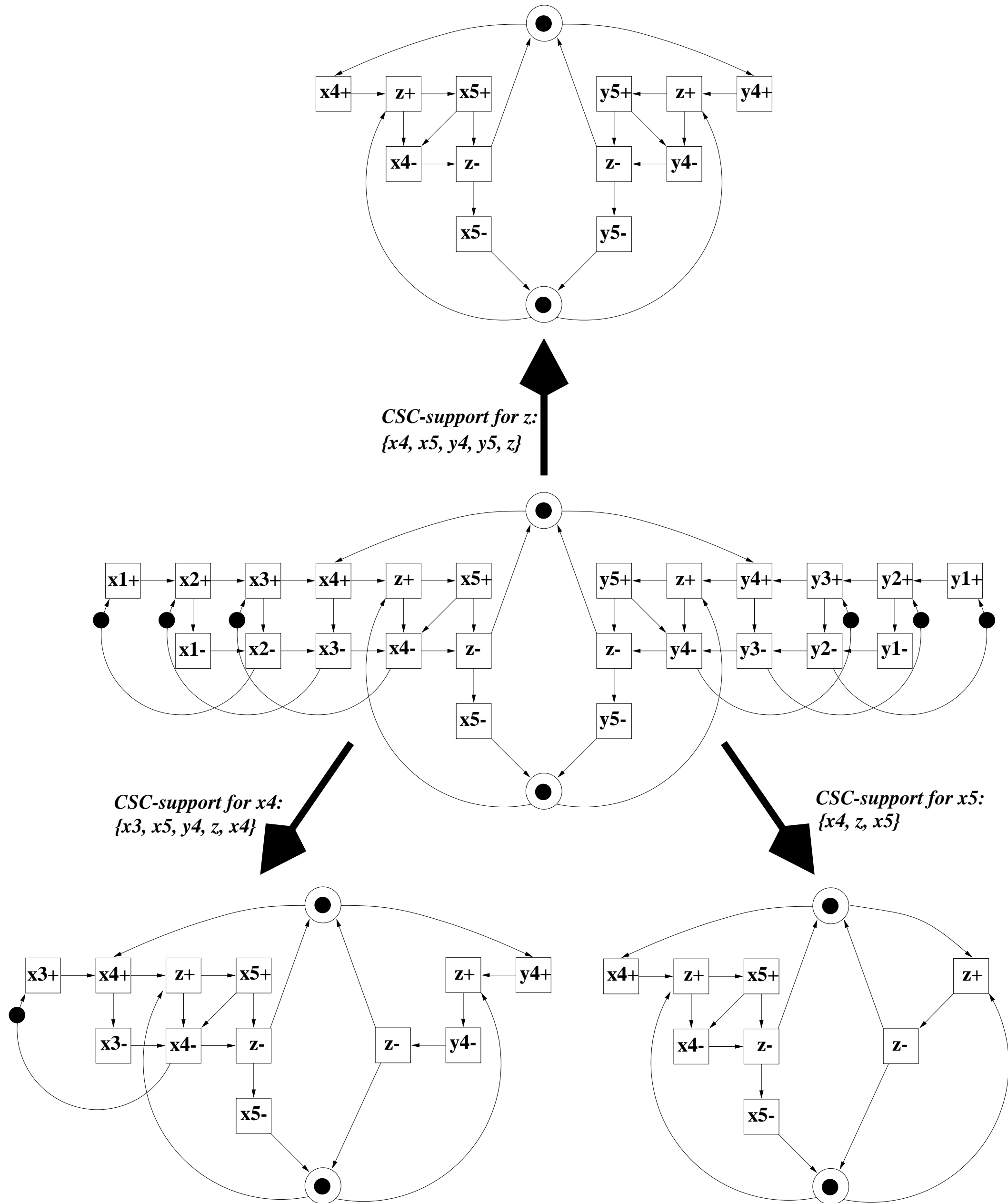


Figure 6.8: CSC support computation and projection for PpArbCsc(2,3) example: signals $x4$, $x5$, z.

structural. This allows to deal with specifications that are impossible to synthesize with state-based methods. In the last step, i.e. when the logic synthesis is performed, state-based methods can be applied because the projections are typically small.

We have measured the quality of the resulting circuits by means of count-

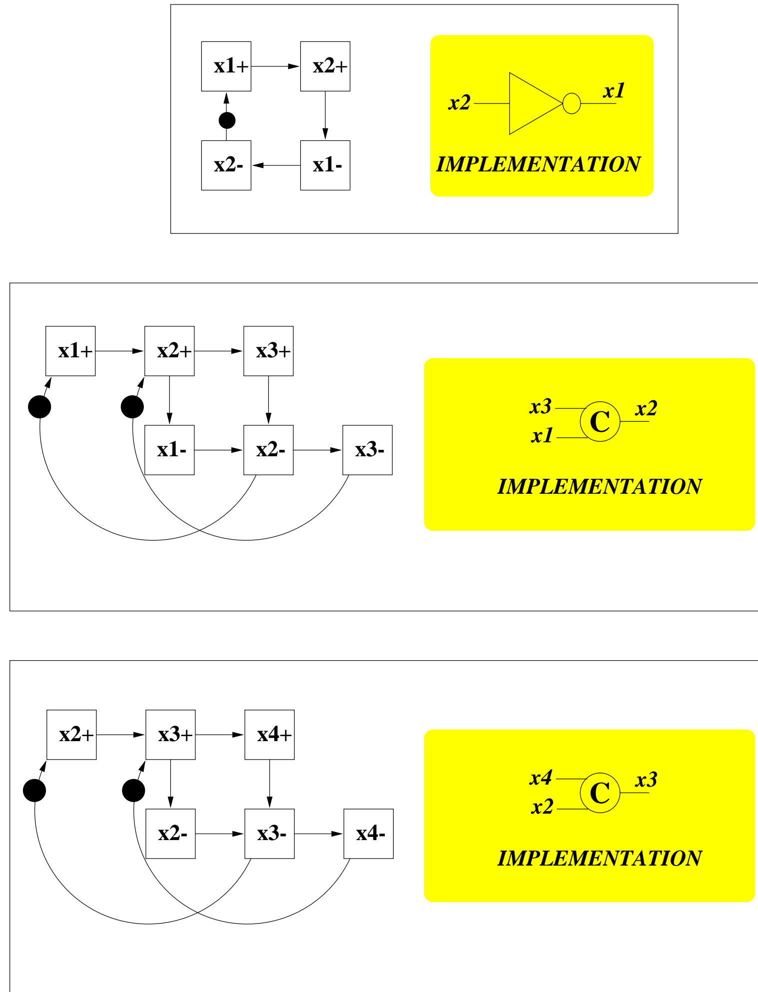
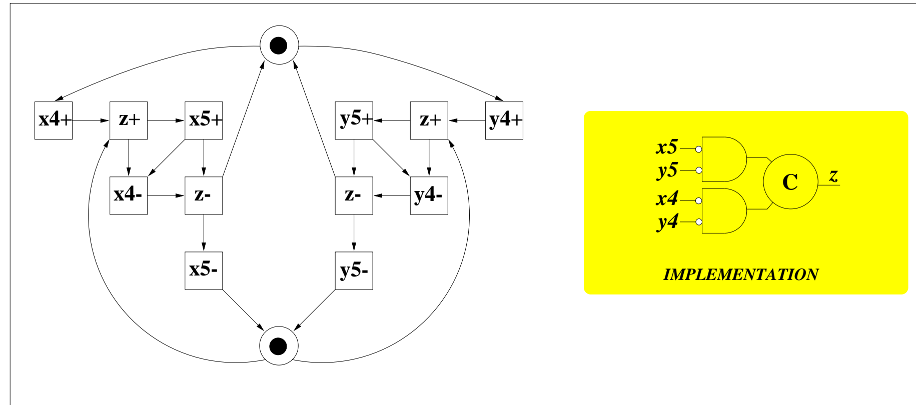
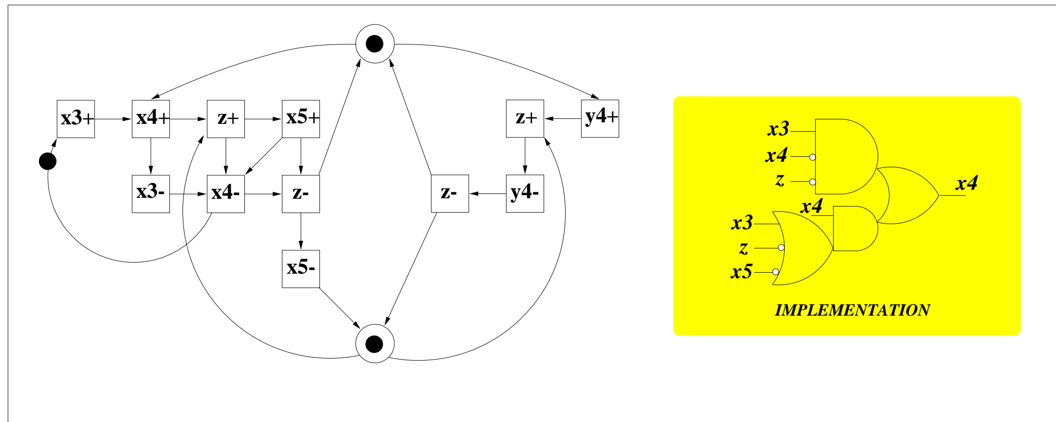


Figure 6.9: Speed-independent synthesis of the PpArbCsc(2,3) example: signals $x1$, $x2$, $x3$.

ing the number of literals. This measure shows that the circuits synthesized by our approach are comparable to those obtained by global optimization techniques. As a future work, it would be interesting to have other parameters for measuring the quality of the synthesized circuits, and to be able to guide the projection and synthesis by these measures.

Moreover, it would also be interesting to do a deep comparison of the method presented in this work with respect to some alternative methods for synthesizing AFSMs. For instance, in [19], algorithms are proposed for this purpose, which are able to synthesize specifications where speed-independent synthesis tools like petrify fail. However, it should be mentioned that the methods proposed there restrict the type specifications to synthesize (no input-output concurrency is allowed), while neither petrify nor the approach





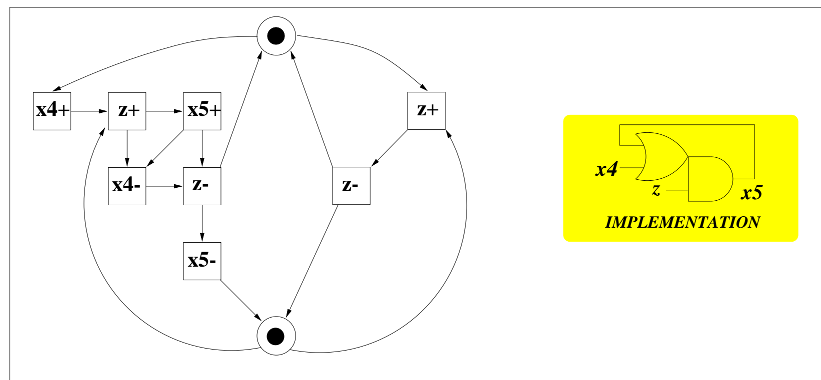


Figure 6.10: Speed-independent synthesis of the PpArbCsc(2,3) example: signals $x4$, $x5$, z.

presented here do this type of restriction. As an example, the VME Bus controller specification used in several chapters of this book can not be synthesized by the methods presented in [19].

benchmark	states	$\|P\|$	$\|T\|$	$\|\Sigma\|$	Lit.		CPU	
					Pfy	ILP	Pfy	ILP
PpWkCsc(2,6)	8192	47	26	19	57	57	5	1
PpWkCsc(2,9)	524.288	71	38	19	87	87	49	2
PpWkCsc(3,9)	2.7×10^7	106	56	28	–	130	mem	3
PpWkCsc(3,12)	2.2×10^{11}	142	74	37	–	117	time	3
PpArbCsc(2,6)	61440	62	36	17	77	77	21	83
PpArbCsc(2,9)	3.9×10^6	110	60	29	107	107	185	59
PpArbCsc(3,9)	3.3×10^9	131	72	34	163	165	10336	289
PpArbCsc(3,12)	1.7×10^{12}	167	90	43	–	210	time	608
TangramCsc(3,2)	426	142	92	38	97	103	56	146
TangramCsc(4,3)	9258	321	202	83	–	247	mem	2 h

Table 6.1: Support computation, projection and synthesis compared to state-based approach.

Chapter 7

Conclusions

The development of formal methods for the synthesis of concurrent systems was the main goal of this work. Structural methods was the tool to implement our goal.

Despite their simplicity, asynchronous circuits are difficult to design and verify. Provided that they are the simplest type of concurrent systems, we focused our attention on this type of systems. Several problems appear in the synthesis of asynchronous circuits. The reason is that, given that there is no global clock synchronizing each component of the circuit, the introduction of errors in the early stages of the specification happens very often. One of the necessary conditions for a specification to be implementable is the encoding condition. When the specification has a correct encoding, the circuit knows which signal must generate and when. Therefore, to guarantee a correct encoding is a crucial problem in the synthesis of asynchronous circuits.

The problem of encoding was the first one that we faced in this work. Assuming that the specification was given as an interpreted Petri net, we developed a structural encoding technique that guarantees a correct encoding in the transformed net. The technique, described in Chapter 4, is based on the insertion of signals into the original Petri net. To the best of our knowledge, it is the first technique that guarantees an encoding for the class of STGs with underlying FCLSPN.

The encoding technique developed brought us to a new problem: if we transform the initial specification, do we have to transform also the environment in order to guarantee that both, the specification and the environment, will understand each other ? The question is in fact more general, and can be asked for any type of reactive system: what are the conditions needed to guarantee that two reactive systems can interact without having errors or deadlocks ? The answer can be found in Chapter 3, where the reactive systems are specified as an automaton and the notion is called I/O Compatibility. We also developed a polynomial time procedure to verify I/O Compatibility, and provided sufficient conditions when some facts of the

systems are known, like observational equivalence.

We developed a kit of transformation rules, applied to a Petri net specifying a reactive system, that preserve the I/O Compatibility between the system and its environment, and adapted the encoding technique to preserve I/O Compatibility. This provides to the designer of a reactive system some freedom to change the initial specification, while preserving its correct functioning on the environment where the system is supposed to work. The theory is described in the first part of Chapter 4.

Afterwards we realized that the design flow that we had in mind needed efficient ways of verifying whether the initial specification has a correct encoding. This would help, for instance, to prevent the application of the encoding technique when the specification already has a correct encoding. Using the marking equation of the Petri net, we have developed ILP models for a fast verification of either USC or CSC. The experimental results show a significant (orders of magnitude) speed-up with respect to existing methods. Moreover, ILP models were introduced for supporting the decomposition (projection) of the initial specification into smaller ones while preserving the implementability conditions. This crucial step allowed us to use, in the final stage of our design flow, state-based algorithms because the resulting projections were typically small. The models were introduced in Chapter 5.

Finally we merged all the theory developed into a full design flow for the synthesis of asynchronous circuits. Consequently we implemented it as the moebius tool. Our approach for synthesis, presented in Chapter 6, provided a considerable speed-up with respect to existing approaches. When compared to state-based methods, which can perform global optimization techniques, the tool proved to obtain circuits with similar area. However, the tool needed considerably less time to perform the synthesis. Very often it happened that our approach was able to synthesize specifications not implementable by state-based methods, due to the state explosion problem.

For each one of the problems faced in this work, more work is expected to be done in the future. First, we are interested in extending the encoding technique to bigger classes than the FCLSPN. This will allow to apply the full design flow to more specifications.

Second, extending also the kit of synthesis rules presented in Chapter 4 will allow to offer more situations where the system can be changed. Three dimensions can be considered for extension: i) to add new rules to the existing kit, ii) to weaken the conditions under which the rules can be applied, and iii) to extend the class of Petri nets where the rules can be applied.

Third, it is interesting to adapt the I/O Compatibility to more complex models. Some work has been doing for adapting I/O Compatibility to the model of Team Automata [29]. Team automata is a formal framework for the specification and analysis of Computer Supported Cooperative Work (CSCW). In a team automata, the type of synchronization between subsys-

tems is also a variable of the model. Several concepts change when trying to adapt the I/O Compatibility notion to team automaton: the concept of deadlock, for instance, is strongly related to the type of synchronization that the components must follow, and therefore, depending on the type of synchronization established, a system can be considered deadlocked or not.

As a fourth research direction, we are interested in the study of the complexity of the encoding problem for general marked graphs. More specifically, we want to find ILP models where the integrality constraint can be avoided and the model, provided that the marking equation characterizes reachability for marked graphs, still characterizes the problem of the encoding but only has polynomial complexity.

Finally it is also interesting to study techniques that alleviate the complexity of the ILP models introduced in Chapter 5. One way of alleviating the complexity is reducing the incidence matrix of the problem. The problem that we want to study is how to reduce the incidence matrix in a way that no spurious solutions are introduced when using the marking equation, to approximate the reachability graph. We have developed some models that, using an SM-cover of the Petri net, reduce drastically the size of the matrix but still some work must be done to guarantee that no new spurious solutions are introduced.

Bibliography

[1] *SIAM Journal on Optimization.*

[2] A. Arnold. *Finite Transition Systems.* Prentice Hall, 1994.

[3] International Semiconductor Industry Association. International technology roadmap for semiconductors. Technical report.

[4] Peter A. Beerel. *CAD Tools for the Synthesis, Verification, and Testability of Robust Asynchronous Circuits.* PhD thesis, Stanford University, 1994.

[5] C. H. (Kees) van Berkel, Mark B. Josephs, and Steven M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, February 1999.

[6] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation.* Cambridge University Press, 1993.

[7] G. Berthelot. Checking Properties of Nets Using Transformations. In G. Rozenberg, editor, *Advances in Petri Nets 1985*, volume 222 of *Lecture Notes in Computer Science*, pages 19–40. Springer-Verlag, 1986.

[8] I. Blunno, A. Bystrov, J. Carmona, J. Cortadella, L. Lavagno, and A. Yakovlev. Direct synthesis of large-scale asynchronous controllers using a petri-net based approach. In *Handouts of the Asynchronous Circuits Design (ACiD) Workshop*, Grenoble, France, January 2000.

[9] Ivan Blunno and Luciano Lavagno. Automated synthesis of micropipelines from behavioral Verilog HDL. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 84–92. IEEE Computer Society Press, April 2000.

[10] Randal Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computer-Aided Design*, 35(8):677–691, 1986.

[11] Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits.* Springer-Verlag, 1995.

[12] J. Carmona and J. Cortadella. On the realization of reactive systems. Report LSI-01-24-R, Universitat Politècnica de Catalunya, May 2001.

[13] J. Carmona and J. Cortadella. Input/Output Compatibility of Reactive Systems. In *Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Portland, Oregon, USA, November 2002. Springer-Verlag.

[14] J. Carmona and J. Cortadella. ILP Models for the Synthesis of Asynchronous Control Circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, San Jose, California, USA, November 2003.

[15] J. Carmona, J. Cortadella, and E. Pastor. A structural encoding technique for the synthesis of asynchronous circuits. In *Int. Conf. on Application of Concurrency to System Design*, pages 157–166, Newcastle Upon Tyne, United Kingdom, June 2001.

[16] J. Carmona, J. Cortadella, and E. Pastor. A structural encoding technique for the synthesis of asynchronous circuits. *Fundamenta Informaticae*, pages 135–154, April 2001.

[17] J. Carmona, J. Cortadella, and E. Pastor. Synthesis of reactive systems: application to asynchronous circuit design. In J. Cortadella, A. Yakovlev, and G. Rozenberg, editors, *Advances in Concurrency and Hardware Design (ACHD)*, volume 2549. Springer-Verlag, 2002.

[18] Tiberiu Chelcea, Andrew Bardsley, Doug Edwards, and Steven M. Nowick. A burst-mode oriented back-end for the Balsa synthesis system. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 330–337, March 2002.

[19] Tiberiu Chelcea and Steven M. Nowick. Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems. In *Proc. ACM/IEEE Design Automation Conference*, June 2002.

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

[21] Edmund M. Clarke, Jeannette M. Wing, Rajeev Alur, Rance Cleaveland, David Dill, Allen Emerson, Stephen Garland, Steven German, John Guttag, Anthony Hall, Thomas Henzinger, Gerard Holzmann, Cliff Jones, Robert Kurshan, Nancy Leveson, Kenneth McMillan, J. Moore, Doron Peled, Amir Pnueli, John Rushby, Natarajan Shankar,

Joseph Sifakis, Prasad Sistla, Bernhard Steffen, Pierre Wolper, Jim Woodcock, and Pamela Zave. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[22] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic synthesis of asynchronous controllers and interfaces.* Springer-Verlag, 2002.

[23] René David. Modular design of asynchronous circuits defined by graphs. *IEEE Transactions on Computers*, 26(8):727–737, August 1977.

[24] R. de Nicola and M. C. B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34(1-2):83–133, November 1984.

[25] J. Desel and J. Esparza. Reachability in cyclic extended free-choice systems. *TCS 114, Elsevier Science Publishers B.V.*, 1993.

[26] J. Desel and J. Esparza. *Free Choice Petri Nets.* Cambridge University Press, Cambridge, Great Britain, 1995.

[27] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* ACM Distinguished Dissertations. MIT Press, 1989.

[28] Doug Edwards and Andrew Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, 2002.

[29] Clarence A. Ellis. Team automata for groupware systems. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge*, pages 415–424, 1997.

[30] Clarence A. Ellis, Simon J. Gibbs, and Gail Rein. Groupware: some issues and experiences. *Communications of the ACM*, 34(1):39–58, 1991.

[31] Joost Engelfriet:. Determinacy - (observation equivalence = trace equivalence). *Theoretical Computer Science*, 36:21–25, 1985.

[32] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, NY, July 1999.

[33] Claude Girault and Rudiger Valk. *Petri Nets for Systems Engineering. A Guide to Modeling, Verification and Applications.* Springer, 2003.

[34] G.J. Milne. CIRCAL: A calculus for circuit descriptions. *Integration, the VLSI Journal*, 1(2–3):121–160, October 1983.

[35] M. Hack. *Analysis of production schemata by Petri nets.* M.s. thesis, MIT, February 1972.

[36] D. Harel and A. Pnueli. On the development of reactive systems. In Krzystof R. Apt, editor, *Logic and Model of Concurrent Systems*, volume 13 of *NATO ASI*, pages 477–498. Springer-Verlag, October 1984.

[37] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

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

[39] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, 1985.

[40] B. Jonsson. Compositional verification of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, 1994.

[41] Mark B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.

[42] V. Khomenko, M. Koutny, and A. Yakovlev. Detecting state coding conflicts in STG unfoldings using SAT. Technical report, Department of Computer Science, Newcastle Upon Tyne, United Kingdom, September 2002.

[43] V. Khomenko, M. Koutny, and A. Yakovlev. Detecting state coding conflicts in STG unfoldings using SAT. In *Int. Conf. on Application of Concurrency to System Design*, June 2003.

[44] Victor Khomenko, Maciej Koutny, and Alex Yakovlev. Detecting state coding conflicts in stgs using integer programming. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 338–345, 2002.

[45] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design.* Series in Parallel Computing. John Wiley & Sons, 1994.

[46] A. Kondratyev, M. Kishinevsky, A. Taubin, J. Cortadella, and L. Lavagno. The use of Petri nets for the design and verification of asynchronous circuits and systems. *Journal of Circuits Systems and Computers*, 8(1):67–118, 1998.

[47] A. V. Kovalyov. On complete reducibility of some classes of Petri nets. In *Proceedings of the 11th International Conference on Applications and Theory of Petri Nets*, pages 352–366, Paris, June 1990.

[48] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[49] Luciano Lavagno and Alberto Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, 1993.

[50] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, Vancouver, British Columbia, Canada, August 1987.

[51] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. In *CWI-Quarterly*, volume 2, pages 219–246, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, September 1989.

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

[53] E. W. Mayr. An algorithm for the general petri net reachability problem. *SIAM Journal on Computing*, 13:441–460, 1984.

[54] Kenneth McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In G. v. Bochman and D. K. Probst, editors, *Proc. International Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer-Verlag, 1992.

[55] R. Milner. *A Calculus for Communicating Processes*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[56] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[57] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–574, April 1989.

[58] Jens Muttersbach, Thomas Villiger, and Wolfgang Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 52–59, April 2000.

[59] Chris J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Dept. of Elec. Eng., Stanford University, October 1995.

[60] Radu Negulescu. *Process Spaces and Formal Verification of Asynchronous Circuits.* PhD thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, August 1998.

[61] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, 1988.

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

[63] E. Pastor, J. Cortadella, A. Kondratyev, and O. Roig. Structural methods for the synthesis of speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, 17(11):1108–1129, November 1998.

[64] Enric Pastor and Jordi Cortadella. An efficient unique state coding algorithm for signal transition graphs. In *Proc. International Conf. Computer Design (ICCD)*, pages 174–177, October 1993.

[65] M. A. Peña and J. Cortadella. Combining process algebras and Petri nets for the specification and synthesis of asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, March 1996.

[66] James L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[67] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn, Institut für Instrumentelle Mathematik, 1962. (technical report Schriften des IIM Nr. 3).

[68] Oriol Roig. *Formal Verification and Testing of Asynchronous Circuits.* PhD thesis, Univsitat Politècnia de Catalunya, May 1997.

[69] L. Y. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets*, pages 199–207, Torino, Italy, July 1985. IEEE Computer Society Press.

[70] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1998.

[71] Alex Semenov, Alexandre Yakovlev, Enric Pastor, Marco Pe na, and Jordi Cortadella. Synthesis of speed-independent circuits from STG-unfolding segment. In *Proc. ACM/IEEE Design Automation Conference*, pages 16–21, 1997.

[72] Manuel Silva, Enrique Teruel, and José Manuel Colom. Linear algebraic and linear programming techniques for the analysis of place/transition net systems. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:309–373, 1998.

[73] Jens Sparsø and Steve Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.

[74] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, 1992.

[75] Peter Vanbekbergen. *Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications*. PhD thesis, Catholic University of Leuven, 1993.

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

[77] Tom Verhoeff. Analyzing specifications for delay-insensitive circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 172–183, 1998.

[78] A. V. Yakovlev and A. M. Koelmans. Petri nets and digital hardware design. In *Lectures on Petri Nets II: Applications. Advances in Petri Nets*, volume 1492 of *Lecture Notes in Computer Science*, pages 154–236, 1998.

[79] Alex Yakovlev. Is the die cast for the token game? In *Lecture Notes in Computer Science: 23rd International Conference on Applications and Theory of Petri Nets, Adelaide, Australia, June 24-30, 2002 / J. Esparza, C. Lakos (Eds.)*, volume 2360, pages 1–70pp. Springer Verlag, June 2002.

[80] Chantal Ykman-Couvreur, Bill Lin, and Hugo de Man. Assassin: A synthesis system for asynchronous control circuits. Technical report, IMEC, September 1994. User and Tutorial manual.

[81] Chantal Ykman-Couvreur, Bill Lin, Gert Goossens, and Hugo De Man. Synthesis and optimization of asynchronous controllers based on extended lock graph theory. In *Proc. European Conference on Design Automation (EDAC)*, pages 512–517. IEEE Computer Society Press, February 1993.

[82] M. Yoeli and A. Ginzburg. Lotos/cadp-based verification of asynchronous circuits. Report CS-2001-09-2001, Technion - Computer Science Department, September 2001.

[83] Kenneth Y. Yun and David L. Dill. Automatic synthesis of extended burst-mode circuits: Part ii (automatic synthesis). *IEEE Transactions on Computer-Aided Design*, 18(2):118–132, February 1999.

List of Figures

1.1 Comparison for area of the synthesized circuits. 15
1.2 Comparison for CPU time for synthesizing every benchmark. 16

2.1 (a) Transition System. (b) Petri net. 22
2.2 Petri net. 24
2.3 Non-free choice net. 27
2.4 An MG-cover (MG1,MG2) and an SM-cover (SM1,SM2) of PN from Figure 2.2. 27
2.5 (a) Petri net, (b) Spurious solution $m = (00020)^T$, (c) Potential reachability graph. 31
2.6 (a) Interface, (b) Timing Diagram. 34
2.7 Transition System specifying the bus controller. 35
2.8 (a) Partition induced by signal `lds`, (b) State graph of the read cycle. States are encoded with the vector (`dsr`, `dtack`, `ldtack`, `d`, `lds`). 35
2.9 Signal Transition Graph specifying the bus controller. 36
2.10 Unbounded and unconsistent STG. 37
2.11 Example *abcd*: (a) Signal Transition Graph, (b) State Graph 40
2.12 Complex gate implementation for the *abcd* example 41

3.1 Connection between different reactive systems (the suffixes ? and ! are used to denote input and output events, respectively). 44
3.2 Conditions 2(a) and 4(a) from the proof of Theorem 3.5.1. . . 51
3.3 Conditions 2(a) and 2(b) from the proof of Theorem 3.5.2. . . 52
3.4 Relation between observational equivalence, input-properness and I/O compatibility. 54
3.5 Two I/O compatible systems that are not input-proper. . . . 54

4.1 (a) Connection between system and environment, (b) mirrored implementation of a concurrent system, (c) valid implementation with concurrency reduction, (d) invalid implementation. 59
4.2 Distributor built from David cells [45]. 60
4.3 Encoding rule applied to the VME Bus Controller example. . 61

4.4 Different possibilities for reducing concurrency. 65
4.5 Conditions 2(a) from the proof of Theorem 4.2.1. 66
4.6 Transformation rule for each transition $t \in \mathcal{T}$. 69
4.7 Encoding for places q and q_i. 70
4.8 Place encoding to guarantee USC. 71
4.9 Initial situation for proof of Lemma 4.3.2. 73
4.10 In the center the initial STG fragment. *Enc*(S): technique of the previous section. *IO-Enc*(S): technique presented in this section to preserve I/O compatibility. 74
4.11 Transformation rule for non-input signals to preserve the I/O Compatibility. 75
4.12 simple join condition to ensure a correct encoding in the modified encoding technique. 76
4.13 (a) *Alloc-Outbound* example, (b) *IO-Enc(Alloc-Outbound)* has CSC. It has not USC due to the complementary sequence (in boldface) between two different markings. 77
4.14 Situation where two markings can have the same code. 78
4.15 (a) Modified *Alloc-Outbound* example, (b) Applying the encoding technique to preserve I/O compatibility does not solves the CSC conflicts 79

5.1 (a) STG, (b) STG with CSC, (c) Projection for signal d, (d) Circuit implementing d. 86
5.2 PpArbCsc(2,3) 90
5.3 Projection of example PpArbCsc(2,3) onto signal x_4. 93
5.4 Art(m, n). 94
5.5 Netlist of handshake components from a Tangram program. . 95
5.6 Tangram program from which the structure of Figure 5.5 can be obtained, as the parallel composition of P1 and P2. 95
5.7 STG for a sequencer in a Tangram program. 96
5.8 STG for a parallelizer in a Tangram program. 96
5.9 STG for a mixer in a Tangram program. 97

6.1 Synthesis of asynchronous circuits. 103
6.2 Greedy removal of signals $sp1$, $sp11$ and $sp2$. 105
6.3 Greedy removal of signals $sp3$, $sp4$ and $sp5$. 106
6.4 Greedy removal of signals $sp6$, $sp8$ and $sp7$. 107
6.5 CSC support computation and projection for the VME Bus Controller example. 108
6.6 Speed-independent synthesis of the VME Bus Controller. . . 109
6.7 CSC support computation and projection for PpArbCsc(2,3) example: signals $x1$, $x2$, $x3$. 111
6.8 CSC support computation and projection for PpArbCsc(2,3) example: signals $x4$, $x5$, z. 112

6.9 Speed-independent synthesis of the PpArbCsc(2,3) example: signals $x1$, $x2$, $x3$. 113
6.10 Speed-independent synthesis of the PpArbCsc(2,3) example: signals $x4$, $x5$, z. 114

List of Tables

5.1 CSC detection for well-structured STGs. 98
5.2 USC detection for well-structured STGs. 99

6.1 Support computation, projection and synthesis compared to state-based approach. 115

List of symbols

General symbols		Page
$\mathbb{Q}$	rational numbers	20
$\mathbb{Z}$	integer numbers	20
$\mathbb{N}$	natural numbers	20
$\mathbb{B}$	binary numbers	20
$x \cdot y$	product of vectors x and y	20
$v\|_P$	projection of the vector v into the index set P	20
$\mathbf{0}$	null vector	20

Sequences		
ϵ	empty sequence	21
$\sigma\,\gamma$	concatenation of sequences σ and γ	21
$\#(\sigma, x)$	number of occurrences of symbol x in the sequence σ	21
$\sigma\|_X$	projection of sequence σ into the set X	21

Transition systems		
$s \xrightarrow{\sigma} s'$	a *sequence* σ that leads from s to s'	22
$\mathsf{En}(s, e)$	event e enabled at state s	22
$L(A)$	language of the TS A	22
$A \times B$	synchronous product of TSs A and B	23

Petri nets		Page
$^\bullet x$	pre-set of the node x	25
$x^\bullet$	post-set of the node x	25
m_0	initial marking	24
$m[\sigma\rangle m'$	marking m' is reachable from marking m	25
$m_0 \stackrel{\sigma}{\to} m$	firing sequence	25
$[m\rangle$	set of reachable markings from marking m	25
$RG(N)$	reachability graph of Petri net N	25
FCLSPN	Free choice, live and safe Petri net	27

Linear algebra		
$\mathbf{N}$	incidence matrix of the net N	30
$\vec{\sigma}$	Parikh vector of sequence σ	31

Asynchronous circuits		
$\lambda(s)$	state encoding function at state s	35
ε	silent event	35
$\mathsf{ER}(x+)$	excitation region of transition signal $x+$	39
$\mathsf{QR}(x+)$	quiescent region of transition signal $x+$	39
$\mathsf{ON}(x)$	on-set of signal x	40
$\mathsf{OFF}(x)$	off-set of signal x	40

Reactive systems		
$A \approx B$	RTS A is observational equivalent to RTS B	50
$A \rightleftharpoons B$	RTS A is I/O compatible to RTS B	47
ϕ_r	transformation for concurrency reduction	62
ϕ_i	transformation for increase of concurrency	63
ϕ_e	transformation for transition elimination	64

Index

asynchronous circuits
- bounded-delay, 33
- control circuits, 33
- delay-insensitive, 33
- speed-independent, 33

complete state coding, 38
conformation, 45
consistency, 38
cycle, 25

David cells, 60
design flow, 103
disabling, 39

encoding technique
- Enc, 69

I/O automata, 21
I/O compatibility
- checking, 49
- definition, 47
- structural, 47

ILP
- model CSC, 88
- model USC, 87
- model support, 89

input-properness, 46

linear programming problem
- complexity, 30
- definition, 30
- integer, 30

livelock, 46

observational equivalence, 50
output persistency, 39

path, 25
Petri net
- boundedness, 26
- definition, 24
- disabling, 26
- firing rule, 25
- free-choice, 28
- home marking, 26
- incidence matrix, 31
- liveness, 26
- marked graph, 28
- marking equation, 32
- Parikh vector, 31
- post-set, 25
- pre-set, 25
- reachability, 25
- reactive, 29
- redundant place, 29
- reversibility, 26
- simple join net, 75
- state machine, 28
- triggering, 26

product
- cartesian, 20
- vectors, 20

reactive
- mirror, 46
- system, 21
- transition system, 23

region
- excitacion, 39
- quiescent, 39

rule
- ϕ_e, 64
- ϕ_i, 63

ϕ_r, 62

sequence
concatenation, 21
definition, 21
empty, 21
signal transition graph, 36
state graph, 35
support, 89
synthesis problem, 21

tangram
handshake components, 95
mixer, 97
parallelizer, 96
program, 95
sequencer, 96
Team automata, 118
trace structure, 45
transition system
definition, 22
deterministic, 23
language, 22
reachability relation, 22
synchronous product, 23

unique state coding, 38

VME Bus example, 34