Technische Universität Kaiserslautern

Doctoral Thesis

Model-based Design of Embedded Systems by Desynchronization

Yu Bai

vom Fachbereich Informatik der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

Dean:
Prof. Dr. Klaus Schneider

Committee Chair of Dissertation:
Prof. Dr. Reinhard Gotzhein

Supervisor:
Prof. Dr. Klaus Schneider

Reviewer:
Prof. Dr. Klaus Schneider,
Prof. Dr. Jean-Pierre Talpin

Date of Defence:
December 17, 2015

D386

UNIVERSITY OF KAISERSLAUTERN

Abstract

Embedded Systems Group
Department of Computer Science

Doktor der Ingenieurwissenschaften

Model-based Design of Embedded Systems by Desynchronization

by Yu Bai

In this thesis we developed a desynchronization design flow in the goal of easing the development effort of distributed embedded systems. The starting point of this design flow is a network of synchronous components. By transforming this synchronous network into a dataflow process network (DPN), we ensures important properties that are difficult or theoretically impossible to analyze directly on DPNs are preserved by construction. In particular, both deadlock-freeness and buffer boundedness can be preserved after desynchronization. For the correctness of desynchronization, we developed a criteria consisting of two properties: a global property that demands the correctness of the synchronous network, as well as a local property that requires the latency-insensitivity of each local synchronous component. As the global property is also a correctness requirement of synchronous systems in general, we take this property as an assumption of our desynchronization. However, the local property is in general not satisfied by all synchronous components, and therefore needs to be verified before desynchronization. In this thesis we developed a novel technique for the verification of the local property that can be carried out very efficiently. Finally we developed a model transformation method that translates a set of synchronous guarded actions – an intermediate format for synchronous systems – to an asynchronous actor description language (CAL). Our theorem ensures that one passed the correctness verification, the generated DPN of asynchronous processes (or actors) preserves the functional behavior of the original synchronous network. Moreover, by the correctness of the synchronous network, our theorem guarantees that the derived DPN is deadlock-free and can be implemented with only finitely bounded buffers.

Acknowledgements

First I would like to thank my professor Klaus Schneider for giving me the opportunity to work on a fascinating research topic, more over for all the supports along my bumpy trip of this thesis. Since when I started this trip, I had a relatively smooth beginning as things went well along the end of my master studies. However, soon similar to many young researchers' experience, I hit the great barrier – being lost in the huge amount of literature and having no clue where to go. It is professor Schneider's guidance from time to time that keeps a light in front of me, and helped me finally find my way out. Besides the scientific results, it is the systematic thinking and discipline I learned through this process from him that will carry me on through all my later career. Second I would like to thank for the useful advice and comments from the professionals that I met during my ph.D studies. In particular, professor Susanne Graf from Verimag gave me lots of useful and insightful suggestions in my works. Professor Gerald Luettgen's warm invitation lead to a set of inspirational discussions that I had and helped me clarified many doubts as well. Of course, I would thank my second supervisor professor Jean-Pierre Talpin from INRIA. It is his group's works that I followed the most and learned the most from. It is an honor to be able to have him as my supervisor. I would also like to thank my dear colleagues in my research group. Dr. Jens Brandt practically tutored me all the way through my early years. I can not adapted myself to the researching area faster without his support and his rich professional experience. Yet his support is beyond professional advice, and I'm truly grateful for that. Daniel, Mike, Manuel, Andreas, thank you for all the support. You are always there whenever I needed you guys. Now all of you have found your positions in your new careers, and I wish you all the best. I would also like to thank the students I have tutored. Working with you is a lot of fun, although teaching you the disciplines is tough. Nevertheless learning true values is never easy, and I hope that you found some fun along the way as well. Finally but most importantly, I would like to thank my parents. Your unconditional love for me is always the most strongest motivation for me to keep moving forward. This thesis is dedicated to you.

Contents

List of Figures

List of Tables

Abbreviations

MoC	**M**odels **o**f **C**omputation
DPN	**D**ataflow **P**rocess **N**etwork
LTS	**L**abeled **T**ransition **S**ystem
FIFO	**F**irst **I**n **F**irst **O**ut
SPN	**S**ynchronous **P**rocess **N**etwork
BDF	**B**oolean **D**ata-**F**low
CPO	**C**omplete **P**artial **O**rder
LID	**L**atency-**I**ntensive **D**esign
ASAP	**A**s **S**oon **A**s **P**ossible
NRSA	**N**o **R**eaction **T**o **A**bsence
BDD	**B**inary **D**ecision **T**ree
SAT	**S**atisfiability
SMT	**S**atisfiability **M**odulo **T**heories
SMV	**S**ymbolic **M**odel **V**erification
SGA	**S**ynchronous **G**uarded **A**ctions
CSGA	**C**ontrol flow-**S**ynchronous **G**uarded **A**ctions
DSGA	**D**ata-flow-**S**ynchronous **G**uarded **A**ctions
FSM	**F**inite **S**tate **M**achine
EFSM	**E**xtended **F**inite **S**tate **M**achine
CAL	**C**al **A**ctor **L**anguage

Symbols

p_i	a process
P	a dataflow process network
ζ	channel state of a DPN
δ	process state of a DPN
Σ	alphabet of a DPN
$\hat{\Sigma}$	alphabet of a single-rated SPN
$\check{\Sigma}$	alphabet of a multi-rated SPN
π	execution of a labeled transition system
$\|\pi\|$	the length of π
γ	action of a labeled transition system, or guard of an SGA
$\Gamma(p)$	the set of executions of p
$\hat{\rho}$	synchronous run of a labeled transition system
$\mathcal{L}(p)$	process p's language
$\Phi(p)$	effective input/output relation of process p
$\Psi(P)$	general input/output relation of DPN P
l	label of a labeled transition system
$\hat{l}$	label of a labeled transition system with scheduling
$\|\|$	synchronous composition
$\|\|\|$	asynchronous composition
$\perp$	unknown symbol in constructive logic
$\rightarrow$	causal preorder
$\rightarrow_p$	set of labeled transition relations of the LTS of p
$\boxdot$	absent signal
ϵ	empty character
$\delta(p)$	desynchronization of process p

$\mathcal{A}$	action of an SGA

Dedicated to my parents.

Chapter 1

Introduction

1.1 Motivation

Embedded systems used to be considered as easier to study because they used to be small, limited and simple. However this picture has faded as we witnessed the rise of advanced hardware platforms such as multi- and many-core processors, heterogeneous distributed application environments as Internet of things, complex and divergent software system requirements from industrial control systems to smart phone applications – all lead to a dramatically increasing effort of development. Nowadays the development of embedded systems is more challenging than usual software development since application-specific hardware and software must be implemented that satisfy functional and additional non-functional specifications.

Model-based design methods have proven beneficial, as code is automatically generated from abstract models that allow the engineers to first concentrate on the functional behavior of their systems. For small embedded systems, model-based design methods based on synchronous models of computation have been established where either a sequential program or an application-specific hardware can be generated from one and the same synchronous model. Synchronous models have many advantages like the ability to model concurrent behaviors, deterministic simulation, applicability to formal verification, and static analysis methods that simplify the prediction of worst case execution/reaction times. However such simplified model is not well suited for the development of distributed applicaitons. As an example, the trend towards using multi/manycore platforms in embedded systems demands the generation of multi-threaded software which poses new challenges to the use of synchronous models: synchronous models have the disadvantage that the single threads would have to synchronize after each reaction step which often reduces the performance of the system to an unacceptable level. The usual convention for developing such distributed applications is to use asynchronous models for modeling the applications directly. For example, dataflow process networks has been successfully applied in signal processing applications. However despite the natural depiction of the target application and the ease of synthesis for target code, a big drawback to use asynchronous models is that major properties of interests like buffer boundedness

as well as deadlock-freeness are in general undecidable. Yet for safety-critical embedded applications, these properties are of crucial importance.

Since synchronous models and asynchronous models along all can not fulfill the task, researchers then tried to bring the two models together to exploit the advantages of each model while trying to avoid the their drawbacks respectively. In particular, techniques for desynchronization have been introduced by Benveniste and others [42]. Desynchronization is developed as a design flow that starts from the synchronous model of a system, and then transforms the synchronous model into an asynchronous model that can be directly used for the synthesis of the target distributed application. This thesis follows this approach. In particular, we developed a theory that provides desynchronization criteria which ensures the correctness of the desynchronization, and implemented the corresponding verification methods. As a validation of our theory, we also implemented a practical model transformation method translating synchronous intermediate codes into actor based coeds modeling DPNs.

1.2 Desynchronization

The results of this thesis is a desynchronization design flow that transforms synchronous networks into dataflow process networks (DPNs) and related techniques that supports this design flow. The input of the design flow is therefore a synchronous network where all components of the network execute their reaction steps at the same time, and during each reaction step, one single value is read from each input port and one single value is written to each output port. In contrast, the resulting DPN of the design flow is *asynchronous*, where components are no longer forced to perform their execution steps at the same time, hence the values communicated between the components have to be stored in FIFO buffers.

A synchronous system can be either *single-rated* or *multi-rated*. We developed corresponding theories covering both cases. As the name indicates, single-rated synchronous systems consist of synchronous components working at the same pace, therefore maintains a universal view in the communication across the components. In particular, during each computation step (or clock cycle), each component reads one signal from each of its input ports and writes to each of its output ports. Desynchronization of a single-rated synchronous network is relatively easy. Yet, there are subtle issues we need to be careful of. In particular, order to make sure that the desynchronized DPN is free from deadlocks, we need to insist on the causal correctness of the original synchronous system as a global criteria. Also, each synchronous component is transformed to a process that runs asynchronously. In order to preserve the functional behavior of the synchronous component, the component needs to be deterministic as a local criteria.

A multi-rated synchronous network consists of components that work in different rates. There is still a global clock of the network, which is the finest clock of all clocks of the components, and each signal connection now posses its own clock period which is a multiple of the clock period of the global clock. Therefore during one global clock cycle, now

a component may not read anything from some of its input ports, and reads something from some other input ports. To desynchronize a multi-rated synchronous network, the global clock that synchronizes all component is removed. Synchronous signal connections between components are still replaced by asynchronous FIFO buffers. But now because of the removal of the global clock, what is transmitted between components is only data tokens. Therefore, the information of "a particular signal is absent during a particular cycle" is now lost. We can equally imagine that previously a special value indicating signal absence was transmitted, but now it is removed from communication.

Since absent values are no longer communicated, each component must be able to decide which values of the input buffers have to be consumed at the beginning of each reaction or have to be retained there for later reactions. Unfortunately this cannot be performed on general synchronous DPNs, but only for DPNs fulfilling certain *desynchronization criteria*. Similar to the desynchronization of single-rated synchronous networks, we need a global criteria as well as a local criteria. The global criteria is still the correctness of the synchronous network , but the local criteria is a property that is stronger than determinisity. In this thesis, we studied the theoretical boundary of the local criteria and try to define the largest set of component that fulfills this criteria. We then developed sufficient conditions of this criteria that can be verified efficiently in practice.

Finally, we practiced our desynchronization theory by developing a compiler that translates synchronous intermediate codes to actor oriented codes that model a DPN. In particular, the compiler takes clocked synchronous guarded actions (SGAs) [68–73] as inputs, and translates the clocked SGAs into CAL actors [101]. SGAs is a general intermediate format of synchronous programs, and in this thesis they are derived from Quartz programs [20]. CAL is an actor-oriented language that is dedicated for modeling dataflow process networks [103].

1.3 Contribution

There are already plenty of work on desynchronizationg of synchronous models. The novelty of this thesis lies in the following points:

- We start from a synchronous model that captures the micro-step semantics, therefore takes care of the causality issues and the preservation of functional behaviors in the level of operational semantics.
- We settled the theoretical boundary for desynchronization and clarified the relationships of related definitions between different levels of abstractions. We proposed a theory that is practical enough for the implementation.
- We implemented a novel method for the verification of the desynchronization criteria, and validated the efficiency of the method by experiments.
- We implemented a model transformation from synchronous intermediate code (clocked synchronous guarded actions) to DPN modeling language (CAL actors)

following the desynchronization theory we developed. The transformation is platform independent and is suited for both software and hardware synthesis.

1.4 Structure of the Thesis

The structure of the thesis goes as follows:

- Chapter 2 introduces the foundations of the thesis. Synchronous process networks and dataflow process networks are presented, and their semantic models are introduced in detail. We further presented some basic results in the area of dataflow process networks, showing the difficulty in their analysis thus motivating the desynchronization design flow.
- Chapter 3 presents the theoretical results of this thesis. It discusses criteria for correct desynchronization of both single-rated and multi-rated synchronous process networks, and presents the corresponding theorems and proofs in detail. It also discussed the theoretical boundary of correctly desynchronizing a single synchronous component.
- Chapter 4 presents the sufficient condition for the local criteria of correct desynchronization as well as the techniques we developed for the verification of this condition. Experimental results followed shows the effectiveness and the efficiency of our method.
- Chapter 5 shows a practical transformation method from clocked synchronous guarded actions to CAL actor actions. This transformation utilizes the theoretical results we developed before, and discussed several major technical difficulties and their solutions in detail. This chapter ends with a section of experimental results that validates our transformation methodology.
- Chapter 6 summarizes the thesis, and suggests some directions for future research.

Chapter 2

Models Of Computations

There has been an abundant set of mathematical tools for the modeling and analysis of concurrent and distributed systems. Some resounding names include *Communicating Sequential Processes*(CSPs)[1], *Calculus of Communicating Systems*(CCS)[2] and Petri Nets[3] for concurrent systems, *I/O Automaton*[4] for distributed algorithms and *Parallel Random Access Machines*(PRAM)[5] for parallel algorithms. This chapter first introduces **Dataflow Process Networks** (DPNs) [6] as a general framework for the modeling of both synchronous and asynchronous systems in this thesis. Then **Synchronous Process Networks**(SPNs) are introduced as a restricted form of DPNs we use to model *synchronous systems*. We chose DPNs not because of a matter of taste, but based on the following reasons:

- **Determinicity**. Embedded systems usually need to work with high predictability, therefore it is crucial for models of embedded systems remain deterministic.
- **Distributivity**. To capture the essence of distributed systems, it is important to provide the concept of processes as well as how they are composed together.
- **Formality**. One major advance in model-based design is the capability for the analysis and reasoning of the systems provided by the mathematical soundness of the system model.
- **Flexibility**. The model should be general enough to capture both synchronous and asynchronous systems.
- **Implementability**. The model should provide a gentle means towards the final system implementation.

In this chapter we study the properties of DPNs from two perspectives: the operational semantics and denotational semantics of DPNs. DPNs are generally asynchronous. In order to model synchronous systems, we introduce a restricted version of DPN– *synchronous process networks*. Based on the foundations introduced in this chapter, the next chapter further studies the problem of desynchronization – how we derive a DPN from an SPN that preserves its functional behavior.

2.1 Dataflow Process Networks

Dataflow Process Networks, like CSP and CCS, describes a set of *processes* that are nodes of the network running concurrently with each other and communicate via their *directed arcs*. Similar as other models, a process of a DPN represents a single processing unit in the network. What's special about DPNs is that the communication represented by the arcs works in a much more constrained way. Each arc connects at most two nodes (processes) and models a *FIFO buffer* between the two nodes, i.e. one *producer* that pushes computed data values into the end of the buffer and one *consumer* that pulls out data values from the head of the buffer. A process executes as long as it has enough values arrived at its input buffers. This requires the FIFO buffers have unbounded size, otherwise a node have to be suspended from production when its output buffer is full even if its input channels have enough values. This also indicates that processes can run in different speeds, and the only factor that limits a process's execution is the data dependency with its producers. For this reason we also call a DPN an *asynchronous DPN*.

2.1.1 Syntax of DPNs

DPNs have naturally a graphical representation, as a DPN can be pictured as a network of nodes where each node is a process and the arc that connects two nodes represents the FIFO buffers between the processes as communication channels. Producers and consumers are identified by the directions of arcs, where the origin of the arc is connected to the producer and the destination connected to the consumer. Let C_P denote the set of channels of a DPN P, and $c \in C_P$ a channel of P. An *alphabet* Σ_c is the set of *values* that can be transmitted over channel c. Σ then denotes the union of all alphabets of channels in C_P, and E^* (E^ω) is the set of all finite (infinite) strings over alphabet E. We denote $(E^* \cup E^\omega)$ by E^∞. We use ϵ to denote the empty character, as well as the empty string or tuples of empty strings. The concatenation of two strings α, β is denoted by $\alpha \cdot \beta$ or simply $\alpha\beta$. $\alpha \sqsubseteq \beta$ denotes that α is a prefix of β. If $\alpha \sqsubseteq \beta$ and $\alpha\gamma = \beta$, then $\gamma = \beta \setminus \alpha$. $\cdot, \sqsubseteq$ and $\setminus$ all can be extended component-wisely to tuples of strings. With a little abuse of notation, we also use the same symbols for function operations. In particular, for $f, g : C_P \to \Sigma^*$, $\forall i \in C_P, (f \cdot g)(i) = f(i) \cdot g(i)$; $f \sqsubseteq g$ iff $\forall i \in C_P, f(i) \sqsubseteq g(i)$ and $\forall i \in C_P, (g \setminus f)(i) = g(i) \setminus f(i)$. $f|_{C'}$ denotes the *project* $C' \to \Sigma^*$ of f over the set of channels C', where $\forall c \in C_P \cap C', f|_{C'}(c) = f(c)$. An arc represents an *input channel* (*output channel*) if it only has a consumer (producer) node. An arc with both a consumer and a producer is an *internal* channel of P. The set of input channels, output channels and internal channels of P are denoted by I_P, O_P and U_P respectively. For a DPN P composed of processes $p_1, \ldots, p_n$, we say P is *syntactically specified* by $\langle \{p_1, \ldots, p_n\}, C_P \rangle$. For example, Figure 2.1 depicts a DPN P with three processes f_1, f_2, f_3

where f_1 and f_2 are producers for f_3. P is syntactically specified by $\langle \{f_1, f_2, f_3\}, \{x_1, x_2, l_1, l_2, y\} \rangle$. The behavior of P is defined by the behavior of the three processes f_1, f_2 and f_3. When the context is clear, we simply denote the DPN as $f_1 ||| f_2 ||| f_3$. Note that a single process

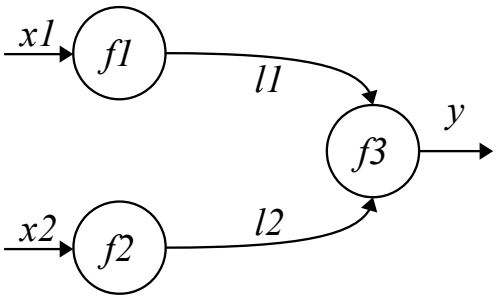


FIGURE 2.1: A DPN with three nodes.

is different from a DPN consisting of a single process, since in the DPN buffers are added. To distinguish the two cases, we occasionally use $\{p\}$ to denote the DPN consisting of the single process p as well. The symbol ||| denotes the asynchronous composition of the processes, i.e. they are connected by FIFO buffers. Each process again has its own input channels and output channels. In the following sections we examine behaviors of a DPN in detail. A DPN can be a closed loop of processes in the sense that every channel has a producer and a consumer. It doesn't make sense to discuss the input and output of such closed DPNs. Therefore in the rest of the thesis we will assume that *all our DPNs contain proper input channels and output channels.*

2.1.2 Operational Semantics of DPNs

When people talk about formal semantics, they usually mean a semantics with regard to some programming language, like the one of PCF [8]. They are developed to describe the mathematical meanings of the language as well as the programs written in the language so that rigorous reasoning and analysis can be done to verify properties of interest of the program, such as termination, correctness, safety and so on. In our context of models of computations, we do not intend to use any particular programming language. However the same idea applies well for models as well. Therefore we use mathematical tools to describe DPNs–our system model of interest. And we do this at different abstraction levels, so that different purposes can be respected. In this section we first introduce the operational semantics we used for DPNs. In order to be independent from any particular programming language, we use *labeled transition systems* (LTSs) to describe each process as well as their composition–which is a DPN. An LTS specifies how computations of a process (or a DPN) are performed step wise, i.e. how the system evolves from one state to the next. Since it captures details of the system's operations, it is straightforward to realize it into a real implementation. However, as we will see later, in general LTS does not serve well as a powerful vehicle for formal analysis of DPNs, as most properties of interest are undecidable for general DPNs. This is mainly because that the model is Turing complete. Therefore, in order to improve analyzability, necessary sacrifice in expresiveness are needed.

Labeled Transition Systems of Processes

The behavior of a process is specified by a *labeled transition system* that describes possible operations that can be performed by the process at each of its local state. Similar to previous literature [7, 9–11], we define the LTS of a process as follows.

Definition 2.1. A labeled transition system $LTS_p : \langle I, O, S, s_0, L, \rightarrow_p \rangle$ of process p is defined by:

- I is the set of input channels of p, O is the set of output channels of p and $I \cap O = \emptyset$;
- S is the set of local states of p, with $s_0 \in S$ the initial state;
- L is the set of *labels*, where each label $l \in L$ is an assignment: $(I \cup O) \rightarrow \Sigma$;
- $\rightarrow_p \subseteq S \times R \times S$ is the set of labeled transition relations.

A transition relation (s, l, s') is also denoted by $s \xrightarrow[p]{l} s'$. Equivalently LTS_p can be simply specified by a set of *firing rules* R_p where each rule $r \in R_p$ has the format:

$$\langle \vec{I} : (s \rightarrow s') : \vec{O} \rangle, \text{ where } \vec{A} \text{ is an assignment } A \rightarrow \Sigma_A \text{ with } A(c) \in \Sigma_c.$$

Intuitively, each firing rule specifies the condition $(\vec{I} : s)$ under which a computation $(s \rightarrow s' : \vec{O})$ to be executed. In particular, $\vec{I}$ of $(\vec{I} : s)$ specifies the required input values over input channels of the process, and s specifies the condition over the current local state. Once fulfilled, $s \rightarrow s'$ indicates that the local state will be updated to s', and $\vec{O}$ specifies the output values for each output channel respectively. We denote the required input values $\vec{I}$ by $\mathsf{r}(r)$, the local state condition s by $\mathsf{s}(r)$, the updated state by $\mathsf{s}'(r)$ and the output values $\vec{O}$ by $\mathsf{w}(r)$ respectively. Then $\exists r \in R_p$ if and only if $\exists (s, l, s') \in \rightarrow_p$ such that $l = \mathsf{r}(r) \cdot \mathsf{w}(r)$ and $\mathsf{s}(r) = s, \mathsf{s}'(r) = s'$. A firing rule r is a *stuttering* rule if its inputs and outputs $\mathsf{r}(r) = \mathsf{w}(r) = \epsilon$ and state transition satisfies $\mathsf{s}(r) = \mathsf{s}'(r)$. r is a *reading rule* if $\mathsf{r}(r) \neq \epsilon$; r is a *writing rule* if $\mathsf{w}(r) \neq \epsilon$. A firing rule can be both reading and writing. If the process only contains one single state, we omit the state update in the rules. A process with finitely many local states and firing rules is called a *finite process*.

Parameterized Firing Rules

For convenience we introduce firing rules with parameters. Unlike an ordinary firing rule r where both $\mathsf{r}(r)$ and $\mathsf{w}(r)$ are assignments from channels to channel alphabets, we allow variables and expressions to be used in $\mathsf{r}(r)$ and $\mathsf{w}(r)$. Variable are named using a, b, c etc., denoting non-empty values. For example: $\langle a : a \rangle$ means to read an input value a and copy this value to the output; $\langle a : g(a) \rangle$ means to read an input value a and produce the output value $g(a)$ to its output, where g is some function; $\langle (a, b) : (a > 0, b < 0) : s \rightarrow s' : b \rangle$ means to read an input value a and b and when $a > 0$

and $b < 0$ and the current state is s, do the state transition and output b. Therefore this rule is enabled at all situations where there are two inputs at the head of the two input channels, the first greater than 0 and the second smaller than 0 and current state is s. We use parameterized firing rules only as a syntax sugar for representing a set of ordinary firing rules in common. Therefore we would like to save the tedious formal specifications, since the examples are already self-explanatory.

Definition 2.2. An execution π of a process p is a sequence of transition relations $s_0 \xrightarrow[p]{l_0} s_1 \xrightarrow[p]{l_1} s_2 \ldots$ where each $s_i \xrightarrow[p]{l_i} s_{i+1} \in \rightarrow_p$. $|\pi|$ denotes the length of π. π is finite if $|\pi| \in \mathbb{N}$, else π is infinite and we denote $|\pi| = \infty$. We use $\Gamma(p)$ to denote the set of executions of p.

From an execution of a process's LTS we can extract its corresponding input-output flows, which is the concatenation of the labels of the transition relations. Similar to a finite state machine, we can define the *language* of a process as follows:

Definition 2.3. A process p's *language* $\mathcal{L}(p)$ is

$$\{w \mid w = l_0 \cdot l_1 \cdot \cdots \cdot l_n, \text{ where } s_0 \xrightarrow[p]{l_0} \ldots \text{ is an execution of } p.\}$$

For execution $\pi = s_0 \xrightarrow[p]{l_0} s_1 \xrightarrow[p]{l_n} \ldots$, we define $\rho(\pi) := l_0 \cdot l_1 \cdot \ldots$ to be the *run* of π.
We also denote the input sequence $\rho(\pi)|_{I_p}$ of π by $\mathsf{r}(\pi)$ and the output sequence by $\mathsf{w}(\pi)$.

Labeled Transition Systems of DPNs

Given the operational semantics of a single process, we can derive the operational semantics of a DPN by composing its processes together. Let a DPN $P = p_1 ||| \ldots ||| p_n$ where each process p_i is specified as $\langle I_i, O_i, S_i, s_0^i, R_i \rangle$. The set of input channels of P is $I_P := \bigcup_i I_i \setminus \bigcup_i O_i$ and set of output channels is $\bigcup_i O_i \setminus \bigcup_i I_i$. The set of internal channels of P is $U_P := \bigcup_i I_i \cap \bigcup_i O_i$. Therefore $C_P := I_P \cup U_P \cup O_P$. For convenience, we also use $P = \{p_1, \ldots, p_n\}$ to represent the DPN consisting of the n processes. A state of a DPN is specified by the process's input channels and its local state together, which we call a *configuration*.

Definition 2.4. A *configuration* of a DPN P is a tuple (ζ, δ) where $\zeta : (I_P \cup U_p) \rightarrow \Sigma^*$ is a *channel state* mapping each input and local channel to a finite string over the channel alphabets, and $\delta = (s_1, \ldots, s_n)$ is a *process state* recording each process's local state of its LTS. We denote s_i by $\delta[i]$.

Definition 2.5. Let $P = p_1 ||| \ldots ||| p_n$ be a dataflow process network. Its labeled transition system LTS_P is defined by the tuple $(P, M_P, m_0^P, Act, \rightarrow_P)$ where M_P is the set of configurations of P, $m_0^P = (l_0, \delta_0)$ the initial configuration where $\forall c \in I_P \cup U_P, l_0(c) = \epsilon$ and $\forall p_i, \delta_0[i] = s_0^i$. $\rightarrow_P \subseteq M \times Act \times M$ is the smallest set of labeled transition relations satisfying the following induction rules with some process p_i:

$$\frac{\forall c \notin I_p, b(c) = \epsilon}{(\zeta, \delta) \xrightarrow{?b} (\zeta \cdot b, \delta)} \quad \text{where } \delta[i] = s$$

where the transition is called an *input transition* and $?b \in Act$ an *input action*, and

$$\frac{s \xrightarrow[p_i]{l} s', l|_{I_{p_i}} = i, l|_{O_{p_i}} = o}{(i \cdot \zeta, \delta) \xrightarrow{!l} (\zeta \cdot (o|_{U_P}), \delta\{p_i/s'\})} \quad \text{where } \delta[i] = s$$

where the transition is called a *fire action* where $\delta\{p_i/s'\}[j] := \delta[j]$ when $j \neq i$ and $\delta\{p_i/s'\}[i] := s'$, and $!l \in Act$ a *fire action*. Its corresponding firing rule is $\langle i : s \to s' : o \rangle$. A fire action with a reading rule (writing rule) is called a *read action* (*write action*).

Given a configuration (ζ, δ), a firing rule r is *enabled* if $\mathsf{r}(r) \sqsubseteq a$ and $\mathsf{s}(r) = s$. When the context is clear, we often denote the transition by $m \xrightarrow{!r} m'$ rather than $m \xrightarrow{!l} m$. Intuitively, an input action simply append more input values to the input channels, while a fire action corresponds to a process firing a rule r by consuming some input values, updating its local state and producing values to output channels according to r. Besides the enabled conditions of firing rules, we have no further constraints on read and write actions. This means a DPN may read any input streams possible in Σ_c^* for each input channel c and fire rules at will. In particular, different firing rules of a process may compete for the same input values, and a nondeterministic choice would be made for the LTS.

Definition 2.6. An execution of a DPN P with $LTS_P : \langle P, M, m_0, Act, \to \rangle$ is a sequence of transition relations $\pi = m_0 \xrightarrow{\gamma_1} m_1 \xrightarrow{\gamma_2} \ldots$, where each $m_i \xrightarrow{\gamma_{i+1}} m_{i+1} \in \to$. We use $\Gamma(P)$ to denote the set of executions of P.

Given an execution π, we can extract the subsequence of π that contains only the input actions, or only the fire actions. In particular let $\pi? = m_{a_0} \xrightarrow{?b_0} m_{a'_0}, m_{a_1} \xrightarrow{?b_1} m_{a'_1}, \ldots$ denote the subsequence of π that contains exactly those input actions of π. Let $\mathsf{i}(\pi) = b_0 \cdot b_1 \cdot \ldots$ denote the *input flow* of π. Similarly, let $\pi! = m_{a_0} \xrightarrow{!r_0} m_{a'_0}, m_{a_1} \xrightarrow{!r_1} m_{a'_1}, \ldots$ be the fire action subsequence of π, then $\mathsf{w}(\pi) = \mathsf{w}(r_0)|_{O_P} \cdot \mathsf{w}(r_1)|_{O_P} \cdot \ldots$ denotes the *output flow* of π, and $\mathsf{r}(\pi) = \mathsf{r}(r_0)|_{I_P} \cdot \mathsf{r}(r_1)|_{I_P} \cdot \ldots$ denotes the *consumed flow*. Note that for an execution π, its input flow might not always equal to its consumed flow. This may happen when at a particular configuration, no firing rule is enabled. By the second induction rule, we can also project the execution over one process p and derive an execution of p. We denote this projected execution by $\pi|_p$. In order to capture different types of executions, we further introduce the following properties:

- (Fairness) π is a *fair execution* iff either it is finite, or it is infinite and an enabled firing rules is either fired or disabled eventually, i.e., $\forall m_i$ such that $\exists r \in R$ is enabled at m_i, either $\exists j \geq i, m_j \xrightarrow{r} m_{j+1}$ in π or r is disabled in m_j.
- (Liveness) π is a *live execution* (*R-live execution*) iff either it is finite, or it is infinite and there is eventually a firing rule (reading rule) enabled along π, i.e. $\forall i \in \mathbb{N}, \exists j \geq i$ and $\exists r \in R$ such that r is enabled at m_j.
- (Maximality) π is a *maximal execution* iff either it is infinite, or it is finite and in the end of the configuration m_n of π, there is no firing rule enabled at m_n.

- (Effectiveness) π is an *effective execution* iff for all input action $m_i \xrightarrow{?b} m_{i+1}$ and all input values of b, it is finally consumed by some fire action, i.e. $\mathsf{i}(\pi) = \mathsf{r}(\pi)$. We say DPN P is *effective* to input assignment i iff there exists an effective execution $\pi \in \Gamma(P)$ and $\mathsf{i}(\pi) = i$. P is effective iff it is effective for all input assignment.
- (Boundedness) π is a bounded execution iff for each internal channel there is an upper bound to the number of values residing in it for all its configurations along π, i.e. $\forall c \in U_P, \exists n \in \mathbb{N}$ such that $\forall m_i = (\zeta, \delta)$, $|\zeta(c)| \leq n$. A DPN P is bounded to input assignment i iff $\exists \pi \in \Gamma(P)$ with $\mathsf{i}(\pi) = i$ and π is bounded. P is bounded if it is bounded for all input assignment.

It is not difficult to see that liveness and fairness are not equivalent. Since one infinite execution might have finitely many enabled firing rules (fair) with infinitely many input actions followed (not live). Or an infinite execution might have one enabled firing rule without firing it (not fair), followed with infinitely many input actions (live).

A fair execution might not be effective, since given an infinite sequence of inputs, a reading rule might be enabled and disabled by some other non-reading action alternatively, without reading any input. A live execution might not be effective because of the same reason. Effective executions are live, since for an infinite execution either there are finitely many input actions and infinitely fire actions are executed; or if there are infinitely many input actions then infinitely many fire actions are needed to consume them. Effective executions may not be fair, since one process may consume all its (finite) inputs while having an enabled firing rule (that does not require any input anymore) that is never fired, while other processes keep on executing and consuming infinitely many values from their input channels.

A maximal finite execution might not be effective, since there might be inputs that no fire action can consume. An effective finite execution might not be maximal, since there might be enabled fire actions that are not fired. An effective execution might not be bounded. These characteristics will be used later for analysis as well as capturing denotational semantics of a DPN later.

If r is enabled at (ζ, δ), $\pi = (\zeta, \delta) \xrightarrow{?b} \ldots \xrightarrow{r'} (\zeta', \delta')$ with r' the first fire action along π and r is not enabled at (ζ', δ'), then we say r is *disabled* by r'.

Proposition 2.7. *If firing rule $r \in R_p$ of process p is enabled at configuration (ζ, δ), then it can not be disabled by $r' \in R_q$, $q \neq p$.*

Proposition 2.7 is easy to prove. r can be disabled only after either its input values are consumed, or its local state is updated. Neither case can be done by a firing action from another process.

2.1.3 Analysis of DPNs

Given a DPN P with initial configuration m_0, a configuration m is said to be *reachable* if there exists an execution $\pi \in \Gamma(P)$ such that $m_0 \xrightarrow{\gamma_0} \ldots \xrightarrow{\gamma_i} m \xrightarrow{\gamma_1}$ is a prefix of π. A

sub-execution starting from m is a sequence of transitions $\pi' = m \xrightarrow{\gamma} \dots$ such that there exists an execution $\pi \in \Gamma(P)$ and π' is a suffix of π.

In reality, it is expected that a DPN is effective to the intended input assignments, as virtually no implementation would be willing to take inputs without consuming them. The reason that P is not effective to an input assignment i is that no matter which execution is chosen, up to some number of transitions no process is able to fire any reading rules. This is either because that processes already terminates (so that no more firing rule is available) or because that some process requires some inputs that are not in the head of the input channels.

Without losing generality, we assume that all processes are non-terminating and there are always some reading actions available. Then if we look closer to these processes, we may distinguish between two cases. The first case is that some process p's reading rule requires some values from an input channel $x \in I_p$ but the channel is empty. The second case is that the reading rule requires some input value from x, however there is some other value at the head of x that can't be consumed by any of its firing rules. These two cases may also happen at the same time at different input channels. If $x \in I_P$, we say that i is *incompatible* with P. This is possible even when x is empty, since there might be other input channels that are not empty, and the emptiness of x is the cause preventing other inputs from consumption. If $x \in U_P$ such that $x \in I_p \cap O_q$ for processes p and q, then if $i(x) \neq \epsilon$ we say that p *and* q *are incompatible*. Finally, the case when $i(x) = \epsilon$ is the most interesting and important one, which we discuss in detail in the following section.

Causality and Deadlock of DPNs

The last case occurs when some process p requires reading from some other process q, however at the same time q also requires reading from p. Consider the following example in Figure 2.2.

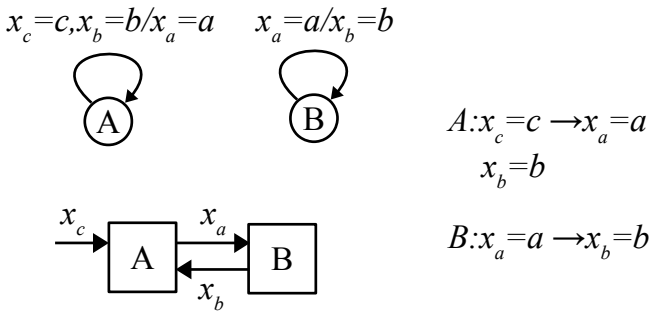


FIGURE 2.2: Causality problem of a DPN with two nodes.

Figure 2.2 shows a DPN $P = A|||B$. Each process's LTS is shown correspondingly. A and B share two channels x_a, x_b and x_c is an exclude input channel for A. A has a

single firing rule $r_A : \langle (b, c) : a \rangle$ and B has a single firing rule $r_B : \langle a : b \rangle$. The DPN starts execution from the initial state where both x_a and x_b are empty. The environment can push values to x_c, however this is actually the only action possible since the DPN immediately falls into a **deadlock**: since r_A requires a value from x_b and r_B requires a value from x_a and yet both buffers are empty, each has to wait for the other and no process can break the mutually waiting situation. In terms of causality, we say that there is a *causal cycle* between A and B, indicating the mutual dependent relation between r_A and r_B. Since the deadlock happened right from the initial configuration, P is unreactive to all possible input assignments.

It is not difficult to see that if we would like to prevent A and B from deadlock, then we have to avoid all firing rules like r_A, r_B (where $\exists x_b \in I_A \cap O_B, x_a \in O_A \cap I_B$ and $(r_A(x_b) = r_B(x_b) \neq \epsilon) \wedge (r_A(x_a) = r_B(x_a) \neq \epsilon)$) from reach. In general, we have to forbid a "chain" of firing rules from being reached, as long as they form a cycle of dependency.

Definition 2.8. For a reachable configuration $m = (\zeta, \delta)$ of a DPN P, m is in *deadlock* if it satisfies the following conditions:

- there exists $Q \subseteq P$, such that for all $\pi \in \mathcal{R}(m)$, there is not read action of any process from Q in π.
- there exists $C \subseteq (U_P \cap U_Q)$ such that for all $c \in C$, $\zeta(c) = \epsilon$.
- there exists input assignment $i : C \to \Sigma^*$ and a reading rule $r \in R_{p_i}$ and $p_i \in Q$ such that for configuration $m' = (\zeta \cdot i, \delta)$, there exists an execution $m' \xrightarrow{\gamma} \ldots \xrightarrow{r}$.

Intuitively, a configuration m is in deadlock means that there is a group of processes $Q = \{p_{k_1}, \ldots, p_{k_m}\}$ such that each p_{k_i} waits for some input from $p_{k_{i-1}}$ and p_{k_1} waits for some input from p_{k_m}. During their waiting, each process can only execute non-reading rules. The waiting situation can only be broken by inserting additional inputs to some of the empty channels along the cycle.

Undecidability in the Analysis of DPNs

It is shown by Buck [12] that *Boolean-controlled Dataflow Graphs*(BDFGs) are Turing complete. It is straightforward to show that our definition of DPNs can be easily used to describe any BDF graph. A BDF graph is a DPN with each process implementing a BDF actor. A BDF actor can be either a *regular dataflow actor* or a *control actor*. A regular dataflow actor consumes from each of its input channels a fixed number of values and produces for each output channel a fixed number of values. In other words, it implements a multi-valued function $(o_1, \ldots, o_n) = f(i_1, \ldots, i_m)$ where each i_j and o_k are variables of a string with a fixed length. It can be implemented by a process as shown in Figure 2.3.

As shown in Figure 2.3, the first sequence of transitions of the LTS reads all input strings $(i_1, \ldots, i_m)$. Then outputs $(o_1, \ldots, o_n)$ are produced. After the last value of o_n is written, the LTS goes back to the initial state and is ready for the next round of

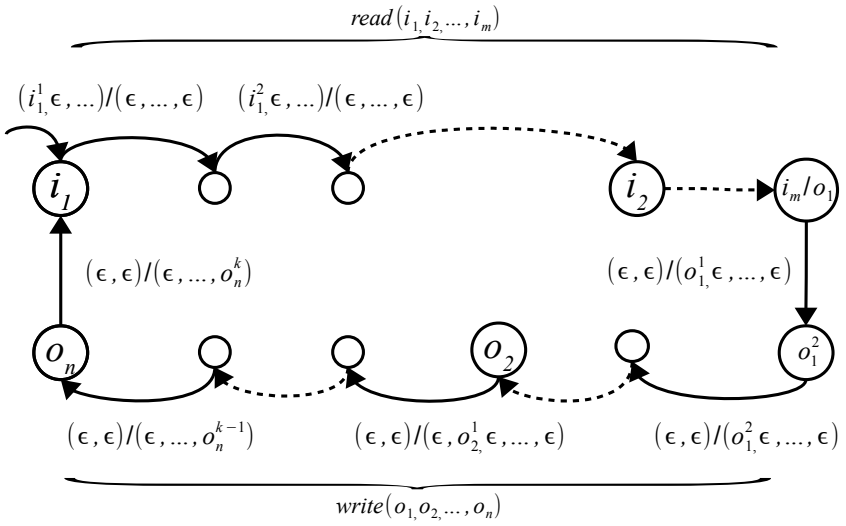


FIGURE 2.3: A process implementing a regular dataflow actor.

computations. A control actor can either be a select actor, or a switch actor. A selector can be implemented by a process with the following firing rules:

$$\langle(\mathsf{true}, a, \epsilon) : a\rangle, \langle(\mathsf{false}, \epsilon, b) : b\rangle$$

and a switch actor can be implemented by a process with the following firing rules:

$$\langle(\mathsf{true}, a) : (a, \epsilon)\rangle, \langle(\mathsf{false}, a) : (\epsilon, a)\rangle$$

For a finite input assignment i, if for all $\pi \in \Gamma(P)$ with $\pi|_{I_P} = i$, π is finite, then we say P terminates with i. Since DPNs are Turing complete, checking whether a DPN *terminates* with a given input is undecidable. Therefore we summarize the above discussions by the following theorem:

Theorem 2.9. *(Buck [12]) Dataflow process networks with a fixed number of finite processes are Turing complete, and checking if such a DPN is terminating with regard to a finite input assignment is undecidable.*

Undecidability of DPN-like MoCs

The discussions of theorem 2.9 by Buck [12] are different from the discussions stating Kahn Process Networks (KPNs) are Turing complete (e.g. in [7]). In a

KPN, a process can be any sequential program, therefore can already mimic a universal Turing machine by itself (which can only be implemented by infinitely many local states) and is much more powerful than a finite process. The additional unbounded FIFO buffers do not add more expressiveness. In this sense our definition of finite processes are more similar to communicating finite state machines in [13], where a finite state machine is used to define a local process, and the network of processes are built by connecting each pair via a full-duplex error-free FIFO channel. Therefore communicating finite state machines can be seen as a DPN of finite processes where each pair of processes have two FIFO channels sending values to each other. It is proven that such communicating finite state machines are Turing complete, and reachability is undecidable. These theoretical results are consistent with our results on DPNs. For completeness we give proofs for our DPNs.

Using theorem 2.9 it is not difficult to prove that reachability of a configuration of a DPN is undecidable.

Theorem 2.10. *Checking if a configuration is reachable for a DPN is undecidable.*

Proof. Consider any DPN $P = p_1 ||| \ldots ||| p_n$ as shown in Figure 2.4.

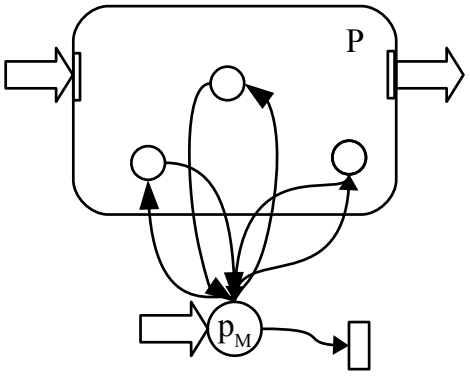


FIGURE 2.4: Checking for p if state 2 is reachable or not is undecidable.

Now we modify P by adding a monitor DPN P_M and modify each p_i as follows:

- Add two channels x_i^M to $(I_{p_i} \cap O_{P_M})$ and y_i^M to $(O_{p_i} \cap I_{P_M})$;
- Replace each firing rule $r \in R_{p_i} : \langle \vec{I} : (s \to s') : \vec{O} \rangle$ by
$$r' : \langle (\vec{I}, x_i^M = \uparrow) : (s \to s') : (\vec{O}, y_i^M = \uparrow_r) \rangle$$
- For each state s, add the rule:
$$r_c^{p_i} : \langle ((\epsilon, \ldots, \epsilon, x_i^M = \downarrow) : (s \to s_c^{p_i}) : (\epsilon, \ldots, \epsilon, y_i^M = \downarrow)) \rangle$$
into R_{p_i} where $s_c^{p_i}$ is a fresh state added to S_{p_i}.

- For each input channel x_i, add firing rule to R_{p_i}:

$$r_{c_j} : \langle(\epsilon, \ldots, x_j = a, \ldots, \epsilon, \downarrow) : (s_c^{p_i} \rightarrow s_c^{p_i}) : (\epsilon \ldots \epsilon, \downarrow_{r_{c_j}})\rangle$$

Let the updated DPN of P be P'. It is straightforward to see that in the presence of $\uparrow$ the behavior of P' is the same as P. We build the process network P_M to monitor the status of P'. In the following we describes intuitively how P_M works, however the full construction of such a DPN is complex and we omit the technical details.

P_M saves all firing rules of processes in P'. In the beginning, the input of P' is copied to P_M. Then P_M send a token value $\uparrow$ to processes of P' in a round robin way to trigger the computations of P. In the start of each round, P_M send $\uparrow$ to one process p_i. Whenever a process p_i fires, it consumes its data values as well as the $\uparrow$ token, produces output values and sens back a token $\uparrow_r$ informing P_M the the rule just fired. Then P_M send $\uparrow$ to p_{i+1} (or p_1 if $i = n$) to start the next round. In this way P_M monitors the configuration of P', since it knows exactly the contents of input channels and local channels of P' as well as each update of local states of processes in P'. Therefore it is able to check at any configuration of P', if there is any enabled firing rule in a process. Note that P_M can always be implemented by a fixed number of finite processes, since such DPNs are already Turing complete.

If in the beginning of one round, a process is not able to fire any rule, P_M will skip p_i and send $\uparrow$ to the next process. If P_M skipped all n processes, then no process in P' can fire. Then P_M sent $\downarrow$ to the processes in a round-robin way. This will enable the processes update to their $s_c^{p_i}$ states and consume the values left in their input channels. P_M also monitors the consumption by collecting $\downarrow_{r_{c_i}}$ tokens. Therefore P_M knows when P's input and local channels are empty. Finally, when all input and local channels of P' are empty, P_M terminates.

Given the above construction, we try to check if the configuration $m_\epsilon = (\zeta, \delta)$ of P' is reachable, where $\forall c \in (I'_P \cup U'_P), \zeta(c) = \epsilon$ and $\delta = (s_c^{p_1}, \ldots, s_c^{p_n})$. Apparently, m_ϵ is reachable in P' if and only if P is terminating. Since checking termination of P is undecidable, checking whether m_ϵ is reachable in P' is undecidable.

□

Based on theorem 2.9 and 2.10 we can further prove that effectiveness, compatibility and causality are all undecidable.

Theorem 2.11. *P is any DPN with a finite number of finite processes and i is a finite input assignment:*

1. *Checking whether P is effective to i is undecidable.*
2. *Checking if processes of P are compatible to i is undecidable.*
3. *Checking if there exists a reachable deadlock configuration to i is undecidable.*

Proof. To prove each undecidable problem we can reuse the proof of theorem 2.10, reducing the problem to reachability.

Proof of 1. Checking effectiveness requires us to find an execution that consumes all inputs of i. Given any DPN P we can create a DPN $Q := P' \cup P_M$ similar to the proof of theorem 2.10. The difference here is that we add one special input value "$\sharp$" to Q, and a special input channel $x_\sharp$ to a process p_M of P_M. For the terminating local state s_T of p_M, we add one firing rule:

$$\langle(\epsilon, \ldots, \epsilon, x_\sharp = \sharp) : (s_T \to s_\sharp) : (\epsilon, \ldots, \epsilon)\rangle$$

where $s_\sharp$ is a fresh state. At the beginning, besides the input assignment i we also push the value $\sharp$ to $x_\sharp$. Therefore the input of Q is $i_\sharp : I_Q \cup \{x_\sharp\} \to (\Sigma \cup \{\sharp\})^*$ where $\forall c \in I_Q, i_\sharp(c) = i(c)$ and $i_\sharp(x_\sharp) = \sharp$. Then it is easy to see that Q is effective to $i_\sharp$ if and only if P is terminating to i. Since by theorem 2.9 termination of P is undecidable, checking effectiveness of Q is undecidable.

Proof of 2. To check if processes of P are compatible to i, we try to find a reachable configuration where at an input channel of a process there exists a value that can never be consumed for all its sub-executions. For any given DPN P, we can construct a DPN $Q := P' \cup P_M$ similar to the proof of theorem 2.10. The difference here is that for the terminating local state s_T of $p_M \in P_M$, we add one firing rule:

$$\langle(\epsilon, \ldots, \epsilon) : (s_T \to s_\epsilon) : (\epsilon, \ldots, x_i^M = \uparrow, \ldots, \epsilon)\rangle$$

where s_ϵ is a fresh state. If this firing rule is executed, the value $\uparrow$ will never be consumed by p_i since P' already terminates. Therefore p_i is not compatible with p_M. This is the case if and only if the configuration m_ϵ (defined in the proof of theorem 2.10) is reachable, which is undecidable by theorem 2.10.

Proof of 3. To show undecidability of deadlock detection, we show that an input assignment i leads to a deadlock configuration of a DPN if and only if the DPN terminates for i. For any DPN P, we build a DPN $Q := P' \cup P_M$ similar to the proof of theorem 2.10. The difference is that for the terminating state s_T of $p_M \in P_M$, we add one firing rule:

$$r_a : \langle(\epsilon, \ldots, y_i^M = a, \ldots \epsilon) : (s_T \to s_a) : (\epsilon, \ldots, x_i^M = b, \ldots, \epsilon)\rangle$$

where s_a is a fresh state, and add one firing rule to p_i:

$$r_b : \langle(\epsilon, \ldots, x_i^M = b, \ldots \epsilon) : (s_{c_i} \to s_b) : (\epsilon, \ldots, y_i^M = a, \ldots, \epsilon)\rangle$$

where s_b is a fresh state. We further modify p_M so that before it terminates, it consumes all values in y_i^M. If Q terminates for i, configuration m_ϵ of P' is reached first. Then p_M flushes channel y_i^M, leads to the configuration where both x_i^M and y_i^M are empty. Finally after all processes of $P_M \setminus \{p_M\}$ reached their terminating states and p_M reached s_T, still no firing rule is enabled. Let the corresponding configuration of Q be m_d. However if we add a value b to channel x_i^M, then r_b is able to fire and a is produced to y_i^M which then enabled r_a. Therefore m_d is a deadlock configuration. Since m_d is reachable if and only if P terminates, checking if m_d is reachable is undecidable. Therefore checking if i leads to a deadlock in Q is undecidable.

□

Finally Buck proved in [12] that checking boundedness of a BDF graph is also undecidable. Therefore checking boundedness for DPNs is also undecidable.

Theorem 2.12. *(Buck [12]) Checking if a DPN is bounded with an input assignment is undecidable.*

Given all these negative results, we can see that it is quite difficult to analyze a DPN in general, which is bad for the design of safety critical systems. In previous works [6] restricted versions of DPNs like static DPNs (where only regular dataflow actors are used) are proposed in the sense that their expressiveness are much less than Turing machines, but boundedness and deadlock are decidable. However, it is also evidence that for practical control applications the usage of select and switch actors are essential, therefore designers have to struggle and make up their minds between expressiveness and analyzability. In section 2.2 we introduce synchronous DPNs that keep the expressiveness while maintaining analyzability. By then, we will see that while preserving the expressiveness of select and switch actors, essentially all questions of interest are decidable. This helps us to rethink the undecidability results of finite DPNs (or BDF graphs). We can see that the undecidability comes from the summation of two aspects: expressive operators and unbounded memory. In particular, given expressive operators like switch and select, *although the FIFO channels are intended to model the unbounded delays during communication, they might be miss-used as unbounded memories.* This endures DPNs the power of a Turing machine. Therefore, instead of limiting the expressiveness of actors, it is really the *notion of communication* (or composition) that should be restricted. Similar observations have been made in [14], where the expressiveness of Lustre [15] is extended to cover recursive functions inside a process of a DPN. This way, like [7] each process performs more like a normal sequential program. Yet the communication between processes js still synchronous in the sense that all the values generated by a producer process are consumed immediately by the consumer process (in functional programming, such phenomenon corresponds to the case when a program can be implemented without using internal lists [16]).

2.1.4 Denotational Semantics of DPNs

While LTSs and corresponding firing rules describe how DPNs run operationally, we might be also interested in their functional characteristics. For example, when we implement an adder, we'd like to know that when we input a, b, the output is always $a + b$ rather than something else. One particular question of interest is: does a DPN implement a function mapping input flows to output flows? It might not be the case, since firing rules might be fired nondeterministically, therefore given the same input flow it might generate different output flows. In this case, the process implements a relation that is not a function. For embedded system applications, we would rather demand that our systems are deterministic such that the resulting DPNs implement functions. This usually constraints the nondeterministic choices inside a process and provides more

predictability of the system behavior which is important to the design of safety critical embedded systems. Functional characteristics of a system are covered by its denotational semantics [17].

Instead of describing how the system works step by step, denotational semantics specifies what the system does in general. In the context of DPNs, previous works by Kahn and others [10, 11, 18, 19] define for each process a *stream function* mapping streams (or histories) of input values to streams of output values. Each function is also an equation specifying the relation between inputs and outputs. A DPN is then a set of equations where channel connections are identified by their channel names. For example, the equation system of the DPN in Figure 2.1 is as follows:

$$\begin{cases} l_1 &= f_1(x_1) \\ l_2 &= f_2(x_2) \\ y &= f_3(l_1, l_2) \end{cases}$$

Naturally, given such an equation system, we expect to compute the output streams once input streams are given. One would also like to derive the function of the whole DPN so that the behavior of the network can be represented by the abstract function instead of the sum of all detailed functions. In deed, intuitively we see that the solution of the equation system with regard to x_1, x_2 and y would then be the denotational semantics (hopefully a function again) we can use for the whole network. In order to solve the equation system, we need to define how functions of processes can be composed in the sense of DPNs. For serial connected processes, the denotational semantics of the DPN can be derived by using classical functional composition. For parallel and cyclic compositions new composition operators are defined accordingly. In this way, a DPN is nothing more than the composition of functions of processes. Important properties of interest (like if the DPN is deterministic) can be reasoned on the level of the equation system of DPNs, without concerning how operationally the DPN works.

Denotational semantics of a system can be proposed separately from its operational semantics. Since they are pure mathematical objects, they serve perfectly as a standard for which the implementation should comply. Therefore a classical design methodology is to design the denotational semantics of a system at first and verify its correctness, then develop the operational semantics correspondingly under the guidance of the denotational semantics. Finally we verify if the operational semantics is faithful to the denotational semantics. If so, the operational model can be implemented with confidence.

Since operational semantics works at a more refined level, it is also natural to derive denotational semantics by abstraction from operational semantics. In particular, the LTSs naturally define the input-output relations of a DPN. The derived denotational semantics is of course faithful to the the operational aspects, but might not be the one we intended. It can then be used to examine the functional characteristics of the implementation, since the corresoinding operational models are the ones implemented. Denotational semantics has its limits when it comes to the operational aspects of interest. For example, we need the operational model to argue about boundedness of

communication channels. But when we analyze the functional aspects of the system, it would be much easier if we can forget about the operational details which are irrelevant.

Since denotational semantics and operational semantics are generally defined in different abstraction levels by different methodologies, a derived problem is to compare their expressiveness: for determined ways of defining denotational semantics and operational semantics, is there an operational model that can not be defined denotationally or vise versa? A closely related question is that, are two systems with the same denotations able to replace each other under all contexts (observationally equivalent)? The problem here is that while sharing the same denotation, two systems might have different operational implementations. It turns out that there are cases when under some context, the two implementations behave differently. If under no case this happens, the denotational semantics is said to be *fully abstract* [8].

In the following, we first discuss different ways to derive the denotations–input-output relations of a process as well as a DPN. Then we show why some of the ways are more preferred in the sense of full abstraction. Kahn's principle [18] shows that continuous stream functions are fully abstract for deterministic LTSs. There are more relaxed type of functions and more restricted type of functions as well, which form a hierarchy of functions. Since this is closely related to different notions of synchronous systems that are used for desynchronization, we'll have a closer look at each type of functions as well in later parts of the thesis.

Relations defined by LTS

The labeled transition systems naturally defines the input output relations of processes and DPNs. Therefore we have the following definitions.

Definition 2.13. The effective input/output relation of a process p is the relation:

$$\Phi(p) := \{(w|_{I_p}, w|_{O_p}) \mid w \in \mathcal{L}(p)\}$$

For $w = (w_i, w_o)$ there must be an execution $\pi = s_0 \xrightarrow[p]{l_1} s_1 \xrightarrow[p]{l_2} s_2 \to \ldots \xrightarrow[p]{l_n} s_n$ such that $l_1 \cdot l_2 \cdot \ldots l_n = w$, therefore $w_i = \mathsf{r}(\pi)$ and $w_o = \mathsf{w}(\pi)$. We use $s_0 \overset{w_i/w_o}{\Longrightarrow} s_n$ to indicate that there is an execution from $s_0 \overset{w}{\Rightarrow} s_n$. reading w_i and generating w_o and reaches s_n in the end. For simplicity, we also denote s. The general input/output relation of a process p is the relation $\Psi(p) := \Phi(p) \cup \Phi'(p)$, where $\Phi'(p)$ is defined as follows:

$$\Phi'(p) := \{(w_I, w_O) \mid w_I \notin \mathcal{L}(p)|_{I_p}, \exists (w'_I, w'_O) \in \Phi(p), w'_I = max(W)\}, \text{ where}$$
$$W := \{w \mid w \sqsubseteq w_I, w \in \mathcal{L}(p)|_{I_p}\}.$$

Definition 2.14. The effective input/output relation of a DPN P is the relation:

$$\Phi(P) := \{(\mathsf{r}(\pi), \mathsf{w}(\pi)) \mid \pi \text{ is an effective, maximal and fair execution of } P.\}$$

The general input/output relation of P is the relation:

$$\Psi(P) := \{(\mathsf{r}(\pi), \mathsf{w}(\pi)) \mid \pi \text{ is a maximal and fair execution of } P.\}$$

Intuitively, $(x, y) \in \Phi(P)$ means that there exists an execution such that all inputs of x are consumed to generate the output y, and output values in y are generated in a fair way, such that if an output value is guaranteed to be generated, it will eventually be generated. When $\Phi(P)$ is a function of $(I \to \Sigma_I^\infty) \to (O \to \Sigma_O^\infty)$, we call it a Φ-*function*. If a process p implements a function, we denote its function by f_p, and its domain and codomain by $dom(f_p), cod(f_p)$. Since we allow nondeterminism inside a process, Φ might not be a Φ-function. Note that LTS_p might still be non-deterministic even Φ is a function. Since for a process, it is possible that $\exists s, s', s \neq s'$ such that for the same $w = (w_i, w_o)$, $s_0 \xRightarrow{w_i/w_o} s$ and $s_0 \xRightarrow{w_i/w_o} s'$. The general I/O relations simply extend the relations to cover all possible inputs, and the corresponding output is the one corresponds to the effective execution where the maximum of the input is consumed.

A subtle thing here we need to pay special attention to is that LTS_p and $LTS_{\{p\}}$ (remember that $\{p\}$ is a DPN consisting of a single process p) induce different I/O relations. This is shown as the example of Figure 2.5.

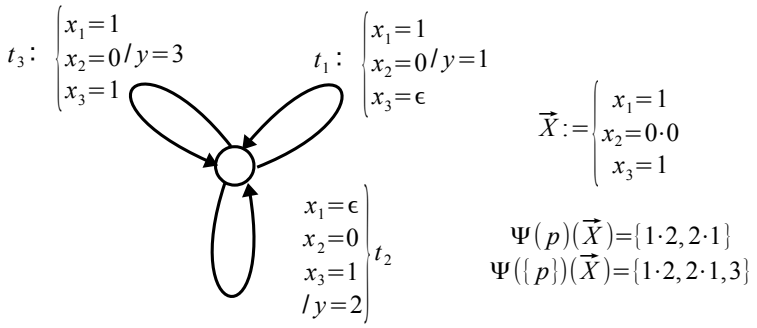


FIGURE 2.5: Compare the I/O relation of LTS_p and $LTS_{\{p\}}$.

For process p of Figure 2.5, the I/O relation derived from LTS_p is different from the I/O relation derived from $LTS_{\{p\}}$. For example, for the input sequence $x = (x_1 = 1, x_2 = 0 \cdot 0, x_3 = 1)$, $\Psi(p)(x) = \{1 \cdot 2, 2 \cdot 1\}$ in which each output sequence can be derived from firing t_1, t_2 in corresponding orders. However $\Psi(\{p\})(x) = \{1 \cdot 2, 2 \cdot 1, 3\}$ in which 3 is derived from executing t_3. This execution is not effective as it leaves one 0 in the input channel x_2, which can not be captured from LTS_p. Intuitively, in order to capture the precise behavior of a process in a DPN environment, we need to further add buffers in to consideration. In section 3.2.1 of the next chapter we will utilize this to capture the precise I/O relations of a process in a DPN.

It is evidence that for each $(r, w) \in \Phi(P)$ and each $p_i \in P$, $(r|_{C_{p_i}}, w|_{C_{p_i}}) \in \Phi(p)$. However, (r, w) is derived from LTS_P directly, rather than the I/O relations of each p_i. If we can derive $\Phi(P)$ from $\Phi(p_i)$ we wouldn't need to construct LTS_P. This compositional way to derive the behavior of P is indeed more preferred. A natural way to define the composition of I/O relations is as follows.

Definition 2.15. Let $P := p_1|||p_2$ be a DPN. $w_1 \in \mathcal{L}(p_1)$ and $w_2 \in \mathcal{L}(p_2)$. Then the composition of w_1 and w_2 is defined by $w := w_1|||w_2$ where $\forall c \in C_{p_1}, w(c) = w_1(c)$ and $\forall c \in C_{p_2}, w(c) = w_2(c)$, provided that $\forall c \in C_{p_1} \cap C_{p_2}, w_1(c) = w_2(c)$. We define $\mathcal{L}(P) = \mathcal{L}(p_1)|||\mathcal{L}(p_2) := \{w|\exists w_1 \in \mathcal{L}(p_1), w_2 \in \mathcal{L}(p_2), w = w_1|||w_2\}$. Then the I/O relation of $\mathcal{L}(P)$ is defined by $\Phi(\mathcal{L}(P)) := \{(w|_{I_P}, w|_{O_P}) \mid w \in \mathcal{L}(P)\}$. It is easy to extend the above definitions to a DPN with more than two processes.

Definition 2.15 provides another way to derive I/O relations of a DPN, which seems to be a rational one as well. The rationality really relies on the question: Is $\Phi(P) = \Phi(\mathcal{L}(P))$? The example of Figure 2.2 shows a counterexample. Consider process A and B separately, it is easy to see that $w_A = \{x_b = b_1, x_c = c_1, x_a = a_1\} \in \mathcal{L}(A)$ and $w_B = \{x_a = a_1, x_b = b_1, x_d = d_1\} \in \mathcal{L}(A)$, and $w = w_A|||w_B = \{x_a = a_1, x_b = b_1, x_c = c_1, x_d = d_1\} \in \mathcal{L}(P)$. Therefore $(x_a = a_1, x_d = d_1) \in \Phi(\mathcal{L}(P))$. However as we've seen that $P = \{A, B\}$ is deadlocked since the initial state, $\Phi(P) = \emptyset$. Therefore $\Phi(P) \neq \Phi(\mathcal{L}(P))$. The following lemma justifies that the equation $\Phi(P) = \Phi(\mathcal{L}(P))$ holds when P has no deadlock configuration.

Lemma 2.16. *If a DPN P has no deadlock configuration, then $\Phi(P) = \Phi(\mathcal{L}(P))$.*

Proof. (sketch) Without losing generality, we assume $P = \{p_1, p_2\}$ as shown in Figure 2.6.

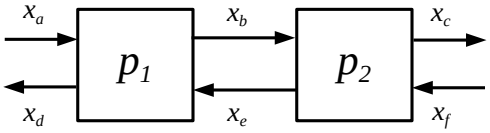


FIGURE 2.6: $P = p_1|||p_2$.

It is evidence that $\Phi(P) \subseteq \Phi(\mathcal{L}(P))$. Therefore if the equation doesn't hold, then there must be some $w = w_1|||w_2 \in \mathcal{L}(P)$ with $w_1 \in \mathcal{L}(p_1)$ and $w_2 \in \mathcal{L}(p_2)$ such that $(w|_I, w|_O) = (w|_{\{x_a, x_f\}}, w|_{\{x_c, x_d\}}) \notin \Phi(P)$. Assume $\pi_1 \in \Gamma(p_1)$ and $\pi_2 \in \Gamma(p_2)$ such that $\rho(\pi_1) = w_1$ and $\rho(\pi_2) = w_2$. Now we try to replay π_1 and π_2 in P. If we success in replaying all sequences of transitions of π_1 and π_2 then P is able to compute $w|_O$, which contradicts the assumption. We do this by first feeding $w|_I$ to x_a, x_f respectively (by firing the input actions accordingly), so that P has all the inputs needed to produce $w|_O$. Then p_1 and p_2 start to fire the sequences of actions of π_1 and π_2. By the assumption, at least one of them has to stop at some point. Without losing generality assume p_1 stops. This must be because that p_1 lacks input values from x_e, otherwise p_1 could have fired its intended rule. There wouldn't be unexpected values in x_e so that p_1 can't consume, since p_2 executes its intended firing rules according to w_2, and $w_1|||w_2$ exists which means the produced values at x_e should be exactly those that are needed by p_1. However, this indicates that p_2 must also be waiting, since otherwise p_2 would have produced the value p_1 needs. This means both x_e and x_b are empty, therefore p_1 and p_2 deadlock at this configuration.

□

The equation might still hold in the presence of deadlocks. This can easily be the case when all the intended outputs have been produced while there are still some internal transitions enabled, and after some internal transitions the network goes into a deadlock configuration. However these cases are not of our interests, since we wouldn't allow deadlocks occur in implementations anyway.

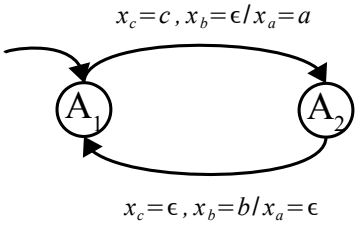


FIGURE 2.7: LTS of process A', where $\mathcal{L}(A) = \mathcal{L}(A')$.

Consider process A' as shown in Figure 2.7. Comparing A' with A in Figure 2.2, it is not difficult to check that $\Phi(\mathcal{L}(A')) = \Phi(\mathcal{L}(A))$, however since $P' = \{A', B\}$ has no deadlock, $\Phi(P') = \Phi(\mathcal{L}(P'))$.

Full Abstraction of Relations of LTSs

Regarding the full abstraction problem, we can see that the denotational semantics derived from $\Phi(\mathcal{L}(P))$ is not fully abstract for the operational semantics defined by LTSs (even if the relations are functions). The counterexample has shown by processes A and A'. They share the same I/O-relations. However comparing $P = \{A, B\}$ and $P' = \{A', B\}$, we've seen that $\Phi(P) \neq \Phi(P')$ since P deadlocks from the start while P' has no deadlock. The source of the problem is exactly that the composition of runs ($|||$) operates on the level of strings, therefore ignores causality of the DPN and may generate I/O relations that are not respecting the causality of the DPN. By lemma 2.16, we can apply the replacement only for DPNs that are deadlock-free.

Relations defined by the Kahn's Principle

As we discussed in the beginning already, instead of deriving I/O-relations from LTSs we can assign to each process of a DPN a stream function, and the I/O-relation of the whole DPN can be computed as the solution of the equation system of the DPN. Kahn demonstrated that when the stream functions are *continuous*, a least solution of the equation system can be computed following the *least fixpoint computation* according to Kleene's fixpoint theorem. Further more, this least solution defines a continuous I/O-function of the DPN, and it coincides with the operational semantics of the LTS of the DPN provided that each LTS of process is faithful to the stream function assigned to the process. This is called *Kahn's principle*. Therefore, least fixpoint semantics is a fully abstract denotational semantics for DPNs of processes implementing continuous functions. This turned out to be an important theoretical foundation for desynchronization as well, since our target DPN are expected to be deterministic. Later we will

also see that continuous functions set an important criteria for the desynchronization of processes when it comes to the implementation. For these reasons, we give a detailed presentation of Kahn's principle, starting from some basic definitions.

Definition 2.17. A binary relation $(S, \leq)$ on a set of elements S is a *partial order* if it satisfies:

- Reflexivity: $\forall x \in S, x \leq x$
- Anti-symmetry: $\forall x, y \in S, x \leq y \wedge y \leq x \implies x = y$
- Transitivity: $\forall x, y, z \in S, x \leq y \wedge y \leq z \implies x \leq z$

A *chain* of $(S, \leq)$ is a sequence of: $d_0 \leq d_1 \leq \ldots$ with each $d_i \in S$. $e \in S$ is a *lower bound* of $M \subseteq S$ iff $\forall x \in M, e \leq x$. $e \in S$ is an *upper bound* of $M \subseteq S$ iff $\forall x \in M, x \leq e$. e is the *greatest lower bound*(infimum) of $M \subseteq S$ iff e is a lower bound of M and $\forall x$ a lower bound of $M, x \leq e$. e is the *least upper bound*(supremum) of $M \subseteq S$ iff e is an upper bound of M and $\forall x$ an upper bound of $M, e \leq x$. We denote the infimum of M by $\inf(M)$ and the supremum of M by $\sup(M)$.

A *complete partial order* (CPO) is a partial order where for each increasing chain $D := d_0 \leq d_1 \leq \cdots \leq d_n \leq \ldots$, there exists $sup(D)$.

Definition 2.18. For two CPOs $(S, \leq)$ and $(T, \sqsubseteq)$, a function $f : S \to T$ is *monotonous* iff $\forall x, y \in S, x \leq y \implies f(x) \sqsubseteq f(y)$. f is *continuous* iff for any increasing chain $D := d_0 \leq d_1 \leq \cdots \leq d_n \leq \ldots, f(\sup(D)) = \sup(f(D))$.

For a function $f : S \to S$, $x \in S$ is a *fixpoint* of f iff $f(x) = x$.

Lemma 2.19. *Every continuous function is mononone.*

Proof. Assume $f : S \to T$ is a continuous function over two CPOs $(S, \leq)$ and $(T, \sqsubseteq)$, and for $x, y \in S, x \leq x$. We need to prove that $f(x) \sqsubseteq f(y)$ holds. Since $x \leq y$, we have $\sup(\{x, y\}) = y$, therefore $f(y) = f(\sup(\{x, y\}))$. By continuity of f and $x \leq y$ a chain we also have $f(\sup(\{x, y\})) = \sup(\{f(x), f(y)\})$. Therefore $f(y) = \sup(\{f(x), f(y)\})$, which implies $f(x) \sqsubseteq f(y)$. □

From now on we denote $\sqsubseteq$ as the prefix order of tuples of strings (e.g., $(\epsilon, 0 \cdot 1) \sqsubseteq (0, 0 \cdot 1)$), as we introduced in the beginning of this section. Following the above definitions, it can be proven that $((\Sigma^\infty)^n, \sqsubseteq)$ is a complete partial order where $\sqsubseteq$ is the prefix order over the domain of n-tuples of strings $(\Sigma^\infty)^n$ [20]. Then for a DPN $P = \{p_1, \ldots, p_n\}$ we can define for each process p_i of P a *stream function* $f_{p_i} : (\Sigma^\infty)^m \to (\Sigma^\infty)^n$ where p_i has m input channels and n output channels. Therefore we can build the *equation system* of P as follows:

$$\begin{cases} \vec{y_1} & = & f_{p_1}(\vec{x_1}) \\ \vec{y_2} & = & f_{p_2}(\vec{x_2}) \\ & \cdots & \\ \vec{y_n} & = & f_{p_n}(\vec{x_n}) \end{cases}$$

where each $\vec{x_i}$ and $\vec{y_i}$ are input channels and output channels of p_i. Note that for some channel c, it may appear on the left hand side and the right hand side of different equations at the same time. For example, the equation system of Figure 2.6 is as follows:

$$\begin{cases} (x_d, x_b) &= f_{p_1}(x_a, x_c) \\ (x_c, x_e) &= f_{p_2}(x_b, x_f) \end{cases}$$

where both x_b and c appear on the left and right hand side of the equations at the same time. It is evidence that this happens if and only if the channel is an internal channel that connects two different processes. With a little change of syntax, we can extract for each output channel of y_i a dedicated equation. Therefore the equation system can be rewritten as follows:

$$\begin{cases} y_1^1 &= f_{p_1}^1(\vec{x_1}) \\ y_1^2 &= f_{p_1}^2(\vec{x_1}) \\ & \cdots \\ y_1^{k_1} &= f_{p_1}^{n_1}(\vec{x_1}) \\ y_2^1 &= f_{p_2}^1(\vec{x_2}) \\ & \cdots \\ y_n^{k_n} &= f_{p_n}^{k_n}(\vec{x_n}) \end{cases} \tag{2.1}$$

where each y_j^i is the i-th output channel of process p_j. Again y_j^i may appear on the right hand side of some equations, provided it is an internal channel. For such an equation system consists of continuous functions, it is proven that there is a unique minimal solution (in the sense of prefix order $\sqsubseteq$) to the equation system.

Theorem 2.20. *(Kahn's Principle [11, 18, 21]) The equation system 2.1 where each $f_{p_j}^i$ is a continuous function has a unique minimal solution. Furthermore, the I/O relation defined by the minimal solution is a continuous function.*

The minimal solution of equantion system 2.1 can be computed by applying the Kleene's fixpoint theorem.

Theorem 2.21. *(Kleene's fixpoint theorem [10, 11, 18]) The minimal solution of equantion system 2.1 can be computed as follows:*

1. *In the start, construct $X_0 : (I_P \cup U_P) \to \Sigma^\infty$ by assigning each $x \in I_P$ a predefined string; assign each $x \in U_P$ the empty string ϵ.*
2. *For the assignment $X_i : (I_P \cup U_P) \to \Sigma^\infty$, compute the assignment $Y_i : (U_P \cup O_P) \to \Sigma^\infty$ by applying each function $f_{p_j}^i$ of the equation system 2.1 once over X_i. Then update the assignment $X_i : (I_P \cup U_P) \to \Sigma^\infty$ to $X_{i+1} := X_i|_{I_P} \cdot Y_i|_{U_P}$.*
3. *Repeat step 2 by the updated assignment X_{i+1} until $Y_{i+1} = Y_i$, i.e. the fixpoint is reached.*

The finally computed Y_i is the least fixpoint of the equation system.

Let's take the DPN of Figure 2.6 as an example. We first assign f_{p_1}, f_{p_2} to p_1, p_2 as follows.

$$\begin{cases} x_d = f^d_{p_1}(x_a, x_e) = a_1 - e_1, \ldots, a_k - e_k, & \text{if } x_a = a_1 \cdot \cdots \cdot a_m, x_e = e_1 \cdot \cdots \cdot e_n \text{ and } k = min(m,n) \\ x_b = f^b_{p_1}(x_a, x_e) = a_1 + e_1, \ldots, a_k + e_k, & \text{if } x_a = a_1 \cdot \cdots \cdot a_m, x_e = e_1 \cdot \cdots \cdot e_n \text{ and } k = min(m,n) \\ x_e = f^e_{p_2}(x_b, x_f) = b_1 + f_1 \cdot \cdots \cdot b_k + f_k, & \text{if } x_b = b_1 \cdot \cdots \cdot b_m, x_f = f_1 \cdot \cdots \cdot f_n \text{ and } k = min(m,n) \\ x_c = f^c_{p_2}(x_b, x_f) = f_1 \cdot \cdots \cdot f_k, & \text{if } x_b = b_1 \cdot \cdots \cdot b_m, x_f = f_1 \cdot \cdots \cdot f_n \text{ and } k = min(m,n) \end{cases}$$

Assume $x_a = 5 \cdot 6 \cdot 3, x_f = 5 \cdot 6 \cdot 3$. The least fixpoint computation is performed in the following steps:

0. The initial assignment $X_0 := \{x_a = 5 \cdot 6 \cdot 3, x_f = 5 \cdot 6 \cdot 3, x_b = \epsilon, x_e = \epsilon\}$.
1. The first round of fixpoint computation: apply functions $f^d_{p_1}, f^b_{p_1}, f^e_{p_2}, f^c_{p_2}$ over X_0 and derive $Y_0 := \{x_d = \epsilon, x_b = \epsilon, x_e = \epsilon, x_c = \epsilon\}$. Update X_0 to $X_1 := \{x_a = 5 \cdot 6 \cdot 3, x_f = 5 \cdot 6 \cdot 3, x_b = \epsilon, x_e = \epsilon\}$ (which is the same as X_0).
2. The second round of fixpoint computation: apply functions $f^d_{p_1}, f^b_{p_1}, f^e_{p_2}, f^c_{p_2}$ over X_1. Since $X_1 = X_0$, we derive $Y_1 = Y_0$ and the fixpoint is reached, which equals to Y_0.

We can construct for each process an LTS as shown in Figure 2.8.

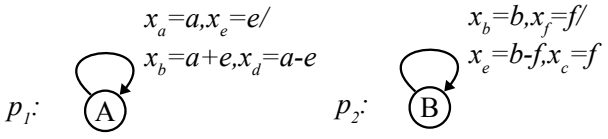


FIGURE 2.8: LTSs of processes p_1, p_2.

It is not difficult to see that for each p_i, the general I/O-relation coincides with the I/O relation $R_{f_{p_i}}$ defined by f_{p_i} (where $(w_i, w_o) \in R_{f_{p_i}}$ iff $w_o = f_{p_i}(w_i)$). More importantly, by following the least fixpoint computation we derive the output $\{x_d = \epsilon, x_b = \epsilon, x_e = \epsilon, x_c = \epsilon\}$ which is the same as if we use the general I/O-relation of the LTS of the DPN. This is the case since the DPN $P = \{p_1, p_2\}$ starts from a deadlock configuration no matter what input stream are there in x_a and x_f. It is also important to see that there are more than one fixpoint of the equation system for the given input X_0, for example $Y := \{x_d = 4 \cdot 4 \cdot 2, x_b = 6 \cdot 8 \cdot 4, x_e = 1 \cdot 2 \cdot 1, x_c = 5 \cdot 6 \cdot 3\}$ is another fixpoint. The corresponding I/O-relation $(w_i = \{x_a = 5 \cdot 6 \cdot 3, x_f = 5 \cdot 6 \cdot 3\}, w_o = \{x_c = 5 \cdot 6 \cdot 3, x_d = 4 \cdot 4 \cdot 2\})$ can be derived from $\Psi(\mathcal{L}(P))$ (but not from $\Psi(P)$), however is not the least fixpoint of the equation system. The following theorem concludes the important fact that the least fixpoints computed are exactly those can be derived from $\Psi(P)$.

Theorem 2.22. *(Kahn's Principle [11, 18]) For a DPN P with each process of P implementing a continuous function, its least fixpoint solutions correspond exactly to the general I/O-relations $\Psi(P)$.*

Full Abstraction of Kahn Processing Networks

By Kahn's principle, the behavior of each process of a DPN can be abstracted by a continuous function. Such abstraction does not change the behavior of the whole DPN, as its I/O-relation computed by the least fixpoint of the equation system coincides with the I/O-relation computed by the operational semantics. Therefore, in computing functional behaviors of a DPN, we can replace any process as well as sub-DPN by its continuous function without losing the behavior of the global DPN. Further more, we can replace processes (DPNs) implementing the same continuous function with each other without breaking the functional behavior. This means that the least fixpoint semantics is fully abstract to DPNs consisted of processes implementing continuous functions.
When considering non-continuous functions or even general relations, Kahn's principle fails. As sometimes one wants a DPN to be non-deterministic (i.e. implementing a general relation that is not a function), there are many previous works trying to establish fully abstract denotational semantics for non-deterministic DPNs [11, 22–26]. Since we only consider deterministic DPNs, our discussion of fully abstract denotational semantics stops here.

2.2 Synchronous Process Networks

In this section we look at a restricted class of dataflow process networks, i.e. *synchronous process networks* (SPN). SPN is the model we used to specify synchronous systems. SPN belongs to the synchronous models of computation [27]. Synchronous MoCs are invented to provide a succinct formal foundation to model and analyze safety-critical embedded systems. Its motivation can be well explained from the following paragraph quoted from [27]:

> The primary goal of a designer of safety-critical embedded systems is convincing him- or herself, the customer, and certification authorities that the design and its implementation is correct. At the same time, he or she must keep development and maintenance costs under control and meet nonfunctional constraints on the design of the system, such as cost, power, weight, or the system architecture by itself. ..... Meeting these objectives demands design methods and tools that integrate seamlessly with existing design flows and are built on solid mathematical foundations. ..... The key advantage of using a solid mathematical foundation is the ability to reason formally about the operation of the system.

Based on these observations several major design decisions were made in the development of *synchronous languages*, including Esterel [28], Lustre [15, 29] and Signal [30]. Our SPN is independent from any synchronous language, but shares the same spirit with synchronous dataflow languages like Lustre and Signal. The most important simplification of synchronous MoCs is the usage of an universal notion of time in the sense of

discrete instants. This means all components of the system preserve the same evolve of time in lock steps. This model of time shares great similarity with synchronous circuits where gates are synchronized by one global clock. By this similarity we can also assign a *logical clock* to the synchronous system, where each instant (or clock cycle) is assigned a natural number. In this way we can talk about a *sequence of instants* (or cycles). This model of time also favors the modeling of reactive systems (many of which are also safety-critical embedded systems) in the sense that a reactive system runs in periods of reactions against the environment, therefore each clock cycle can be mapped to a cycle of reaction to the environment. The dictatorship of a global logical clock forces the components to run in lock steps. This in turn greatly simplifies the composition of synchronous systems. In particular, synchronous composition is much more compact in the sense that all components have to contribute a clock cycle which combined together to form a global cycle of the system. Instead, asynchronous models of computation suffer from interleaved semantics that is caused by asynchronous computations, which is often the source of complexity for analysis. This is exemplified by the difficulty in the analysis of DPNs in section 2.1.3. Indeed, DPN is quite an expressive model largely because unbounded sized FIFO buffers are used. However, anaylsis is still difficult even for some models using synchronous rendezvous communications (e.g. CCS) simply because asynchronous interleaving still exists. For this reason, partial-order based methods are developed to cope with the overly complexed interleaving semantics of the system model (e.g. partial-order reduction in model checking [31]).

Since synchronous composition forces components to communicate within one cycle, the communication requires *no buffer* at all. This leads to another important simplification. In particular, during one clock cycle, all the variables of the system poses a single value. There is no time elapsed during one cycle, which makes every event within the cycle running "instantaneous"–this is called *perfect synchrony* [32]. However events can not happen at any order–they have to obey causal rules, i.e. the events are partially ordered according to their data dependencies. The execution of these events together form the execution of one clock cycle. Therefore these events are also called "*micro-steps*", and each clock cycle is also called a "*macro-step*". Only after all micro-steps are executed, may the system finish the current macro-step and move to the next macro-step. Therefore, only finitely many micro-steps are allowed within each macro-step. It is usually demanded that the number of micro-steps are even bounded, so that only a limited amount of resource is needed. If such requirement is satisfied, the synchronous system is often called *reactive*. Finally, synchronous systems are *deterministic* in the sense that for a given input and a local state, the output and the next state are determined uniquely.

Largely because of the simplified notion of composition, synchronous MoCs are able to provide powerful tools for analysis which makes it much more preferred for modeling safety-critical embedded systems. The following remark shortly introduces the major classes of synchronous languages. In the following section we introduce the operational semantics of SPNs.

Synchronous Programming Languages

Following the synchronous paradigm, several major synchronous programming languages as well as their variants have been developped in the past three decades. Most of the concepts introduced in this thesis are invented during the development of these languages, therefore we feel the need to give them at least some informal introduction, so that interested readers can follow the threads to their sources.

- Imperative synchronous languages: Esterel [28] is the representative language of this class, and it is also the most distinguishing synchronous language as it set up the foundation of synchronous paradigm. It is an imperative language, therefore it provides normal control statements like if-then-else and while loops. It is particularly designed for development of control-intensive embedded system applications, therefore it also has sophisticated preemptive statements like abort [33]. As a synchronous language, in order to explicitly specify different macro steps, instantaneous statements and non-instantaneous statements are distinguished for which an instantaneous statement is executed within one macro-step. For example, a typical assignment is instantaneous. The most important non-instantaneous statement is pause which marks the end of a macro-step, therefore separates different instantaneous statements into different macro-steps.

  Esterel follows the perfect synchrony paradigm introduced before, therefore is deterministic, i.e. for each cycle every variable has exactly one value, and the successive state is completely determined by the current state. Interestingly, although originally targeting sequential (single-threaded) code synthesis executed on a single-CPU machine, Esterel is a multi-threaded language. The composition of threads follows synchronous composition. Such explicit support of (synchronous) communication is also a trade mark of essentially all synchronous languages, as synchronous composition is one crucial feature of synchronous MoCs. In order to stay reactive, no recursion is allowed. Other variants of synchronous imperative languages include Quartzs [20] (which is the synchronous language developed by the research group where the author of this thesis works at) and Statechart [34, 35].

- Functional synchronous data-flow languages: Lustre [15] is the representative language of this class. Different from Esterel, Lustre's design is based on the simple data-flow model which is widely applied in signal processing systems: systems of equations that form a data-flow network. Each node can be seen as a functional operator mapping a sequence of inputs to a sequence of outputs, which is called a *sequence operator*. Each non-input sequence is defined in terms of some other sequences, making the language functional. Similar to BDF graphs of PTOLEMY [36], Lustre provides a set of primitive sequence operators along with ordinary sequence operators including: previous (pre), followed by ($-\ >$), down-sampling (when) and

memory access (current). Different from BDF graphs, the sequence operators are synchronously composed. Although also named "synchronous data-flow", static dataflow graphs (SDGs, the restricted BDF graphs) do not follow synchronous paradigm, in the sense that the composition of actors are asynchronous. Synchrony of SDGs instead indicates the existence of a periodic scheduling so that buffers are bounded.

Because of the functional natural of the language, it is straightforward to derive an operational semantics that can be used for code synthesis. This also makes Lustre deterministic just like Esterel. The original semantics of Lustre also matches the least fixpoint semantics of Kahn's processing networks [15, 37]. Therefore, all nodes of the synchronous process network defined by Lustre implement continuous functions (to be precise, they are even sequential functions which we will introduce in the next chapter). Other similar language include ReactiveML [38], Lucid Synchrone [39].

- Declarative synchronous data-flow languages: Signal [30] is the representative language of this class. Like Lustre, Signal also follows data-flow style of modeling. The difference is that Signal tend to work at a higher abstract level as a specification language. In particular, although similar sequence operators are defined, the equation system of Signal is referred as a specification of the relation between sequence of inputs and outputs. Therefore, a Signal program is quite denotational and may specify some behaviors that are implementable. This of course is a problem when comes to implementation, but might be desired as a specification language. Also, it might be the case that information of input sequences can be derived from some output sequences, as equations in Signal do not have directions (unless Lustre, where equations are treated as functions, therefore are always directed from inputs to outputs).

  Another notable feature of Signal is the *polychronous* modeling of time, i.e. time instants of different variables are partially ordered. This means that clocks of two different variables might not be comparable, i.e. they are neither present/absent at the same time nor one is present and the other is absent at the same time. Such modeling of time follows naturally a multi-clocked distributed system where clocks of different systems may be not comparable. MRICDF [40] is a variant of Signal.

It is important that although sharing the same principle of synchrony, different languages have different perspective on time. In particular, both Lustre and Signal allow a signal to be *absent* during one cycle in the sense that there is no value possessed by the signal. Although Esterel also allows a signal to be absent, still the signal can be read by others and its value is simply taken from the previous cycle. We will have a more detailed discussion about this topic in section 2.2.2.

2.2.1 Single-rated SPNs

LTS of synchronous processes

A synchronous process network is a network of *synchronous processes*. Like a process of a DPN, a synchronous process can be specified by its labeled transition system.

Definition 2.23. A labeled transition system $LTS_p : \langle I, O, S, s_0, L, \rightarrow_p \rangle$ of a synchronous process p is an LTS with the following additional constraints:

- L is the set of *labels*, where each label $l \in L$ is an assignment: $(I \cup O) \rightarrow (\Sigma \setminus \{\epsilon\})$;
- $\rightarrow_p \subseteq S \times R \times S$ is the set of labeled transition relations, where for any two transitions $s \xrightarrow[p]{l} s_m$ and $s \xrightarrow[p]{l'} s_n$, $l = l' \implies s_m = s_n$.

As shown in the definition, a synchronous process is a special process in the sense that in each transition the process must read exactly one value from each of its input channels and write one value to each of its output channels. For simplicity we also denote $\Sigma \setminus \{\epsilon\}$ by $\hat{\Sigma}$. Furthermore, the LTS is deterministic in the sense that for the same state with the same label of inputs, the label of outputs and successive state are uniquely determined. Similar to processes, LTS_p can be simply specified by a set of firing rules R_p. In the rest of the thesis, we assume that all synchronous processes are deterministic.

The definitions of executions and runs of a process also applies for a synchronous process. Here we define a *synchronous run* in order to reflex the specialty of synchronous processes that are equipped with logical clocks.

Definition 2.24. A *synchronous run* of a synchronous process $p : \langle I, O, S, s_0, L, \rightarrow_p \rangle$ according to its execution $\pi := s_0 \xrightarrow[p]{l_0} s_1 \xrightarrow[p]{l_1} s_2 \rightarrow \dots$ is a function $\hat{\rho}(\pi) : \mathbb{N} \rightarrow L$ such that $\forall i \in \mathbb{N}, \hat{\rho}(i) = l_i$.

Therefore $\hat{\rho}(i)$ is the variable assignment of the ith cycle. We denote the set of synchronous runs of synchronous process p by $\hat{\mathcal{L}}(p)$.

LTS of SPNs

The major difference between SPNs and general DPNs is that the communication channels of SPNs are synchronous signal connections rather than asynchronous FIFO buffers. This means that communications of SPNs are *instantaneous*. Therefore the composition of synchronous processes is again a synchronous process. First we define the synchronous composition of labels:

Definition 2.25. Given two channel assignments $l_1 : C_1 \rightarrow \hat{\Sigma}$ and $l_2 : C_2 \rightarrow \hat{\Sigma}$, the *synchronous composition* of l_1 and l_2 is defined by $l = l_1 || l_2 : (C_1 \cup C_2) \rightarrow \hat{\Sigma}$ where $\forall c \in C_1, l(c) = l_1(c)$ and $\forall c \in C_2, l(c) = l_2(c)$, provided that $\forall c \in C_1 \cap C_2, l_1(c) = l_2(c)$.

Definition 2.26. For an SPN P consists of two synchronous processes p_1 and p_2 where each $LTS_{p_i} : \langle I_i, O_i, S_i, s_0^i, L_i, \rightarrow_i \rangle$, the labeled transition system of P is defined by $LTS_{p_1}||LTS_{p_2} := \langle I_P, U_P, O_P, S_P, s_0, L_P, \rightarrow_P \rangle$ where:

- I, U, O are the set of input, local and output channels of P;
- $S_P \subseteq (S_1 \times S_2)$ is the set of local states of P, $s_0 = (s_0^1, s_0^2)$ is the initial state;
- $L_P \subseteq L_1||L_2$ is the set of labels, where each label $l \in L$ is an assignment: $(I_P \cup U_P \cup O_P) \rightarrow \hat{\Sigma}$;
- $\rightarrow_P \subseteq S_P \times L_P \times S_P$ is the set of labeled transition relations;

where $L_1||L_2 := \{l_1||l_2 \mid \exists l_1||l_2, l_1 \in L_1, l_2 \in L_2\}$ and $(s_1, s_2) \xrightarrow[P]{l_1||l_2} (s_1', s_2') \in \rightarrow_P$ iff $s_1 \xrightarrow[p_1]{l_1} s_1' \in \rightarrow_{p_1}$ and $s_2 \xrightarrow[p_2]{l_2} s_2' \in \rightarrow_{p_2}$. Then for two synchronous runs $u_1 = l_1 \cdot l_2 \cdot \cdots \cdot l_n$ and $u_2 = l_1' \cdot l_2' \cdot \cdots \cdot l_n'$, $u_1||u_2 = (l_1||l_1') \cdot \cdots \cdot (l_n||l_n')$. The synchronous composition can be easily extended to sets of synchronous runs.

We denote the SPN P consisting of processes $p_1, \ldots, p_n$ by $p_1|| \ldots ||p_n$. Intuitively, the symbol $||$ indicates the synchronous composition of the processes. The definition of executions and reachable states of an SPN is similar to those of a DPN. Comparing to executions of DPNs, however, all executions of SPNs are effective, fair, live and bounded. All finite executions of SPNs are maximal. This is simply because that communications of SPNs are instantaneous and synchronous, and each process produces a value to each output channel that will be consumed during the same cycle by another process. Notice that the LTS of an SPN does not necessarily encode all possible input sequences, but only those that can be derived from each process. However it is also decidable to check whether a given input sequence can be accepted by an SPN, since this can be boiled down to membership of languages of finite state machines which is decidable [41].

Input-Output Relations of SPNs

Similar to DPNs we can derive the effective and general I/O-relations from synchronous processes and SPNs. It is not difficult to see that for SPN $P := p_1|| \ldots ||p_n$, $\Phi(P) = \Phi(\mathcal{L}(P))$. Also, the I/O relation of a synchronous process is a function. This is easy to see from the determinism of synchronous processes. However, the derived I/O relations of an SPN might not be a function, even each synchronous process is deterministic. A counterexample is shown in Figure 2.9.

As shown in Figure 2.9, both process A and B are deterministic. However their synchronization $P = A||B$ is not. Since by definition we can derive the LTS of P on the left, and given input $x_a = 1$, the output x_d can be either 4 or 5. Indeed if we try to compute the least fixpoint of the corresponding equation system from f_A and f_B given the input $x_a = 1$, we would see that $x_d = f_P(1) = \epsilon$. This shows from another perspective that the synchronous composition is too abstract. Again we see that the Kahn semantics of DPNs may help us in justifying the validity of our definitions.

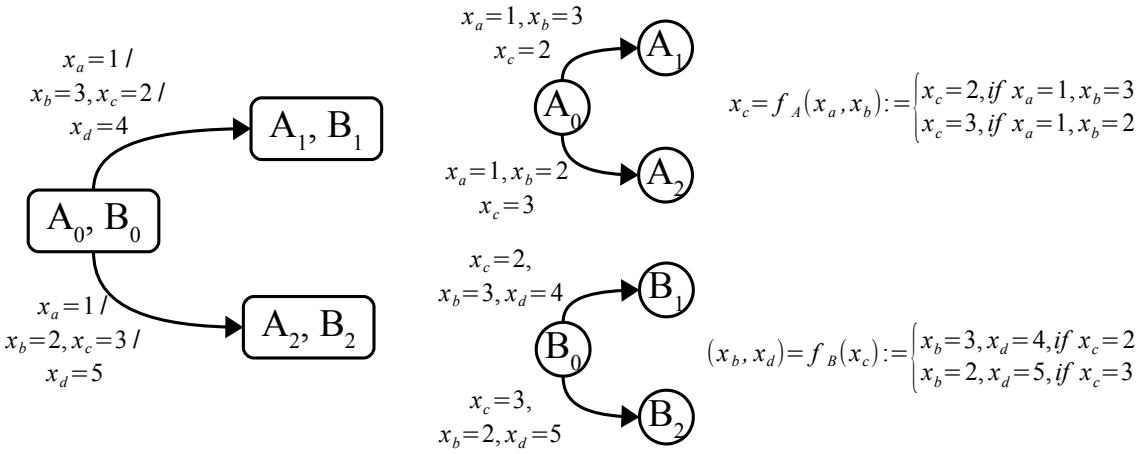


FIGURE 2.9: An SPN $P = A||B$ that doesn't implement a function, although both A and B implement functions.

Full Abstraction of SPNs

Our notion of SPN as the synchronous composition of synchronous processes follows the convention of previous literature [27, 42, 43]. It is important to notice that the denotational semantics derived by this definition is **not** fully abstract in a similar way compared to deadlocks of DPNs. Yet the equation $\Phi(P) = \Phi(\mathcal{L}(P))$ holds. This is because our operational model did not specify the full picture of the synchronous system, i.e. it is somehow lying between fully operational and fully denotational. In [27] this problem is rephrased as:

> ... functional systems not being closed under synchronous, concurrent composition...

In particular, definition 2.23 didn't specify the data dependencies between micro-steps of processes within each clock cycle. Data dependencies turn out to be necessary information to implement an "executable" synchronous process network, and if the dependency relations are flawed, the SPN will fall into a deadlock situation similar to DPNs. Therefore it is important to distinguish the two views of synchronous systems. Macro-steps abstracts from data dependency because of perfect synchrony, and provides a succinct way of synchronous composition. However in order to implement an executable system, it is necessary to adopt the micro-step view and capture the operational details happened within each macro-step, so that causal relations (or data dependencies) between micro-steps are respected. We'll have a closer look at this problem in the next section.

Causality of Synchronous Process Networks

As discussed in the introduction of this section, within one cycle events happened based on their data dependency relations. The execution of these events together form the

execution of one clock cycle. Therefore these events are also called "micro-steps", and each clock cycle is also called a "macro-step". Look back at definition 2.23 and 2.26, we can see that state transitions are all defined in the level of macro-steps. Equivalently speaking, for each synchronous process there is only one micro-step within one macro-step. This view leads to problematic situations as shown in the example of Figure 2.10.

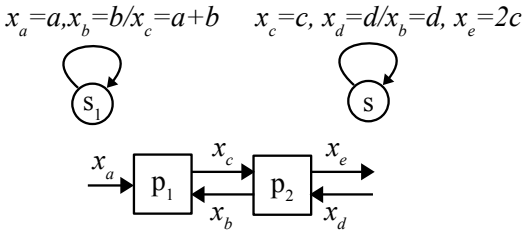


FIGURE 2.10: An SPN $P = p_1||p_2$.

As shown in the figure, we can see that each process of P has one parameterized firing rule: p_1 has the rule $r_1 : \langle (x_a = a, x_b = b) : (x_c := a + b) \rangle$ and p_2 has the rule $r_2 : \langle (x_c = c, x_d = d) : (x_b := d, x_e := 2 \times c) \rangle$. The input channels of P are x_a and x_d. In order to execute r_1, both values of x_a and x_b are required. Since x_b is the result of r_2, during each cycle r_1 have to wait until r_2 finishes. However, in order to fire r_2 both values of x_d and x_c are required, among which x_c is the result of r_1. This turn out to constraint the execution of r_2 after r_1. As a result, none of p_1 and p_2 is able to execute. When we examine the problem by Kahn's least fixpoint semantics, it is clear to see that the functions we can derive from the LTSs would require the domain of inputs all be non-empty sequence of values. Therefore if from the beginning some internal channels are empty (in our example x_c and x_b) and they form cycles, then they would stay empty and hence the output of the corresponding functions (e.g. f_{p_1} and f_{p_2}) would produce only empty strings.

This situation is similar to a deadlock configuration in a DPN. However in a synchronous system, a macro-step of execution is often the composition of several micro-steps. Take r_2 as an example, we may find out that the assignment $x_b := d$ only depends on the input condition $x_d = d$ and $x_e := 2 \times c$ only depends on the condition $x_c = c$. Therefore we may decompose r_2 into two smaller firing rules: $r_2^d : \langle x_d = d : x_b := d \rangle$ and $r_2^e : \langle x_c = c : x_e := 2 \times c \rangle$, and r_2 is a combination of r_2^d and r_2^e. As a result, we may be able to fire r_2^d first and produce a value to x_b, so that r_1 can fire, after which r_2^e can be fired. In this refined micro-step view, there is no causal problem for P. Also, it is easy to see that if an SPN is acyclic, then it has no causal problem. The following remark shortly discussed causality in different synchronous languages.

Causality of Synchronous Languages

The concept of micro-steps is originaly introduced by [44], referring to the atomic actions occurred during one clock cycle (macro-step). Nevertheless micro-steps exsit in essentially all synchronous languages. They are typically introduced by

the structural operational semantics of the languages [14, 20, 30, 45]. Therefore similar causal problem exists, i.e. when some micro-steps' executions are mutually dependent (the dependency relation of micro-steps is cyclic). Different synchronous languages used different techniques to solve causality problems [27]:

- Constructiveness. In Esterel, micro-steps can be syntactically depending on each other, however the cycle must not be a real one, in the sense that it should never happen in the real world. To check whether the cycle is real, one need to solve the equation system and check if the solution is unique, which is of course of high complexity. Instead, Esterel adopted the *constructive logic* and try to construct the assignment of the (unknown) outputs from the (known) inputs in a least fixpoint computation. This can be shown intuitively by a really easy example. Consider the following two micro-step assignments:

$$X := I \wedge Y$$
$$Y := \neg I \wedge X$$

  Apparently the two micro-steps are syntactically depending on each other. However, when $I = \mathsf{true}$, we immediately derived $Y = \mathsf{false}$ from the first equation (by lazy evaluation), then $X = \mathsf{false}$ follows. Similar case happens when $I = \mathsf{false}$. Therefore we can construct the results of X and Y without guessing the other's value.

- Acyclic. In Lustre, causality problem is syntactically avoided by the forbidden of instantaneous cycles in the topology of the dataflow network. Therefore the dataflow network of Lustre remains fully functional. This is a simple solution for implementation, but rejects cases like the one above that could be accepted by Esterel compilers. Yet Lustre allows non-instantaneous cycles in the network. This is done by inserting delayed operators during the cycle.

- Relations. In Signal, it is simply allowed to have either no solution or multiple solutions of the equation system, therefore non-causal behaviors might be included. This might be interested for partial designs or high-level specifications. However in order to synthesize the design into a real implementation, a unique functional solution needs to be found. In [30] Signal's behavioral semantics is encoded in a three-valued sub-algebra, translating the Signal programs into a set of implicit equations. Then a so-called *static clock calculus* can be extracted from this equation system as an abstraction of the original program semantics. Algebraic algorithms are developed to solve the static clock calculus, and it is proven that if there is a unique functional solution of the abstraction, this solution works also for the program and can be implemented without causal problem.

Systematically we can decompose a firing rule into a sequence of three steps: reading one input, updating a local state and writing one output, each corresponds to a micro-step event. Then we can define the LTS of a process where a transition relation only resembles one of the three steps. Such a refined definition is adopted by [7, 46] (for DPNs and synchronous LTSs respectively) and allows direct causal analysis. However we found it over-specified, since in most synchronous MoCs there is not one single order for reading, computing and writing events during one cycle (e.g. all the major synchronous languages we mentioned before except Lustre). However, if we want to keep determinism of the synchronous processes, we would be forced to use one particular total order for the events (followed from the state transition relations). Otherwise, we have to encode all possible orders allowed, not only making the LTS nondeterministic but also much bigger.

Moreover, while "micro-steps" of a so-called μLTS [46] are explicitly modeled, the notion of a "macro-step" is blurred. Since a macro-step can be anything happened between two clock ticks, and such clock ticks are nothing but special micro-steps (in which nothing but a clock-tick can occur), it is no more trivial to examine a synchronous reaction step as a single unit, as well as how reactions are synchronously composed. This has a direct impact on desynchronization: micro-step based criteria instead of macro-step based criteria are adopted, which makes the definition as well as analysis much more complicated.

Based on the above discussion, we still chose LTS as our basic semantical tool. In order to consider causality, we further decorate each state transition relation of a synchronous process by a *causality preorder* [43], which defines a preorder of data dependencies (so that we can model cyclic dependencies). As a synchronous LTS specifies only causality preorders, there can be more than one scheduling of the micro-step events. In this way we still need only one LTS for the specification of macro-step state transitions of the process, but each state carries the scheduling information needed for the micro-steps.

Definition 2.27. For a synchronous process p with I, O its sets of input and output channels, a *causality preorder* is a preorder $(I \cup O, \rightarrow) \subseteq I \times O$. For $(x, y) \in \rightarrow$, we say that y depends on x, denoted by $x \rightarrow y$.

The *supreme* of two causality preorders t_1 and t_2 is the preorder $t_1 \vee t_2$ which is the least extension of both t_1 and t_2. A causality preorder is *cyclic* if there exists a sequence $x_i \rightarrow x_j \rightarrow \cdots \rightarrow x_i$.

We associate for each state transition relation of the LTS of p a causality preorder. The corresponding LTS is thus a *scheduled LTS*. The definitions of LTS of synchronous processes and SPNs can be extended accordingly. Now for the computation of each output, the process needs to read the inputs it depends based on the causal preorder. Once all inputs required are read, the output can be produced. We call such a step of computation of an output a *micro-step*, where as a transition a *macro-step*. Therefore the number of micro-steps within a macro-step equals the number of output variables.

Definition 2.28. A *scheduled synchronous LTS* $LTS_p^s : \langle I, O, S, s_0, \hat{L}, \rightarrow_p \rangle$ of a synchronous process p, is an LTS with the following additional constraints:

- $\hat{L} \subseteq L \times T$ is the set of labels, where L is the set of channel assignments and T the set of scheduling specifications;
- $\rightarrow_p \subseteq S \times \hat{L} \times S$ is the set of labeled transition relations;

For two labels $\hat{l}_1 := (l_1, t_1), \hat{l}_2 := (l_2, t_2)$, the synchronous composition $\hat{l}_1 || \hat{l}_2$ is again a label: $(l_1 || l_2, t_1 \vee t_2)$. The scheduled LTS of $p_1 || p_2$, denoted by $LTS^s_P := LTS^s_{p_1} || LTS^s_{p_2}$, is an LTS of P with the additional constraint where the labeled transition relation $(s_1, s_2) \xrightarrow{\hat{l}_1 || \hat{l}_2} (s'_1, s'_2)$ iff $s_1 \xrightarrow{\hat{l}_1} s'_1 \in \rightarrow_{p_1}$ and $s_2 \xrightarrow{\hat{l}_2} s'_2 \in \rightarrow_{p_2}$.

A scheduled LTS of a synchronous process (or SPN) is *causally correct* iff for each transition relation $s \xrightarrow{(l,t)} s'$, t is acyclic.

Definition 2.29. Given a transition of a synchronous LTS $(s, \hat{l}, s')$ of process p where $\hat{l} = (l, t)$, the *micro-step* of output $y \in O_p$ is $\hat{l}_y := (l_y, t_y)$ where $t_y := \{x \rightarrow y \mid x \rightarrow y \in t\}$ and $l_y := l|_{V_y}$ where $V_y := \{x \mid x \rightarrow y \in t_y\} \cup \{y\}$. Then $\hat{l} = ||_{y \in O_y} \hat{l}_y$. For convenience we also denote $\hat{l}_y$ by $\hat{l}|_y$. For simplicity, we also use $\hat{l}$ to denote the set of micro-steps.

Lemma 2.30. *Given an SPN $P = p_1 || \ldots || p_n$. If P is causally correct then $\Phi(P)$ implements a function.*

Proof. First, it is easy to see that each p_i implements a function $\Phi(p_i)$. If P is acyclic then $\Phi(p_i)$ can be derived by function composition and is a therefore a function. If P contains cycles, then without losing generality we assume the processes $p_{i_1}, p_{i_2}, \ldots, p_{i_k}$ forms the only cycle of P as shown in Figure 2.11.

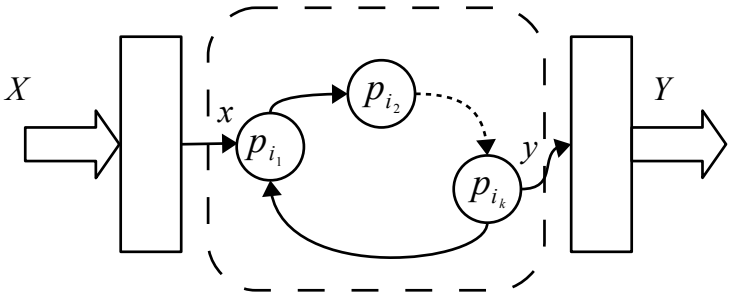


FIGURE 2.11: A cycle in SPN P.

Then assume that there exists an input of X generating an input of x such that different outputs of y can lead to different outputs of Y. However if P is causally correct, then the micro-steps of the cycle of processes can only be executed deterministically, and given the same input of x the output of y is determined uniquely, which contradicts the assumption. □

The causal correctness of an SPN is comparable to the deadlock situation in a DPN. However while checking deadlock is undecidable for DPNs with finite processes (section 2.1.3), it is decidable to check for causal correctness for SPNs with finite synchronous processes. This is easy to see: since for finite synchronous processes, the scheduled LTS of the SPN is the composition of the scheduled LTSs of the processes which is also finite.

Therefore we only need to check if there is a cyclic transition relation, of which there are finitely many.

Proposition 2.31. *Checking if an SPN of finite scheduled synchronous processes is causally correct is decidable.*

Remark. Equivalently, we can add causality preorders for each firing rule. For the example of Figure 2.10, let $\hat{r}_1 : \langle (x_a = a, x_b = b) : (x_c := a + b) : t_1 = \{x_a \to x_c, x_b \to x_c\}\rangle$ and $\hat{r}_2 : \langle (x_c = c, x_d = d) : (x_b := d, x_e := 2 \times c) : t_2 = \{x_d \to x_b, x_c \to x_e\}\rangle$ be the rules after decorated with causality preorders t_1, t_2. Later we will introduce a refined synchronous specification, called the *synchronous guarded actions* [20, 47]. We'll see that we can derive a scheduled LTS from a set of synchronous guarded actions. For dataflow synchronous languages like Signal, *scheduling specifications* are used along with dataflow language-specific operators to derive scheduled LTSs [43].

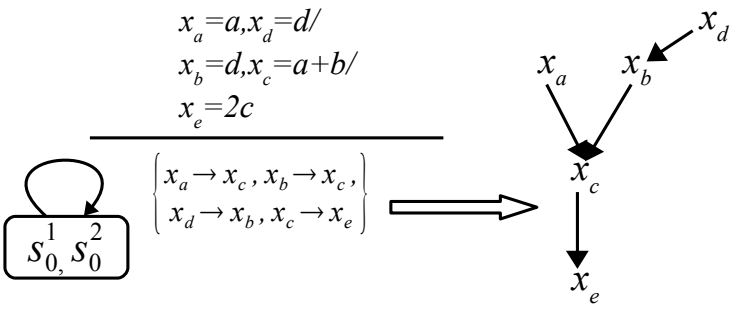


FIGURE 2.12: (a) The scheduled LTS of $P = p_1||p_2$, (b) $t_1 \vee t_2$

Figure 2.12(a) shows LTS^s_P of the example in Figure 2.10. It is not difficult to check that there is only one state in LTS^s_P with one state transition relation (by using parameterized labels). The causality preorder of the transition is $t_P = t_1 \vee t_2$ which is shown in Figure 2.12 (b). It is easy to see that P is causally correct, since t_P is acyclic. There are three micro-steps (for x_b, x_c, x_e respectively), and we can derive the possible schedules for them from t_P: x_b should be produced before x_c for example. This corresponds to possible ways to decompose the firing rules into smaller ones. We will discuss such decomposition in desynchronization. Also we can derive the corresponding equation system that computes the local values x_b, x_c and the output value of x_3 for each macro step from the refined LTSs as follows:

$$\begin{array}{lll} x_c = & (a+b), & \text{if } x_a = a, x_b = b, \\ x_b = & d, & \text{if } x_d = d, \\ x_e = & 2c, & \text{if } x_c = c, \end{array}$$

Therefore by the least fixpoint semantics, it is straightforward to see that given the input sequences of $x_a = a_1 \cdots a_n, x_d = d_1 \cdots d_n$, the output is $x_3 = (2(a_1 + d_1)) \cdots (2(a_n + d_n))$.

Causality is not Compositional

Causality correctness has been a big problem for the modular compilation of synchronous programs [27]. There has been a great effort to develop compilers of synchronous programs that can support modularity [43]. Since in general, we see that the composition of causally correct SPNs might not be causally correct (take the example of Figure 2.10). Therefore it is essential that the compilation preserves the information of scheduling specifications (or its equivalent forms). With the support of scheduling specifications it is then possible to reason about causality correctness of an SPN in a compositional way [43]. However it is not our goal to improve compositional compilation in this thesis. Indeed, what we do is the other way around–we decompose a synchronous system into components, and put the components into an asynchronous environment with the hope that they run consistently regarding the synchronous system.

Since we refined LTSs by causal preorders, the notion of deterministic has to be re-examined. In particular, consider a synchronous LTS shown in Figure 2.13(a) where there are two transitions starting from s.

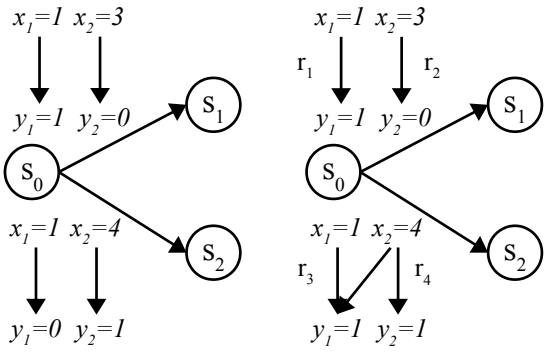


FIGURE 2.13: (a) Two unfaithful transitions; (b) Superfluous microstep r_3 w.r.t. r_1.

By the previous definition the LTS is deterministic, however we notice that starting from the same state s and given the same input $x_1 = 1$, the micro-step of the above transition assigns $y_1 := 1$ but the micro-step of the transition below assigns $y_1 := 0$. The process can choose between the two assignments based on the input value of x_2, however this means the assignment to y_1 should also depend on x_2, which is not reflected by the causal preorder. Consider another situation as shown in Figure 2.13(b). For computing $y_1 = 1$, r_3 requires both $x_1 = 1$ and $x_2 = 4$ while r_1 only requires $x_1 = 1$, indicating that $x_2 = 4$ is not really required for the computation. We call such pairs of transitions (macro-steps) *unfaithful* to each other, as they do not reflect the true dependence relations. For a scheduled LTS to be deterministic when executing micro-steps, we require that at each state all enabled transitions are faithful.

Definition 2.32. Let $l : I_p \to O_p$ be a label of synchronous process p, $\hat{l}_y = (l_y, t_y)$ a micro-step. $\hat{l}_y$ is *compatible* with l iff $l|_{I_t} = l_y|_{I_t}$ where $I_t = \{x \mid x \to y \in t_y\}$. Given two

labels of a scheduled LTS LTS_p^s $\hat{l}_1 = (l_1, t_1)$ and $\hat{l}_2 = (l_2, t_2)$, $\hat{l}_1$ is *faithful* to $\hat{l}_2$ iff for all $y \in O_y, \hat{l}_y = \hat{l}_1|_y = (l_y, t_y)$, $\hat{l}_y$ is compatible with l_2 implies $\hat{l}_y = \hat{l}_2|_y$.

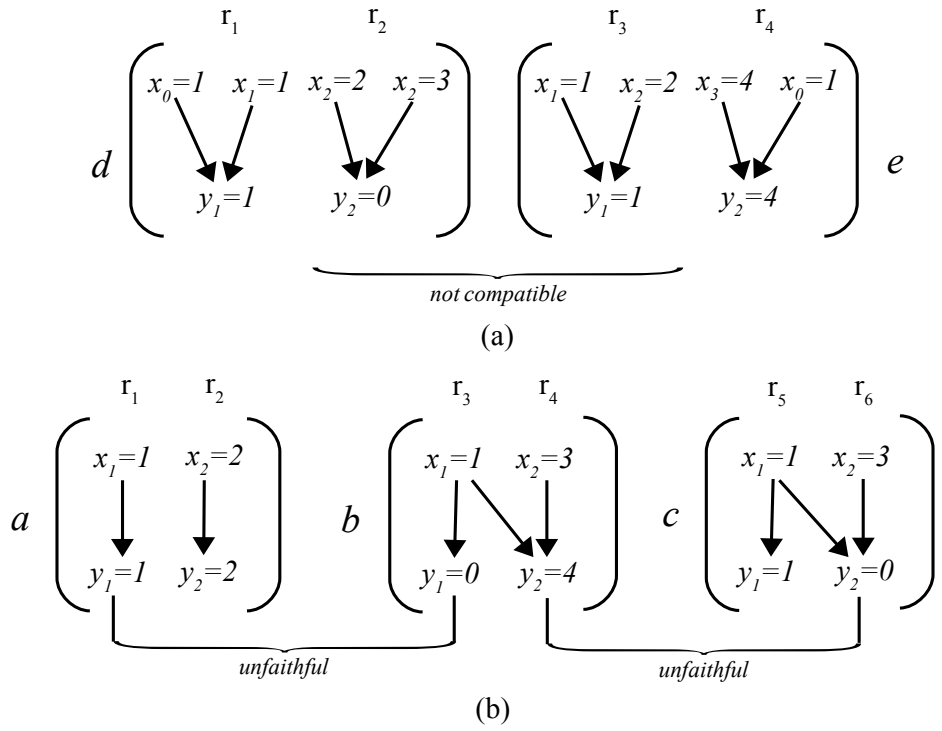


FIGURE 2.14: Unfaithful transition labels.

Figure 2.14 shows some more examples of unfaithful transition labels. In Figure 2.14(a) transition d is unfaithful with transition e because micro-step r_1 is compatible with label of e but is not $l_e|_{y_1}$. Similarly, e is unfaithful with d because r_3 is compatible with label of d but is not $l_d|_{y_1}$.

In Figure 2.14(b) transition a is unfaithful with transition b because the micro-step r_1 is compatible with b but the assignment to y are different. Similarly, transition b is unfaithful with c since given the same assignments of inputs x_1, x_2 the assignments to y_2 are different. Transition a and c are faithful with each other.

Definition 2.33. LTS_p^s is *deterministic* iff for all its reachable state s and transition $(s, \hat{l}, s')$:

- for each output y there is exactly one micro-step computing y in $\hat{l}$,

and for all transitions $(s, \hat{l}, s')$ and $(s, \hat{l}', s'')$:

- If $l|_{I_p} = l'|_{I_p}$, then $\hat{l} = \hat{l}'$ and $s' = s''$;
- $\hat{l}$ and $\hat{l}'$ are mutually faithful.

Intuitively, determinism of a scheduled LTS ensures that during one macro-step, the same inputs can only trigger the same micro-steps and there is no non-deterministic

choice between different micro-steps, therefore each output is deterministically computed. Further, the triggering condition of the same computation of an output should be minimal and consistent. It is also straightforward to see that determinism of a finite scheduled process is decidable, since there are only finitely many states, and there are only finitely many micro-steps need to be checked for each state.

Proposition 2.34. *Checking if a finite synchronous process is deterministic is decidable.*

Note that technically whatever we did here can be applied for a process of DPN as well. However we do not do this because we take a synchronous process as working on a more abstract level where one state transition is a combination of several micro-step events. Instead processes of DPNs work on a more refined level, and in a process we treat each state transition as an *atomic* event which can not be decomposed anymore. Moreover, micro-steps of SPNs are separated by clock ticks, and those happen within one clock cycle are considered to happen all at the same time. In DPNs however, there is no difference between a micro-step and a macro-step, as each transition is atomic and they are executed only following their data dependency. We will see that this type of treatment favors our desynchronization method in the sense that desynchronization can be seen as a decomposition of synchronous behaviors into asynchronous behaviors, where each macro-step is decomposed into its corresponding micro-steps. Furthermore, instead of running in synchronous pulsations, the micro-steps after desynchronization are not synchronized anymore.

Compatibility of Synchronous Processes

In the beginning of section 2.1.3 we introduced the concept of compatibility. Intuitively, two processes are compatible if at any reachable configuration, for the outputs produced by either process, the other is able to react and consume them. Since reachability is undecidable for DPNs, compatibility of processes are also undecidable (theorem 2.11). For synchronous processes, compatibility can be boiled down to checking the existence of synchronous composition of two labels regarding a reachable state, therefore is decidable. We formally define the compabibility of synchronous processes as follows.

Definition 2.35. For two synchronous processes p_1, p_2 and their synchronous LTSs LTS_{p_1}, LTS_{p_2}, let $P = p_1||p_2$ be their synchronous composition. p_1 and p_2 are *compatible* if and only if $\forall w_I \in \mathcal{L}(p_1)|_{I_P}||\mathcal{L}(p_2)_{I_P}, \exists w \in \mathcal{L}(p_1||p_2)$ such that $w_I = w|_I$.

Intuitively $\mathcal{L}(p_1)|_{I_P}||\mathcal{L}(p_2)_{I_P}$ defines the allowed sequences of inputs for P. Therefore, for any allowed input sequence w_I, p_1 and p_2 are compatible if and only if there exists a label sequence in $\mathcal{L}(p_1||p_2)$ whose input sequence is w_I. In the functional point of view, the function f_P that P implemented is defined for all $w_I \in \mathcal{L}(p_1)|_{I_P}||\mathcal{L}(p_2)_{I_P}$.

Proposition 2.36. *For finite synchronous processes, compatibility is decidable.*

The proof of proposition 2.36 is straightforward. Since each LTS_{p_i} is finite, $\mathcal{L}(p_1)|_{I_P}||\mathcal{L}(p_2)_{I_P}$ and $\mathcal{L}(p_1||p_2)$ can also be represented by two finite LTSs. Then $\mathcal{L}(p_1||p_2)|_I$ also corresponds to a finite LTS. Then to check compatibility is to check the language equivalence of $\mathcal{L}(p_1||p_2)|_I$ and $\mathcal{L}(p_1)|_{I_P}||\mathcal{L}(p_2)_{I_P}$ of which each is a regular language.

It is important to see that compatibility is a *global property*. This means for $P = p_1||p_2||p_3$, even if each pair of p_i, p_j is compatible, p_1, p_2 and p_3 together can still be incompatible. A counterexample is shown in the following Figure 2.15 for an SPN $P = A||B||C$. Although any two processes of A, B and C are compatible, the three together are not compatible since for the only input $\{x_a = 1, x_b = 2, x_c = 3\}$ there is no run of P.

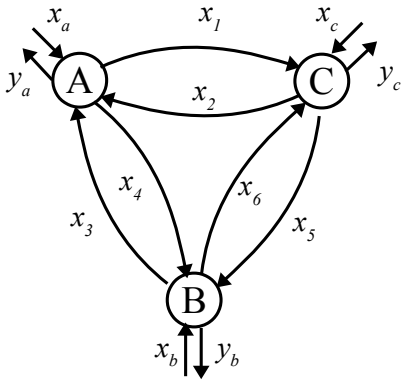


FIGURE 2.15: An SPN $P = A||B||C$.

Figure 2.16 shows the situation for the SPN $P = A||B||C$. It is easy to check that A, B, C are all mutually compatible, and $\mathcal{L}(A)|_{I_P}||\mathcal{L}(B)|_{I_P}||\mathcal{L}(C)|_{I_P} = \{(x_a = 1)||(x_b = 2)||(x_c = 1)\}$. There is no single run taking the input $x_a = 1, x_b = 2, x_c = 3$ in $\mathcal{L}(A||B||C)$. More interestingly, even if p_1 and p_2 are not compatible, p_1, p_2 and p_3 might be compatible! An example is shown in Figure 2.17. In Figure 2.17(a), P consists of three synchronous processes where "+" sums up two input values from x'_a and x'_b and outputs the value to x_c only when both inputs are present. If both inputs are absent it produces a $\boxdot$. Other input patterns are undefined. Cp copies one value from its input channel to its output channel for each cycle, no matter if it's a present data value or a $\boxdot$. As the adder "+" requires both operands to be present or absent, whenever during one cycle x_a is present and x_b is absent the whole SPN P fails, therefore processes of P are not compatible. However this is no problem for P' where an additional process $Cp2$ duplicates its input value to its two output channels each cycle. Hence both values of x_a and x_b are either present or absent during the same cycle. Therefore for any input of $Cp2$ there is a corresponding output at x_c and the processes are compatible.

In previous works a comparable property called *isochrony* has been proposed as one of the criterion of correct desynchronization [42]. We will compare compatibility and isochrony in Chapter 3 in detail. Compatibility is closely related to the functional characteristic of a process. In particular, we say a process (or SPN) is *completely specified* if for any input sequence $w \in (\hat{\Sigma}^*)^n$ there is a corresponding run. Otherwise we say that the process (or SPN) is *partially specified*. Apparently those processes that implement total functions over the input domains are completely specified.

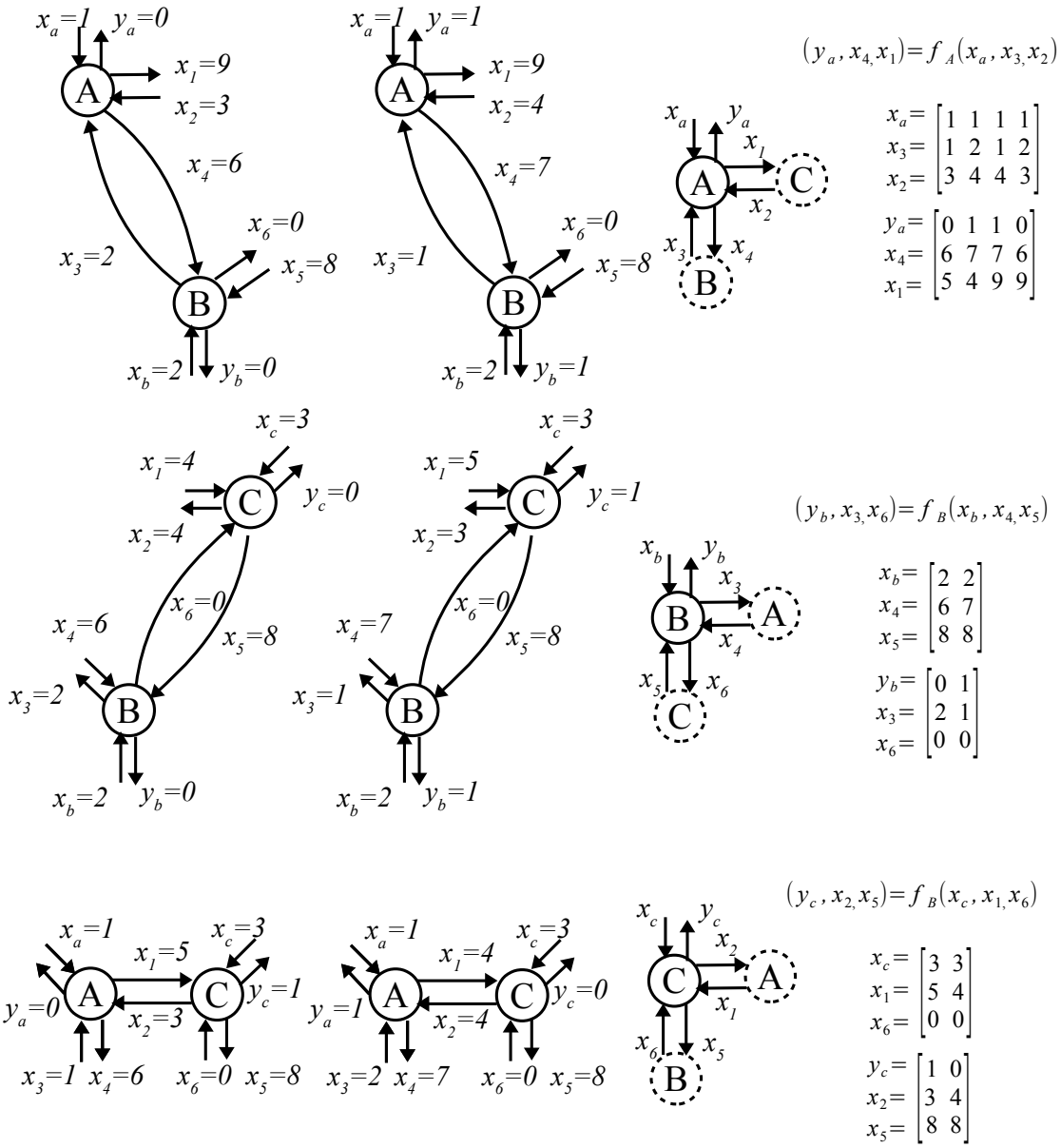


FIGURE 2.16: (a) Runs of A, B, C (b) Runs of $A||B, B||C, A||C$.

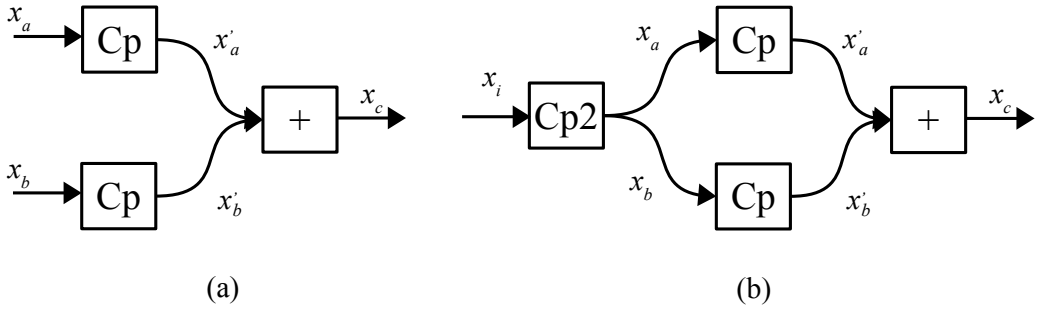


FIGURE 2.17: (a) An $P = cp||cp||+$ with processes that are not compatible, (b) An SPN $P' = cp||cp||cp2||+$ with processes that are compatible.

Completing partially defined Processes

It is easy to complete a partially defined process. Just like completing a partially defined finite state machine, we only need to add some default behavior once an input assignment has no enabled firing rules. For example, many hardware discription languages including VHDL [48] adopted such simple solution, as well as Esterel and Quartz due to the strong influence from hardware description languages. Also, it is trivial that for completely specified processes, their composition is always completely specified.
However, complete specification is not always desired. This is not simply because of the partial specification desired during design phase. As we will see in the following section, multi-rated or multi-clocked synchronous systems are naturally partially specified. This applies for Lustre and Signal, and in general all synchronous dataflow languages.

2.2.2 Multi-rated SPNs

For a synchronous process introduced in the last section, during one cycle there is a data value read from each input channel and produced to each output channel. Such synchronous processes are called *single-rated* in the sense that all channels are processed in the same rate. We relax this constraint in this section by introducing *multi-rated* synchronous processes. As the name indicates, for a multi-rated synchronous process, during one cycle some input channel might transmit nothing, i.e. the input value of that channel is *absent*. Similarly, an output value might also be absent at some output channel. We encode the absent of data value by a special symbol "$\boxdot$". In this way, we can transform a multi-rated synchronous process to a single-rated synchronous process, i.e. each process still reads (writes) one value from (to) each input (output) channel and the only difference is that now the data domain is extended to include $\boxdot$. The synchronous LTS of a multi-rated synchronous process can then be defined correspondingly.

Definition 2.37. A labeled transition system $LTS_p : \langle I, O, S, s_0, L, \rightarrow_p \rangle$ of a multi-rated synchronous process p is an LTS with the following additional constraints:

- L is the set of *labels*, where each label $l \in L$ is an assignment: $(I \cup O) \rightarrow (\hat{\Sigma} \cup \{\boxdot\})$;

We denote $\hat{\Sigma} \cup \{\boxdot\}$ by $\check{\Sigma}$. The multi-rated versions of definitions of LTS for SPNs and scheduled LTSs can be derived similarly. The notion of absent ($\boxdot$ used by this thesis, $\perp$ used by synchronous dataflow languages) is traditionally introduced by synchronous dataflow languages. For dataflow applications, it is usually the case that different dataflow processes operate at different speeds (e.g. because of down sampling). Therefore when trying to synchronize data sequences of different processes together, one needs to model the case that at one particular cycle, one channel has a valid data value while some other channel has not. This indicates that different processes could be loosely coupled, i.e. they do not communicate to synchronize their computations all the time. Instead, they only synchronize when necessary–when one process needs the result from some other process.

Clocks of Synchronous Languages

- Imperative synchronous languages: Single-rated synchronous languages like Esterel and Quartz often invoke the synchronous circuits view of the system. This view results in tightly coupled synchronous processes and is not efficient for distributed implementations. It is important to notice that although a signal can be present or absent in Esterel, the MoC of Esterel is still single-clocked in the sense the absent signals maintain their values from the last cycle and can be read by others. Therefore absence simply means that the signal is not updated.

  Multi-rated synchrony is introduced to Esterel and Quartz via multi-clocks / clock refinement in later works [49–51], however are some how treated in an ad-hoc style (with the preference to hardware synthesis) and is not directly comparable to the $\perp$ of synchronous dataflow languages. In particular, the notion of multi-clock signals introduced in [49] still invokes the circuit view where each module is controlled by a single clock, i.e. all channels of the module should be processed with the same clock where in contrast our multi-clocked processes is able to deploy different clocks channel-wise. The clock-refinement of [50] introduces a hierarchical view of macro-steps, where a macro-step can be refined by several smaller sub-steps in which each can be treated again as a macro-step. This refined view of macro-steps can be helpful for hierarchical design but complicates the operational semantics of synchronous programs and related analysis. Finally the multi-clocked extension of Esterel in [51] introduced the override of clocks. But at a particular control position only one clock is visible, therefore again constraints the signals to process by a unique clock. It further allows clocks to be arrived as events from outside in the sense of discrete-event models of computation [52], therefore distorted the synchronous model of computation.

- Synchronous dataflow languages: because of up-sampling and down-sampling sequence operators, synchronous dataflow languages are naturally multi-rated. One immediate consequence is that processes are naturally partially specified. Consider the following equation system:

$$\begin{cases} a = x_1 \ \mathit{when} \ i \\ b = x_2 \ \mathit{when} \ j \\ y = a + b \end{cases}$$

  Because $+$ is defined only when both a and b are present, the SPN has no reaction when $i = \mathsf{true}$ and $j = \mathsf{false}$, i.e. when only one of a and b is present. Moreover, we can not simply fix the problem by providing default values to absent signals, since this changes the semantics of the network– unlike ESTEREL, a value of a signal in dataflow languages are treated as a

token carrying some useful message. Presence of some signal means that its value is produced by some process and consumed by some other process. Instead, in Esterel presence means an update, therefore even some signal is not updated, its value can still be read by others.

An important difference between clocks of Lustre and Signal is that clocks of Lustre are well-aligned and can be inferred functionally (clocks of outputs are defined by clocks of inputs). In particular, all processes (nodes) share the same *basic clock* [15], and any clock can only be a down-sampling of this basic clock. Therefore, although some nodes might react on the absence of some signal (e.g. operator current), it ultimately depends on the presence of some signal with a finer clock. Signal instead has no such constraint. As introduced before, Signal defines the relations of sequences of inputs and outputs, therefore clocks of outputs might determine clocks of inputs. Further, Signal adopted the polychronous modeling of time, and clocks of different signals might not be comparable at all (neither up-nor down-sampling). Some actions might depend on the absence of some signal, and unlike Lustre this can not be reduced to some presence condition of a finer clocked signal / basic clock.

The synchronous LTS introduced in this thesis adopted the well-aligned multi-clocked view of Lustre, but has no constraint on the existence of a basic clock. We also found the polychronous view of Signal inconvenient when it comes to the synchronous composition of LTSs, as non-comparable clocked signals can only be modeled by interleaving states in the composition, which leads to a potential state explosion of the transition system. Moreover, such a view has more of the taste of asynchronous systems. Instead we'd like to maintain a purely synchronous semantics for our synchronous models.

Notion of Absence: "⊡" versus "⊥"

We use "⊡" to denote absence while other synchronous dataflow languages like Signal use "⊥" instead. We chose ⊡ because in classical Scott-domain ⊥ means "unknown". Indeed a variable can be present but its value is unknown, which has a totally different meaning than that the variable is absent. ⊥ is also used to denote unknown in constructive reasoning of Esterel and Quartz. Therefore for clarification, we'd like to pay respect to the classical usage of ⊥ and reserve it to "unknown", and use ⊡ for absent instead.

Clock-Consistency of SPNs

The concept of *clock-consistency* is originally introduced by synchronous dataflow programming languages [27, 37]. Synchronous dataflow programs are comparible to BDF graphs [12], however in BDF graphs actors are executed asynchronously, while in synchronous dataflow programs actors are like synchronous processes and are synchronously

composed, therefore run synchronously together. Associated with each actor, a corresponding clock-operator is defined to specify the clock relations of the inputs and outputs of the actor–how input values are paired together and associated for the computation of an output value. As a simplest example, for the pluss operation $c = a + b$, c is present (i.e., the clock of c is true) if and only if when both a and b are present. Naturally, something like $3 + \boxdot$ is undefined and illegal for $+$. Such cases are called *clock-inconsistent*, indicating that there is a miss-match between some input channels so that the operation is undefined.

In a functional point of view, we may treat each synchronous process p as a function f_p mapping sequence of input values to sequence of output values (as we did in section 2.2.1). Then in domain $\check{\Sigma}$ of multi-rated synchronous processes, f_+ is a partial function, since it is undefined for cases like $3 + \boxdot$. Therefore if we treat $\boxdot$ merely as a data value, clock-consistency can be generalized to compatibility. Indeed, if we define a plus operator $+^{\mathbb{N}}$ that is defined only for natural numbers, something like $3 + (-1)$ or $1.5 + 2.3$ are then undefined as well, and such cases are troublesome if there are processes producing negative numbers or fractional numbers to the inputs of $+^{\mathbb{N}}$. Instead, if there is an updated version f'_+ that is defined for all possible inputs, then it is easy to see that f'_+ is clock-consistent with all other processes. Therefore it is natural to deduce that if p_1 and p_2 are compatible, then $p_1||p_2$ is clock-consistent. Without bothering with the formal definition of clock-consistency, we would rather define it as a synonym of compatibility.

Definition 2.38. An SPN $P = p_1||\ldots||p_n$ is *clock-consistent* if and only if $p_1, \ldots, p_n$ are compatible.

Checking Clock Consistency of Synchronous Programs

Synchronous programs in Esterel and Quartz are single-clocked and completely specified, therefore by definition they are always clock-consistent. Lustre encodes clocks of signals as a data type [14, 15] and checking clock-consistency are reduced to checking type-safety in classical functional programming languages [53].
Although using the polychronous modeling of time, clock-consistency is also a problem for Signal programs. In particular, if two Signal processes share the same signal connection, then they must agree on the communication (called synchronization in [30]) of this connection. As introduced in section 2.2.1, Signal abstracts the combinational behavior of programs in static clock calculus. Clock-consistency is verified by solving the static clock calculus.

2.3 Summary

In this chapter, we studied dataflow process networks (DPNs) from two perspectives–their operational and denotational semantics. Operational semantics captures the details of how a DPN is executed regarding each computation step, while denotational semantics shows the bigger picture regarding the relations of input sequences and output sequences.

With the help of these semantical tools, we are able to derive important characteristics of the models of computations of our interest.

Although powerful in expressiveness and straightforward modeling of distributed systems, most of the analysis problems are undecidable for DPNs. Synchronous process networks (SPNs) as a special sub-class of DPNs abstracts the asynchronous composition to synchronous composition and insists on perfect synchrony for each macro-step. This makes finite composition of finite synchronous processes again a finite system, therefore making the same undecidable analysis problems for DPNs decidable for SPNs. Among those a particular problem of interest is causal correctness of SPNs, which can be seen as the corresponding problem of deadlock in DPNs. It is important to notice that the operational semantics of SPNs using LTSs is not fully abstract, since causal relations of micro-steps are not captured (because of the abstraction of perfect synchrony). This is fixed by adding causal-preorders for state transition relations. Because of the abstraction, determinisity of a synchronous LTS is under-specified comparing to determinisity of a synchronous scheduled LTS, where as determinisity of scheduled LTSs further constraints on the determnistic execution of micro-steps.

Finally, we introduced multi-rated SPNs as a generalization of SPNs–where a special symbol $\boxdot$ is used denoting the absence of a signal. We further discussed clock-consistency of multi-rated SPNs, and shows that when treating $\boxdot$ as just another data-value, clock-consistency is just a case of special interests of compatibility for single-rated SPNs.

Chapter 3

Desynchronization of Synchronous Systems

In the introduction of the thesis we have seen that due to the synchronous semantics, components of a system are tightly coupled. In particular, the values of shared variables are synchronized during each cycle regardless if the value is needed for computation or not. As a result, directly implementing such a synchronous system into a message-based distributed system might suffer from great communication penalty (especially when components in the original system are not tightly coupled). In this chapter we'll develop a theory helping us correctly desynchronizing a synchronous system into a corresponding distributed system (correct in the sense that the behavior of the synchronous system is *operationally* preserved). Moreover, besides the guarantee of correctness the desynchronized system is decoupled so that unnecessary communications are saved from transmission.

As we've studied the models of computations in the last chapter, now we can examine the problem of desynchronization again in more details and discover some more insight in solving the problem. Firstly, it is clear now given that SPN and DPN as MoCs for synchronous and distributed systems respectively, our goal is to desynchronize an SPN into a DPN. In chapter 2 we introduced two subclasses of SPNs: single-rated and multi-rated. Technically speaking, single-rated SPNs can be seen as a special case of multi-rated SPNs where the alphabet does not include $\boxdot$. Apparently, $\boxdot$ plays a critical role in synchronizing reactions of processes, therefore for decoupling purpose they should be removed from communication. For single-rated SPNs we may simply forget them. In the following we first develop a correct desynchronization procedure for this special case, and then try to generalize it for multi-rated SPNs. It may seem to be easy in the first place for desynchronizing single-rated SPNs, since we don't remove anything from the synchronous communications, i.e. the data communicated are totally preserved after desynchronization, which is essential for a correct desynchronization. However there are still other problems we need to take care of.

Since an SPN is a synchronous composition of synchronous processes, the natural idea is to replace the synchronous composition by asynchronous composition, i.e. technically

replacing the synchronous signal connections by asynchronous unbounded FIFO buffers. Also we need to desynchronize a synchronous process into an asynchronous process. Remember that a transition of the LTS of a synchronous process is a composition of several micro-steps, while a transition of the LTS of an asynchronous process is atomic. Therefore a direct mapping from synchronous transitions to asynchronous transitions would ignore the internal dependency relations of the micro-steps and may cause causality problems. Hence a correct desynchronization should take care of the causality problem, so that the derived DPN is free from deadlocks.

Once we developed a correct desynchronization for single-rated SPNs, we expect to make minimal adaptions so that it works for the multi-rated case. For multi-rated SPNs, the only additional thing we would do is to remove the $\boxdot$ from transmission as well as computation. However in some cases this would change the process to behave nondeterministically, therefore changing the behavior regarding the original SPN. As a result, we need to find out the cases when $\boxdot$ can be safely removed. This turned out to be one of the biggest problem for multi-rated SPNs.

Finally, notice that a single-rated SPN is naturally tightly coupled. Without further treatment, the desynchronization would lead to a tightly coupled DPN. However this might be simply caused by the synchronous semantics rather than the natural functional behavior of the system. For example, programs of Esterel and Quartz are completely specified and as a result many default behaviors may be generated accordingly even if they are not required by any computation. Thus, it might be possible to optimize the single-rated SPN so that these redundant computations can be removed. In the next chapter, we'll introduce techniques to synthesize $\boxdot$ to single-rated SPNs, so that we can derive a multi-rated SPN and exploit the multi-rated nature to derive a loosely coupled DPN.

Through out this chapter, we will examine our theory of desynchronization from both the perspective of denotational semantics as well as operational semantics. Denotational semantics provides us clean and simple ways stating the criteria. However it is the arguments respecting operational semantics that guarantee the implementability of the desynchronized DPN.

3.1 Desynchronization of Single-rated SPNs

As discussed in the introduction, desynchronization of an SPN consists of two parts: (1) replacing the synchronous composition by asynchronous composition; (2) desynchronize the synchronous process to an (asynchronous) process. For multi-rated SPNs, desynchronization also involves removing $\boxdot$ from communication as well as computation which is apparently unnecessary for single-rated SPNs. Therefore the desynchronization of single-rated SPNs can be seen as a special case for multi-rated SPNs, and in this section we first develop a desynchronization theory for single-rated SPNs.

The idea of desynchronizing single-rated SPNs is simple: we translate each synchronous process into an asynchronous process, then compose the processes asynchronously to construct the corresponding DPN. If our desynchronization is correct, then the synchronous runs of a synchronous process should correspond to the runs of the derived process. Then it is not difficult to see that the synchronous runs of the SPN correspond to the runs of the derived DPN. Therefore the *crucial* part of our procedure is to correctly desynchronize a synchronous process. From the denotational semantics point of view, let $\delta(p)$ be the desynchronized process of synchronous process p, then we expect that $\mathcal{L}(p) = \mathcal{L}(\delta(p))$ and this should imply $\mathcal{L}(p_1||\ldots||p_n) = \mathcal{L}(p_1)||\ldots||\mathcal{L}(p_n) = \mathcal{L}(\delta(p_1))|||\ldots|||\mathcal{L}(\delta(p_n))$. In order to make sure that the synchronous runs of the SPN can be preserved operationally, we further need to guarantee that $\Phi(p_1||\ldots||p_n) = \Phi(\delta(p_1)|||\ldots|||\delta(p_n))$, since $\Phi(p_1||\ldots||p_n) = \Phi(\mathcal{L}(p_1||\ldots||p_n))$, by lemma 2.16 in section 2.1.4, we know that the equality holds only when $\delta(p_1)|||\ldots|||\delta(p_n)$ is deadlock-free. In the following, we'll develop a desynchronization of synchronous processes such that as long as the original SPN is causally correct, the desynchronization makes sure that the derived DPN is deadlock-free. This together with the equivalence of $\mathcal{L}(p) = \mathcal{L}(\delta(p))$ make sure that the synchronous runs are operationally preserved after the desynchronization.

3.1.1 Desynchronization of Synchronous Processes

Despite the high similarity between definition 2.1 of processes and definition 2.23 of synchronous processes, we can not map the transition of a synchronous process directly to a transition of a process. This is because such direct mapping omits the data-dependencies of micro-steps carried by the causal preorders. As an example, remember SPN $p_1||p_2$ of Figure 2.10 in section 2.2.1. For convenience we show it again in Figure 3.1.

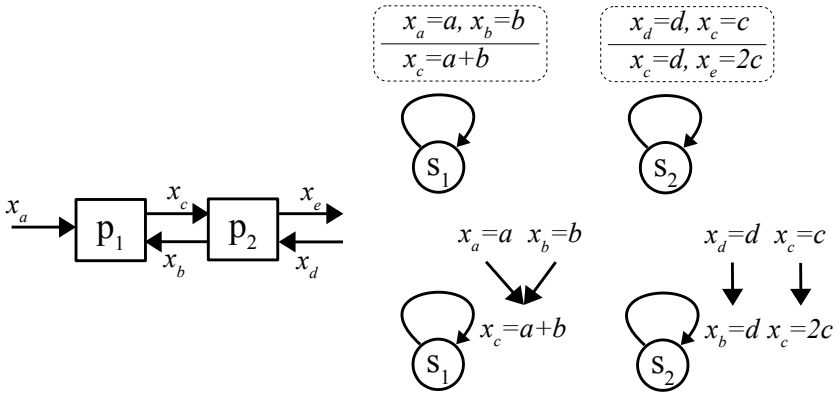


FIGURE 3.1: Causal problem in desynchronization.

If we map the transitions of the synchronous processes directly to transitions of a process, we can equally derive two parameterized firing rules:

$$r_{p_1} : \langle (x_a = a, x_b = b) : (s_0^1, s_0^1) : (x_c = a + b) \rangle$$
$$r_{p_2} : \langle (x_c = c, x_d = d) : (s_0^2, s_0^2) : (x_b = d, x_e = 2 \cdot c) \rangle$$

It is easy to see that the initial local state of the DPN $p_1|||p_2$ is (s_0^1, s_0^2), and all reachable configurations of $p_1|||p_2$ with this local state deadlocks. However as we discussed in section 2.2.1, with the causal preorders of the two transitions, the SPN $p_1||p_2$ is causally correct and its set of behaviors is not empty. Intuitively, it is also easy to see that if we replace the synchronous signal connections by asynchronous FIFO buffers, if we schedule the micro-steps following the causal orders, the DPN has no deadlock. For example, $r_2 \to r_1 \to r_3$ is a valid schedule. Since this schedule even works without buffering in the SPN, it should work with FIFO buffers of any size in the DPN.

From the example it is clear now that it is rather the micro-steps that should be mapped to transitions of a process during desynchronization. In order to capture the computation of micro-steps, we need to add intermediate states between the successive local states of a synchronous transition relation and record the progress of the executions of the micro-steps.

Definition 3.1. Let p be a single-rated synchronous process and $LTS_p : \langle I, O, S, s_0, \hat{L}, \to_p \rangle$ is the scheduled LTS of p. Then the *desynchronized process* of p is a process $\delta(p)$ and $LTS_{\delta(p)}$ is $\langle I, O, S_\delta, s_0^\delta, L_\delta, \to_p^\delta \rangle$ with the following definitions:

- $S_\delta := \{(s, \iota) \mid s \in S, (s, \hat{l}, s') \in \to_p, \iota \subseteq \hat{l}\}$,
- $s_0^\delta := (s_0, \emptyset)$
- L_δ is the set of labels where $l \in L$ is a partial assignment from $I \cup O$ to Σ,
- $\to_p^\delta$ is defined by the following induction rules:

$$\frac{s \xrightarrow{\hat{l}} s', \hat{\mu} \subseteq \hat{l}, \hat{l}_y \in (\hat{l} \setminus \hat{\mu})}{(s, \hat{\mu}) \xrightarrow{l'_y} (s, \iota \cup \hat{l}_y)} \quad \text{data flow transition}$$

  where $\forall x \in dom(l_y) \cap dom(\mu), l'_y(x) = \epsilon$ and $\forall x \in dom(l_y) \setminus dom(\mu), l'_y(x) = l_y(x)$.

$$\frac{s \xrightarrow{\hat{l}} s', \hat{\mu} = \hat{l}}{(s, \hat{\mu}) \xrightarrow{c} (s', \emptyset)} \quad \text{control flow transition}$$

  where $c : I \cup O \to \Sigma$ is the assignment $\forall x \in I \cup O, c(x) = \epsilon$.

Intuitively, $LTS_\delta(p)$ decomposes each transition (macro-step) of LTS_p into a set of data flow transitions and a control flow transition. Each data flow transition corresponds to a micro-step, and they are executed according to the data-dependency of the causal preorders. Each local state of $LTS_\delta(p)$ not only records the local state of the synchronous process p but also remembers the inputs that are read already, as well as the executed micro-steps. After all micro-steps of a macro-step are executed, the control flow transition is performed corresponding to the update the local state of the synchronous process. An example of desynchronized transitions is shown in Figure 3.2.

As shown in Figure 3.2, while in the synchronous case there is only one path leading from s_0 to s_1, there are two paths leading from $(s_0, \emptyset)$ to $(s_1, \emptyset)$ (more importantly, to $(s_0, \{r_1, r_2\})$). This is because of the different order of execution of $\{r_1, r_2\}$. These

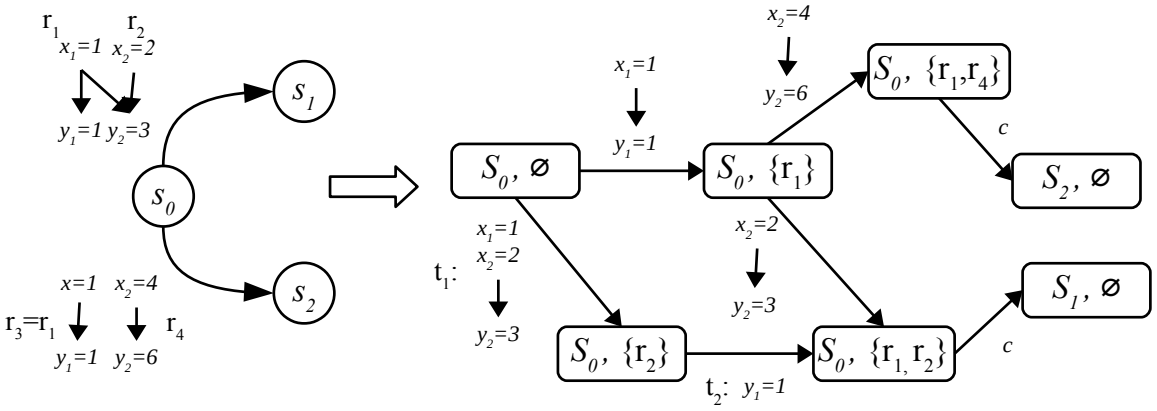


FIGURE 3.2: Desynchronization of macro-step transitions.

multiple paths reflect the fact that the micro-steps happened during one macro-step are concurrent with each other, therefore any interleaving of these data transitions is possible. Note that nevertheless the transitions of the two paths are not different permutations of the labels of r_1, r_2. In particular, after $x_1 = 1, x_2 = 2$ are both read by t_1 (a data flow transition corresponding to r_2), $x_1 = 1$ is not read by t_2 although it is required by r_1. Such omission of reading is important, since during each macro-step each input channel is read only *once*. However the same value can then be used to trigger more than one micro-step until the end of the macro-step. In the sense of data consumption, there is only one value consumed in each input channel during one macro-step. In order to preserve the consumption of input values, the reading of the same input channel of a micro-step should be saved once the value has been read by some other micro-step. Practically, after the process copied the value from input port to its local memory, other micro-steps don't have to fetch it again from the input channel. More importantly, they also can not fetch it anymore, as it is already consumed.

The following lemma justifies our definition of desynchronized process in the sense that denotationally, f_p and $f_{\delta(p)}$ implement the same function regarding $dom(f_p)$.

Lemma 3.2. *For a deterministic synchronous process p, $\forall x \in dom(\Phi(p)), \Phi(p)(x) = \Phi(\delta(p))(x)$.*

Proof. (sketch) First by section 2.2.1 $\Phi(p)$ is a function. Moreover, by determinicity of p for each input sequence x there is a unique execution π such that $(\rho(\pi))|_I = x$ correspondingly. Therefore we only need to prove that there exists $\pi \in \Gamma(p)$ if and only if there exists $\pi' \in \Gamma(\delta(p))$ such that $\rho(\pi) = \rho(\pi')$. This can be proven by the following argument.

By definition 3.1, there is a transition of LTS_p: $s \xrightarrow{l} s'$ if and only if there is a sequence of transitions in $LTS_{\delta(p)}$: $\pi = (s, \emptyset) \xrightarrow{l_1} (s, \{r_1\}) \xrightarrow{l_2} (s, \{r_1, r_2\}) \rightarrow \ldots \xrightarrow{c} (s', \emptyset)$. Further more, although π might not be unique, it is not difficult to see that $l = l_1 \cdot l_2 \cdot \cdots \cdot c$ always hold. □

It is obvious that for every synchronous execution $\pi \in \Gamma(p)$, there exists at least one execution in the desynchronized LTS in $\Gamma(\delta(p))$. We denote this set of executions by $\delta(\pi)$. For convenience, we also call each execution $\pi' \in \delta(\pi)$ a *macro-step execution* as π resembles a macro-step level execution in LTS_p.

3.1.2 Desynchronization of Synchronous Communication (Single-Rated)

Given the desynchronization of a process, it is natural to extend it to an SPN. In particular, for $P = p_1||p_2||\dots||p_n$, the desynchronization of P can be derived as: $\delta(P) = \delta(p_1)|||\delta(p_2)|||\dots|||\delta(p_n)$. As we discussed in the beginning of this section, by lemma 3.2 and the definition of the asynchronous composition $|||$ we would like to derive a property similar to: $\mathcal{L}(p_1||\dots||p_n) = \mathcal{L}(p_1)||\dots||\mathcal{L}(p_n) = \mathcal{L}(\delta(p_1))|||\dots|||\mathcal{L}(\delta(p_n))$. Note that the exact property is not this equation, as $\delta(P)$ works in the level of micro-steps hence definitely has more runs. But these runs should be prefixes of corresponding runs of macro-step executions, therefore a more reasonable discussion should be restricted to inputs in $dom(\Phi(p))$. However, even with this restriction, the property wouldn't guarantee that $\Phi(P) = \Phi(\delta(P))$, which is shown by the counterexample of Figure 3.1. In this sub-section we prove that the desynchronization we introduced in definition 3.1 guarantees that $\Phi(P) = \Phi(\delta(P))$ holds with the restricted domain as well, provided that P is causally correct and each p_i is deterministic.

In the following we first prove some properties that are of use to the prove of our goal.

Lemma 3.3. *For SPN $P = p_1||\dots||p_n$, if each p_i is deterministic then $\forall x \in dom(\Phi(\mathcal{L}(P)))$, $\Phi(\mathcal{L}(\delta(P)))(x) = \Phi(\mathcal{L}(P))(x)$.*

Proof. Notice that $dom(\Phi(\mathcal{L}(P))) \subseteq dom(\Phi(\mathcal{L}(\delta(P))))$. The lemma is easily proven by lemma 3.2 and the definition of $|||$ as well as $||$. □

Theorem 3.4. *For an SPN $P = p_1||\dots||p_n$, if P is causally correct and each p_i is deterministic, then $\delta(P)$ is deadlock free.*

Proof. Without losing generality, we may assume that P is strongly connected, since a deadlock can only happen in a strongly connected subset of processes of a DPN. If Then we prove that this configuration leads to or comes from a state where each process resides in $(s_i, \hat{\mu}_i)$ such that $(s_1, s_2, \dots, s_n)$ is reachable in LTS_P. This can be concluded from the following two statements:

(1) For each process $\delta(p_i)$, given the same input sequence x there is a unique sequence of local states $(s_i^0, \emptyset), (s_i^1, \emptyset), \dots$ visited, and the sequence $s_i^0, s_i^1, \dots$ is exactly the sequence of local states visited by LTS_{p_i}.

(2) Given an input sequence of $\Phi(P)$, for each reachable local state $((s_1^{j_1}, \hat{\mu}_{j_1}), (s_2^{j_1}, \hat{\mu}_{j_2}), \dots, (s_m^{j_m}, \hat{\mu}_{j_m}))$ of processes of Q, $\forall j_p, j_q \in \{j_1, \dots, j_n\}$, $j_p = j_q$.

(1) holds given the fact that each p_i is deterministic. Since for a process to reach a different local state $(s', \emptyset)$ from a state $(s, \emptyset)$, by definition 3.1 there must be different data flow transitions fired given the same values at its input channels. This can only happen if different micro-steps are enabled given the same input assignment. However this contradicts with the fact that p_i is faithful. By the determinisity of p_i and definition 3.1 we know that the sequence of states corresponds exactly to the sequence visited in LTS_{p_i}.

(2) holds because of the of the way processes evolve. In particular, for $\delta(p_i)$ to update its local state from $(s_i^j, \emptyset)$ to $(s_i^{j+1}, \emptyset)$, by definition 2.23 and definition 3.1 it reads from each input channel one value and writes to each output channel one value. Since Q is strongly connected, for one process p_a in Q to finish one sequence of transitions of $\delta(p_a)$ resembling a macro-step transition in LTS_p, there must be a value produced to the channel that belongs to the channel cycle of the deadlock. As all channels of the cycle are empty, this means all other processes must iteratively consume their inputs so that the produced value of p_a is consumed. From this we can conclude that $\forall j_p, j_q \in \{j_1, \ldots, j_n\}, max(j_p - j_q) \leq 1$. For the case $max(j_p - j_q) = 1$, take one process p_b of the processes $P \backslash \{p_a\}$, then it stuck in the middle to finish its following data flow transitions. However it is not waiting for some inputs from p_a, since as we discussed before p_b (or its predecessors) has consumed the value already, which is the reason why its input channel of the cycle is empty. Therefore p_b must be waiting on some other inputs channels that is not in the cycle, and this cycle is not the reason for the deadlock which contradicts our assumption. As a result $j_{p_a} = j_{p_b}$.

By (2) we can further prove that for each reachable local state $((s_1^{j_1}, \hat{\mu}_{j_1}), (s_2^{j_2}, \hat{\mu}_{j_2}), \ldots, (s_n^{j_n}, \hat{\mu}_{j_n}))$ of processes of P, $\forall j_p, j_q \in \{j_1, \ldots, j_n\}$, $max(j_p - j_q) \leq 1$. Let $j_Q = k$ for processes of Q. If there is a process $p_c \in P \backslash Q$, then we can prove that $|j_Q - j_{p_c}| \geq 1$. Since P is strongly connected, for $j_{p_c} - j_Q \geq 2$, there must be at least 2 values consumed from the input channel of p_c from some process $p_Q$of Q, which is only possible if $j_{p_c} - j_{p_Q} \leq 1$. Similarly for $j_Q - j_{p_c} \geq 2$ it requires $j_Q - j_{p_c} \leq 1$. Both lead to contradictory.

Finally, if $j_Q - j_{p_c} = 1$ then we may let p_c to finish its macro-step round of transitions, since the inputs values from p_Q are already produced provided that p_c does not have to wait for any other processes (which might lead to a bigger deadlock). If $j_{p_c} - j_{pQ} = 1$, then we may let p_c go one macro-step back. Either way we can deduce that $(s^{j_1}, s^{j_2}, \ldots, s^{j_n})$ is reachable in $LTS_{\delta(P)}$ and $j_1 = \cdots = j_n$. This together with (1) conclude that $(s^{j_1}, s^{j_2}, \ldots, s^{j_n})$ are reachable in LTS_P, which can be easily proven by the induction on the number of macro-steps.

Given the above arguments, we prove the theorem by contradiction. Assume there is a configuration of $LTS_{\delta(P)}$ that deadlocks at local state $\zeta = (s^{j_1}, s^{j_2}, \ldots, s^{j_n})$, then some process p_a stuck to finish its rest sequence of transitions resembling a macro-step. However this is impossible, since if P is causally correct then the causal preorder of state ζ of LTS_P should be acyclic, which indicates that there must be a schedule that allows p_a to finish its transitions at ζ. □

Theorem 3.4 is important for the desynchronization design flow, as it shows that once the starting synchronous model is designed properly, the desynchronized system can be

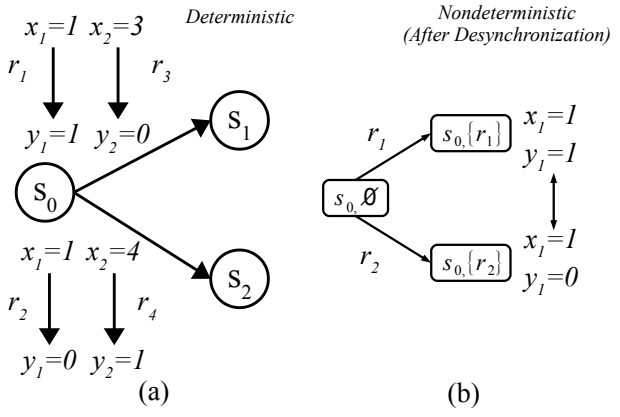
implemented correctly by construction. Because of the synchronous abstraction of the system's communication, the design of the synchronous model can be verified both effectively and efficiently, preventing us from struggling with the computationally infeasible (even theoretically undecidable) analysis of the DPNs directly.

Theorem 3.5. *For an SPN $P = p_1||\ldots||p_n$, if P is causally correct and each p_i is deterministic, then $\forall x \in dom(\Phi(P))$, $\Phi(P)(x) = \Phi(\delta(P))(x)$.*

Proof. Given that P is causally correct, by lemma 2.30 $\Phi(P)$ is a function. Since P is causally correct, $\delta(P)$ is deadlock free by theorem 3.4. By lemma 2.16 we can derive that $\Phi(\delta(P)) = \Phi(\mathcal{L}(\delta(P)))$. By lemma 3.3 and $dom(\Phi(P)) = dom(\Phi(\delta(P)))$ we have $\forall x \in dom(\Phi(P))$, $\Phi(\mathcal{L}(\delta(P)))(x) = \Phi(\mathcal{L}(P))(x)$. Since for SPNs $\Phi(P) = \Phi(\mathcal{L}(P))$, we have $\Phi(P)(x) = \Phi(\delta(P))(x)$. □

Notion of Determinism

Note that the definition of determinism of synchronous processes (definitin 2.33) is more complicate than the classical definition of determinism of LTSs / FSMs []. Instead, the classical definitions only require that for the same state and the same label, the same successive state should be reached. The complexity of our definition is nevertheless necessary, as the scheduled LTSs carry not only state transitions (i.e. information of macro-steps), but also the causal preorders (i.e. information of micro-steps). Therefore the LTS needs to be deterministic in the level of micro-steps, and indeed by theorem 3.5 we can see the importance of such refined level of determinism. If we would have only required the classical determinism, theorem 3.5 would not hold any more, and the desynchronization of single-rated SPNs are not guaranteed to be correct by construction. A concrete example ins given in the following figure:



Note that by classical definition the LTS of (a) is deterministic, since the two transitions encodes different inputs. However after desynchronization, given the same input $x_1 = 1$ the system is able to either chose to fire r_1 or r_2, which leads to different outputs as well as different successive state. Actually, even without

desynchronization the problem still exists, as during one macro-step, the micro-steps should be executed following only causal preorders. Therefore given the same input, as both r_1 and r_2 are enabled, by the synchronous semantics they should be able to executed at the same time, which leads to a write conflict in y_1.

Corollary 3.6. *For an SPN $P = p_1||\ldots||p_n$, if P is causally correct and clock-consistent, each p_i is deterministic, then for all input sequence $x \in \Phi(p_1)|_{I_1}||\ldots||\Phi(p_n)|_{I_n}$, $\Phi(P)(x) = \Phi(\delta(P))(x)$.*

This corollary can be simply derived by the definition of clock-consistency / compatibility (definition 2.35). Since by compatibility, $dom(\Phi(P)) = \Phi(p_1)|_{I_1}||\ldots||\Phi(p_n)|_{I_n}$. The practical advantage thereby is that, if we know that P is additionally clock-consistent, there is no need to constraint the input of $\delta(P)$ to $dom(\Phi(P))$ explicitly by adding constraints to inputs of each $\delta(p_i)$. However notice that in general it is not the case that $dom(\Phi(P)) = \Phi(p_1)|_{I_1}||\ldots||\Phi(p_n)|_{I_n}$, since by definition 2.26, an execution is meaningful only when the synchronous composition of transitions exists. This in general would restrict the domain of $\Phi(P)$ to a proper subset of $\Phi(p_1)|_{I_1}||\ldots||\Phi(p_n)|_{I_n}$. For an intuitive understanding, Figure 3.3 shows an extreme example when such shrinking of input domain happens when considering the synchronous composition.

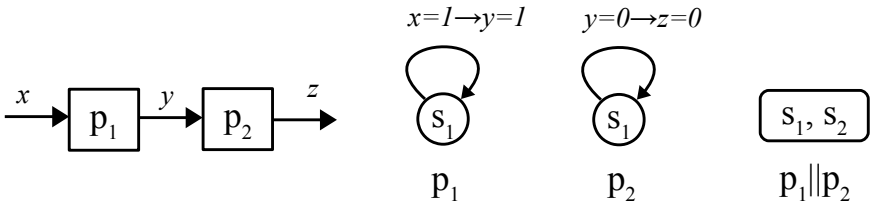


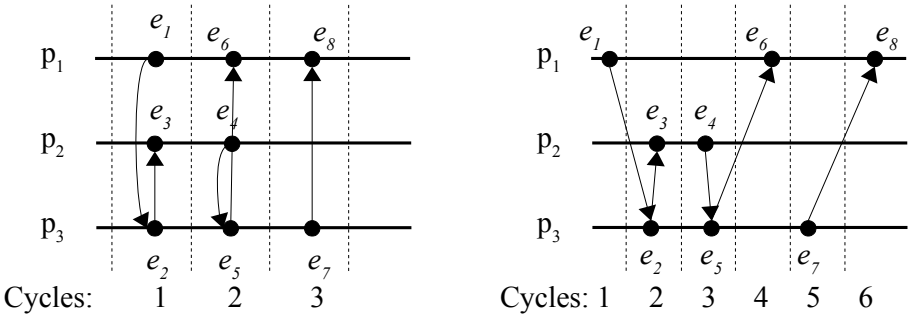
FIGURE 3.3: $dom(\Phi(p_1||p_2)) = \emptyset$.

Since the only output from p_1 is $y = 1$ while the only input expected by p_2 is $y = 0$, the only two transitions of p_1 and p_2 has no synchronous composition, hence the synchronous composition of the two processes has no behavior at all. However such an SPN is trivially causally correct and p_1 and p_2 are trivially deterministic, therefore theorem 3.5 applies. Nevertheless, for the desynchronization, the input $x = 1$ is forbidden since such an input would lead the DPN to a state where p_2 can not react at all, therefore if p_1 repeats its own actions iteratively, the buffer y between p_1 and p_2 would contain all the 1s produced by p_1, and in reality for a bounded buffer y would overflow.

Desynchronization and Latency-Insensitive Designs

The desynchronization of single-rated SPNs we introduced in this section can be seen as a generalization to the latency-insensitive designs (LID) proposed for synchronous circuits [54, 55] and the followed up works on elastic circuits [56–58]. The

motivation of LID is similar to desynchronization, namely to cope with the communication latency in hardware circuits. The problem for classical synchronous circuits design is that the clock frequency is determined by the critical path of the circuit. Thereby in order to raise the clock frequency for better performance, the critical path must be shortened which requires complicated design and re-design iterations. The idea of LID is to discretize the critical path into several sections and pipeline the communication, so that instead of patrolling the critical path in one cycle, it can be done in several cycles. The following message sequent charts intuitively demonstrate the idea of latency-insensitive designs.



The left figure above shows a synchronous execution consists of three cycles, where during each cycle processes p_1, p_2, p_3 communicate with each other. The duration of such a cycle is determined by the longest communication patter of the processes for classical synchronous circuits. In order to shorten the period, LID stretches the events along the time line as shown in the right figure above, therefore events that used to belong to the same cycle now reside in successive cycles. For each event crossing two cycles, a relay-station is used to store and forward the event towards its destination. Such stretching breaks down the critical path (e.g. e_1, e_2, e_3 are performed in two cycles instead of one), but still maintains the normal paths (e.g. e_2, e_3 are still performed in one cycle), therefore may raise the average performance of the system.

In desynchronization, this is generalized by inserting unbounded sized FIFO buffers between processes. The difference between LID and desynchronization is that while relaxing the original system synchronization, LID still promises a synchronous system as the result. Practically, although dedicated buffers (relay-stations [54]) are inserted between different circuits (so called perls), the global system is still synchronized by one physical clock (but a faster one). Similar to the desynchronization of single-rated SPNs, no communication is reduced after the desynchronization. However, while our desynchronization does not add any communication, dummy values are added by LID in order to keep the perls synchronized. The synchronous implementation is later relaxed by introducing asynchronous hand-shake protocols (SELF protocol [58]) by elastic circuits, yet this is more like a technical variation of LID. In the next section we will see that when desynchronizing a multi-rated SPN, the absent values ($\boxdot$) are removed from communication. This may change the functionality of the original SPN, therefore

need to be taken special care of. Instead, LID and elastic circuits didn't examine the semantical redundant values nor try to remove them.

Case Study

Back to our example of Figure 3.1 in the beginning of the section, we can derive the desynchronized LTS of p_1 and p_2 shown as in Figure 3.4.

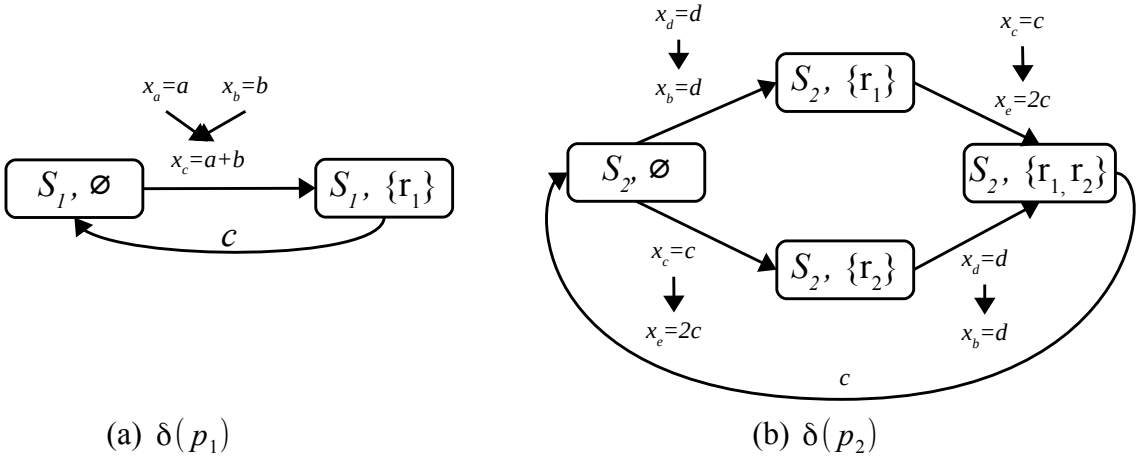


FIGURE 3.4: The labeled transition systems of (a) $\delta(p_1)$ and (b) $\delta(p_2)$ of Figure 3.1.

From the perspective of Kahn's least fixpoint semantics, we can derive the corresponding equation system from the refined LTSs of Figure 3.4 as follows:

$$\begin{aligned} x_c = &\quad (a_1 + b_1) \ldots (a_i + b_i), && \text{if } x_a = a_1 \ldots a_m, x_b = b_1 \cdot \ldots b_n, i = min(m, n), \\ x_b = &\quad d_1 \cdot d_2 \ldots d_n, && \text{if } x_d = d_1 \cdot d_2 \ldots d_n, \\ x_e = &\quad 2c_1 \ldots 2c_n, && \text{if } x_c = c_1 \cdot c_2 \ldots c_n, \end{aligned}$$

Therefore by the least fixpoint semantics, given the input sequences of $x_a = a_1 \cdots a_n, x_d = d_1 \cdots d_n$, the output $x_3 = (2(a_1 + d_1)) \cdots (2(a_n + d_n))$. Compare to the analysis of the example after Figure 2.12, we can see that the desynchronization is consistent with the original synchronous behavior.

3.1.3 Implementation Issues

Theorem 3.5 shows that the desynchronization by definition 3.1 is correct regarding the operational semantics, therefore it promises us for a possible implementation of a DPN preserving the behavior of the SPN. Moreover, as discussed in section 2.1.2, the desynchronized processes work in a data driven way, i.e. the transitions (i.e. firing rules) are enabled once enough input values arrive at the input channels (as soon as possible). Hence unlike the SPN, the desynchronized DPN now works autonomously without

any central control, and each process's execution simply depends on the completion of computations of others processes.

In order to implement the desynchronized processes, we need to collect the firing rules regarding the data flow transitions as well as the control flow transitions, and schedule them properly. The scheduling is again not difficult to implement, as any schedule that follows the data dependency of the causal preorder is a valid one, therefore promises a correct implementation. Scheduling of transitions can be done by a *wrapper*. A wrapper periodically reads values from input buffers and checks if any data-transition can be executed. Once a value is read from each input channel and no further data-transitions can be read, the control transition can take place and this completes the resembling of a macro-step transition. We will have a detailed discussion of how to implement schedulers of processes in the following chapters.

3.2 Desynchronization of Multi-rated SPNs

From the last section we've already developed an effective desynchronization method for single-rated SPNs. As multi-rated SPNs can be seen as single-rated SPNs with the extension of the vocabulary by including $\boxdot$, this method also works for multi-rated SPNs. However, we would like to push our method a bit more so that the multi-rated nature can be exploited in deriving better performance of the derived DPN. This can be shown by an example in Figure 3.5.

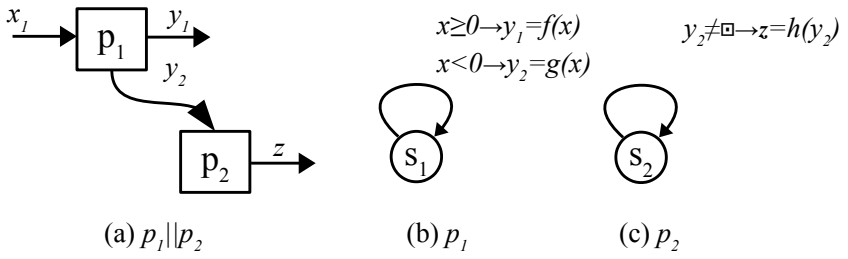


FIGURE 3.5: A multi-rated SPN $p_1||p_2$.

For simplicity we omitted the rules generating $\boxdot$, for example $(x < 0) \rightarrow y_1 = \boxdot$ is omitted. Note that p_2 computes a present value for z only when y_2 is present, i.e. when $(x \leq 0)$ holds for p_1. Otherwise, $\boxdot$ is produced by p_1 and sent to p_2 solely for synchronization purpose. However, such synchronization does not provide any contribution in producing a present value of z. Therefore during desynchronization, we may safely remove the $\boxdot$ transmitted between p_1 and p_2, without changing the functional behavior of the original SPN, namely whenever a present value of y_2 is produced, a present value of z is produced. For a possible extension to definition 3.1, we may remove the transition $(y_2 = \boxdot \rightarrow z = \boxdot)$ from $LTS_{\delta(p_2)}$ and remove the data flow transitions producing $y_2 = \boxdot$ from $LTS_{\delta(p_1)}$, therefore all usages of $\boxdot$ are removed. However as we've seen in the introduction, such a simple extension might not always work, as shown in examples in Figure 3.6.

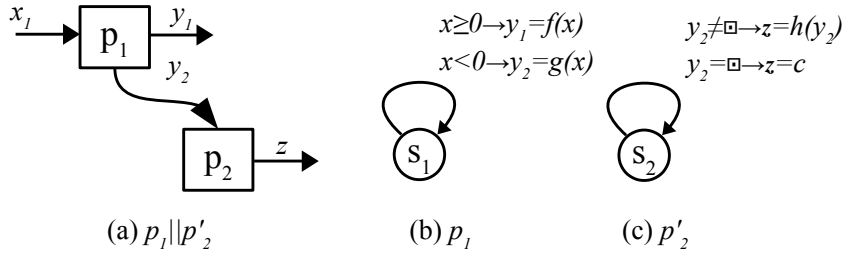


FIGURE 3.6: A multi-rated SPN $p_1||p_2'$.

The problem with $\delta(p_1||p_2')$ is that if we try to remove $\boxdot$ from computations of p_2, then the micro-step $y_2 = \boxdot \rightarrow y = c$ would be transformed to $y = c$, therefore the computation of y would have no condition at all, which is different from the original behavior of $p_1||p_2'$. It is clear that in this case, we can not remove $\boxdot$ because it appears in the input of the micro-step for which the computation of a present value of z depends on. A smarter strategy is therefore to remove $\boxdot$ conservatively, i.e. when they are used for the computation of some present output, then they should be untouched. However the example in Figure 3.7 shows that this strategy might be too conservative for some cases.

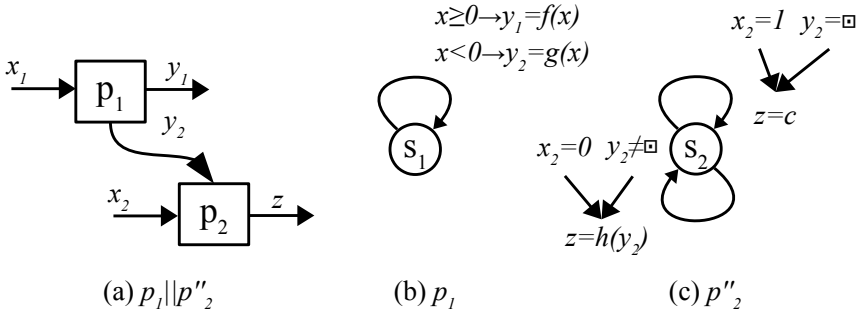


FIGURE 3.7: A multi-rated SPN $p_1||p_2''$.

For p_2'', even if we remove $\boxdot$ from both micro-steps, it would be no problem for p_2'' to react to p_1, since we can *re-insert* $\boxdot$ by looking at the value of x_2. In particular, $x_2 = 0$ indicates that $y_2 \neq \boxdot$ and $x_2 = 1$ indicates that $y_2 = \boxdot$. Therefore the decision of which micro-step to execute can be made once x_2 is read, and if $x_2 = 0$, one value of y_2 should also be read from p_1. This example is quite interesting, as it demonstrates that even when $\boxdot$ has been used for generating some present value, the computation is not necessarily depending on it. In particular, as long as the $\boxdot$ can be re-inserted correctly, the original input sequence can be reconstructed and by determincity of the process the correct sequence of transitions can be performed. Yet another interesting example is shown in Figure 3.8.

As shown in the transition system of process Copy2, whenever x_1 is present it only lead to the presence of y_1. This is symmetric for x_2 and y_2. Therefore the transitions computing y_1 and y_2 are independent with each other, and firing the transitions in any order would not change the functional behavior of the process. Depending on the arrival order of values at x_1 and x_2, $\boxdot$ are inserted correspondingly and different transitions

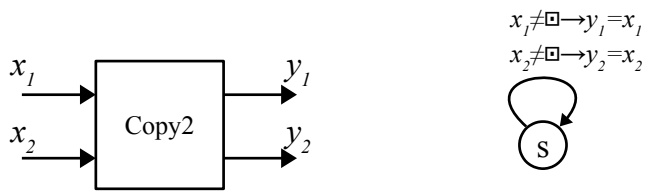


FIGURE 3.8: Process Copy2.

are executed. In case both arrived, either transition can be chosen to be executed first. Although the order of execution might be different, given the same sequence of present values to x_1 and x_2 the same sequence of y_1 and y_2 still can be derived because of the independency between the two transitions.

In the following we first formalize the desynchronization of a multi-rated synchronous process, then we identify a set of properties that are both sufficient and necessary conditions for a process to preserve its functional behavior. Finally we try to find out whether is it decidable to check if a process have these properties after desynchronization.

Definition 3.7. Let $l : C \to \check{\Sigma}$ be a label of an LTS of a multi-rated process. The desynchronization of l is defined as $\delta(l)$ where $\forall x = \boxdot, \delta(l)(x) = \epsilon$ and $\forall x \neq \boxdot, \delta(l)(x) = l(x)$. Then the desynchronization of run $\pi = l_1 \cdot l_2 \cdot \ldots l_n$ is defined as $\delta(\pi) = \delta(l_1) \cdot \delta(l_2) \cdot \ldots \delta(l_n)$.

Definition 3.8. Let p be a multi-rated synchronous process and $\langle I, O, S, s_0, L, \to \rangle$ is derived by definition 3.1. Then $\delta(p)$ is the desynchronized process of p with L'_δ replacing L and $\to_\delta$ replacing $\to$ in which each label $(l : C_l \to \check{\Sigma}) \in L$ is replaced by $\delta(l) : C_l \to \Sigma$.

Intuitively the only extension to desynchronizing a single-rated process is that the $\boxdot$ of labels is replaced by ϵ. Note that the derived process may work quite differently from a desynchronized process of a single-rated process, since now in order to resemble a macro-step transition the process does not have to read from each input channel one value and write to each output channel a value (which used to be the case for the single-rated processes). This turned out to be crucial for a correct reconstruction. Since for single-rated porcess desynchronization, the reconstruction can be easily figured by counting the number of inputs read from each channel. As soon as one value is read from each input channel, it is known that all data transitions are executed and its time for the final control flow transition. The determinicity of the synchronous process ensures that the data transitions are executed correctly (theorem 3.5). However this is different for multi-rated processes, as now it is not trivial to figure out the data flow transitions by simply counting on the arrived inputs, since for some cases some input channels are untouched during a macro-step transition. More importantly, by replacing $\boxdot$ by ϵ, micro-steps of a synchronous process might not be faithful to each other anymore. Figure 3.9 shows such an example.

Notice that replacing $\boxdot$ by ϵ makes the three micro-steps of p not faithful to each other anymore. As for the input label $x_1 = 1, x_2 = 0, x_3 = 1$ all three micro-steps are enabled. As an example, for input sequence $x_1 = 1 \cdot \boxdot, x_2 = 0 \cdot 0, x_3 = \boxdot \cdot 1$, the

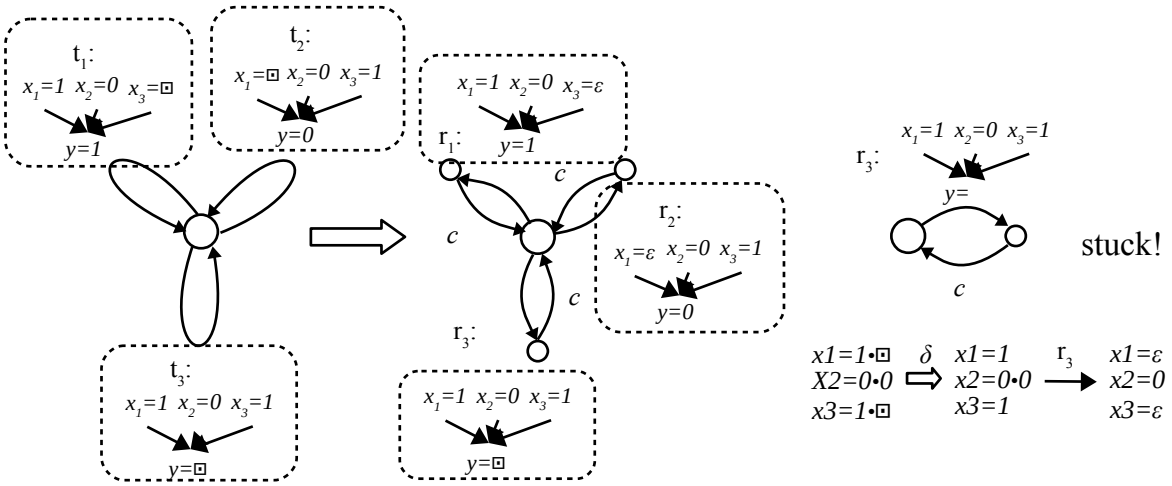


FIGURE 3.9: Desynchronization of a multi-rated SPN consisting of a single process p.

synchronous execution corresponds to $s \xrightarrow{t_1} s \xrightarrow{t_2} s$ deterministically. However, after desynchronization the input sequence becomes: $x_1 = 1, x_2 = 0 \cdot 0, x_3 = 1$, and by the arrival of input values, since all three data-transitions are enabled, $\delta(p)$ is free to chose any of them to execute. As Figure 3.9(c) shows, if r_3 is chosen, then $x_1 = 1, x_2 = 0, x_3 = 1$ are consumed and $x_2 = 0$ is left in the buffer. According to the (macro-step) transition $s \xrightarrow{t_3} s$ of LTS_p no data transition is needed anymore and a control transition can already complete the resembling of t_3, which is different from the synchronous execution. However this leads to the output of $y = \epsilon$ and the configuration $(s, (x_1 = \epsilon, x_2 = 0, x_3 = \epsilon))$, both different from the case if r_1, r_2 were executed which should be the correct sequence that corresponds to the synchronous execution. Therefore, we see that removing ⊡ makes the process p nondeterministic, and the process is not able to reconstruct the correct sequence of executions.

3.2.1 Theoretical Boundary

In addition to desynchronization of single-rated SPNs, the desynchronization of multi-rated SPNs further tries to remove ⊡ from communication and computation. However the removal should not change the functional behavior of the original SPN in the sense that for the same sequence of present input values, the same sequence of present output values should be generated. This requires the desynchronized process to be able to figure out the position of the removed ⊡ and re-insert them properly so that the same sequence of transitions can be executed as before the removal of ⊡. We call this process of figuring out the correct positions of ⊡ *resynchronization*. Similar to single-rated processes, wrappers are needed for multi-rated processes to read inputs and schedule transitions. Further, the wrapper also needs to carry out the resynchronization. The resynchronization of a wrapper for single-rated processes is quite simple, as there is no need to re-insert ⊡. As shown by the example of Figure 3.9, the resynchronization of multi-rated processes is more complicated, thererfore the wrapper construction for

single-rated processes can not be applied for multi-rated processes. In the following, we first formalize the concept of resynchronization to make its meaning precise and clear. Then we go on and study what conditions are required for a successful resynchronization.

Definition 3.9. For a multi-rated synchronous process p, we formalize *resynchronization* as a two-player infinite game with perfect information: $\mathcal{G}(p)$ between players I and II. The game plays as follows:

- the game starts with the initial configuration of $LTS_{\{\delta(p)\}}$ and initial input sequence $x_0 = \epsilon$;
- player I and II make alternative moves with player I make the first move;
- each time player I makes a move, it either calls stop, or chooses an input sequence x' such that $\exists x_c \in dom(\Phi(p)), x \cdot x' \sqsubseteq x_c$ where x is the current input sequence, and put $\delta(x')$ into the input channels of p by executing the corresponding input actions, therefore updating the configuration of $LTS_{\{\delta(p)\}}$. Then it updates the input sequence from x to $x \cdot x'$.
- each time player II makes a move, it executes a sequence of read / write transitions of $LTS_{\{\delta(p)\}}$ and updates the configuration accordingly.

Player II is said to have a *winning strategy* if no matter how player I plays, after a number of rounds (necessarily infinitely many) when I calls stop, the input sequence x generated by player I satisfies $x \in dom(\Phi(p))$ and sequence of execution π that player II performed produces the correct output, i.e. $\mathsf{w}(\pi) = \delta(\Phi(p)(x))$. We say that p is *resynchronizable* if and only if player II has a winning strategy.

From the examples we studied previously, the key to correct resynchronization for multi-rated processes is to:

(1) Either re-insert the $\boxdot$ deterministically, so that a unique sequence of input values can be reconstructed, and by the determinicity of the process a unique sequence of output can be produced (e.g. Figure 3.7), i.e.:

$$\forall x, x' \in dom(\Phi(p)), \delta(x) = \delta(x') \implies x = x'$$

(2) Or re-insert the $\boxdot$ nondeterministically and utilize the independency of transitions so that a unique sequence of output is generated (e.g. Figure 3.8), i.e.:

$$\forall x, x' \in dom(\Phi(p)), \delta(x) = \delta(x') \implies \delta(\Phi(p)(x')) = \delta(\Phi(p)(x))$$

If we examine the above two properties in detail, we can see that property (1) is a special case of property (2), as $x = x'$ implies $\Phi(p)(x') = \Phi(p)(x')$. Besides (2) the process should also be able to distinguish with different input sequences:

(3) For the case when $\delta(x) \neq \delta(x')$, the process needs to recognize which of x and x' is the correct input sequence, by using finitely many computations. I.e.:

$$\exists k \in \mathbb{N}, \forall x, x' \in dom(\Phi(p)), \delta(x) \neq \delta(x') \implies max(Q) = k, \text{ where}$$

$x = x_1 \cdot x_2 \cdot \ldots x_{n-1} \cdot x_n$ and $x' = x'_1 \cdot x'_2 \cdot \ldots x'_{n-1} \cdot x'_n$ such that $\forall i < n, \delta(x_i) = \delta(x'_i)$ and $\exists s_0, s_1, \ldots, s_{n-1}$ in LTS_p such that $\forall s_i, s_i \xrightarrow{x_i/w_i} s, s_i \xrightarrow{x_i/w_i} s' \implies s = s' = s_{i+1}$, and $Q := \{|\delta(x_1)|, |\delta(x_2)|, \ldots, |\delta(x_n)|, |\delta(x'_1)|, |\delta(x'_2)|, \ldots, |\delta(x'_n)|\}$.

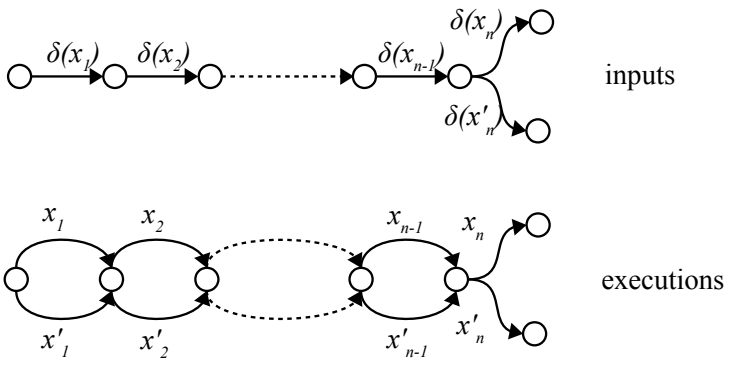


FIGURE 3.10: Illustration of property (3).

In the following we prove that property (2) and (3) are both sufficient and necessary for resynchronizability.

Theorem 3.10. *For a multi-rated process p, p is resynchronizable if and only if property (2) and (3) hold for p.*

Proof. ($\Rightarrow$) Assume properties (2) and (3) hold, we prove that player II has a winning strategy. This winning strategy can then be used to build a conceptual wrapper for $\delta(p)$ making the decisions of which transitions to execute, while player I plays the role as the environment that pushes input values any time it wants to the input channels of $\delta(p)$. Once the game starts, player II plays its own rounds, assume the current input sequence in its input buffers is x and the local state of p is s_i. Player II simply waits for player I to make its moves, until $|x| \geq k$ or player I calls stop. Then player II makes its own move correspondingly as the following two cases:

a. If player I calls stop, then $|x| < k$ and player II can simply try to to resynchronize x to a synchronous input sequence a that leads to a maximum and effective execution. By property (2), any other synchronous input sequence a' such that $\delta(a) = \delta(a')$ should lead to an execution that produces the same present outputs.

b. If $|x| \geq k$, then player II tries to extract the prefix $x' \sqsubseteq x$ with $|x'| \leq k$ from its input channels such that for all resynchronization a and a' with $\delta(a) = \delta(a') = x'$, $\forall s, s', s_i \xrightarrow{a/w} s, s_i \xrightarrow{a'/w'} s' \implies s = s'$. Again two cases are possible. Either by property (3) there is a state $s_{i+1} = s = s'$. Or a, a' must both be prefixes of

some synchronous inputs b, b' such that $\delta(b) = \delta(b')$. By property (2) choosing any resynchronization of x won't change the produced present outputs.

Finding s_{i+1} can be finished in finitely many computations since x' is finite and p is finite. If s_{i+1} is found, player II executes the corresponding transitions that resembles the macro-step transitions consuming a, generates the outputs and updates the configuration. By property (2) $\delta(w) = \delta(w')$. Otherwise player I simply chose any resynchronization of x and try to consume a maximum prefix of x.

The above strategy is a winning strategy for player II, since for any input, once player I calls stop, player II can proceed until case (a). This may take as many rounds as needed, but player II is able to finish the game once player I calls stop. Further, the execution π performed by player II satisfies $\mathsf{w}(\pi) = \delta(\Phi(p)(x))$, since it either finds the correct resynchronization by case (a) or it produces a correct prefix of the outputs by case (b).

($\Leftarrow$) Assume that property (2) fails for p. Then there exists $x, x' \in dom(\Phi(p))$ such that $\delta(x) = \delta(x')$ but $\delta(\Phi(p)(x')) \neq \delta(\Phi(p)(x))$. If player I pick $\delta(x)$ as the input sequence, then player II may lose as if it choose x as the resynchronized input sequence, player I would have used x' which leads to a different output sequence. The case for player II choosing x' is symmetric.

Assume that property (3) fails for p. Then for each $k \in \mathbb{N}$, there exist $x, x' \in dom(\Phi(p))$ such that $max(Q) > k$. Note that $max(Q)$ might even be infinite. For such case, whenever player I pushed some input values x into the input channels, player II always has to wait, since if it choose to resynchronize x, it might made the wrong choice. This is illustrated by the example of Figure 3.11.

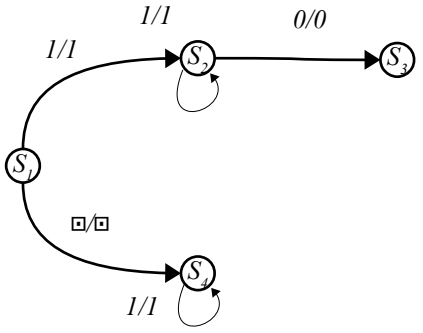


FIGURE 3.11: Illustration of the case when player II waits forever.

As if player I chooses to push only 1s to the input buffer, player II waits forever. Otherwise if it chooses $s_1 \xrightarrow{1/1} s_4 \xrightarrow{1/1} s_4 \rightarrow \ldots$, player I may then push a 0 to the input buffer, and player II is never able to consume the 0 from state s_4.

For the case k is finite, player II need to wait for finitely many steps to find out that the input sequence x is a prefix of some synchronous input sequences such that $\delta(b) \neq \delta(b')$. However player II may still need to wait forever, since it can not tell whether it is on the way to one of two different input sequences $\delta(b) \neq \delta(b')$ and $x \sqsubseteq \delta(b)$, or it is rather $x \sqsubseteq \delta(b)$ and $\delta(b) = \delta(b')$. For the second case, player II would have to wait forever.

□

For the process of Figure 3.9, player II has no winning strategy. For finite input sequences, player II can find a correct resynchronization by counting the number of "0" it received after player I called stop. Since for all transition player I would send a 0 to channel x_2, but only a 1 is sent to x_1 when t_1 is executed and only a 1 is sent to x_3 when t_2 is executed, we can derive the following invariant for the input sequences:

$$\begin{aligned} n_0 &= n_{t_1} + n_{t_2} + n_{t_3} \\ n_{t_1} &= n_{x_1} - n_{t_3} \\ n_{t_3} &= n_{x_3} - n_{t_3} \end{aligned}$$

where n_0 is the number of 0s in channel x_2, n_{x_1} and n_{x_3} are the number of 1s in x_1 and x_3 and $n_{t_1}, n_{t_2}, n_{t_3}$ are the number of executions of transitions t_1, t_2 and t_3 respectively. Any solution that satisfies the equation system corresponds to a valid resynchronization. However, if player I sends infinitely many inputs values, player II would have to wait forever. For any decision it makes, player I is able to adjust the number of 0's follows so that player II fails to complete an effective execution. Also, it is clear that player I needs to send the special stop signal. Otherwise player I would never know whether there will be more inputs coming, hence has to wait forever. Unfortunately, property (2) is undecidable even for finite synchronous processes.

Theorem 3.11. *For a multi-rated synchronous process p, it is undecidable that whether* $\forall x, x' \in dom(\Phi(p)), \delta(x) = \delta(x') \implies \delta(\Phi(p)(x')) = \delta(\Phi(p)(x))$.

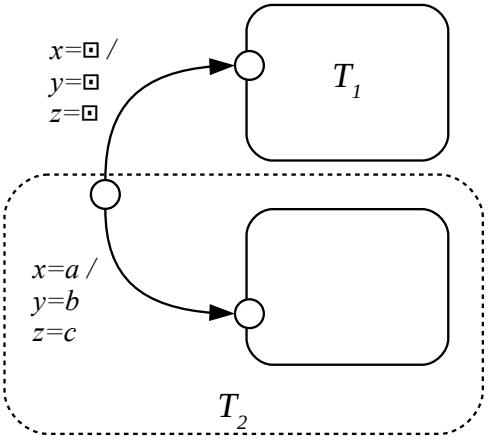


FIGURE 3.12: Prove undecidability of property (2) by construction.

Proof. We use the Fisher-Rosenberg theorem on rational languages, and reduce our problem to one of the undecidable problems there.

Theorem 3.12 ([59])**.** *Let X, Y be alphabets with at least two letters. For rational subsets $A, B \subseteq X^* \times Y^*$, it is undecidable that whether (1) $A = B$ (2) $A \subseteq B$.*

We construct a multi-rated synchronous process that is deterministic and show that to decide whether it holds for property (2) we need to solve the comparison of rational languages. The example is shown in Figure 3.12, where there is one input channel x and two output channels y, z for process p. The two different transitions of LTS_p starting from the initial state leads to to deterministic single-rated processes T_1, T_2. Note that the transition leads to T_1 only reads and writes $\boxdot$. Without losing generality, we can assume that $\mathcal{L}(T_1)|_{\{x\}} = \mathcal{L}(T_2)|_{\{x\}}$. Therefore if $\mathcal{L}(T_1)|_{\{y,z\}} = \mathcal{L}(T_2)|_{\{y,z\}}$ by determinicity of T_1 and T_2 we can deduce that for the input sequence x, x' such that $\delta(x) = \delta(x')$, $\delta(\Phi(p)(x')) = \delta(\Phi(p)(x))$. However since the output languages of T_1 and T_2 are rational relations, and by theorem 3.12 comparing their equivalence is undecidable.

□

ASAP Winning Strategy for Player II

The winning strategy we introduced for player II allows it to accumulate some inputs before it starts execution, even if some transitions are already able to be executed. The reason is that in order to decide which is the correct resynchronization, player II needs to examine a finite history of input sequence. Another important issue is that for finite input sequences, player I needs to send a "stop" signal to player II indicating the end of the input sequence. Therefore the stop signal should never be encoded by $\boxdot$, or it would be removed from communication and player II would wait forever. In practice for better reaction time player II may need to react as soon as possible (ASAP) in the sense that once a transition is enabled, it is executed immediately. Such an ASAP strategy has the benefit that no internal memory is needed for storing input history. The key of ASAP strategy is that whenever player II executes a transition, it will not regret the decision, since player I has no chance to produce a different output. Based on this intuition we propose property (4) as follows:

$$\begin{aligned}&(4)\ \forall x \in dom(\Phi(p)), \forall \pi \in \Gamma(\{\delta(p)\}), \mathsf{r}(\pi) \sqsubseteq \delta(x) \implies \\ &\exists \pi' \in \Gamma(\{\delta(p)\}), \pi \sqsubseteq \pi' \wedge \mathsf{r}(\pi') = \delta(x) \wedge \mathsf{w}(\pi') = \delta(\Phi(p)(x))\end{aligned}$$

Intuitively, property (4) states that if player II chooses an execution consuming a prefix of x, then it should be able to finish consuming the whole sequence of x and produce the correct output sequence. Therefore for any input sequence, player II can schedule the available transitions any way possible without regret, since it is always able to complete an execution that consumes all future inputs and produce the correct outputs. We can decompose property (4) into the conjunction of two properties.

Lemma 3.13. *Property (4) is equivalent to the conjunction of the following two properties:* $\forall x_0 \in dom(\Phi(p))$ *with* $x = \delta(x_0)$*:*

(4.1) $\forall \pi' \in \Gamma(\delta(p)), \mathsf{r}(\pi') = x' \sqsubseteq x \implies (\exists \pi \in \Gamma(\delta(p)), \pi' \sqsubseteq \pi \wedge \mathsf{r}(\pi) = x)$*;*

(4.2) $\forall \pi, \pi' \in \Gamma(\delta(p))$ *with* $\mathsf{r}(\pi) = x$, $\mathsf{r}(\pi') \sqsubseteq \mathsf{r}(\pi) \implies \mathsf{w}(\pi') \sqsubseteq \mathsf{w}(\pi)$*;*

Proof. (sketch) First notice that properties (4.1) and (4.2) are about process $\delta(p)$, rather than the DPN $\{\delta(p)\}$. However this is no problem, since both properties only talk about prefixes of executions, and there is a prefix π' of an execution π in $\Gamma(\delta(p))$ if and only if there is a prefix ψ' of execution ψ in $\Gamma(\{\delta(p)\})$ with $\pi = \psi|_p$ and $\pi' = \psi'|_p$. Property (4.1) states that whenever player II chooses one execution path reading the prefix x' of x, it should be able to extend the path by reading the rest of x. Property (4.2) states that no matter which path player II choose, the produced result should be consistent. Since there must be one execution that is consistent to the synchronous execution player I performs, the produced output sequence should be equivalent to $\delta(\Phi(p)(x_0))$.

□

Theorem 3.14. *Player II has an ASAP winning strategy if and only if property (4) holds for $\{\delta(p)\}$ (or equivalently, property (4.1) and (4.2) hold for $\delta(p)$).*

Proof. (sketch) ($\Rightarrow$) This direction is trivial.

($\Leftarrow$) Assume player II has an ASAP winning strategy. If (4.1) fails, then there exists an execution π with $\mathsf{r}(\pi) = x' \sqsubseteq x$ that player II can not finish reading $x \setminus x'$. If player II choose this path, and player I pushes the rest of the input sequence of x into the input channels, player II would have no way to proceed. If (4.2) fails, then if player II chooses an execution path that generates an output sequence y, there must exists a synchronous execution that player I can perform which generates output sequence y' with $\delta(y') \neq y$.

□

Theorem 3.15. *For a multi-rated synchronous process p, it is undecidable to check if $\{p\}$ satisfies property (4).*

Proof. (sketch) Note that similar to theorem 3.11, in order to verify property (4.2) we need to solve the comparision of rational relations. □

Figure 3.13 shows a synchronous process whose desynchronization has an ASAP winning strategy. Note that different choices of player II may lead to different control states of the LTS, however the functional behavior stays the same. The undecidability is intuitively caused by such divergence, which leads to the requirement of comparison of rational languages produced by different transducers.

Decidability of ASAP Winning Strategy for Player II

In this section we show that with an additional property called *confluence*, it is decidable whether a desynchronized process allows ASAP winning strategy. Previously we mainly focused on the capability of the desynchrinzed process to reproduce the correct sequence of outputs. Therefore as shown by the example of Figure 3.13, different executions may lead to different local states. It is more often required that for the same input sequence of inputs, different executions should lead to not only producing the same sequence

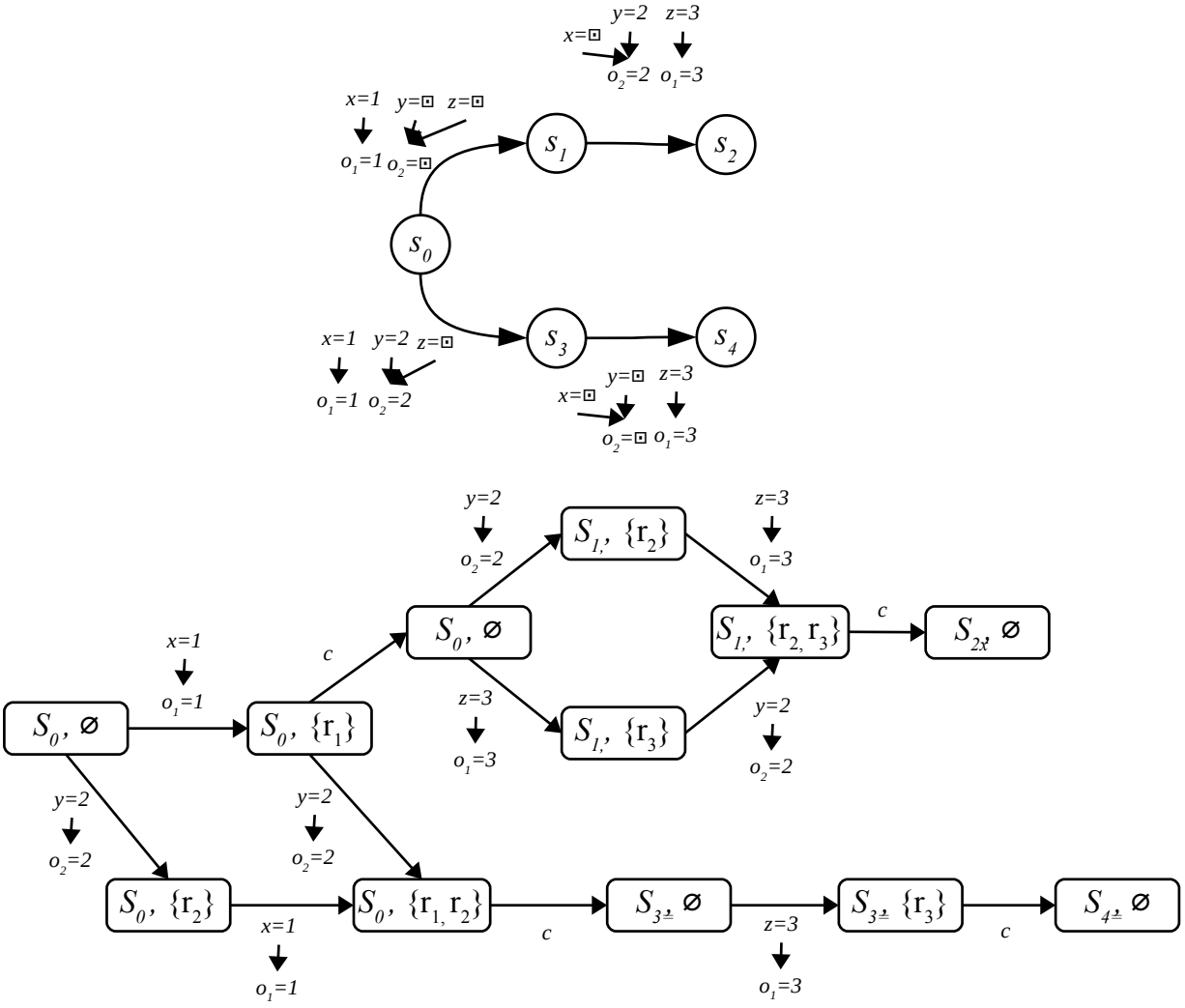


FIGURE 3.13: A multi-rated process with observational equivalent executions.

of outputs but also the same final state. This is often called *confluence* [7, 60]. In the following we show that confluence implies property (4.1) and (4.2), therefore for a confluent process player II has an ASAP winning strategy.

Before we formalize the definition of confluence, we first introduce some assistant concepts. For two transitions $(s, l, s_1), (s, l', s_2)$ originated from state s, we say that l and l' are *consistent* if for $D = dom(l) \cap dom(l'), \forall c \in D, l(c) = l'(c)$. For example, $l := \{x_2 = 2, x_3 = 3\}$ and $l' := \{x_1 = 1, x_2 = 2\}$ are consistent. Consistency can be extended to concatenations of labels in the obvious way.

Definition 3.16. For multi-rated synchronous process p, its desynchronization $\delta(p)$ is confluent if only $\delta(p)$ satisfies the following condition (property 5):

(5) $\forall \pi, \pi' \in \Gamma(\delta(p)), \pi = s_0 \xrightarrow{\rho(\pi)} s_1, \pi' = s_0 \xrightarrow{\rho(\pi')} s_2$, if π is a macro-step execution and $\mathsf{r}(\pi') \sqsubseteq \mathsf{r}(\pi)$ then $\exists \psi, s_2 \xrightarrow{\rho(\psi)} s_1 \wedge \rho(\pi' \cdot \psi) = \rho(\pi)$.

Figure 3.14 illustrates the intuition of confluence.

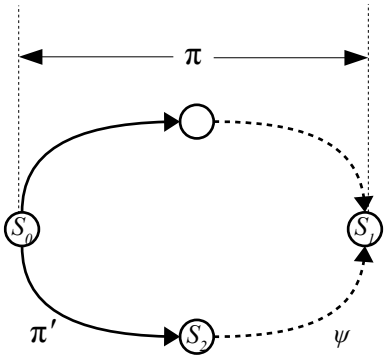


FIGURE 3.14: Confluence.

Lemma 3.17. *Property (5) implies property (4).*

Proof. By lemma 3.13 we prove property (4.1) and (4.2) are both implied by property (5). For property (4.1), given $\pi := s_0 \xrightarrow{\rho(\pi)} s_1, \pi' := s_0 \xrightarrow{\rho(\pi')} s_2$ with π any macro-step execution of $\delta(p)$ and $\mathsf{r}(\pi') \sqsubseteq \delta(x)$, by property (5) there exists ψ such that $\psi := s_2 \xrightarrow{\rho(\psi)} s_1$ and $\mathsf{r}(\pi' \cdot \psi) = \mathsf{r}(\pi)$. Further by property (5) $\mathsf{w}(\pi \cdot \psi) = \mathsf{w}(\pi' \cdot \psi)$ which also proves property (4.2). □

Note that property (4.1) and (4.2) do not imply confluence. Figure 3.13 is one counterexample.

Corollary 3.18. *For a multi-rated synchronous process p, if its desynchronization $\delta(p)$ is confluent, then player II has an ASAP winning strategy for $\mathcal{G}(p)$.*

In the following, we develop a condition that is equivalent to confluence and we show that this condition is decidable, therefore proving that confluence is decidable.

Definition 3.19. For multi-rated synchronous process p, let $t_1 := (s_0, R_0) \xrightarrow{l_1} (s_i, R_1), t_2 := (s_0, R_0) \xrightarrow{l_1'} (s_j, R_1')$ be two transitions of its desynchronization $\delta(p)$ and $l_1|_{I_p}$ and $l_1'|_{I_p}$ are consistent. We say transition t_2 *converges to* t_1 if the following condition holds:

- Let $\pi := (s_0, R_0) \xrightarrow{l_1} \ldots \xrightarrow{l_m} (s_n, \emptyset)$ be a sequence of transitions such that $\forall \pi' := (s_0, R_0) \xrightarrow{l_1} \ldots \xrightarrow{l_k} (s_k, \emptyset)$ with $\pi' \sqsubseteq \pi$, $\rho(\pi')|_{I_p} \sqsubseteq l_1'|_{I_p}$. Then there must be a sequence of transitions $\pi' := (s_0, R_0) \xrightarrow{l_1'} \ldots \xrightarrow{l_{m'}'} (s_n, \emptyset)$. We call π a *least extended macro-step execution* and t_2 converges to t_1 at π with π'.

$\delta(p)$ is *locally confluent* if and only if for all transitions $t_1 := (s_0, R_0) \xrightarrow{l_1} (s_i, R_1), t_2 := (s_0, R_0) \xrightarrow{l_2} (s_j, R_2)$, $l_1|_{I_p}, l_2|_{I_p}$ are consistent implies that for any least extended macro-step execution π, t_2 converges to t_1 at π with π' and $\rho(\pi) = \rho(\pi')$.

Transition convergent is illustrated in Figure 3.15.

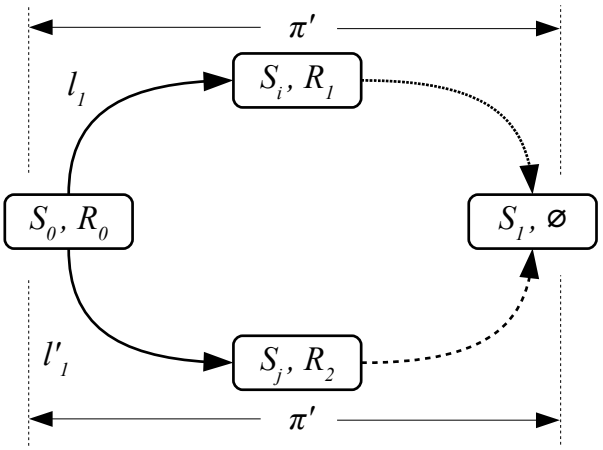


FIGURE 3.15: Local confluence.

Theorem 3.20. *For multi-rated synchronous process p, $\delta(p)$ is confluent if and only if $\delta(p)$ is locally confluent.*

Proof. ($\Leftarrow$) This direction is trivial.

($\Rightarrow$) Assume $\delta(p)$ is locally confluent and a macro-step execution $\pi := (s_0, \emptyset) \xrightarrow{l_1} \ldots \xrightarrow{l_m} (s_n, \emptyset)$ and $\pi' := (s_0, \emptyset) \xrightarrow{l'_1} \ldots \xrightarrow{l'_k} (s_l, R_k)$ and $\mathsf{r}(\pi') \sqsubseteq \mathsf{r}(\pi)$. We prove by induction over the length of π' that $\delta(p)$ is confluent. When $k = 1$ it is trivial that there exists $\psi \sqsubseteq \pi$ such that ψ is a least extended macro-step execution. By local confluence there exists ϕ leading from π' to the end of ψ. As $\psi \sqsubseteq \pi$, it obviously converges to the end of π. Assume $\delta(p)$ is confluent when $k = l$, we prove that it also holds for $k = l + 1$. Assume $\pi' := (s_0, \emptyset) \xrightarrow{l'_1} \ldots \xrightarrow{l'_j} (s, R) \xrightarrow{l'_{j+1}} (s', R')$. By the assumption there must be a sequence of transitions $\phi = (s, R) \xrightarrow{l'_i} \rightarrow \ldots (s_n, \emptyset)$ and $\rho(\pi' \cdot \phi) = \rho(\pi)$. Therefore $l'_i|_{I_p}$ and $l'_{j+1}|_{I_p}$ must be consistent. By the induction base there must be a sequence of transitions leading from (s', R') to some state prior or equal to $(s_n, \emptyset)$. This completes the induction step.

□

It is not difficult to see that local confluence is decidable for finite processes. In particular, the definition of local confluence only involves pairs of transitions with input-consistent labels which are finitely many, and for each transition, its number of least extended macro-step executions is also finite, and the length of each least extended macro-step execution is also finite.

ASAP scheduling strategy and Deterministic Desynchronization

A previous work that is closely related is the concurrent NRSA condition introduced in [60]. The authors of the paper identified several important properties regarding the desynchronized process, namely: monotonous, deterministic and

confluent. Monotonous and deterministic are properties of the function of the desynchronized process. Comparing to our work, property (4.2) ensures that our desynchronized processes are both monotonous and deterministic. Confluent is a structural property of the transition system of the desynchronized process which demands that no mater which order the process reads the inputs, the same outputs can be produced and the same state is reached. This property is similar to the definition of confluence we introduced. The main result is that a condition called concurrent NRSA (no reaction to signal absence) is proven to be both necessary and sufficient for a desynchronization that is monotonous, deterministic and confluent with an ASAP winning strategy.
Concurrent NRSA is proposed as a property of a synchronous LTS, however as the desynchronized process shares the same LTS with the synchronous process, it can also be verified against the desynchronized process. The drawback is that the authors assume that inputs should be arrived in the start of each macro-step, which is not applicable for complex causal situation in DPNs of scheduled synchronous LTSs. We assume that the main result still holds for our case, however concurrent NRSA should be rather verified against the desynchronized process to keep the if-and-only-if preserved, since it is rather the function and the structure of the desynchronized process that matters. From this we can see that the unified LTS presentation of both synchronous and desynchronized process makes the arguments simpler, but also blurs some important questions and limits its application.
Concurrent NRSA is comparable to local confluence we introduced as a local property that is decidable. The difference is that our definition of local confluence is developped based on the notion of macro-step executions, which takes a refined perspective of a macro-step in the desynchronized transition system. Instead the concurrent NRSA does not take care of such refined view because of the unified representation of the transition system.

3.2.2 Desynchronization of Synchronous Communication (Multi-Rated)

The previous sub-section studied the desynchronization of a single multi-rated process in detail. With the results we've developed, we now go on and study the core problem–desynchronization of a multi-rated SPN. Like the desynchronization of single-rated SPNs, we expect to develop a theorem comparable to theorem 3.5, which gives us sufficient conditions of a correct desynchronization. More over, the condition should be necessary for the synchronous SPN (but might not be necessary for a correct desynchronization). Since single-rated processes are special forms of multi-rated processes, a natural starting point is theorem 3.5. We then try to make necessary modifications to adapt the theorem for multi-rated SPNs. The first modification obviously comes from the results we derived from the last sub-section, i.e. the desynchronization should promise each process with a winning strategy. This requires property (2) and (3) to hold for all synchronous processes. Also note that the desynchronized process might not react as soon as possible (since the wrapper might hold the inputs until it can make the right

decision), the behavior of the DPN should consider the wrappers of the processes as well. We denote the wrapped desynchronized process by $\omega(p)$. Therefore the desynchronization of the SPN P is denoted by $\omega(P) = \omega(p_1)|||\dots|||\omega(p_n)$. Another important point is that the asynchronous composition of processes is based on the as-soon-as-possible execution strategy of processes. Since by the second induction rule of definition 2.5, once a transition is enabled, it can be fired immediately. Therefore in order to realize the winning strategy (which requires waiting), the operational semantics of the LTS of $\omega(P)$ needs to be refined. Here we'd rather give an intuitive explanation (as we will soon notice that the operational semantics of general wrapped processes are not of our interests):

- The wrapped process $\omega(p)$ consists of three parts: an internal buffer, a scheduler and $\delta(p)$;
- The scheduler performs the winning strategy, therefore *reads* enough inputs into its buffer until it can make the right *decision* on how to schedule its transitions (assuming the buffer has enough storage). Then it *executes* the determined transitions, consumes corresponding values from the local buffer and generates outputs.
- After the determined transitions are executed, the wrapper starts the new round of read-determine-execution.

By the above modification, each wrapped process $\omega(p)$ is now a process that executes as soon as possible, therefore may be used for the asynchronous composition of DPNs. Now we may state the modified claimed theorem as follows:

- For an SPN $P = p_1||\dots||p_n$, if P is causally correct and each p_i satisfies property (2) and (3), then $\forall x \in dom(\Phi(P))$, $\Phi(P)(x) = \Phi(\omega(P))(x)$.

Unfortunately, the following example of Figure 3.16 shows that the above claim does not hold.

The structure of the SPN (and corresponding DPN) is shown in Figure 3.16 (a), where the SPN consists of two synchronous processes p_1, p_2. Figure 3.16 (b) and (c) are the LTSs of p_1 and p_2 accordingly. For simplicity, we used the same labels for the corresponding local states of p_1 and p_2 that are synchronized in the SPN. It is not difficult to see that both processes are deterministic. Moreover, both satisfy property (2) and (3), since both have only finitely many executions. Note that even both process may continue their executions, the part of the transitions system shown in the figure still satisfies property (2) and (3), as p_2 has two branches which reads present values: $y_1 = 1 \cdot 1$ and $y_1 = 1 \cdot 2$ respectively, and p_2 has two branches which reads present values: $\{x_1 = 1, x_2 = 2, x_3 = 3\}, \{x_1 = 1, x_2 = 2, x_3 = 2\}$ respectively. Therefore the wrapper of p_1 can wait until it reads the second value from y_1 to tell which branch to proceed, while the wrapper of p_2 can wait until it reads the first value of x_3 to decide which branch to proceed. Also the SPN $p_1||p_2$ is causally correct. The scheduled LTS of the SPN is shown in Figure 3.17.

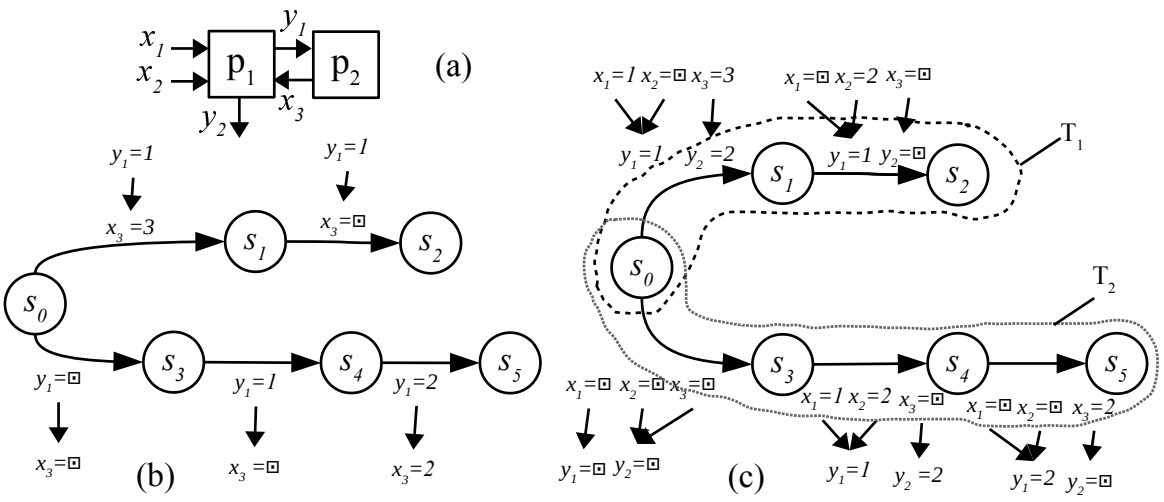


FIGURE 3.16: A counterexample of the claim.

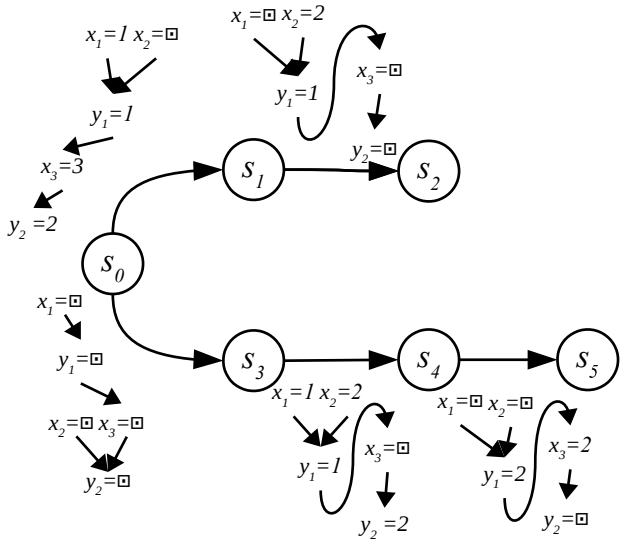


FIGURE 3.17: The SPN of $p_1||p_2$.

It is not difficult to see that the SPN is causally correct since no transition's causal preorder is cyclic. However, as we desynchronize the SPN, $\omega(p_1)$ would wait for the value of x_3 to distinguish between the two macro-step executions T_1 and T_2, since inputs from x_1, x_2 are the same for T_1 and T_2. However, the value of x_3 is produced by $\omega(p_2)$ and in order to produce the value of x_3, $\omega(p_2)$ needs to read values from y_1, which is produced by $\omega(p_1)$. Therefore both processes end up waiting each other, and the DPN deadlocks from the start.

The example in Figure 3.16 shows that even though each process has a winning strategy and hence is able to resynchronize correctly, because of the waiting nature of the winning strategy the global DPN may fall into deadlocks. However we know that the ASAP winning strategy does not suffer from waiting. Can it provide a deadlock-free

desynchronization? The counterexample in Figure 3.18 shows that even for confluence processes there might still be deadlocks.

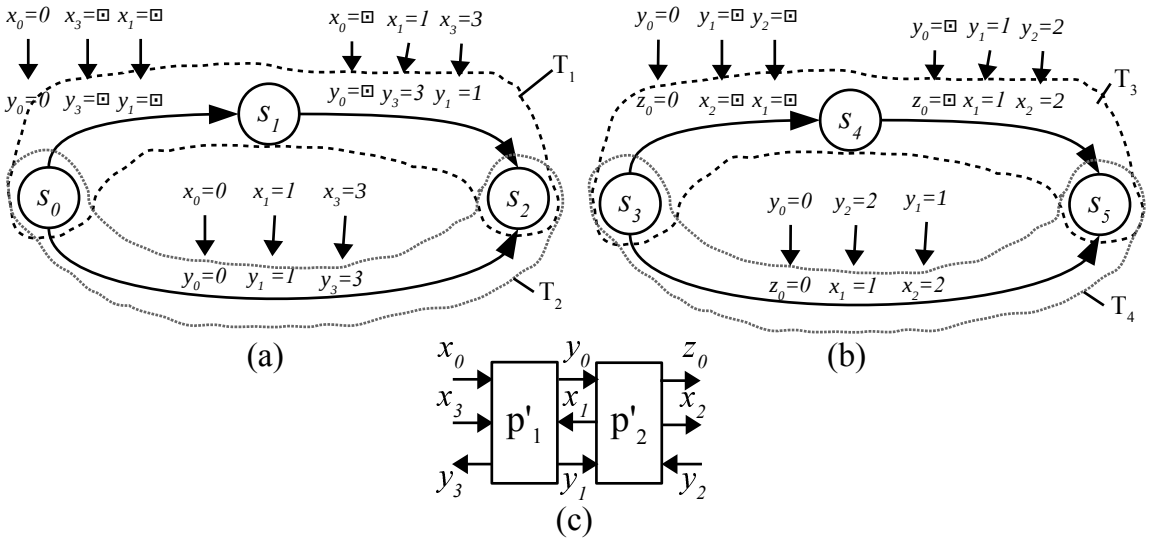


FIGURE 3.18: The SPN of p_1' and p_2'.

As shown in Figure 3.18, both process p_1 and p_2 are confluence, therefore have ASAP winning strategy. When running synchronously, the execution T_1 of p_1' is synchronized with the execution T_3 of p_2', while the execution T_2 is synchronized with T_4 respectively. However after desynchronization, since there will be no more data transitions from $(s_0, \{x_0 = 0 \rightarrow y_0 = 0\})$ to $(s_1, \emptyset)$ as well as from $(s_3, \{x_0 = 0 \rightarrow z_0 = 0\})$ to $(s_4, \emptyset)$, both $\delta(p_1')$ and $\delta(p_2')$ are free to follow either macro-step executions. In particular, if $\delta(p_1)$ chooses to resemble T_2 while $\delta(p_2')$ chooses T_3, then after $\delta(p_2')$ reaches $(s_4, \emptyset)$, the two processes both have to wait for the other, since in order to procuce $y_1 = 1$ $\delta(p_1')$ needs $x_1 = 1$ which is produced by $\delta(p_2')$, which in turn needs $y_1 = 1$. When we examine the behavior of processes from Kahn's least fixpoint semantics, we can find out that the I/O relations derived from the corresponding LTSs are not functions. For example, for p_1', given the input sequence $x = (x_0 = 0, x_1 = 1, x_3 = \epsilon)$, $\Phi(\delta(p_1'))(x) = \{(y_0 = 0, y_1 = 1), (y_0 = 0, y_3 = 3)\}$. Therefore the least-fixpoint semantics can not be applied.

This example reveals a different problem than the example in Figure 3.16. In particular, even if the desynchronized process can react as soon as possible, still because of the internal nondeterministic behavior caused by desynchronization, it may choose an execution path that leads to causal problems during the interaction with other processes. We call the choice of such wrong execution path by the process *deviation*.

In order to solve the problem of deviation, we can either constraint the choice of transitions for the wrapper of the desynchronized process, so that when more than one transition is possible, only one particular transition is chosen always. Since any choice is valid, the particular fixed choice is definitely fine. However this equally means that the other transitions (as well as corresponding executions) are eliminated from the transition system. Instead we can constraint the structure of the transition system, so that difference choices of transitions should not harm the causality of the system.

Causally Confluent Processes

Let's have a look at the example of Figure 3.18 again. If we examine the LTSs of the processes in detail, we'll see that the cause of the deadlock is that by matching the macro-step execution of T_2 with T_3, the causal preorders of the data flow transitions $(x_1 = 1 \rightarrow y_1 = 1)$ and $(y_1 = 1 \rightarrow x_1 = 1)$ form a cycle. However when T_1 matched with T_3 there is no cycle of the causal preorders. Although from the macro-step perspective, both executions T_2 and T_3 consume the same inputs and produce the same outputs, from the micro-step perspective the consumption and production of the values are in different orders. If the two paths would posses the same micro-steps, then the causal problem would be gone.

Definition 3.21. For the desynchronization $\delta(p)$ of multi-rated synchronous process p, given an execution $\pi := (s_0, R_0) \xrightarrow{l_1} (s_1, R_1) \rightarrow \ldots \xrightarrow{l_n} (s_n, R_n)$, we define $\mathcal{M}(\pi) := \bigcup_{i \in \{0 \ldots n\}} R_i$. Then p is *causally confluent* if $\delta(p)$ is confluent and satisfies:

(6) For all macro-step executions $\pi := s_0 \xRightarrow{\rho(\pi)} s_1$ and $\pi' := s_0 \xRightarrow{\rho(\pi')} s_2$ executions of $LTS_{\delta(p)}$, $\rho(\pi) = \rho(\pi')$ implies $\mathcal{M}(\pi) = \mathcal{M}(\pi')$.

Theorem 3.22. *For an SPN of multi-rated synchronous processes $P = p_1||p_2||\ldots||p_n$, if P is causally correct and each $\delta(p_i)$ causally confluent, then $\delta(P)$ is deadlock free.*

Proof. We prove by contradiction. Since P is causally correct, for each synchronous execution π_0 of P, there exists a macro-step execution π_0' of $\delta(P)$ such that $\delta(\rho(\pi_0)) = \rho(\pi_0')$. Hence there must be a process p_i such that for some input sequence x which leads to a non-deadlock execution $\pi = \pi_0|_{p_i}$ of $\delta(p)$, there exists another execution π' reading the prefix x' of x that leads to a deadlock. Then it must be the case that transitions of π' are different from π. Without losing generality, we may assume π is a macro-step execution. By the assumption that each p_i is causally confluent, there must exist macro-step executions π_0, π_0' such that $\pi_0 \sqsubseteq \pi$ and $\pi_0' \sqsubseteq \pi'$ such that $\mathcal{M}(\pi_0) = \mathcal{M}(\pi_0')$, and π_0' is maximal in the sense that $\forall \pi_1' \sqsubseteq \pi'$, $\pi_0' \sqsubseteq \pi_1'$ implies that π_1' is not a macro-step execution. Similarly, without losing generality we assume π_0 is also maximal. This is illustrated in Figure 3.19.

Then there exists a dataflow transition t_1 along π with the corresponding micro-step π' can not perform, as shown in Figure 3.19. Assume the sequence of transitions ψ further extends from t_1 to the end of π. However since p_i is confluent, there must be a sequence of transitions ψ' extending from π' that converges with π. Further, by p is causally confluent we have $\mathcal{M}(t_1 \cdot \psi) = \mathcal{M}(\psi')$, therefore t_1 must be somewhere along ψ'. Since t_1 should be the only data transition that reads the corresponding inputs, t_1 must be enabled at the end of π' as well. By the assumption of deadlock, the reason that t_1 is disabled must be that there is another transition t_1' along an execution φ_0' with $\varphi_0' \sqsubseteq \varphi$ of some other process $\delta(p_j)$, and that t_1' produces the input that is used by other processes that finally produces the values t_1 waits for. Since t_1' waits for the output of t_1, t_1 and t_1' both involved in a causal cycle. This is illustrated in Figure 3.20.

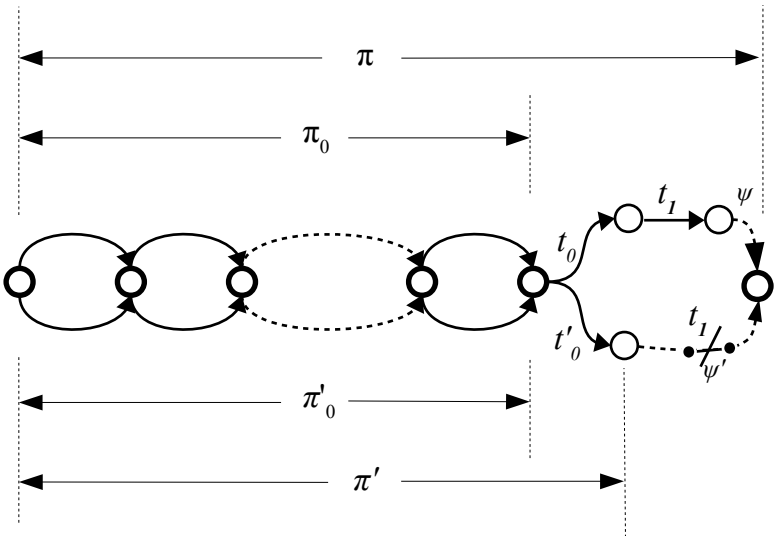


FIGURE 3.19: Illustration of causally confluent.

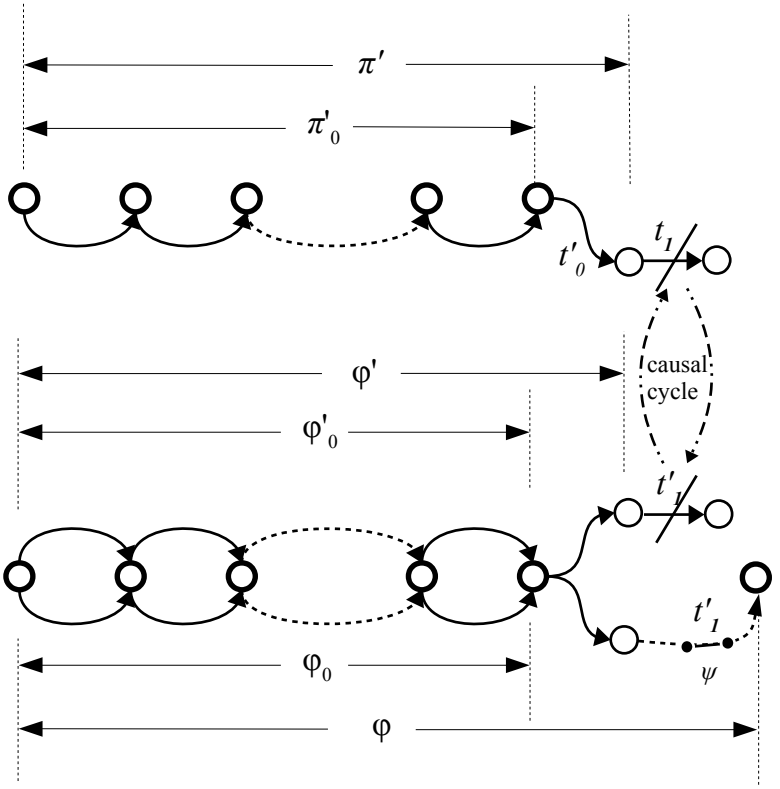


FIGURE 3.20: Illustration of causally confluent.

Without losing generality, we assume t_1, t_1' forms a causal cycle. Assume φ is the macro-step execution of p_j so that when composed with π together there is no deadlock. Since all processes are confluent, for each $\delta(p_i)$ given the same input sequence the same output sequence are produced even the executions might be different. Therefore given the same input sequence $\delta(x)$, $\rho(\varphi') \sqsubseteq \rho(\varphi)$. By the same argument we discussed for t_1 and π, we can conclude that there exists $\varphi_0 \sqsubseteq \varphi$ with $\rho(\varphi_0) = \rho(\varphi_0')$ and t_1' is executed along φ. Further, by causal confluence $\mathcal{M}(\varphi_0) = \mathcal{M}(\varphi_0')$ and $\rho(\varphi') \sqsubseteq \rho(\varphi)$, t_1' must also be executed right in the least macro-step extension along φ_0. Therefore t_1' is executed with t_1. However this contradicts the assumption that t_1 and t_1' forms a causal cycle. □

Theorem 3.23. *For an SPN of multi-rated synchronous processes $P = p_1||p_2||\dots||p_n$, if P is causally correct and each $\delta(p_i)$ causally confluent, then $\forall x \in dom(P), \Phi(\delta(P))(\delta(x)) = \delta(\Phi(P)(x))$.*

Proof. From the proof of theorem 3.22 we can further conclude: for all input sequence $x \in dom(\Phi(P))$ and maximal fair executions $\pi, \pi' \in \Gamma(\delta(P))$ such that $\mathsf{r}(\pi) = \mathsf{r}(\pi') = \delta(x)$, $\mathsf{w}(\pi) = \mathsf{w}(\pi')$. Further, since $\delta(P)$ is deadlock free, π and π' are efficient. Since P is causally correct, for synchronous execution π_0 of P with $\mathsf{r}(\pi_0) = x$, there must be a maximal and efficient execution π_0' of $\delta(P)$ and $\delta(\rho(\pi_0)) = \rho(\pi')$. Thus we have $\mathsf{w}(\pi) = \mathsf{w}(\pi') = \mathsf{w}(\pi_0')$, which proves the theorem. □

Corollary 3.24. *For an SPN of multi-rated synchronous processes $P = p_1||p_2||\dots||p_n$, if P is clock-consistent and causally correct, and each $\delta(p_i)$ causally confluent, then $\forall x \in \Phi(p_1)|_{I_1}||\dots||\Phi(p_n)|_{I_n}, \Phi(\delta(P))(\delta(x)) = \delta(\Phi(P)(x))$.*

Similar to local confluence, we now introduce the definition of local causal confluence. Then we prove that a process is causally confluent if and only it it is locally causally confluent. Since local causal confluence is decidable, this also proves that causal confluence is decidable.

Definition 3.25. For the desynchronization $\delta(p)$ of multi-rated synchronous process p, $\delta(p)$ is *locally causally confluent* if and only if it is locally confluent and satisfies the following condition:

- For all transitions $t_1 := (s_0, R_0) \xrightarrow{l_1} (s_i, R_1), t_2 := (s_0, R_0) \xrightarrow{l_2} (s_j, R_2)$, assume π_1 a least extended macro-step execution of t_1. t_2 converges to t_1 at π_1 by π_2 implies $\mathcal{M}(\pi_1) = \mathcal{M}(\pi_2)$.

Theorem 3.26. *For multi-rated synchronous process p, p is causally confluent if and only if it is locally causally confluent.*

Proof. ($\Leftarrow$) This direction is immediate.

($\Rightarrow$) Assume p is locally causally confluent but not causally confluent. Then there exists macro-step executions π and π' of $LTS_{\delta(p)}$ such that $\rho(\pi) = \rho(\pi')$ but $\mathcal{M}(\pi) \neq \mathcal{M}(\pi')$. Then there must be maximal macro-step executions $\pi_0 \sqsubseteq \pi$ and $\pi_0' \sqsubseteq \pi'$ such that $\rho(\pi_0) = \rho(\pi_0')$ and $\mathcal{M}(\pi_0) = \mathcal{M}(\pi_0')$, but for all macro-step executions π_1 and π_1' with

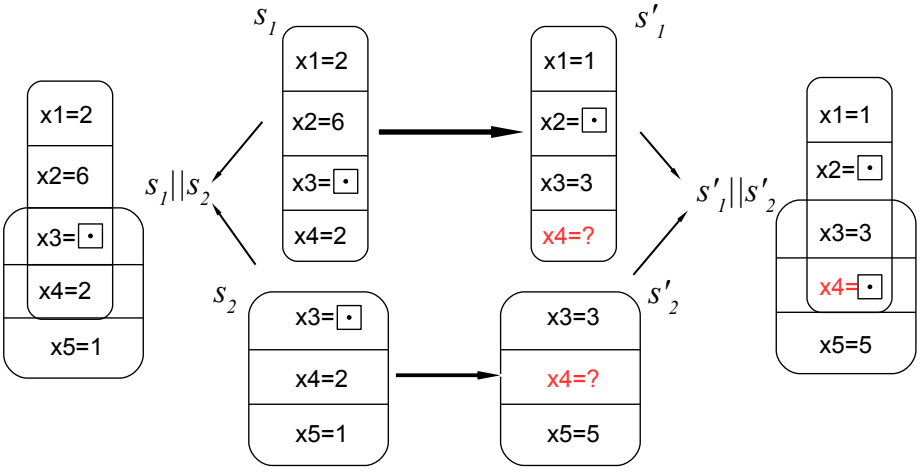
$\pi_0 \sqsubseteq \pi_1 \sqsubseteq \pi$ and $\pi'_0 \sqsubseteq \pi'_1 \sqsubseteq \pi'$, $\mathcal{M}(\pi_1) \neq \mathcal{M}(\pi'_1)$. Assume $\pi_0 = s_0 \xrightarrow{\rho(\pi_0)} s_1$ and $\pi'_0 = s_0 \xrightarrow{\rho(\pi'_0)} s_2$. Since p is confluence, without losing generality we assume that $s_1 = s_2$.

Now assume $t_1 := s_1 \xrightarrow{l} s_i$ with $\pi_0 \cdot t_1 \sqsubseteq \pi$ and $t_2 := s_1 \xrightarrow{l'} s_j$ with $\pi'_0 \cdot t_2 \sqsubseteq \pi'$. Then for the least macro-step extension q_1 of t_1 such that $\pi_0 \cdot q_1 \sqsubseteq \pi$ and the least macro-step extension q_2 of t_2 such that $\pi'_0 \cdot q_2 \sqsubseteq \pi'$, $\mathcal{M}(\pi_0 \cdot q_1) \neq \mathcal{M}(\pi'_0 \cdot q_2)$. However this contradicts with the assumption that p is locally confluent.

□

Comparison of Desynchronization Criterion

In [42], two properties are desired as criterion of correct desynchronization. The first property states that for a single synchronous component, after desynchronization it should be able to resynchronize its corresponding synchronous reactions from the asynchronous environment. The second property states that for the global system (network), the system behavior of the desynchronized system should be flow-equivalent with the original synchronous system's behavior, i.e. the runs of the asynchronous system and the runs of the synchronous system after the absent removed should be the same. In order to satisfy the properties, two sufficient conditions are proposed. For satisfying the first property, endochrony is introduced, which is a sufficient condition of causal confluence as we will see in the next chapter. For the second property, isochrony is introduced, which is comparible with clock-consistency.
However, isochrony has quite a technical definition, and needs some detailed explanation. Intuitively, as shown in the following figure:



Isochrony requires that for any two states s_1, s_2 of two synchronous components, if $s_1||s_2$ exists and $s_1 \to s'_1$ and $s_2 \to s'_2$ valid transition relations of the Synchronous LTSs of the two components, and further more s'_1 and s'_2 coincide on their present

variables (in the above figure it's the value of x_3), then this should imply that s'_1 and s'_2 should coincide on all their shared variables (x_3 and x_4), which means that the existence of $s'_1||s'_2$.

Isochrony is quite a technical definition. The implication assumes the synchronizability of two predecessor states s_1 and s_2, therefore for those that are not synchronizable isochrony does not consider them. Instead clock-consistency is more restricted, as it requires that as long as the un-shared inputs of the two components should always lead to a synchronizable global state. However once predecessors are synchronizable, isochrony then requires the existence of synchronizable successors, which is also implied by clock-consistency. Therefore clock-consistency is a sufficient condition of isochrony. However as our theorems shows, we do not require clock-consistency as long as we restrict the input sequence of the desnychronized system to the flows of the input sequences of the original synchronous system. Different from previous desynchronization criteria, we took implementability as our first goal, therefore we took causality correctness in to consideration. This is different from the preliminary version of the desynchronization main theorem we developed in [61] where we still considered language equivalence. Also, instead of two properties for correct desynchronization, we have only one property: the system after desynchronization should preserve the functional behavior of the original system. This applies to both cases: either the system is a network or is a single component. By functional behavior, we have a clear notion: for the same input sequence of data values, the desynchronized system should be able to derive the same sequence of output data values. Clock-consistency only appears as a stronger requirement: when we want to preserve all input behaviors component wise.

The theoretical benefit of isochrony is that it is both a sufficient and necessary condition for the second property given that each component is endochrony. However it is also arguable that how much sense does the second property make alone, as it only ensures language equivalence. But what is needed in practice is the preservation of I/O functional relation, which is stronger than language equivalence. For example, if for input sequence $x_1 = \boxdot, 1$ we can derive an output sequence $y_1 = 1, 2$ and for input sequence $x_2 = 1, \boxdot$ we have $y_2 = 3, 4$, then language preservation requires that for $x' = 1$, both (x', y_1) and (x', y_2) exist after desynchronization and no new behavior added. However this could mean that the desynchronized system may have the freedom to rebuild either y_1 or y_2 from x', which is still OK but does not preserve the original I/O relation. Indeed, it only makes sense when the first property is ensured. With the first property, it is not easy to prove that the two properties together actually ensures I/O functional relation preservation, which is nevertheless not explicitly stated in previous works.

Another restriction of previous desynchronization criteria is that the models considered are all based on Synchronous LTS, which does not take care of causality issues in micro-step values. This leads to the restriction as discussed before, and brings the "illusion" that the desynchronized system is a GALS system [62, 63],

however in our case the desynchronized components also react in a fully asynchronous style.

Case Study

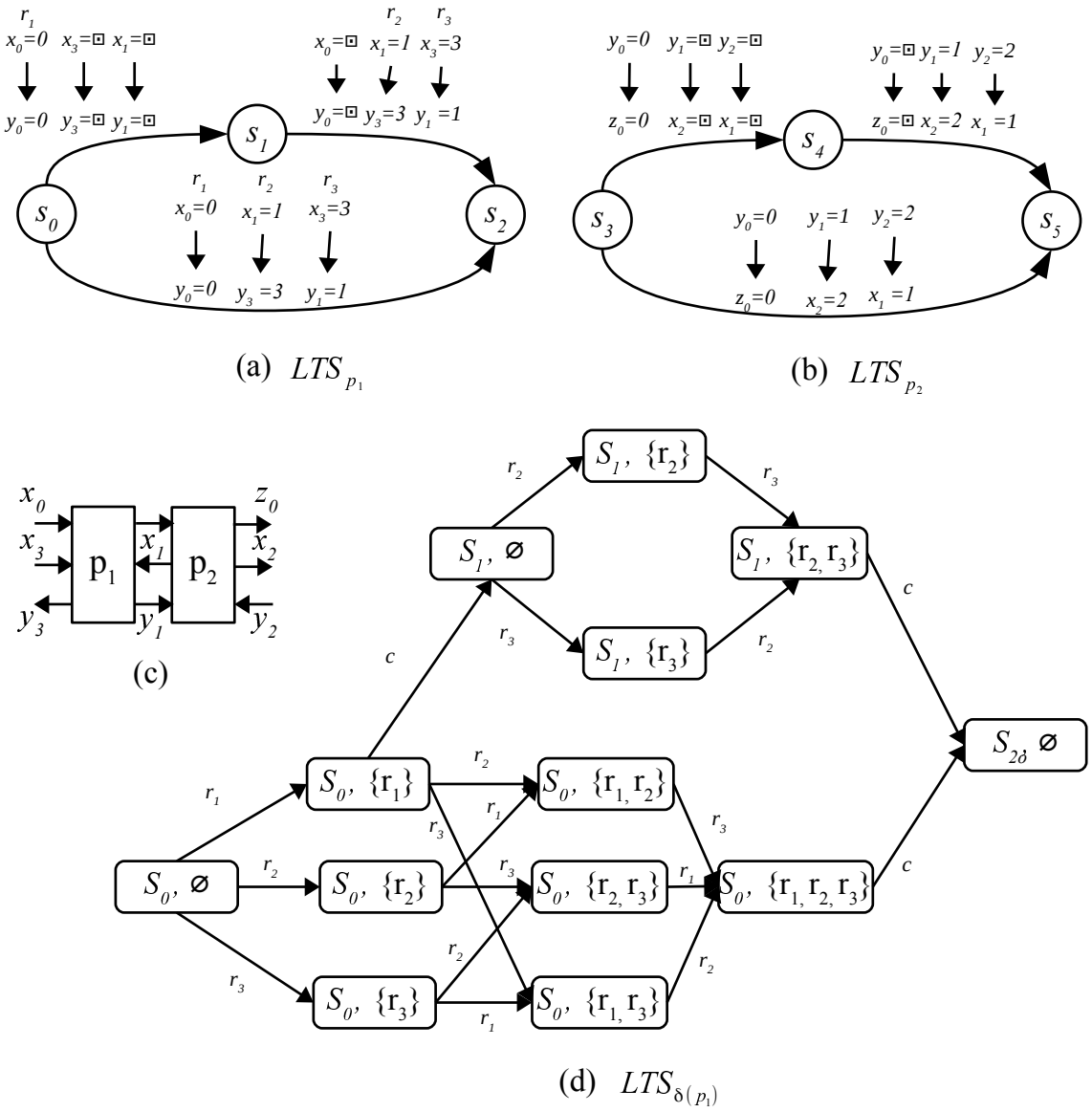


FIGURE 3.21: The LTSs of (a) p_1 and (b) p_2, the SPN of (c) $p_1||p_2$ and the LTS of (d) $\delta(p_1)$.

Consider the example of Figure 3.21, which is a modified version of the example of Figure 3.18. It is not difficult to see that both p_1 and p_2 are causally confluent. The desynchronization of p_1 is shown in Figure 3.21 (d). We can see that the behavior of $\delta(p_1)$ consists of the different permutations of executions of the micro-steps r_1, r_2 and r_3. From the transition systems we can derive the corresponding I/O relations of $\delta(p_1)$

and $\delta(p_2)$, and indeed they are both continuous functions. For example, the function of $\delta(p_1)$: $(y_0, y_3, y_1) = f_{p_1}(x_0, x_1, x_3)$ is defined as follows:

$$\begin{aligned} y_0 = 0 \quad & \text{if } x_0 = 0, \text{ otherwise } y_0 = \epsilon \\ y_3 = 3 \quad & \text{if } x_1 = 1, \text{ otherwise } y_3 = \epsilon \\ y_1 = 1 \quad & \text{if } x_3 = 3, \text{ otherwise } y_1 = \epsilon \end{aligned}$$

and the function of $\delta(p_2)$: $(z_0, x_2, x_1) = f_{p_2}(y_0, y_1, y_2)$ is defined as follows:

$$\begin{aligned} z_0 = 0 \quad & \text{if } y_0 = 0, \text{ otherwise } z_0 = \epsilon \\ x_2 = 2 \quad & \text{if } y_1 = 1, \text{ otherwise } x_2 = \epsilon \\ x_1 = 1 \quad & \text{if } y_2 = 2, \text{ otherwise } x_1 = \epsilon \end{aligned}$$

Given the input sequence of $p_1||p_2$: $x = \{x_0 = 0 \cdot \boxdot, x_3 = \boxdot \cdot 3, y_2 = \boxdot \cdot 2\}$, the corresponding input sequence of present values are $\delta(x) = \{x_0 = 0, x_3 = 3, y_2 = 2\}$, and the least fixpoint computation of the output of $\delta(p_1||p_2)(x)$ is carried out as follows:

$$\begin{aligned} &\text{round 1:} \quad y_0 = 0, y_3 = \epsilon, y_1 = 1, z_0 = \epsilon, x_2 = \epsilon, x_1 = 1; \\ &\text{round 2:} \quad y_0 = 0, y_3 = 3, y_1 = 1, z_0 = 0, x_2 = 2, x_1 = 1; \\ &\text{round 3:} \quad \text{Fixpoint reached.} \end{aligned}$$

3.2.3 Summary

This chapter introduced the main theoretical result of this thesis. In particular, we started from the desynchronization of single-clocked synchronous systems, and formally defined what we mean by desynchronization. We derived theorem 3.5 as the criteria for desynchronization of single-clocked systems. The major problem revealed for single-clocked systems is the preservation of causal relations, as this in turn ensures that the desynchronized system is free from deadlocks.

In order to decouple synchronous components from each other, we further considered desynchronization of multi-clocked synchronous systems, and try to remove the absent signal communications (i.e. $\boxdot$). This however makes theorem 3.5 not efficient anymore, and may change the functional behavior of a component. We models the asynchronous behavior of a desynchronized component with the environment as an infinite two-player game, and further studied the theoretical boundary for correctly desynchronizing a single synchronous component, then showed that it is undecidable to decide in general whether such a correct desynchronization exists. For ASAP winning strategy, however, local confluence is enough and is decidable. Yet we can not rely on local confluence, as it does not preserve causal relations and may leads to deadlocks for the DPN to be created. For this reason, we found a sufficient condition – local confluence – which is also decidable, and indeed preserves the causal relations as well. Theorem 3.23 is therefore the final criteria we used for desynchronization of a synchronous network.

Chapter 4

Verification of Desynchronization Criteria

We presented the theory of correct desynchronization in the last chapter. In particular, for multi-rated SPNs theorem 3.23 states two properties that ensures the behavior preservation of desynchronization. Globally the synchronous process network needs to be causally correct, and locally each synchronous process should be causally confluent. Both properties are either stated on the corresponding labeled transition systems: causally correctness can be verified on the LTS of the SPN while causal confluence can be verified on the LTS of each desynchronized process. In this chapter we show that how these verifications can be implemented and performed efficiently. In particular, while causal confluence is stated on the level of the transition system of the desynchronized process, we try to develop sufficient conditions that can be verified on the level of synchronous transition systems. This is important, as we have seen that after desynchronization the state space of a process is typically largely extended because of the explicit modeling of the intermediate states caused by the execution of interleaving micro-step level data transitions. Similar to the spirit of partial order reduction [31], maintaining the verification on the level of synchronous processes allows us to avoid exploring redundant interleaving executions, so that the verification can be efficiently implemented.

Another contribution of this chapter is that we show that we can derive a multi-rated SPN out of a single-rated SPN. This is in many perspectives desired. Since single-rated SPNs are typically tightly coupled, a direct desynchronization would still result in a logically tightly coupled system. However as we already discussed in section 2.2.1, for many synchronous programs the single-rated behavior is resulted from the completeness of the semantics of the language, even if the functional nature of the system is not tightly coupled. For distributed implementations such completeness needs to be intentionally broken in order to decouple system components. Furthermore, the decoupling we introduced ensures that each synchronous process is causally confluent by construction, therefore no more verification is needed.

For efficient implementation purpose, we introduce a symbolic and semi-symbolic representation of the labeled transitions systems, utilizing an intermediate format of synchronous programs called *synchronous guarded actions*.

4.1 Synchronous Guarded Actions

In this section we introduce a new formalism called *synchronous guarded actions* describing the operational semantics of our synchronous systems. Synchronous guarded actions (or SGAs) are a variant of guarded commands, which is originally invented by Dijkstra [64] and has many important applications because of its simplicity [65–67]. It is also demonstrated that SGAs and clocked-SGAs (a generalizeation of SGAs for multi-rated synchronous systems) can be used as a common formalism to represent various types of concurrent systems [68–73].

4.1.1 Synchronous Guarded Actions for Single-rated Synchronous Systems

A synchronous guarded action $\langle \gamma \implies \mathcal{A} \rangle$ consists of two parts: the guard γ and the action $\mathcal{A}$. Guard γ is a boolean expression and action $\mathcal{A}$ is an assignment that updates the state of the system. As the name suggests, the guarded action is executed synchronously. In the beginning of every cycle (or macro-step), γ is evaluated. When γ evaluates to true, the action $\mathcal{A}$ is executed. An action can be either an *immediate action* $\langle y = \tau \rangle$ which assigns the value evaluated from τ to y for the current macro-step, or a *delayed action* $\mathsf{next}(y) = \tau$ which assigns the value of τ of the current macro-step to y at the next macro-step. The behavior of a synchronous process can be specified by a set of SGAs. Figure 4.1 shows a set of SGAs specifying the synchronous process p.

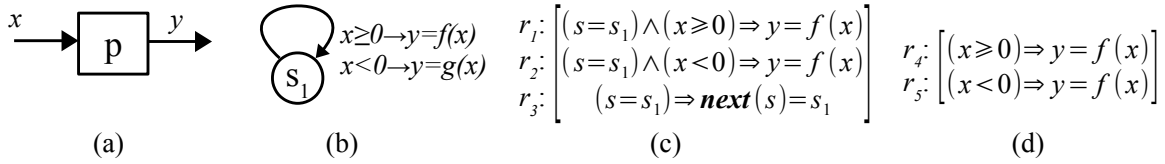


FIGURE 4.1: (a) Synchronous process p, (b) its LTS and (c) its SGAs. (d) Another set of SGAs that specifies the same I/O behavior as (c).

The SGAs of p consists of three SGAs r_1, r_2 and r_3. The actions of r_1 and r_2 update the value of output y, and we call them *dataflow guarded actions* (DSGAs). r_3 updates the local state of p, and we call such guarded actios *control flow guarded actions* (CSGAs). It is obvious that SGAs specifies the LTS of p. In particular, in the beginning of each cycle, r_1 and r_2 test the condition of the local state, reads and test the input x and execute the update of y accordingly. Since x can be in only one case, only one action of r_1, r_2 can be executed. On the other hand, since s_1 is the only state, the action of r_3 is always executed. Since the update of s is redundant, we can also remove this state variable. Figure 4.1 (d) shows the simplified SGAs that has the same I/O behavior as

Figure 4.1 (c), where the DGAs r_4 and r_5 are the only SGAs, and no CGA is needed. From the SGAs it is easy to see that process p is completely specified, since $(s = s_1)$ always evaluates to true and $(x \geq 0) \wedge (x < 0)$ also evaluates to true. Therefore for any input from x, a corresponding output to y can be generated by the SGAs. An SPN can be specified by a set of SGAs as well. In particular, for an SPN $P = p_1 || \ldots p_n$, the SGA of P is simply the union of the SGAs of each p_i.

SGAs v.s. Firing Rules

Note that the SGAs are similar to the firing rules we introduced in section 2.1.2. However the firing rules are only a symbolic way of representing the labeled transition systems for general processes, while the SGAs are specially suited for describing synchronous processes / SPNs. In particular, synchronous guarded actions are always deterministic, because there is no choice among activated guarded actions. Instead, all of the activated actions must be fired. Hence, any system is guaranteed to produce the same outputs for the same inputs. However, forcing conflicting actions to fire simultaneously leads to problems. Instead, firing rules are executed nondeterministically, i.e. when more than one firing rule is enabled, any one may be chosen to execute. As we have seen before, this indeed may lead to different outputs given the same inputs.
Causal problems occur in both formalization. Since different guarded actions require to read inputs in the same time, there may be causal cycles that prevent any of them from firing. Because of the same problem, a DPN may be in deadlock since different firing rules are waiting for the output from each other.

SGAs in Quartz

The implementation of this thesis is based on the Averest framework developed by the Embedded Systems Group in the University of Kaiserslautern [1]. Averest is a framework for the specification, verification, and implementation of reactive systems. It can be used to build various tools targeting software in embedded systems, concurrent programs in general and hardware design. The core of Averest is the synchronous programming language Quartz [20]. The compiler in Averest is able to compile Quartz programs into a set of synchronous guarded actions, which is taken as our starting point of the implementation of desynchronization. As we mentioned in section 2.2.2, Quartz specifies complete systems by adding default actions. This means that for an output variable, if in a cycle no guarded action is executed to give it an assignment, there will be a default value assigned to it. The default value depends on the *storage type* of the variable. In particular, variables of Quartz programs can be divided into *events* and *memorized*. The value of an event variable is only affected by an actively executed guarded action, therefore if it is not assigned by any guarded action, its default value will be a system-defined default value (e.g. false for a boolean variable). Otherwise if it is a memorized value, its default value is the value of the variable in the previous cycle.

[1] http://www.averest.org

From SGAs to scheduled LTS

It is not difficult to show that SGAs can be used to specify a synchronous process as well as an SPN. In this section we show that following some structural rules, SGAs can be equally treated as a symbolic representation of synchronous scheduled LTSs. In Quartz, there are four kinds of variables: *inputs* $\mathcal{V}_{in}$, *outputs* $\mathcal{V}_{out}$, *in/outs* $\mathcal{V}_{io}$ and *locals* $\mathcal{V}_{loc}$. The specialty of an *in/out* variable enables a synchronous process to read its own output, which is forbidden for processes. However this is not a hard restriction, as any such self-loop of a single process p with in/out variable x can be implemented by $p'||cp_x$ where cp_x simply copies x from its input to its output, as shown by Figure 4.2.

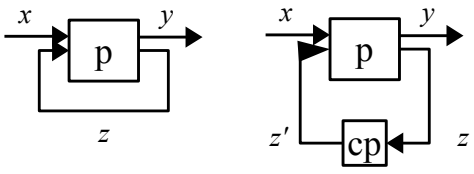


FIGURE 4.2: Treatment of in/out variables.

Another restriction is that no different processes may share the same input variable / output variable / local variable. This means processes are not structurally hierarchical, but are rather parallel composed with each other. Intuitively, for an SPN $P = p_1||p_2||\ldots||p_n$, the corresponding Quartz source program would be the synchronous parallel composition of n Quartz modules, where each module implements one process p_i of P.

Note that the above constraints also defines a valid *decomposition* of a set of SGAs in order to derive an SPN of processes. In particular, any partition of a set of SGAs that satisfies the above constraints forms an SPN, where each partition identifies a synchronous process.

The above restrictions ensure that structural properties of an SPN (and in general a DPN) are respected. Now we show that a set of SGAs can be easily used to derive the state transition relations of a synchronous process. This completes the purpose showing that SGAs are expressive enough to model both synchronous processes as well as SPNs, since an SPN of several synchronous processes is just a partitioned set of SGAs. Without losing generality, we assume that the given SGAs are completely specified, so for any given input assignment all local and output variables are actively assigned by some guarded action. Now for the translation we need to identify the inputs and outputs of the firing rules as well as the local states. We map input (local event and output) variables of the SGAs onto input (output) variables of firing rules, and local memorized variables onto state variables that can be used to encode states of synchronous processes. In case there is no local memorized variable of the SGAs, we can simply introduce a constant as the representation of the only state of the process. Therefore the assignment of input, output and local event variables can be mapped to the label of a transition, the current value assignment of the local memorized variables mapped to the current state and the next assignment of the local memorized variables mapped to the successive state. Note that if a local memorized variable has no active update, then its value is simply remains unchanged in the successive state.

The execution of each SGA is mapped to a micro-step transition. Hence causal preorders can simply be derived from the data dependencies indicated by the guarded actions. For an executed guarded action $\langle \gamma(x_1, \ldots, x_k) \implies y = f(x_{k+1}, \ldots, x_n) \rangle$, the corresponding causal preorder is: $\{x_1 \to y, \ldots, x_n \to y\}$.

Refined Causal Preorders

The causal preorders can be derived in a refined way, utilizing the lazy evaluation of the guards of SGAs. For example, for the following SGA:

$$x_1 \vee x_2 \implies y = \mathsf{true}$$

With the input assignment $x_1 = \mathsf{true}, x_2 = \mathsf{false}$ the action $y = \mathsf{true}$ is executed. By the data dependency we can derive the causal preorder $\{x_1 \to y, x_2 \to y\}$. However either $x_1 = \mathsf{true}$ or $x_2 = \mathsf{true}$ is already able to trigger the action. Therefore we can refine the causal preorder to $\{x_1 \to y\}$. In this way, the causal preorder is refined and could avoid more causal cycles. In general we can do this for all computational operators, and derive the minimal input information needed to perform a computation. However with the following SGAs we can derive the same causal preorder:

$$\begin{aligned} x_1 &\implies y = \mathsf{true} \\ x_2 &\implies y = \mathsf{true} \end{aligned}$$

Therefore by changing the syntax, we can achieve the same effect. For simplicity, we assume that the synchronous guarded actions are syntactically arranged such that the most refined causal preorders can be derived.

It is also not difficult to see that for every synchronous process, there is a set of SGA that specifies its LTS. In particular, we can map the input / output variables of a synchronous process onto the input / output variables of SGAs, and we create a local memorized variable s storing the local state of the process. Therefore for each transition $s_0 \xrightarrow{\hat{l}} s_1$, we can derive the corresponding SGAs $\mathcal{G} = \mathcal{I} \cup \{\langle (s = s_0) \implies \mathsf{next}(s) = s_1 \rangle\}$ where $\{(x_1 = a_1 \to y_j = b_j), \ldots, (x_m = a_m \to y_j = b_j)\} \in \hat{l}$ if and only if there is an immediate SGA $\langle (s = s_0) \wedge (x_1 = a_1) \wedge \cdots \wedge (x_m = a_m) \implies y_j = b_j \rangle \in \mathcal{I}$.

4.1.2 Clocked-Synchronous Guarded Actions for Multi-rated Synchronous Systems

We extend the SGAs of the last section so that multi-rated synchronous processes can be modeled. Remember that the key of generalization of multi-rated synchronous processes is the introduction of $\boxdot$ to the data domain. We introduce a *clock predicate* clk to variables of the SGAs to encode the presence of the variables, so that $\mathsf{clk}(x) = \mathsf{false}$ represents the situation that $x = \boxdot$. In practice we can use $\mathsf{clk}(x)$ as a boolean variable. Therefore a variable x now consists of two parts: the data value and its presence (or clock). For simplicity, we still use the variable's name x to denote the value of x.

However, x is only meaningful when $\mathsf{clk}(x) = \mathsf{true}$. Therefore it makes no sense to calculate $x + 1$ when $\mathsf{clk}(x) = \mathsf{false}$.

As discussed in section 2.2.2, multi-rated synchronous processes are typically partially specified. However the Quartz compiler would add default behaviors to complete the SGAs, which is against the multi-rated nature. In the following example Figure 4.3, we show that the completion may further destroy the decoupling by desynchronization, in the sense that no $\boxdot$ can be removed.

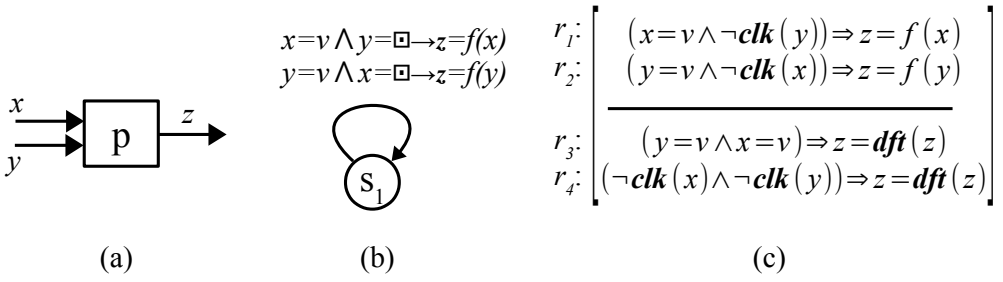


FIGURE 4.3: A multi-rated synchronous process (a), its LTS (b) and completed SGAs (c).

In the example, we assume that the domain of both x and y is the singleton $\{v\}$ where v is a constant. $dft(x)$ in an SGA denotes the system-defined default value of x. The behavior of process p is quite simple. In each cycle, based on the presence of x or y, z is computed using the present value correspondingly. Furthermore, exactly one of x and y is present, corresponding to one guard of r_1 and r_2. We call these SGAs that are directly generated from the source code *explicit*. However, such a system is not complete, since the case when both x and y are present and both are absent are not specified. After completion, r_3 and r_4 will be added to the SGAs, where a default value of x is produced for the input cases not specified by r_1 and r_2. We call such guarded actions added by completion *implicit*. More generally, we can derive the following predicate that holds during each cycle for each output variable y_k:

$$y_k = \begin{pmatrix} \mathsf{case} \\ \gamma_1 : \tau_1 \\ \vdots \\ \gamma_m : \tau_m \\ \mathsf{else}\ dft(y_k) \end{pmatrix}$$

where each pair of (γ_i, τ_i) corresponds to an explicit SGA, and the else part corresponds to the implicit SGAs. Since the SGAs are completely specified, there must be a SGA $\bigwedge_i \neg\mathsf{clk}(x_i) \implies y = \tau$ where the guard corresponds to the case when all inputs are absent. This prevents $\boxdot$ from removed if for the corresponding action $y = \tau$, $\tau \neq \boxdot$. Since if we remove the $\boxdot$ otherwise, the process would never know exactly how many value of y should be produced. A first solution is to define $def(y) := \boxdot$. However this does not solve the whole problem. Because for what is worse, the completion of SGAs actually changed the synchronous behavior of the process. Consider r_3 of Figure 4.3 (c).

Its guard also allows the process to consume a value from both x and y and produce a value of z, where previously in the partially specified process producing a value of z only costs one value from either x or y. Without the information of the additional $\boxdot$, there is no way for the process to distinguish r_3 from r_1 and r_2. However, for the original system of Figure 4.3 (b), there is no problem to remove the $\boxdot$. Since although the process can not decide deterministically which of r_1 and r_2 to fire first, actually firing them in either order is fine, as they all produce the value $f(v)$ to output channel y.

Based on the above argument, we demand that the added default part by the Quartz compiler be ignored. As a result, the LTS as well as the allowed input sequence is totally defined by the explicit guarded actions, which is our intention. The effect is that the composition of modules might not be compatible anymore. Therefore when compatibility is demanded, we need to assume that the composition of the partially defined modules should be compatible.

Finally, for simplicity we assume that $\mathsf{clk}(x)$ can only appear in a guard or the right-hand-side of an action. Therefore no guarded action can set a variable to absence. Instead, a variable is absent if none of its guard evaluates to true. This does not mean that we allow all additional input sequence that is accepted by the implicit guards. Instead, if for a particular cycle the allowed input values is φ and the condition for any action to output y is ψ, then ψ implies $\mathsf{clk}(y)$ is true and $\varphi \wedge \neg\psi$ implies $\mathsf{clk}(y)$ is false, i.e. $\varphi \wedge (\mathsf{clk}(y) \leftrightarrow \psi)$ is an invariant for the cycle.

4.1.3 Symbolic Representation of Labeled Transition Systems

We now develop the symbolic representation of the LTS of a synchronous process from its corresponding set of SGAs. This symbolic representation is the target of our analysis later, thus sets the basis of an efficient analysis of the correct criteria for desynchronization. Before we start the construction of the LTS, we first do some further optimizations for the SGAs. In particular, as shown in [20], we can assume that for each variable $x \in \mathcal{V}_{loc} \cup \mathcal{V}_{out}$ there are either only immediate assignments $(\gamma_1, x = \tau_1), \ldots, (\gamma_m, x = \tau_m)$ or only delayed assignments $(\gamma'_1, \mathsf{next}(x) = \tau'_1), \ldots, (\gamma'_n, \mathsf{next}(x) = \tau'_n)$. If that should not be the case, then one can introduce a new local variable x_c (called the carrier variable of x) that captures the delayed assignments, i.e., these are replaced with $(\gamma'_1, \mathsf{next}(x_c) = \tau'_1), \ldots, (\gamma'_n, \mathsf{next}(x_c) = \tau'_n)$ and adding the new guarded action $(\bigwedge_{i=1}^{m} \neg\gamma_i, x = x_c)$. In the following, we can therefore say that a variable $x \in \mathcal{V}_{loc} \cup \mathcal{V}_{out}$ is an immediate variable or a delayed variable (depending[2] on whether it has immediate or delayed assignments).

Now assume for a synchronous process p, its corresponding SGAs are shown as Figure 4.4. Then the symbolic representation of the transition system can be constructed by Figure 4.5. In particular, it is easy to see that each SGA can be treated as an implication stating that once the guard evaluates to true, the equivalence of the left-hand-side and right-hand-side of an action holds. This applies to both immediate and delayed assignments, where for a delayed assignment, a special variable can be used to encode

[2]For $x \in \mathcal{V}_{loc} \cup \mathcal{V}_{out}$, we assume that there is at least one assignment.

$$\left\{\begin{array}{c} \langle \gamma_1^1 \Rightarrow x_1 = \tau_1^1 \rangle, \ldots, \langle \gamma_{n_1}^1 \Rightarrow x_1 = \tau_{n_1}^1 \rangle, \\ \vdots \\ \langle \gamma_1^m \Rightarrow x_m = \tau_1^m \rangle, \ldots, \langle \gamma_{n_m}^1 \Rightarrow x_m = \tau_{n_m}^1 \rangle, \\ \langle \rho_1^1 \Rightarrow \mathsf{next}(y_1) = \eta_1^1 \rangle, \ldots, \langle \rho_{p_1}^1 \Rightarrow y_1 = \eta_{p_1}^1 \rangle, \\ \vdots \\ \langle \rho_1^q \Rightarrow \mathsf{next}(y_q) = \eta_1^q \rangle, \ldots, \langle \rho_{p_q}^q \Rightarrow y_q = \eta_{p_q}^q \rangle, \end{array}\right\}$$

FIGURE 4.4: The set of SGAs of process p.

$\mathsf{next}(x)$. $\mathcal{B}_x$ encodes all the allowed input assignments of any macro-step for triggering an assignment for x. As the SGAs specified a partial function, the valid input assignment of the system is hence encoded by $\bigwedge_{x \in \mathcal{V}_{loc} \cup \mathcal{V}_{out}} \mathcal{B}_x$.

Given $\mathcal{R}$, the transition relations of the LTS can be encoded as: $\mathcal{R}_{\|} :\Leftrightarrow (\mathcal{R} \wedge \mathsf{next}(\mathcal{R}))$. $\mathsf{next}(\mathcal{R})$ refers to the successive states of the transition system, where each variable x in $\mathcal{R}$ is replaced by its next version: $\mathsf{next}(x)$ in $\mathsf{next}(\mathcal{R})$. Technically, there would be nested next operators inside $\mathsf{next}(\mathcal{R})$ (for $\mathcal{T}_x$ and $\mathcal{C}_x$ of delayed variables), therefore we need to pull them out separately, so that they are not influenced by the outer next operator. A state s assigns values to the variables $\mathcal{V}$ of the considered system, and s' assigns to the next variables $\mathcal{V}'$. Therefore $\mathcal{R}_{\|}$ relates two states s_1, s_2 whenever $s_1 \wedge s_2'$ satisfies $\mathcal{R}$. We therefore write $(s_1, s_2) \in \mathcal{R}_{\|}$ if there is a transition between s_1 and s_2 and for a variable $x \in \mathcal{V}$, we write $s_1(x)$ to denote its value in state s_1 (same for clocks $s_1(\mathsf{clk}(x))$). Note that the symbolic representation we build is equally the *Kripke Structure* [74] of the synchronous process. In particular, the state encoded by the symbolic representation is *not* a state of an LTS, but rather encodes both the state and label of the LTS. Therefore when we mention the state of our symbolic representation, we mean the state of the Kripke Structure.

The initial condition $\mathcal{I}_x$ of variable x is defined by the system. Therefore it is encoded as $(\mathsf{clk}(x) \leftrightarrow b) \wedge (x \leftrightarrow v)$, where b is the boolean constant indicating the presence of x and v the constant value assigned to x. The initial condition of the entire system is the conjunction $\mathcal{I}_{\|} :\Leftrightarrow \bigwedge_{x \in \mathcal{V}_{loc} \cup \mathcal{V}_{out}} \mathcal{I}_x$.

As an example, consider the Quartz code shown in Figure 4.6 which shows the synchronous module that assigns its output `y` one of the inputs `x2` or `x3` depending on whether the first input `x1` is true or not. It therefore has the set of variables: $\mathcal{V}_{in} = \{\mathtt{x1}, \mathtt{x2}, \mathtt{x3}\}, \mathcal{V}_{out} = \{\mathtt{y}\}$ and the following guarded actions:

- `clk(x1) & clk(x2) & x1 -> y=(x2, true)`
- `clk(x1) & clk(x3) & !x1 -> y=(x3, true)`

The valid behavior $\mathcal{R}$ is therefore the conjunction of the following formulas:

- `clk(y) = clk(x1) & (clk(x2) & x1 | clk(x3) & !x1)`
- `clk(x1) & clk(x2) & x1 -> y=x2`

The symbolic description of valid behaviors of a variable $x \in \mathcal{V}_{loc} \cup \mathcal{V}_{out}$ having clocked guarded actions $\{\langle \gamma_1 \Rightarrow x = \tau_1 \rangle, \ldots, \langle \gamma_m \Rightarrow x = \tau_m \rangle\}$ is defined by $\mathcal{R}_x$ as follows:

$$\mathcal{B}_x :\Leftrightarrow \underbrace{\bigvee_{i=1}^{m} \gamma_i}_{\text{valid behaviors}} \quad , \mathcal{T}_x :\Leftrightarrow \underbrace{\left(\bigwedge_{i=1}^{m} \gamma_i \implies x = \tau_i \right)}_{\text{explicit actions}},$$

$$\mathcal{C}_x :\Leftrightarrow \underbrace{\left(\mathsf{clk}(x) = \left(\bigvee_{i=1}^{m} \gamma_i \right) \right)}_{\text{clock def.}}, \mathcal{R}_x :\Leftrightarrow \mathcal{B}_x \wedge \mathcal{C}_x \wedge \mathcal{T}_x$$

The symbolic description of valid behaviors of a variable $x \in \mathcal{V}_{loc} \cup \mathcal{V}_{out}$ having clocked guarded actions $\{\langle \gamma'_1 \Rightarrow \mathsf{next}(x) = \tau'_1 \rangle, \ldots, \langle \gamma'_n \Rightarrow \mathsf{next}(x) = \tau'_n \rangle\}$ is defined by $\mathcal{R}_x$ as follows:

$$\mathcal{B}_x :\Leftrightarrow \underbrace{\bigvee_{i=1}^{n} \gamma'_i}_{\text{valid behaviors}} \quad , \mathcal{T}_x :\Leftrightarrow \underbrace{\left(\bigwedge_{i=1}^{n} \gamma'_i \implies \mathsf{next}(x) = \tau'_i \right)}_{\text{explicit actions}},$$

$$\mathcal{C}_x :\Leftrightarrow \underbrace{\left(\mathsf{next}(\mathsf{clk}(x)) = \left(\bigvee_{i=1}^{n} \gamma'_i \right) \right)}_{\text{clock def.}}, \mathcal{R}_x :\Leftrightarrow \mathcal{B}_x \wedge \mathcal{C}_x \wedge \mathcal{T}_x$$

With the help of the above definitions, we can define the valid behavior of the synchronous system as follows:

$$\mathcal{R} :\Leftrightarrow \left(\bigwedge_{x \in \mathcal{V}_{loc} \cup \mathcal{V}_{out}} (\mathcal{T}_x \wedge \mathcal{C}_x) \wedge \bigvee_{x \in \mathcal{V}_{loc} \cup \mathcal{V}_{out}} \mathcal{B}_x \right)$$

FIGURE 4.5: Valid behavior of a synchronous system.

```
module site(clocked bool ?x1,?x2,?x3,!y) {
    loop{
        if(clk(x1) & clk(x2) & x1) {
            y = (x2,true);
        }
        if(clk(x1) & clk(x3) & !x1) {
            y = (x3,true);
        }
        pause;
    }
}
```

FIGURE 4.6: Quartz Code of Sequential If-then-else.

- `clk(x1) & clk(x3) & !x1 -> y=x3`
- `clk(x1) & (clk(x2) & x1 | clk(x3) & !x1)`

and the transition relations $\mathcal{R} \wedge \mathsf{next}(\mathcal{R})$. Without introducing new notations, we also use $\mathcal{R}_{\|}$ to denote the transition system that is induced from $\mathcal{I}_{\|}$ and $\mathcal{R}_{\|}$.

By now we discussed how to derive the symbolic representation of the LTS of a single synchronous process. By definition 2.26, the symbolic transition relation of the LTS for an SPN $P := p_1 || \ldots || p_n$ is then simply the conjunction of the symbolic transition relations of all processes of P:

$$\mathcal{R}_{||}^{P} :\Leftrightarrow \bigwedge_{i=1}^{n} \mathcal{R}_{||}^{p_i}.$$

This is similar for the initial condition of P.

4.2 Verification of Global Properties

In the last section we discussed how to build the symbolic representation of the transition system. This is the basis to the verification of our synchronous process. We show in this section that how the global properties like clock consistency and causal correctness can be formulated and verified against the LTS we built. Before going into the details, we first introduce some basic symbolic constructions. They can be used as tools for our bigger verification goal. In particular, we introduce how to compute existential successors. The existential successors of a (set of) state is defined as follows:

$$\mathsf{succ}_{\exists}(Q_1) := \{s_2 \mid \exists s_1, (s_1, s_2) \in \mathcal{R} \wedge s_1 \in Q_1\}$$

Where R is the transition relation. Similarly, there are symbolic methods to compute universal successors, existential predecessors and universal predecessors. We won't be needing them, therefore we omit their definitions. They are the basic building blocks to many verification algorithms [75–77]. More interested readers may reference [74] for details.

Reachable States

One of the most classical problem in verification is to compute the reachable states of a system. By computing the reachable states, we then are able to prove clock-consistency, endochrony (which will be introduced later) and other safety properties of our synchronous systems.The basic idea of computing reachable states $\mathcal{R}$ of LTS_p is simply described by the following inductive definition:

- Let s_0 be the initial state of LTS_p, then $s_0 \in \mathcal{R}$;
- If $\exists s', (s, l, s') \in \mathcal{T}$ and $s \in \mathcal{R}$, then $s' \in \mathcal{R}$;

The iteration of the above computation will terminate if the LTS is finite. Then for our symbolic representation with initial state encoded by $\mathcal{I}$ and transition relation $\mathcal{R}$, the formula $\Phi_{\mathcal{R}}$ representing the reachable states can be calculated as follows:

- Initially let $\Phi_{\mathcal{R}} := \mathcal{I}$; let $\varphi := \mathcal{I}$
- Update the successive states $\varphi' := \mathsf{succ}_{\exists}(\varphi)$, then $\Phi_{\mathcal{R}} := \Phi_{\mathcal{R}} \vee \varphi$, where φ is the un-primed version of φ';

Encoding Micro-step Information

Since we will be arguing about micro-steps within a macro-step, we would like to encode the execution of micro-steps also into our symbolic representation. In particular, for each SGA $\langle \gamma \Rightarrow \mathcal{A} \rangle$ we introduce a fresh flag variable g and define: $g \Leftrightarrow \gamma$. For the symbolic transition relation $\mathcal{R}$ we further add the flags of SGAs:

$$\mathcal{R}' := \mathcal{R} \wedge (\bigwedge_i (g_i \Leftrightarrow \gamma_i)).$$

Therefore whenever g evaluates to true we know that the corresponding SGA is executed.

4.2.1 Verification of Clock Consistency

Comparison of LTSs

By definition 2.35, we can deduce that the processes are compatible (i.e. the SPN is clock-consistent) if and only if the parallel composition of transition systems encoding the input sequences of each process is equivalent to the transition system encoding the input sequences of the SPN. This requires us to build the transition systems encoding only the input sequences, as well as comparing transition systems.

As we are already able to compute the symbolic representation of each reachable state, it is not difficult to extract the projection of the state over a particular set of variables. In particular, let φ_s be the formula encoding a state s, then the formula encoding $s|_{\mathcal{V}}$ is: $\exists(\mathcal{V}_p \backslash \mathcal{V}).\varphi_s$ where $\mathcal{V}_p$ is the set of variables of p. Now assume given SPN $P = p_1 || \ldots || p_n$, and $\mathcal{V}_i = I_P \cup \mathcal{V}_{p_i}$ (where I_P is the input variables of P). Then we can formulate the verification of clock consistency based on the computation of reachable states as follows:

- Check
$$\bigwedge_{i=1}^{n} \mathcal{I}_i|_{\mathcal{V}_i} \Leftrightarrow \mathcal{I}_P|_{I_P}$$
where $\mathcal{I}_i$ is the initial states of p_i and $\mathcal{I}_P$ the initial states of P.
- For each current reached set of states $\mathcal{S}_i$ of p_i and $\mathcal{S}_P$ of P, check
$$\bigwedge_{i=1}^{n} \mathsf{succ}_{\exists}(\mathcal{S}_i)|_{\mathcal{V}_i} \Leftrightarrow \mathsf{succ}_{\exists}(\mathcal{I}_P)|_{I_P}$$

Check for False States

The verification of clock-consistency can be easier, once we noticed the following fact: For a state where two processes do not agree on the value (or clock) of some variable x, the symbolic representation of this state would be false. Therefore instead of comparing the two LTSs one simply needs to concentrate on the reachable states of the SPNs and check if there is a possibility to reach a false state. The following example shows this insight.

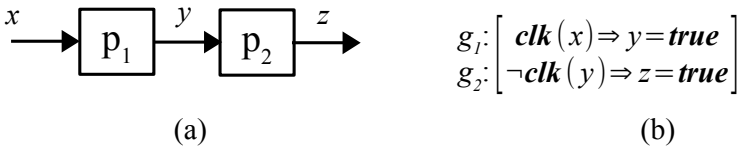


FIGURE 4.7: (a) The SPN $p_1||p_2$, (b) SGAs of p_1 and p_2.

Figure 4.7 (a) shows an SPN of two processes p_1 and p_2 and the each process consists of only one SGA g_i. x, y and z are all boolean event variables. It is obvious that the two processes are not compatible, since $\mathsf{clk}(y)$ is true, but process p_2 only accepts when $\mathsf{clk}(y)$ is false. There is only one symbolic (set of) state, and following the definitions of Figure 4.5 we have:

$$\left\{\begin{array}{c}\mathcal{B} :\Leftrightarrow (\mathsf{clk}(x) \wedge \neg\mathsf{clk}(y)) \\ \mathcal{D} :\Leftrightarrow \mathsf{true}\end{array}\right., \left\{\begin{array}{c}\mathcal{T}_y :\Leftrightarrow (\mathsf{clk}(x) \implies y) \\ \mathcal{C}_y :\Leftrightarrow (\mathsf{clk}(y) \leftrightarrow \mathsf{clk}(x))\end{array}\right., \left\{\begin{array}{c}\mathcal{T}_z :\Leftrightarrow (\neg\mathsf{clk}(y) \implies z) \\ \mathcal{C}_z :\Leftrightarrow (\mathsf{clk}(z) \leftrightarrow \neg\mathsf{clk}(x))\end{array}\right.$$

Given the above basic components the formula is:

$$\mathcal{B} \wedge (\mathcal{T}_y \wedge \mathcal{C}_y) \wedge (\mathcal{T}_z \wedge \mathcal{C}_z) :\Leftrightarrow \left(\begin{array}{c}(\mathsf{clk}(x) \wedge \neg\mathsf{clk}(y)) \wedge \\ (\mathsf{clk}(x) \implies y) \wedge (\mathsf{clk}(y) \leftrightarrow \mathsf{clk}(x)) \wedge \\ (\neg\mathsf{clk}(y) \implies z) \wedge (\mathsf{clk}(z) \leftrightarrow \neg\mathsf{clk}(x))\end{array}\right)$$

Notice that by $\mathsf{clk}(x) \leftrightarrow \mathsf{true}$ and $\mathsf{clk}(y) \leftrightarrow \mathsf{clk}(x)$ we know that $\mathsf{clk}(y) \leftrightarrow \mathsf{true}$, however this contradicts with $\neg\mathsf{clk}(y)$ in $\mathcal{B}$ – this is exactly the problem of incompatibility. Therefore the whole formula equals to false. However note that a false state can also be reached under other situations. For example, if the system suffers write conflicts. In this case, it is rather the value of some local / output variable that is not consistent. For example the SGAs: $\{\gamma \implies y = \mathsf{true}, \gamma \implies y = \mathsf{false}\}$ under the same condition assigns y to different values, therefore would cost the state formula equal to false as well.

4.2.2 Verification of Causal Correctness

In section 2.2.1 we already introduced the techniques synchronous language compilers deal with causality. Here we briefly show that the verification of causal correctness can be implemented based on the reachable state computation. In particular, we assume a (set of state) is encoded by formula φ_s. Following standard boolean logic, this formula represents all variable assignments that satisfy this states. Therefor for the presence (i.e.

clock predicate) as well as data value of each local and output variable can be derived from a variable assignment s. However, as shown in the same section, the outputs might not be computed operationally because of causal cycles. Instead, constructive logic can be used to reason about the computation of outputs step-by-step using only the already known information, i.e. in a constructive way. This procedure follows the causal preorders (provided that our SGAs syntactically reflects the constructive logic) and therefore outputs can be successfully constructed only when there is no causal cycle. There are plenty of works done in this field previously, the interested readers may reference works of []. We won't go into details of causal analysis, but we would like to briefly show the basic procedure and give an example to show how the verification works.

The procedure of causal analysis is based on the reachable state computation as well. In particular, for each set of states reached we derive a corresponding formula. Similar to verification of clock consistency, by existential quantification over non-input variables we can derive the formula encoding the allowed inputs (both clock and data value), in particular for each input variable, of the set of states. Then we perform a fixpoint computation by starting from setting only inputs as the known formula, while both clocks and values of local variables are unknown. By evaluating the valid behaviors, clock-constraints as well as guarded actions via constructive logic, we update the status of the clocks and data values for the local and output variables. This is repeated until no more change of status for the local /outputs. If all local/outputs status are known (as some boolean formula), the corresponding macro-steps of the set of states are causally correct. More importantly, from the example it is straightforward to see that the verification of clock-consistency can be naturally embedded in the verification of causal correctness.

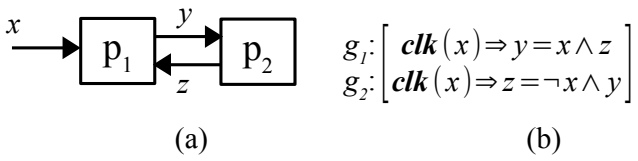


FIGURE 4.8: (a) The SPN $p_1||p_2$, (b) SGAs of p_1 and p_2.

Figure 4.8 (a) shows an SPN of two processes p_1 and p_2 and the each process consists of only one SGA g_i. x, y and z are all boolean event variables. There is only one symbolic (set of) state, and following the definitions of Figure 4.5 we have:

$$\begin{cases} \mathcal{B} :\Leftrightarrow \mathsf{clk}(x) \\ \mathcal{D}_x :\Leftrightarrow (\mathsf{clk}(x) \implies \mathsf{clk}(x)) \\ \mathcal{D}_y :\Leftrightarrow (\mathsf{clk}(x) \implies \mathsf{clk}(y)) \\ \mathcal{D}_z :\Leftrightarrow (\mathsf{clk}(x) \implies \mathsf{clk}(z)) \end{cases},$$

$$\begin{cases} \mathcal{T}_y :\Leftrightarrow (\mathsf{clk}(x) \implies (y \leftrightarrow x \wedge z)) \\ \mathcal{C}_y :\Leftrightarrow (\mathsf{clk}(y) \leftrightarrow \mathsf{clk}(x)) \end{cases}, \begin{cases} \mathcal{T}_z :\Leftrightarrow (\mathsf{clk}(x) \implies (z \leftrightarrow \neg x \wedge y)) \\ \mathcal{C}_z :\Leftrightarrow (\mathsf{clk}(z) \leftrightarrow \mathsf{clk}(y)) \end{cases}$$

Now we may start the fixpoint computation of local / output variables based on the maximal ternary extended constructive logic introduced in [78], where $\varphi_{\hat{x}}$ denotes the evaluated formula for $\mathsf{clk}(x)$, φ_x the formula for x, $\perp$ the symbol of unknown and $\top$

for inconsistent value. Also, the implementation is realized by dual-rail encoding of the data domain [78]:

- Round 1

$$\left\{\begin{array}{c}\varphi_{\hat{x}} = (\mathsf{true}, \mathsf{false}) \\ \varphi_x = (\neg x, x)\end{array}\right., \left\{\begin{array}{c}\varphi_{\hat{y}} = (\mathsf{false}, \mathsf{false}) \\ \varphi_y = (\mathsf{false}, \mathsf{false})\end{array}\right., \left\{\begin{array}{c}\varphi_{\hat{z}} = (\mathsf{false}, \mathsf{false}) \\ \varphi_z = (\mathsf{false}, \mathsf{false})\end{array}\right.,$$

- Round 2

$$\left\{\begin{array}{c}\varphi_{\hat{x}} = (\mathsf{true}, \mathsf{false}) \\ \varphi_x = (\neg x, x)\end{array}\right., \left\{\begin{array}{c}\varphi_{\hat{y}} = (\mathsf{true}, \mathsf{false}) \\ \varphi_y = (\mathsf{false}, x)\end{array}\right., \left\{\begin{array}{c}\varphi_{\hat{z}} = (\mathsf{true}, \mathsf{false}) \\ \varphi_z = (\mathsf{false}, \neg x)\end{array}\right.,$$

- Round 3

$$\left\{\begin{array}{c}\varphi_{\hat{x}} = (\mathsf{true}, \mathsf{false}) \\ \varphi_x = (\neg x, x)\end{array}\right., \left\{\begin{array}{c}\varphi_{\hat{y}} = (\mathsf{true}, \mathsf{false}) \\ \varphi_y = (\mathsf{false}, x \vee \neg x) \\ = (\mathsf{false}, \mathsf{true})\end{array}\right., \left\{\begin{array}{c}\varphi_{\hat{z}} = (\mathsf{true}, \mathsf{false}) \\ \varphi_z = (\mathsf{false}, \neg x \vee x) \\ = (\mathsf{false}, \mathsf{true})\end{array}\right.,$$

- Round 4 = Round 3, fixpoint reached.

Now we explain one step from round 1 to round 2 in detail, and the rest steps follows similarly. Before start, we check if $\mathcal{B} = \mathsf{false}$ or not. In case it is, the process is then clock-inconsistent. For the first round, initially we can fix the value of $\mathsf{clk}(x)$ to be true, since $\mathcal{B} \Rightarrow \mathsf{clk}(x)$ is trivially true. Hence the dual rail encoding $(\mathsf{true}, \mathsf{false})$ for $\varphi_{\hat{x}}$. The value of x is simply encoded by the dual rail boolean variables $(\neg x, x)$. Also since y and z are local variables, their clock as well as data values are all unknown, hence the dual rail encoding $(\mathsf{false}, \mathsf{false})$. Now to derive round 2, from $\mathcal{C}_y$ we deduce that $\mathsf{clk}(y) = \mathsf{true}$ and similarly $\mathsf{clk}(z) = \mathsf{true}$. Now if we check their values, we can see that they are consistent with $\mathcal{D}_y$ and $\mathcal{D}_z$. In case this is not the case, we immediately derive a clock-inconsistent case. We next go on with the guarded actions to compute values of y and z. By $\mathcal{T}_y$ and $\mathcal{B} \Rightarrow \mathsf{clk}(x)$ we know that y should be assigned as $x \wedge z$, and following the dual-rail encoding we have $\varphi_x \wedge \varphi_z = (\neg x \wedge \mathsf{false}, x \vee \mathsf{false}) = (\mathsf{false}, x)$. Similarly, we have $\varphi_z = (\mathsf{false}, \neg x)$.

After the fixpoint is reached, we can see that both y and z are false, therefore no variable is unknown, and the process is causally correct. In the mean time we also verified the evaluation of the state formula, so that it is not a false state. Therefore the process is also clock-consistent.

The verification of causal analysis has been implemented by the Quartz compiler, and can be performed very efficiently.

4.3 Verification of Local Properties

Following theorem 3.5 and theorem 3.23, we need to further ensure that each process is either determinism or causally confluent to guarantee the correctness of the desynchronization for single or multi-rated SPNs. Since determinism and causal confluence are required for each individual synchronous process, we call them *local properties*.In this section we discuss how these local properties are verified.

4.3.1 Verification of Determinism

By theorem 3.5 each synchronous process is required to be determinism. Assume given a synchronous process p and its SGAs are $\mathcal{G}$. By definition 2.33, the process should satisfy three conditions, which can be verified as follows:

- $\forall y \in \mathcal{V}_p \setminus I_p, \forall \langle \gamma_i \Rightarrow y = \tau_i \rangle \in \mathcal{G}$ and $\langle \gamma_j \Rightarrow y = \tau_j \rangle \in \mathcal{G}$, check if $\models \neg(\gamma_i \wedge \gamma_j)$. Similarly, the property corresponds to delayed assignments of y should be verified as well.
- $\forall g : \langle \gamma \Rightarrow y = \tau \rangle \in \mathcal{G}$ where g is the flag of the SGA, $\forall \mathcal{S} \in \mathcal{R}(p)$ where $\mathcal{R}(p)$ the reachable set of states of p, $(\mathcal{I}(\mathcal{S}) \models \gamma) \Rightarrow g$ where $\mathcal{I}(S) :\Leftrightarrow \exists O_p.\mathcal{S}$.
- $\forall \mathcal{S}_1, \mathcal{S}_2 \in \mathcal{R}(p)$ where $\mathcal{R}(p)$ the reachable set of states of p, $(\mathcal{I}(\mathcal{S}_1) \Leftrightarrow \mathcal{I}(\mathcal{S}_2)) \Rightarrow (\mathcal{S}_1 \Rightarrow \mathcal{S}_2)$ where $\mathcal{I}(S) :\Leftrightarrow \exists O_p.\mathcal{S}$.

where $\mathcal{S} \models \psi$ states that the state $\mathcal{S}$ satisfies the formula ψ. Its semantics is that for all variable assignment l that satisfies the formula $\mathcal{S}$, l should also satisfy the formula ψ. This can be verified by checking if $\mathcal{S} \Rightarrow \psi$ is valid. We briefly explain the above verified properties:

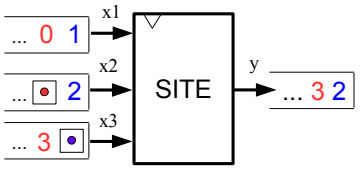
- The first property ensures that at any state, two (both immediate or both delayed) SGAs that assign to the same variable should never be enabled at the same time. Therefore there is only a single SGA assigns to a variable, i.e. no write conflict.
- The second property checks if for an SGA $\langle \gamma \Rightarrow \mathcal{A} \rangle$, its guard γ is enabled at state $\mathcal{S}$ (i.e. $\mathcal{I}(\mathcal{S}) \models \gamma$), then its flag g is true, ie. it is fired in this state. Therefore the corresponding micro-step is faithful to the label of $\mathcal{S}$.
- The finial property checks if for the same local state and same input values, the same actions are taken. If so, they must arrive at the same successive state.

4.3.2 Endochrony

In section 3.2.2 we discussed the desynchronization of multi-rated SPNs. By theorem 3.22 we need to ensure that the desynchronization of each synchronous process

$$r_1 : x1 \wedge clk(x2) \wedge \neg clk(x3) \Rightarrow y = x2$$
$$r_2 : \neg x1 \wedge clk(x3) \wedge \neg clk(x2) \Rightarrow y = x3$$
$$c : state = s_0 \Rightarrow \mathsf{next}(state) = s_0$$

(a) CGAs of SITE



(b) Possible Run of SITE

FIGURE 4.9: Sequential If-Then-Else (SITE)

should satisfy causal confluence (definition 3.21). Although causal confluence argues about potentially infinite executions, by theorem 3.26 we only need to check for local confluence (definition 3.25), which only needs to argue about finite executions. However, local causal confluence is defined based on the LTS of the desynchronized process, therefore we need to derive the LTS of the desynchronized process in order to verify it. As the desynchronized process considers the interleaving of different micro-steps, the LTS might be considerably larger than the LTS of the synchronous system. Further, because our property need to quantify over executions of finite length (i.e. least extended macro-step executions), it is not easy to be solved by symbolic methods compared to the verification of causality and clock consistency. Therefore if we would verify local confluence directly, we then face the similar awkward situation as the verification of concurrent systems. In deed, partial-order based methods [31] are invented exactly for the purpose to avoid the redundant interleaving of executions. Here we would also like to avoid to examine the interleaving behaviors, but we would rather like to utilize the compactness of our synchronous LTS and avoid the complexity of the desynchronized LTS. We do this by developing a criteria based on the synchronous scheduled LTSs, which is a sufficient condition for local causal confluence.

To this point, let us examine the synchronous component Sequential If-Then-Else(SITE) again in Figure 4.9, where Figure 4.9 (a) shows the two clocked-synchronous guarded actions of SITE, and Figure 4.9 (b) shows a possible synchronous run. The behavior of SITE is quite simple: during each cycle, based on the truth value of $x1$ it will either assign y by the value of $x2$ (when $x1 = \mathsf{true}$) or assign y by the value of $x3$ (when $x1 = \mathsf{false}$). Therefore, in the first cycle shown in Figure 4.9 (b) y is assigned to 2. Figure 4.10 (a) and (b) shows the synchronous and asynchronous labeled transition systems of SITE respectively.

The interesting point of SITE is that during each cycle, only one of r_1 and r_2 is able to fire. This is ensured by the value of $x1$, as r_1 is only enabled when $x1 = \mathsf{true}$ and at the same time $r2$ is disabled. A symmetrical situation applies when $x1 = \mathsf{false}$ when r_2 is enabled and r_1 is disabled. As a result, the guards of r_1 and r_2 are mutually exclusive. This leads to a very desired property after desynchronization. As shown in Figure 4.10 (b), even after desynchronization, to execute runs of the LTS we still only need a deterministic wrapper. This is evidence since the two transitions r_1 and r_2 distinguish from each other by the value of $x1$. Therefore there would be no ambiguity in

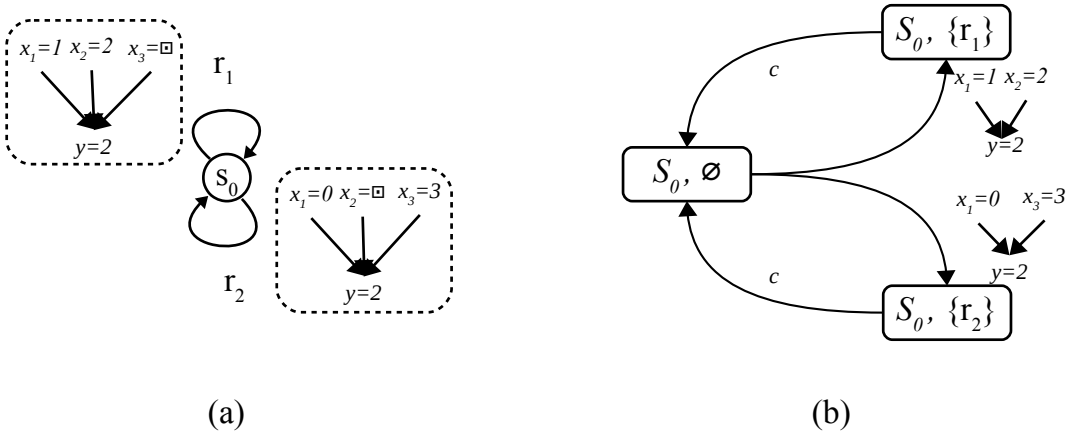
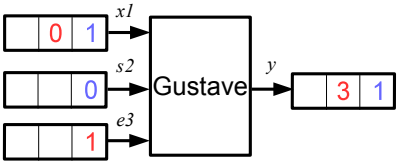


FIGURE 4.10: (a) Synchronous LTS of SITE, (b) Desynchronized LTS of SITE.

$$r_1 : x_1 \wedge clk(x_1) \wedge \neg x_2 \wedge clk(x_2) \wedge \neg clk(x_3) \Rightarrow y = 1$$
$$r_2 : x_2 \wedge clk(x_2) \wedge \neg x_3 \wedge clk(x_3) \wedge \neg clk(x_1) \Rightarrow y = 2$$
$$r_3 : x_3 \wedge clk(x_3) \wedge \neg x_1 \wedge clk(x_1) \wedge \neg clk(x_2) \Rightarrow y = 3$$
$$c : state = s_0 \Rightarrow \mathsf{next}(state) = s_0$$

(a) clocked SGAs of Gustave



(b) Possible Run of Gustave

FIGURE 4.11: Gustave Function

executing the data flow transitions. The intuition behind SITE is that the ⊡ arises only as a **by-product** of the synchronous semantics. i.e. they are there because the values of the variables are not needed but since the clock ticked once, there has to be something fulfilled for the time slot. Since they do not play any active roll in neither control-flow nor data-flow of the computations of the component, removing them would naturally cause no problem and the same functional behavior should be preserved. Furthermore, from the perspective of the desynchronized component, it is able to deduce the correct decisions (in the sense of functional behavior preservation) on whether or not to consume any particular given input data by only looking at the values of the head of the arrived data of each input channel. For the case of SITE, it always reads a value from $x1$, and then based on the value of $x1$ it either reads a value form $x2$ or a value from $x3$ – in this case we even have a deterministic order in reading the inputs. As a more complicated (and more relaxed) example, consider the Gustave function[95]. Figure 4.11 (a) and (b) shows the clocked-SGAs and a possible synchronous run of Gustave funciton, and Figure 4.12 shows the desynchronized LTS of Gusteve function.

Notice that unlike SITE, for Gustave function there is no deterministic order in reading the inputs. In the worst case, one needs to read the head value of all three input channels to figure out which guarded action / data flow transition to take. As an example, consider Figure 4.11 (b). When the first value 1 of input $x1$ arrived, we still can not determine firing r_1 would be the correct choice, since in r_2 x_1 is absent, this value 1 of x_1 might as

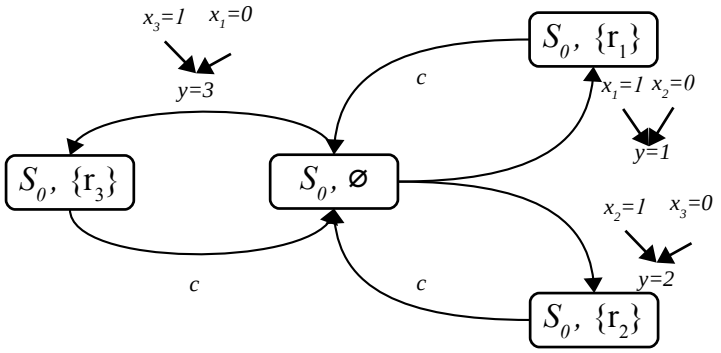
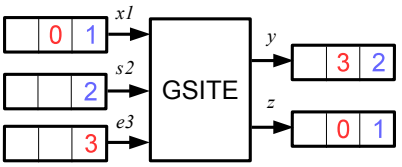


FIGURE 4.12: Labeled transition system of desynchronization of Gustave function.

$$r_1 : x1 \wedge clk(x2) \wedge \neg clk(x3) \Rightarrow y = x2$$
$$r_2 : \neg x1 \wedge clk(x3) \wedge \neg clk(x2) \Rightarrow y = x3$$
$$r_3 : \mathsf{true} \Rightarrow z = x1$$
$$c : state = s_0 \Rightarrow \mathsf{next}(state) = s_0$$

(a) SGAs of GSITE



(b) Possible Run of GSITE

FIGURE 4.13: Generalized Sequential If-Then-Else (GSITE)

well correspond to a later execution of r_1, and the current action might as well be r_2. If the secondly arrived value is 1 from x_3, the confusion is then solved since we know that in order to fire r_2, the value from x_3 should be 0 and in order to fire r_3, the value from x_1 should be 0. However if the value arrived at x_3 would have been 0, the confusion remains as we still don't know which of r_1 and r_2 should be executed. This is only solved when the value of x_2 arrived – if $x_2 = 0$ we know r_1 should be fired, otherwise it is r_2 to be executed. Such argument can be easily adapted to other cases, therefore during any cycle, there is no fixed order for us to read the input values. Yet from Figure 4.12 we can see that in order to execute runs of the LTS after desynchronization we still can build a deterministic wrapper. This is because between each pair of $\{r_1, r_2, r_3\}$, there is a different input variable whose value distinguishes the two guarded actions and making the guards mutually exclusive. Nevertheless, unlike SITE we now need to read from each input channel the first arrived value, but consume only two of them deterministically.

The previous two examples both result in deterministic wrappers after desynchronization. In the following example we show that it might not always be the case. Figure 4.13 (a) and (b) shows the clocked-SGAs and possible synchronous runs of the example of Generalized-SITE. Figure 4.14 shows the LTS of the desynchronized GSITE.

The only difference between SITE and GSITE is that there is one more guarded action r_3 in GSITE. Notice that this guarded action is executed in every cycle, as it copies the value of $x1$ to the output z. However as shown in Figure 4.14, in order to execute runs of the LTS of the desynchronized GSITE in an ASAP fashion, we can either fire

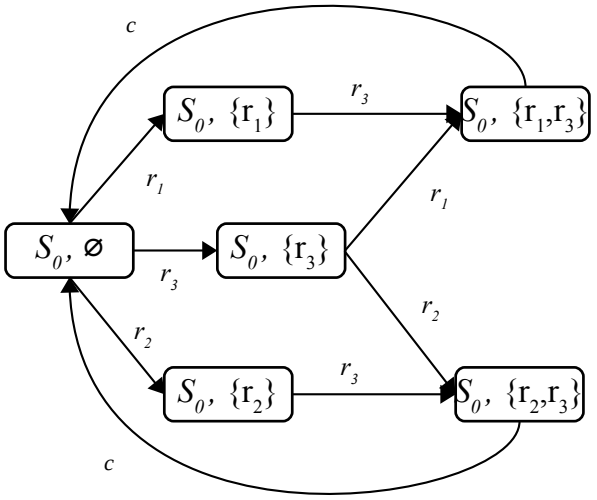


FIGURE 4.14: Labeled transition system of desynchronization of GSITE.

r_3 or one of r_1 and r_2. The order of the execution is not relevant to the final output sequences produced, as any order would correspond to a correct output sequence of the synchronous GSITE given the same input sequences. This leads to a possible non-deterministic wrapper implementation, as our wrapper may fire either r_3 or one of r_1 and r_2 (or even in the same time) based on the arrival of corresponding inputs. Note that although we build a non-deterministic wrapper, we still read a fixed set of input values in each cycle. The only nondeterminicity comes from the different possible orders executing the dataflow transitions / guarded actions. Also note that there is a potential race condition between r_3 and r_1 or r_2 on reading $x1$. However this can be simply solved by first storing the value of $x1$ into a local variable and use it as a shared variable after.

Based on the above discussions, it is not difficult to see that all three examples satisfy causal confluence, therefore allow us to build wrappers that resynchronize inputs while preserving the original functional behavior. Furthermore, some may even allow a non-deterministic wrapper implementation. Therefore to derive a deterministic wrapper is not the criteria of our sufficient condition for causal confluence. Nevertheless, all three examples have some thing in common – for each synchronous cycle, by only looking at the value of each input variable, the clocks of the variables are determined uniquely. As an example, let's consider the synchronous run of SITE in Figure 4.9 (b). In the first cycle, the input assignments are $\{x1 = 1, x2 = 2, x3 = \boxdot\}$ and it is clear that since $x1 = 1$, the clock of $x2$ must be true and clock of $x3$ must be false. In the following, we formalize the intuitive observation and introduce the definition of endochrony.

Definition 4.1. A synchronous component p is called *endochronous* if and only if its synchronous labeled transition system LTS_p satisfies the following condition:

- For all reachable states s_1 and s_2 of LTS_P, for all input and local variable x: $s_1(x) \Leftrightarrow s_2(x)$ implies for all input variable x, $s_1(\mathsf{clk}(x)) \Leftrightarrow s_2(\mathsf{clk}(x))$.

Note that the labeled transition system follows the definition of section 4.1.3, therefore each state of the LTS encodes the full information of a synchronous cycle, i.e. both assignment of local state variables as well as data-flow transitions (assignments of inputs and outputs). Intuitively, the formula indicates that by only looking at the value assignment of the input variables in each cycle, the clock assignments of the input variables can be determined uniquely. Therefore we can build a wrapper that deterministically reads the inputs. It is not difficult to see that endochrony implies causal confluence.

Theorem 4.2. *If a synchronous component p is endochronous, then it is also locally causally confluent.*

Proof. Since by endochrony at each state of the desynchronized LTS, given an assignment to the input variables, it is determined for each input value that whether or not it is consumed. Therefore it is easily deduced that the data-flow transitions are also determined unambiguously. Therefore within one macro-step level sequence of transitions, the possible different execution paths already converges, and it is the same set of data-flow transitions that are executed. □

Corollary 4.3. *For an SPN of multi-rated synchronous processes $P = p_1||p_2||\ldots||p_n$, if P is causally correct and each $\delta(p_i)$ endochronous, then $\forall x \in dom(P), \Phi(\delta(P))(\delta(x)) = \delta(\Phi(P)(x))$.*

By theorem 3.26, we know that locally causally confluence is equivalent to causally confluence. Therefore endochrony is also an efficient condition for causal confluent. Now as a sufficient condition of causal confluent, endochrony is however defined in the level of synchronous transition systems, therefore allows us to directly verify against the synchronous system.

It is important to see that endochrony only means deterministically reading of input values. It never constraints the execution of data-flow transitions / guarded actions. This is examplified by GSITE. Therefore we can still exploit some concurrent executions with endochronous component. In practical this means that the wrapper we build can fire data-flow transitions nondeterministically – based on the arrival of input values.

Sequentiality v.s. Endochrony v.s. Weak Endochrony

The definition of endochrony has quite a checkered history. It is first proposed in a paper by Benveniste et.al. [42]. The definition there states endochrony as the existence of an order in reading input variables, such that by the already read input value sequence, the next input variable to be read can be determined deterministically. It is easy to see that SITE fulfills this definition, as in each cycle we always starts reading $x1$ and based on the truth value of $x1$ either $x2$ or $x3$ is then read after. Since $x1$ is always read in each cycle, it is also called the *master clock* of SITE. However, it is also not difficult to check that Gustave function does not fulfill this definition. In later literature like [79], the definition of endochrony

changes to the statement: the equivalence of any flows of two synchronous runs of a synchronous component implies the equivalence of the two synchronous runs modulo silent actions. As an example, consider:

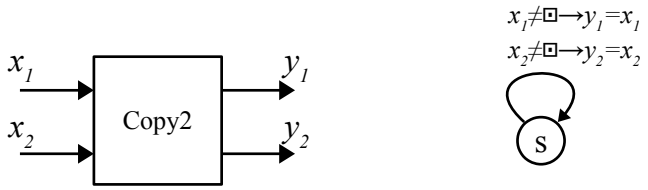
x_1=*1, ⊡, 2, ⊡, 3*	x_1=*1, ⊡, 2, ⊡, ⊡, 3*	x_1=*1, 2, 3*
y_1=*2, ⊡, 4, ⊡, 5*	y_1=*2, ⊡, 4, ⊡, ⊡, 5*	y_1=*2, 4, 5*
A synchronous run π_1	A synchronous run π_2	Flow of π_1 (or π_2)

where π_1 and π_2 are two synchronous runs of some component. The flow of π_1 and π_2 are simply the sequence of values after ⊡ are removed. It is trivial that the flows of π_1 and π_2 are equivalent. By definition in [79], this should imply that π_1 and π_2 are *clock-equivalent*, i.e. they only different in silent-reactions, which is indeed the case (where π_2 has one more silent-reaction at the 4th cycle). This definition also includes Gustave function. Also it is not difficult to prove that this definition is equivalent to our definition 4.1.

An important difference between our work and previous works on endochrony is that in pervious works only macro-step semantics of synchronous systems are considered (e.g. the model used in [42] is Synchronous LTS). This of course greatly simplifies the analysis, however requires that all inputs should be arrived before any action can take place. Therefore a component may only receive the input values once. Instead, our analysis are based on scheduled LTSs and considered the additional causal relations between components, therefore allows a component to read input values in a data-driven style, i.e. an arrived input value is only read when it is required for computation.

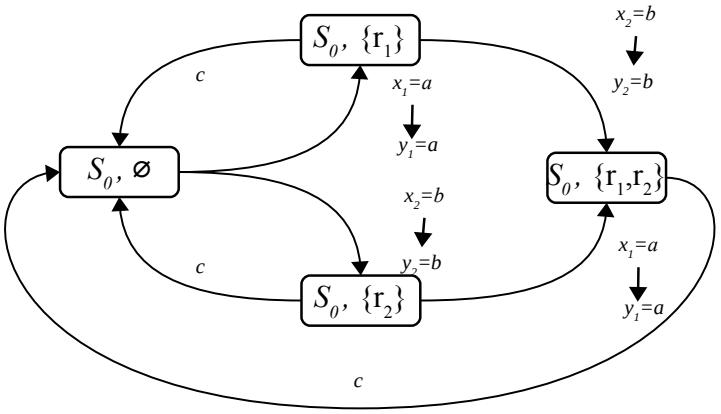
Also notice that when considering macro-step semantics only, endochrony implies the determinicity of the desynchronized process. However it is not the case for us, as we've already seen that deterministic reading of inputs does not necessarily mean deterministic execution of data-flow transitions / guarded actions. In examples like GSITE, we may still enjoy some concurrency within the desynchronized component.

In later works [80–84], *weak endochrony*, literally a weaker version of endochrony is proposed. This sequence of works followed the early works of [42] and tries to exploit the concurrency in synchronous components. As a simple example, consider the component Copy2:



It is not difficult to see that Copy2 is not endochronous. Since for the input assignment $\{x_1 = 1, x_2 = 2\}$, there are three different possible clock assignments: $\{\mathsf{clk}(x_1) = \mathsf{true}, \mathsf{clk}(x_2) = \mathsf{true}\}$, $\{\mathsf{clk}(x_1) = \mathsf{true}, \mathsf{clk}(x_2) = \mathsf{false}\}$,

$\{\mathsf{clk}(x_1) = \mathsf{false}, \mathsf{clk}(x_2) = \mathsf{true}\}$. However, let's have a look at the LTS after desynchronization:



It is not difficult to check that the desynchronized LTS is causally confluent. Indeed Copy2 is more relaxed than endochronous components in the sense that it can non-deterministically choose to end the current macro-step as soon as possible, yet the enabled but not fired data-flow transitions are still enabled at the beginning of the next macro-step. Firing these transition within the current macro-step or in the following macro-step do not change the squence of outputs produced. This way we can still fire the enabled transitions as soon as possible while preserving the functional behavior. Of course, the definition and verification of weak-endochrony are more complex, as instead of concentrating single states of an LTS, weak endochrony requires us to consider successive states. Also note that the major works of weak endochrony are also based on the assumption of macro-step semantics. Causality is considered in [46, 85] by micro-step automata, however the orders of micro-steps are specified explicitly. Instead, the execution orders of SGAs are specified implicitly, and can vary under different situations.

4.3.3 Verification of Endochrony

The main result of this section is also published in [86]. As introduced in the last section, since the definition of endochrony is defined based on synchronous labeled transition systems, we can verify the validity of the formula directly against the symbolic transition system of the synchronous component. In particular, the following property is verified for validity:

- $(\mathcal{R} \wedge \mathcal{R}' \wedge \forall x \in \mathcal{V}_{in}.x \Leftrightarrow x') \implies (\forall x \in \mathcal{V}_{in}.\mathsf{clk}(x) \Leftrightarrow \mathsf{clk}(x'))$.

where $\mathcal{R}$ is the formula encoding the reachable states of the synchronous component, and $\mathcal{R}'$ is the formula with each free variable x in $\mathcal{R}$ renamed by x'.

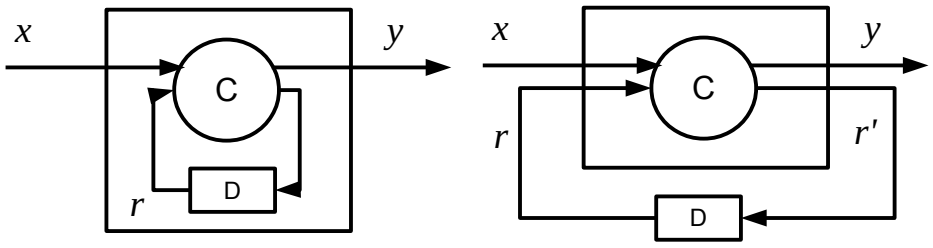


FIGURE 4.15: Overestimation of the reachable states.

Overestimation of Reachable States

The computation of reachable states $\mathcal{R}$ is labor-intensive. For the ease of computation we can perform various techniques for overestimation. A direct and easy technique we used is to simply replace $\mathcal{R}$ by the transition relation $\mathcal{T}$ of the symbolic transition system. The intuition is shown in Figure 4.15.

As shown in the figure, a synchronous component can be seen as a piece of synchronous circuit, where C is the combinatorial part of the circuit and D is the memory. To compute the reachable states of the system means to remember all configurations reachable in the memory. Instead, we simply assume that all configurations of D are possible, therefore only verify the combinatorial part C. This way we lose the precise values of local variables, therefore enlarges the reachable states.

Increase Preciseness by Control Flow Constraints

Only consider the transition relation might be too pessimistic. Here we introduce a way to increase the preciseness of the verification without adding too much burden to the reachability computation. The SGAs of a synchronous component generated by Averest are compiled from an imperative synchronous Quartz program. With such programs, control flows are encoded by the control-flow SGAs. By analyzing these control-flow SGAs we can further derive some invariants helping us to constrain the control flow state variables, therefore reduce the overestimation of reachable states. Figure 4.16 shows a Quartz program Copy2. Figure 4.17 (a) shows its corresponding SGAs and Figure 4.17 (b) shows the *Extended Finite State Machine (EFSM)* of the program.

The program Copy2 consists of two *threads* synchronously composed together (by the synchronous composition operator ||), where each thread performs an infinite loop. Each thread repeatedly performs two cycles (indicated by the two `pause` statements): in the first cycle each thread copies a value from its input channel to its output channel, and in the second cycle each thread remains silent. $l1, l2, l3$ and $l4$ are *location variables* that marks the control flow locations of the program. In a Quartz program, each macro-step corresponds to a control flow location. It is important to notice that because of synchronous composition, the two threads run in lock steps, i.e. when the first thread is in location $l1$, the second thread must reside in location $l3$. However this information

```
module Copy2(clocked bool ?x1,?x2,!y1,!y2) {
    loop{
        if(clk(x1)) {
            y1 = (x1,true);
        }
        l1: pause;
        l2: pause;
    }// end of loop
     ||
    loop{
        if(clk(x2)) {
            y2 = (x2,true);
        }
        l3: pause;
        l4: pause;
    } // end of loop
}
```

FIGURE 4.16: Quartz Code of Copy2.

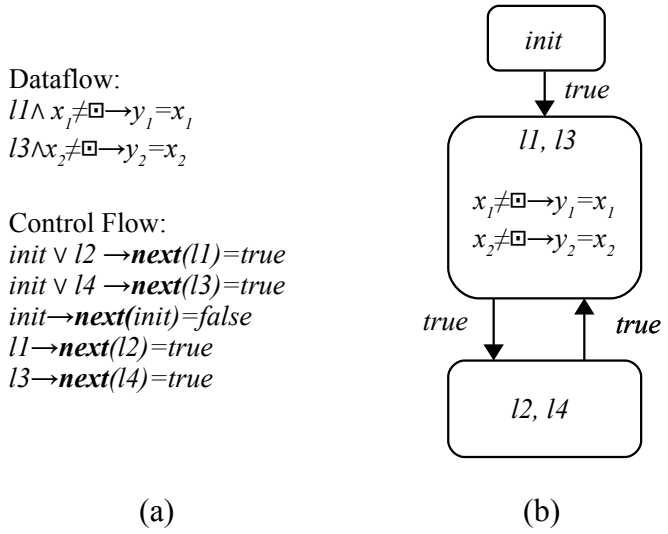


FIGURE 4.17: (a) SGAs of Copy2 and (b) Extended Finite State Machine of Copy2.

is not explicitly encoded by the SGAs as shown in Figure 4.17 (a). In order to extract this information, we can build the EFSM from the SGAs.

An EFSM of a synchronous program is similar to a control-flow graph of a sequential program. It shows explicitly the control structure of the synchronous program. As shown in Figure 4.17 (b), we can see directly from the EFSM that whenever the first thread is in location $l1$, the second thread must reside in location $l3$. We further knows that when the program starts to run, $\{l1 \wedge l3, l2 \wedge l4\}$ are the only two possible configurations of the location variables. These control flow invariants can be used as additional constraints for the verification of endochrony. In order to build the EFSM, we can perform a symbolic execution [87, 88] of the SGAs. In particular, we only record the states of location variables explicitly, and left the partially evaluated guarded actions (according to the

assignments of location variables) in each location state. The branching conditions for each location state are also partially evaluated based on the location states. For example, we know that before the program Copy2 starts, only the initial label *init* (which is not seen from the source code) is true. Therefore the starting state of the EFSM's location variable assignment would be $\{init = \mathsf{true}, l1 = l2 = l3 = l4 = \mathsf{false}\}$. We use this location variable assignment to evaluate the guards of the dataflow SGAs, and found that no guard is possibly true, therefore no dataflow is encoded in this state. By the control flow SGAs, we further knows that in the next macro-step, both $l1$ and $l3$ are true (i.e. the two threads start simultaneously their first macro-step), therefore in the second macro-step the location assignment is $\{l1 = l3 = \mathsf{true}, init = l2 = l4 = \mathsf{false}\}$. Further ,we found that the guards of the two dataflow SGAs are possibly true, therefore we encode the partially evaluated SGAs inside this control state as well. The symbolic execution always terminates, as there are only finitely many location variables, and the symbolic traversal of the location state space is performed over the finitely many control flow SGAs. For more details the reader may reference [89–92].

Fixing Non-Endochronous Components

Since not every synchronous component is endochronous, plus that the verification may perform an overestimation and therefore create false-alarms, we need further methodology to deal with the case when the the verified formula is not valid. In order to verify that a formula is valid, we may employ a SAT solver and check if its negation is satisfiable. If not satisfiable, then equally the formula is valid. Otherwise, we can further derive a model that satisfies the negation of the formula. This model is an variable assignment to the free variables of the formula that makes its evaluation to true. According to the formula of definition 4.1, there must exist at least one input variable x such that the data value of x are equal between two states, but the clocks of x are different. We call such variables *clock-sensitive*. We can extract this variable, and encode its clock as a data input in the synchronous transition system so that its clock information is transmitted explicitly as data and therefore avoids being removed by desynchronization. Then we verify endochrony of the modified transition system, and hopefully the modified system would be endochronous. If not, we repeat the above procedure until the verification successes. The algorithm is listed as follows.

When terminated, the above algorithm returns a set of input variables $\mathcal{U}$, which are the variables whose clock signals should be transmitted through all cycles. Note that the algorithm 1 always terminates, since there are only finitely many input variables, and the worst case is that clocks of the whole set of input variables need to be transmitted, therefore the formular to check becomes trivial. This algorithm nevertheless is pessimistic, as it simply add the clock transmission of the clock-sensitive variables through all cycles, even if the cause of the problem might only exists in a particular cycle. A better way is to add the decorated clock transmissions based on particular control states rather than all macro-steps. Also, whenever the negation formula is satisfied, there might be more than one clock-sensitive variable. Choosing a better suited one might make the next verification immediately unsatisfiable, therefore we might also need a smarter strategy

Algorithm 1 Fixing non-endochronous components

Input: Symbolic Reachable States $\mathcal{R}$, Formula checking endochrony φ
Output: if successfully terminates, the the set of variables whose clocks should always be transmitted

```
1: procedure ForceEndo(R, φ)
2:     U ← ∅
3:     while (true) do
4:         if IsNotSatisfiable(¬φ) then
5:             return U
6:         else
7:             M ← GetSatModel(¬φ)
8:             U ← U∪ ExtractSensClkVars(M)   ▷ Extract one clock-sensitive variable
9:             φ ← UpdateEndoFormula(R,U)
10:        end if
11:    end while
12: end procedure
```

in choosing the right candidates for additional clock transmissions. We leave these topics as future works.

Since we might need to modify and verify the endochronous property multiple times, it is important that we have a very efficient verification technique. In the following section we show that by SAT solving the verification can be done very efficiently.

Experimental Results

We evaluated our method of checking endochrony on a set of examples, including both endochronous and non-endochronous clock-driven components. As outlined in the beginning of this section, we implemented a set of algorithms to check if formula

$$(\mathcal{R} \wedge \mathcal{R}' \wedge \forall x \in \mathcal{V}_{in}.x \Leftrightarrow x') \implies (\forall x \in \mathcal{V}_{in}.\mathsf{clk}(x) \Leftrightarrow \mathsf{clk}(x')).$$

is valid, where $\mathcal{R}$ is over-approximated by the symbolic transition relation of the transition system. Note that the quantifier over states can be eliminated by instantialization over all input variables. Then we invoke a SAT solver to check the validity of the formula. In case it is valid, we know that the corresponding component is endochronous. In particular, we employed both a BDD based solver (NuSMV [93]) and a SAT solver (Z3 [94]) to verify our property. In order to keep a fair comparison, we boolified all non-bool programs so that a pure boolean formula of (2) can be generated, and then we push this boolean formula into NuSMV and Z3 respectively to check its validity. The whole set of experiments is carried out on a laptop with an 2.9 GHz Intel Core i7 processor and 8 GB DDR3 memory. Table 5.1 shows the preliminary result of our verification.

Our test suit contains both small examples and practical applications that are written in Quartz [20]. The first four test cases are sequential and parallel versions of Or and

TABLE 4.1: Experimental Results — Check Endochrony

Example	**BDD (NuSMV)**		**SAT (Z3)**		**Endochrony**
	time	nodes	time	clauses	
Seq-Or	0.01	1	0.01	20	Yes
Seq-ITE	0.01	1	0.01	25	Yes
Par-Or	0.01	268	0.01	30	No
Par-ITE	0.01	1,053	0.01	55	No
NWE	0.01	368	0.01	36	No
Gustave	0.02	1	0.01	44	Yes
Filter	0.12	14,645	0.01	258	No
OpDecode	722.05	1	0.01	269	Yes
Heating	9.91	1	0.01	119	Yes

If-Then-Else operations, where the sequential versions are endochronous. Gustave function [95] is a typical example that is endochronous but not sequential. As far as we know, none of the existing methods for checking endochrony covers Gustave function. NWE is the example we discussed before in Figure 3.9. The rest three examples are based on practical applications. Filter is a component that only chooses to read from a subset of its inputs at each cycle. Since the choice depends on absent of signals that makes it non-endochronous. OpDecode models the decoding stage of a processor. It has input values i_1, i_2 and i_3 that form a "clock-tree" where i_1 is the master clock and i_2, i_3 are its sub-clocks. In particular, if $i_1 = \mathsf{true}$, then i_2 determines the rest of present values, otherwise i_3. Heating is a component that reads indoor and outdoor temperatures as well as movements inside a house, and based on different conditions it either increases the indoor temperature or does nothing. Inside the table, whether or not the component is endochrony is shown in the right-most column. Performances of the two solvers are shown in the middle columns. For both solvers we record the time (in seconds) taken to compute the final result. For NuSMV, we record the size of the BDDs generated, and for Z3 we record the number of clauses made. It is noticeable that once a formula is valid, its BDD reduces to a single node which is true. However, for some endochronous example it still takes a long time to compute this single node (e.g. OpDecode). On the other hand, the SAT solver Z3 performs quite well in all cases. In order to test scalability of our method, we further generated a set of parameterized test suit. The result is shown in Table 4.2.

In particular, we chose Parallel-Or as our template and two parameters (m, n), as shown in the first column. The first parameter m specifies the number of rules that are generated as well as the number of input variables, and the second parameter n bounds the variables by the range of natural numbers $[0, n-1]$. Each generated rule only choose to read one of the m input variables and assigns it to the single output. Also for each program, there is an additional rule specifying if all inputs are present and 0 then 0 is the output. Therefore, all parameterized programs are not endochronous. Besides data collected similar to Table 5.1, we also record the number of free variables in formulas in the second column. The data in Table 4.2 shows that our method scales quite well for SAT as it can deal with programs having 20,000 boolean variables within three seconds,

TABLE 4.2: Experimental Results — Parameterized Par-OR

Param.	**No. vars**	**BDD (NuSMV)**		**SAT (Z3)**	
		time	nodes	time	clauses
(3,2)	50	0.01	268	0.01	30
(4,4)	150	282.53	1,108,464	0.01	301
(4,8)	214	-	-	0.01	891
(4,32)	342	-	-	0.02	1,399
(8,32)	634	-	-	0.02	2,615
(16,32)	1,218	-	-	0.03	5,046
(32,32)	2,386	-	-	0.04	9,941
(64,32)	4,772	-	-	0.12	19,706
(128,32)	9,394	-	-	0.34	39,203
(256,32)	18,738	-	-	1.61	78,222
(256,64)	22,330	-	-	2.42	92,607

while the BDD solver soon become unavailable for cases with 200 variables. In particular, from parameters $(4, 8)$, we run our tests on NuSMV for 10 minutes and terminate without getting any result.

Other Methods Verifying Endochrony

As the introduction of endochrony is tightly related to the problem of generating distributed threads of the Signal [30] programs, checking endochrony is originally solved by Signal compilers. A fundamental difference between our methods and the methods of Signal is that Signal is a declaritive synchronous language equipped with clock calculus. Therefore the verification of endochrony reduces to the verification of the existence of a master-clock in the clock hierarchy (i.e. if the clock hierarchy forms a tree), which is actually equivalent to checking sequentiality – which is more restricted than endochrony [96]. MRICDF [40, 97] adopted SAT-based techniques to compute the master clock, in particular by checking existence of a base clock is reduced to check if there exists a prime implicate of a boolean formula, but is still based on the clock calculus of Signal and still check for endochrony. Instead, our method works for the intermediate format SGA of synchronous programs [72], and does not require the help of a clock calculus. In particular, we reduced checking endochrony to an equivalent SAT problem, so that efficient symbolic model checking procedures can be applied. This distinguished our work from all previous works. It is also noticeable that components of Lustre [15] are by definition endochronous, since each Lustre component requires explicitly a master clock as a trigger of the computation of the component.
A benifit from Signal based methods is that they can use the same method to verify if a component is weakly endochronous or not. This is simply because of the fact that a weakly endochronous component has multiple clocks as their roots of the clock hierarchy. However unlike our method, none of these methods is able to verify the endochrony of Gustave function – since in this situation all input variables would be merged to form a single clock root, therefore all inputs would

be required to have the same presence at the same cycle.
Here we would like to argue that the need of verification of weak endochrony is rather motivated by the search of a proper decomposition of the global synchronous system. Take Copy2 as an example, which is weakly endochronous. It is clear that the two threads are logically independent with each other, as none of the threads uses the data from the other. Therefore instead of forcing them to run in lock steps, a better decoupling is to treat each thread as a single node in the DPN and let them run totally independently from each other. Note that this logical partition doesn't mean that the two threads would be physically separated. They may still be running on the same computer, nevertheless it would be rather a better idea to really implement them as two threads, rather than one thread. Indeed, in later works of program distribution [98–100], root clocks of the clock forest of a Signal-like program are used as identification of threads for multi-threaded code generation. In our context, although it is possible to encode weak endochrony as a SAT problem, we do not tackle it because we beleive that weak-endochrony is more related to partitioning, and we would rather assume any given valid partition of the global synchronous system.

4.4 Summary

In this chapter we introduced a sufficient condition for causal confluence – endochrony, and discussed in detail its verification. The motivation to introduce endochrony is to keep the verification in the level of synchronous transition systems, so that no interleaving semantics introduced by desynchronization is needed to be bothered. This keeps the verification working on a very compact formalization and therefore leads to a very efficient verification method. In particular, we reduced the verification to a SAT solving problem and implemented the verification methods correspondingly. Further more, once we found a component is not endochronous, we can add additional communication to make the component endochronous. This however is in the cost of communication efficiency.

Our verification method is based on the transition system of the synchronous model, therefore is independent of the underlying implementation details of the synchronous programs. In previous works, however, verification methods are all based on clock-calculus, which are not directly available for imperative synchronous programs written in Esterel and Quartz. Experimental results shows that our verification method is quite efficiency and scalable. Added with the consideration that typically a component of a synchronous system is small to middle sized, our method should be sufficient with most practical scenarios.

Chapter 5

Practicing Model Transformation

In previous chapters we have developed step by step a theory of correct desynchronization with the special care of causality. In particular, the global property causality correctness is ensured by the synchronous language compilers, and the local property endochrony can be verified very efficiently. For those non-endochronous components, by calculating the set of clock-sensitive input variables we can add their clock-transmission and make the modified components endochronous. Therefore in this chapter, we can already assume that our synchronous system is causally correct and each component is endochronous, thus allows us a correct desynchronization using the as soon as possible scheduling strategy for the desynchronized system.

In this chapter, we show that how the theory of model transformation is implemented in practice, therefore validating our statements in previous chapters by real examples. In particular, we choose the synchronous guarded actions as our description format for synchronous components, and the RVC-CAL [101] actor language as the description format for asynchronous components. In the following we first briefly introduce the syntax and semantics of RVC-CAL, then we discuss in detail how a set of SGAs can be translated into a set of RVC-CAL actions following our desynchronization theory. Finally a case study is show for the validation of the model transformation techniques.

5.1 CAL-Actor Language

CAL is an actor language created as a part of the Ptolemy II project [102] at the UC Berkeley. An actor is simply an entity that can be composed with other actors to form a concurrent system. A foundation for actor computation shares great similarity of Kahn's Processing Networks [18], therefore with DPNs. The following descriptions are quoted from the CAL language reference [103]:

> Intuitively, an actor is a description of a computation on sequences of tokens (atomic pieces of data) that produces other sequences of tokens as a result. It has input ports for receiving its input tokens, and it produces its output

> tokens on its output ports. The computation performed by an actor proceeds as a sequence of atomic steps called firings. Each firing happens in some actor state, consumes a (possibly empty) prefix of each input token sequence, yields a new actor state, and produces a finite token sequence on each output port. Several actors are usually composed into a network, a graph-like structure (often referred to as a model) in which output ports of actors are connected to input ports of the same or other actors, indicating that tokens produced at those output ports are to be sent to the corresponding input ports.

Other important facts include:

- The connection between actors is an abstraction of the history of sequence of values that are sent by the producer actor and received by the consumer actor, therefore the connection may be treated as an unbounded FIFO buffer, thus sharing the same assumption as channel connections in DPNs.
- The connection structure between actors is only a topology structure, i.e. it does not enforce a particular scheduling of the execution of the actors. Instead actors are executed autonomously, i.e. the only constraint limiting the execution of an actor is whether or not it has enough input data.

Therefore we can simply treat actor as a synonym of process of a DPN, and actor networks as DPNs. Of course, actors support hierarchical design and an actor itself can be a DPN composed of several actors. However in our context of desynchronization, we maintain a flattened view of the DPNs and treat an actor as a single entity that can not be decomposed anymore.

As the name of the language indicates, the major task in writing CAL programs is to write actors. The network of actors are formed naturally by interconnecting the shared ports of the actors. In the following we introduce briefly the syntax and semantics of actors, and the semantics of the network follows naturally.

RVC-CAL

Our target language is RVC-CAL, which is a restricted subset of CAL. While being abstract enough to model DPNs, CAL is designed as a programming language [103] and therefore is expressive enough for modeling very practical applications. However it is also platform-independent and implementation independent, as it can be synthesized to both VHDL and multi-threaded C/C++ software [101]. According to [101], RVC-CAL allows only the usage of four primitive data types (**bool**, **float**, **int**, **uint**) and two advanced data types (**List**, **String**) that we doesn't support or need for now. It also requires that all variables are typed and concrete, and no advanced features like channel selections are needed. For more details the readers can reference [104]. For us, it is important to know that we only use the four basic types, and we do conform with the constraint that variables are concretely typed. For further details of the CAL language, interesting readers may refer to the language reference [103].

```
actor <name_of_actor> () <type>(size=<n>) <input_channel_name>, ...
     ==> <type> (size=<n>) <output_channel_name> , ... :

     <local_variable_declarations>

     <action_label>: action <input_channel_name>: [<input_var_name>], ...
                        ==> <output_channel_name>: [<output_expr>],...
     guard <boolean_expr>
     <output_var_declaration>
     do
        <var>:=<expr>
        ...
        <var>:=<expr>
     end


    schedule fsm <init_state> :
    <curr_state>(<action_label>) --> <next_state>;
    ...
    <curr_state>(<action_label>) --> <next_state>;
    end

end
```

FIGURE 5.1: CAL Actor Structure.

5.1.1 Syntax of CAL

As pointed out in the CAL language reference, the syntax and semantics of the language covers the following four essential factors:

- The port signature of an actor (its input ports and output ports), as well as the kind of tokens the actor expects to receive from or be able to send to them.
- The code executed during a firing, including possibly alternatives whose choice depends on the presence of tokens (and possibly their values) and/or the current state of the actor.
- The production and consumption of tokens during a firing, which again may be different for the alternative kinds of firings.
- The modification of state depending on the previous state and any input tokens during a firing.

In the following we only give an informal description of the actor's syntax. Also, we won't cover all syntax structures (statements, expressions etc.) of the language, but only those that are used during our model transformation. Figure 5.1 shows the basic structure of an actor, where key words are in bold font and user-defined words are in angle brackets. As shown in the figure, an actor definition begins from its name and port declarations, where as the port declarations defines a list of ports. Each port declaration consists of the the type of data values that are transmitted in the corresponding channel

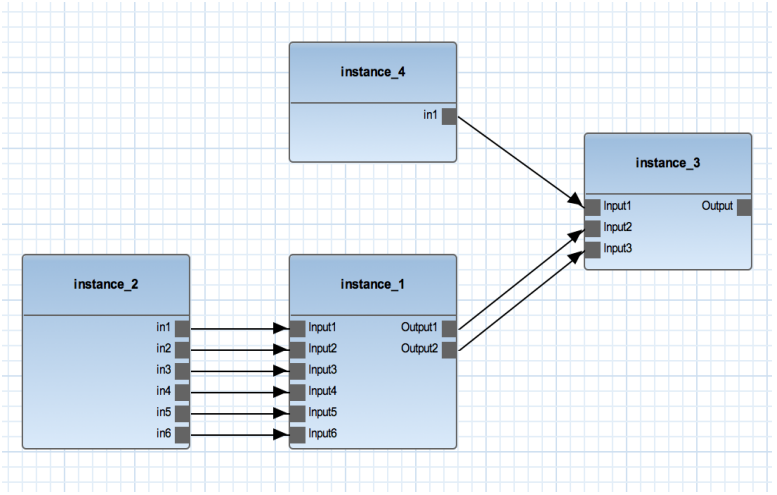
```
actor Add () int(size=8) Input1, int(size=8) Input2 ==> int(size=8) Output:
    add: action Input1: [a], Input2: [b] ==> Output: [c]
    guard true
    var int(size=8) c
    do
      c:= a+b;
    end

    schedule fsm add :
    add (add) --> add;
    end

end
```

FIGURE 5.2: CAL Actor Adder.

connected to the port as well as the size of each value, followed with the name of the port. Input ports and output ports are separated by the symbol "==>". Inside an actor, a programmer can define a list of local variables as local memory used during the computations of the actor. The computations of an actor are performed by actions.

An actor can define multiple actions. Action declarations follow the declaration of local variables. Each action declaration starts with a unique label of the action, followed by the key word "action", then a list of read-write patterns of input and output ports. In particular, the read-write pattern specifies for each port the number of tokens read from / write to the port each time the action is executed. In our context, for each port at most one token is read / write whenever the action is executed, since actions are translated from SGAs and by synchronous semantics during each execution only a fixed value is read / written for each variable. Notice that because of the binding style of CAL, the input tokens are declared as variables, but the output tokens can be declared as an expression. This output expression can be an expression using the input variables defined in the input port patterns. If it uses new variables, the variables must be declared immediately in the next line. The `guard` section defines the guard of the action, which is a boolean expression. The free variables of the boolean expression can be both local and input variables. the `do-end` section defines the assignments (or actions) of the local and output variables.

CAL provides explicit methods to schedule actions. One of the methods we used during our model transformation is the finite state machine (FSM) based scheduling. The scheduling is defined in the section `schedule fsm`. The first word follows the key words defines the initial state of the FSM, which should correspond to the some action's label. The name of states of an FSM can either be an action label, or a prefix of some actions' labels. In the latter case, the prefix must be separated with the suffixes by a ".". Following the initial state, the state transition relations are defined, where the starting state and the destination state are separated by "`-->`". After the starting state one need to further specify which specific action is fired (since different actions might share the same prefix) by using the action's label. Figure 5.2 shows a simple adder specified in CAL.




```
<?xml version="1.0" encoding="UTF-8"?>
<XDF name="mytest_ntw">
    <Instance id="instance_1">
        <Class name="mytest.mytest"/>
    </Instance>
    <Instance id="instance_2">
        <Class name="mytest.input"/>
    </Instance>
    <Instance id="instance_3">
        <Class name="mytest.site"/>
    </Instance>
    <Instance id="instance_4">
        <Class name="mytest.binput"/>
    </Instance>
    <Connection dst="instance_1" dst-port="Input1" src="instance_2" src-port="in1"/>
    <Connection dst="instance_1" dst-port="Input2" src="instance_2" src-port="in2"/>
    <Connection dst="instance_1" dst-port="Input3" src="instance_2" src-port="in3"/>
    <Connection dst="instance_1" dst-port="Input4" src="instance_2" src-port="in4"/>
    <Connection dst="instance_1" dst-port="Input5" src="instance_2" src-port="in5"/>
    <Connection dst="instance_1" dst-port="Input6" src="instance_2" src-port="in6"/>
    <Connection dst="instance_3" dst-port="Input2" src="instance_1" src-port="Output1"/>
    <Connection dst="instance_3" dst-port="Input3" src="instance_1" src-port="Output2"/>
    <Connection dst="instance_3" dst-port="Input1" src="instance_4" src-port="in1"/>
</XDF>
```

FIGURE 5.3: Graphic representation of a CAL actor network and the corresponding .xdf file.

CAL network

A network of actors forms a CAL network. The syntax of a CAL network should define the instances of actors as well as their interconnections. In RVC-CAL this is specified by an .xdf file which is an XML-format file. RVC-CAL provides a graphic editor to build CAL actor networks, and the corresponding .xdf file can be generated from the graphical network automatically. Figure 5.3 shows the graphical representation of an actor network as well as the corresponding .xdf file. As shown in the .xdf file, first four instances of for different actors are declared, followed by their interconnections.

5.1.2 Semantics of CAL

The execution of an actor is carried out by the execution of its actions. As already indicated from the syntax, an action in CAL is quite similar to a guarded action. In our context, an action is *enabled* only if the following conditions are fulfilled:

1. The current state of the FSM matches the prefix of the action's label.
2. There are sufficient input tokens available to bind the input pattern variables to appropriate values – in our context, each specified input channel should contain at least one token.
3. Given such a binding, all guard expressions (the Boolean expressions) evaluate to `true`.
4. There is sufficient room for the output tokens produced by this firing – in our context, each specified output channel should have at least one empty token slot.

Whenever all four of the above conditions are fulfilled, an action is enabled can then be executed. If more than one action is enabled, the CAL scheduler is free to chose any of them to execute. The execution will consume the tokens specified in the input pattern of the action from the input channels, evaluate the output expression and output the value to the output channels. Therefore the semantics of an actor is quite similar to the semantics of a process of a DPN as we introduced in section 2.1.2, and it is not difficult to see that we can directly derive the LTS of an actor. Note that the above conditions are not the complete enable conditions of CAL actions. But since we never use other scheduling strategies, in our context they are already complete. Condition number 2, 3, 4 are clear. Condition 1 needs some more explanation.

Finite State Machine based Scheduling

FSM based scheduling is an easy way to encode explicit control states in firing actions. The set of states of the FSM is a set of prefixes of action labels. The actor maintains the current state, and only when an action's prefix matches the current state can the action be enabled. For example, if the current state is `st1` and we have three actions with labels `st1.a`, `st1.b` and `st2.a`, Then the first two actions are possibly enabled (depending on other enabling conditions as well) while the last action is definitely not enabled. FSM specifies the initial state as well as state transition relations. A state transition has the format `curr_state(action_label) --> next_state;`, where `curr_state` specifies the current state, `<action_label>` specifies the executed action's label and `<next_state>` specifies the new state after the transition updating the current state.

Figure 5.2 shows an adder specified as a CAL actor. The name of the actor is `Add`, and its function is to take one token value from each of its input channels (`Input1`, `Input2`) and calculate the summation and output it to the output channel `Output`. Note that the types of the input and output tokens are integer with eight bit width. There is only one action in the actor with label `add`. Its input pattern shows that when this action is executed, it needs one value from each input channel. The two values are referred as a and b, and can be used in the output expression, guard expression as well as the expressions inside the action. The output expression is a fresh variable c, therefore needs to be declared immediately after the guard. The guard expression is simply `true`, therefore adds no further constraint. The `do` part assigns the output c by

the sum of a and b. The FSM scheduler simply specifies a single state `add`, and when action `add` is executed the state remains the same – therefore the FSM also doesn't add any further scheduling constraint and can be removed. From previous discussions, we can conclude that when the current state is `add` (which is always the case), at least one token is arrived at each input channel and at least one empty token slot is available at the output channel, action `add` is executed.

Semantics of CAL Actor Networks

The semantics of a CAL actor network is basically the same as the semantics of a DPN. There is one important difference that should be pointed out. A DPN is an ideal model in the sense that all interconnections of processes are unbounded FIFO buffers. This is not the case for CAL actor networks, since an actor network is specified in the implementation level. Therefore for CAL's actor networks, all interconnections are FIFO buffers with finite space. However the finiteness of buffer storage will not cause any problem for our model translation. In particular, by our theory of desynchronization we know that once the SGAs fulfilled corollary 4.3, we know that a one-place buffer for each actor's channel is already enough to preserve the functional behavior of the synchronous system.

5.2 Translation from SGAs to CAL Actions

In this section we discuss in detail how to translate a set of SGAs to a set of CAL actions that form a CAL actor. We basically follow the procedure discussed in definition 3.8 and definition 3.1. In section 4.1.1, 4.1.2 we've already seen that the execution of one synchronous guarded action corresponds to a state transition in the desynchronized LTS. Therefore a naive translation is to translate each SGA into an action of a CAL actor. However there are many technical details in between which make the translation from SGAs to an actor not that trivial. The major technical problems are listed as follows:

1. Sharing guards: there might be more than one guarded actions that are sharing the same guard. We should only perform the evaluation of the guard once, not only because of efficient reasons, but also because that the guards that are implemented by CAL actions might consume input tokens, therefore the tokens shared by different actions should be read only once by the guard.

2. Race condition on inputs: similar to the sharing of guards, inputs might be shared by different actions as well. If more actions sharing the same guard are enabled, following the synchronous semantics they should all be fired. However if the actions share the same inputs, the input tokens should be read only once.

3. Deterministically reading inputs: in order to preserve the functional behavior of the synchronous component (SGAs), the actor should compute the same outputs given the same inputs. We already knew that if the synchronous component is

endochronous, we can consume the inputs deterministically and as soon as possible. This needs to be practiced for the CAL-actions.

4. Control flow scheduling: for the desynchronized LTSs, control flow transitions are always scheduled after data flow transitions, although the control flow transitions might not have data dependency from the data flow transitions. This additional constraint should be followed by corresponding CAL-actions. Note that in theory each state of the desynchronized LTS **knows** a priori the set of data-flow transitions to be fired. However this is not the case in reality! Therefore we need an algorithm that determines the right timing to execute the control flow actions not depending on a priori knowledge, which is not trivial.

5. Delayed assignments: in SGAs there are delayed assignments that only take effect in the starting of the next macro-step. This needs to be reflected by the corresponding CAL actions. In particular, for a delayed assignment, the corresponding CAL-action should be scheduled after all immediate assignments. More importantly, as CAL has no notion of steps, the delayed assignments will take immediate effect once they are done. Therefore these assignments should not affect the actions that logically belong to the current macro-step.

6. Reaction to absence of local variables: unless input variables whose values are treated as tokens, local variables are treated as shared memories of actions. Therefore they also inherent the reaction to absence from the synchronous semantics, i.e. whenever no assignment to a local variable is executed, the value of the variable should be assigned to a default value.

Based on the above issues, we decide to translate the guard and the action of an SGA in to two CAL actions respectively. This translation logically partitions the set of CAL actions in to two groups – one group of actions corresponds to the guards that play the roll of the wrapper of an actor, which monitors the input tokens and triggers the corresponding actions – which form the other logical group of CAL actions.

General Translation Scheme

Figure 5.4 shows the general scheme of the transformation. The upper half of the figure shows the derivation of variables used in CAL actors, and the lower half shows the derivation of CAL actions, both indicated by the black arrows. The un-pointed variables / CAL actions are additional parts that are needed for the transformation. For the variables, input and output ports of an actor are derived directly from the input and output variables of SGAs, and local variables of an actor are derived directly from the local variables of SGAs as well. Additional variables are needed for CAL actions. For solving the race condition of input tokens between different actions, additional *input buffers* are needed for storing the input tokens read from the input ports. For storing the updated local variable for the delayed assignments, temporal storage for the *delayed local* variable assignments is needed as well.

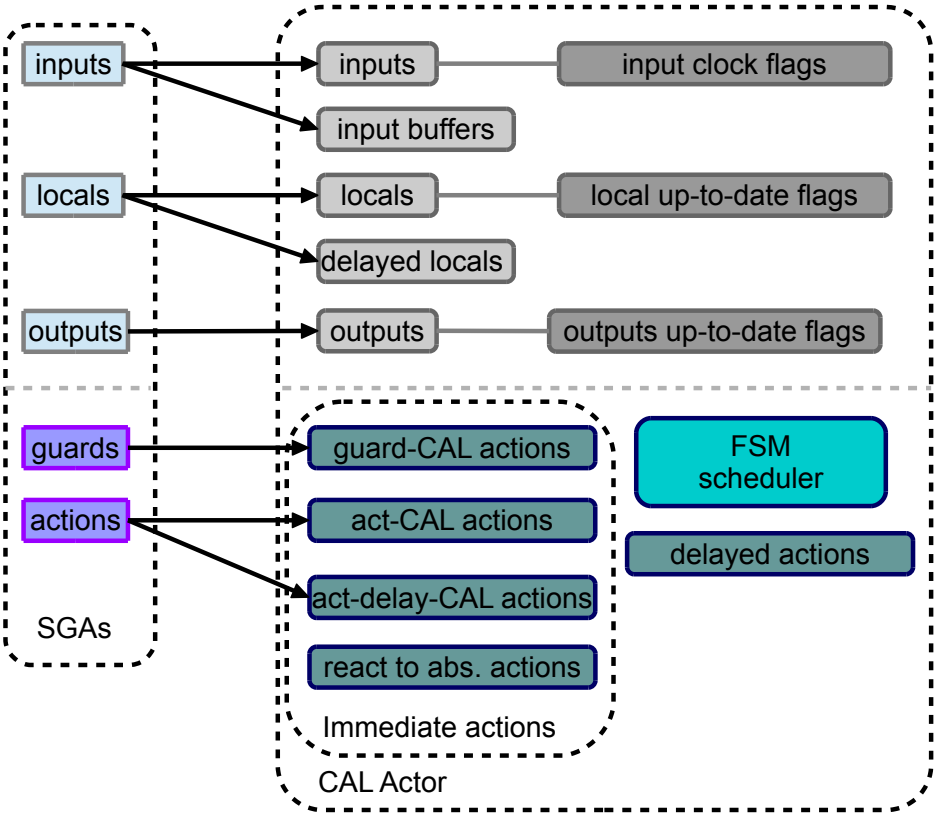


FIGURE 5.4: Model Translation Scheme.

One major problem for the desynchronized component to perform the computations correctly is to decide when to read an input token / use the value of a local variable. In order to indicate this information, flags are used. In particular, *input clock flags* are indicators for the consumption of input tokens, *local / output up-to-date flags* are indicators for the validity of the content of a local or an output variable.

For the part of the guarded actions, as can be seen from the figure that guards and actions of SGAs are translated separately into different CAL-actions. In particular, we translate each guard (possibly shared by different assignments) of an SGA into a *guard-CAL action*, and each action of an SGA into an *act-CAL action*. For different types of assignments, we further distinguish them into different CAL actions. Since these CAL actions all need to read input tokens / local variable in order to perform their computations, they need to know when is the right timing to read them. However as clocks are removed, this must be done by other mechanisms. Also since local variables are treated as shared memories rather than tokens, they always posses a value and can be read at any time. Therefore we must make sure that when they are read, their values are up-to-date. The proper timing of reading inputs and local variables are managed by the guard-CAL actions. Also, immediate and delayed actions should be treated differently as delayed actions should only take effect in the next logical macro-step. Finally, the actor needs to know that when is the right time to execute the control flow transitions to evolve to the next macro-step. Apparently, this should take place only after all outputs are updated (if they can be updated at all). For this purpose, the actor also needs

to maintain the up-to-date flags of the outputs, so that it knows an output is either updated or there should be no update at all during a particular macro-step. In the following sections we will introduce the translations to CAL actions and discuss the above issues in more detail.

5.2.1 Translate Guards to Guard-CAL Actions

For endochronous components we know that given an input token at each input port, there is only a single way to consume them. However, in order to schedule the actions, we need to transform this knowledge into the decision of when does a particular guard evaluate to true. The difficulty here is that if a guard reads some inputs, then the corresponding guard-CAL action consumes the input tokens once the guard evaluates to true. However, the input tokens should only be consumed when the guard evaluates to true. This means we have no way to test when does the guard evaluate to false – otherwise by testing its negation is true, we would equally consume the input tokens. Fortunately, for endochronous components we have a proper solution. Intuitively, if a guard reads some inputs, then we can deduce that some other guards must be false by the mutual exclusiveness on the clock variables, and their flags can be unset accordingly. Another possibility is that some sub-expression of the guard is falsified by the update of some local variable, therefore in turn falsifies the whole guard. We claim that for an endochronous component, a guard can only be falsified by the two reasons we just stated. This is justified by proposition 5.1.

Proposition 5.1. *Let $\mathcal{G}$ be the set of clocked SGAs of a synchronous component p with $\mathcal{T}$ its transition system. Then p is endochronous implies for any guard γ_1, at any state s if $(T, s) \not\models \gamma_1$, then there must exists another guard γ_2 with a sub-expression e, such that $(T, s) \models \gamma_2$ and: $\neg e \implies \gamma_1$ and $e \implies \gamma_2$.*

Proof. Before we begin the proof we should assume that for all guards of p they are all satisfiable, i.e. there exists a state s such that some inputs and local variables approved the guard. Furthermore, inputs should have no constraints. Therefore at any states, if the guard is not falsified by assignment of some local variables, then there exists input assignments that approve the guard. Otherwise, if the guard is invalid we should simply remove the guarded action from the system.

First by Figure 4.5 we know that at least one of the guards should be true at each macro-step. Assume that at state s we have $(T, s) \not\models \gamma_1$, therefore γ_1 is falsified. Without losing generality, we assume that γ_1 reads one input x. If there is not γ_2 with sub-expression e such that $(T, s) \models \gamma_2$ and $\neg e \implies \gamma_1$ and $e \implies \gamma_2$, then it must be the case that the assignment of x falsifies γ_1. However since there is no other guard reads x, it must be the case that $\mathsf{clk}(x) = \mathsf{false}$ in state s. As by our assumption, there exist cases that $\mathsf{clk}(x) = \mathsf{true}$ and γ_1 is satisfied as well. We assume that v is such a possible value. Since when $\mathsf{clk}(x) = \mathsf{false}$, the value of x in T is irrelevant and can be any value, therefore it can be v as well. Therefore we can find another state s' with the same value assignments of local and input variables as s, but the clocks of inputs are different as already shown before, which is a contradict with the assumption that p is endochronous.

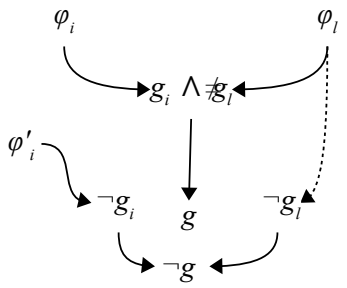


FIGURE 5.5: Scheme setting guard flags.

□

Since the free variables of sub-expression e can only contain input and local variables, our statement is justified. The scheme of setting guard flags is shown in Figure 5.5. As shown in the figure, the guard flag g of a guard is determined by the conjunction of g_i and g_l which are the flags indicating the validation by the input variables and local variables respectively. Assume the guard is $\varphi_i \wedge \varphi_l$, where the truth value of φ_i depends on inputs and local variables and the truth value of φ_l only depends on local variables, thus the truth of φ_i (φ_l) determines the truth value of g_i (g_l). Now φ_l can be evaluate by the actor as long as all local variables it reads are up-to-date. Instead, the truth value of g_i is indicated by both φ_i and a mutually exclusive sub-expression φ'_i of some other guard.

Properly Scheduling Control Flow Transitions

Proposition 5.1 is important for the actor to schedule its control flow correctly. In particular, in section 3.2 we can see that the desynchronization of synchronous LTSs leads to the refined LTSs where in each state, not only the state variable's assignments are recorded but the executed data-flow transitions are also remembered. The control flow transition is then executed whenever there is no more data-flow transitions enabled. This information is kept in the desynchronized LTS, but in our implementation of actors there is no direct way to remember the executed data-flow transitions, therefore it is rather nontrivial to decide when to execute the control-flow transition, so that we can start the next sequence of macro-step transitions. However, if we can evaluate the truth value of every guard, then we can simply wait until all guards are evaluated, then we can further guarantee that all possible actions that can be executed within the current macro-step are already executed and then move on to the next macro-step.

Based on the above discussion, we can calculate for each guard γ a set of guards $\mathcal{F}(\gamma) := \{\gamma' \mid \gamma \implies \neg\gamma'\}$, i.e. the approval of γ falsifies all guards in $\mathcal{F}(\gamma)$. This can be easily done by checking for pairs of guards γ, γ' if $\gamma \implies \neg\gamma'$ is valid.

Race Condition between Guards

It is possible that two guard may compete for the same input token. This happens when in the synchronous system two guards sharing the same input both evaluates to true during the same macro-step. Accordingly, in the desynchronized system when both guards evaluates to true, they would all try to read the same input value from the port. However since only one of them succeeds, the other is disabled after, which violates the synchronous semantics. To solve such race conditions we need to save the input token to a local variable, so that the guards can share the same value again. However, the problem is that we still need a guard-CAL action that reads the inputs, and this should only happen when one of the guards is true. Once a particular guard is true, the other guards should use the local variables instead of the input tokens. However since any of the guards might be true, we can not fix a particular form of the guard-CAL actions to make sure that it either reads inputs, or reads local variables.

The solution here is to implement a twin of guard-CAL actions, where for the shared variables, one monitors the input tokens, the other monitors the validity of the corresponding local variables. Any execution of the actions would results in the validation of the guard flag. However only one of the twin actions is executed. Once a guard-CAL action consumes the inputs, it should then disable the other guard-CAL action that is waiting on the shared input token and enable the action that is waiting on the validation of the local variable. Also note that the principle above can easily be extended to cover the case when more than two guards sharing the same inputs and can be enabled in the same macro-step.

Based on the above discussions, the translation from a guard to the corresponding guard-CAL actions should first collect the following information for each guard:

- list of inputs it reads;
- list of local variables it reads;
- set of guards that can be falsified by this guard;
- set of guards that can be satisfied with this guard and sharing same inputs;

Figure 5.6 shows the template used for translating from guards to guard-CAL actions with a twin of actions, where the first action (`Grd < grd_id > _RI`) waits on the input tokens while the second action (`Grd < grd_id > _RL`) waits on the local variable's update (and its list of input patterns are the same as `Grd < grd_id > _RI` except for the missing shared input ports). A flag `< twin >` is used to disable either of the actions. For example, when the action `Grd < grd_id > _RI` is enable, it will set the flag `< twin >` so that the guard of the action `Grd < grd_id > _RL` is falsified, therefore the action is disabled. Similarly, if another action sharing the input variable is enabled and consumes is variables, it would then set the input flags (`< set_input_read >`;), which in turn falsifies the guard of the action `Grd < grd_id > _RI` and helps approving the guard of `Grd < grd_id > _RL` by setting the up-to-date flags of the local variables (`< set_inp_loc_up_to_date >`;). Finally, the action sets its flag of guard to true

```
Grd<grd_id>_RI: action <input_channel_name>: [<input_var_name>], ...
                    ==>
 guard (<grd_boolean_expr>) and
       (<grd_flag_unset>) and
       (not <twin>) and
       (<loc_up_to_date>) and
       (<inp_unread>)
 do
    <assign_inp_to_local>;
    <set_twin>;
    <set_input_read>;
    <set_inp_loc_up_to_date>;
    <set_grd_flag>;
    <unset_mutex_grd_flags>;
 end

Grd<grd_id>_RL: action <input_channel_name>: [<input_var_name'>], ...
                    ==>
 guard (<grd_boolean_expr'>) and
       (<grd_flag_unset>) and
       (not <twin>) and
       (<loc_up_to_date'>) and
       (<inp_unread'>)
 do
    <assign_inp_to_local>;
    <set_twin>;
    <set_input_read>;
    <set_inp_loc_up_to_date>;
    <set_grd_flag>;
    <unset_mutex_grd_flags>;
 end
```

FIGURE 5.6: Translation template from guards to guard-CAL actions.

(< `set_grd_flag` >;), indicating the validity of the guard and unset the flags of those to be falsified to false (< `unset_mutex_grd_flags` >;). Note that by Proposition 5.1, once sufficient inputs are read by the actor, all flags of guards should have a value. Because we need to distinguish between true, false and "not assigned" for each guard flag, we use an integer rather than a boolean to encode the three states, and adapt corresponding tests on the flag. Instead, for flags of input consumption as well as local variable's update, we can simply use boolean variables.

5.2.2 Translate Actions to Act-CAL Actions

Actions of a clocked SGA is translated to an act-CAL action. The translation is relatively easier than the translation of guards. An action first needs to remember the guard flag of its guard, and wait for the time the flag is set to valid. The validity of its flag means not only that its guard is evaluated to true, but also that all input tokens it reads are arrived. Additionally, it also needs to wait for the validity of the local variables it reads (if any). Once these two conditions are fulfilled, it can carry out the assignment. Then

```
Act<ga_id>: action ==>
        <output_channel_name>: [<output_var_name>], ...
 guard (<grd_flag_valid>) and
       (<loc_up_to_date>)
 <output_var_declaration>
 do
    <assign_val_to_local>;
    <assign_val_to_output>;
    <set_loc_up_to_date>;
    <set_output_up_to_date>;
    <set_delay_up_to_date>;
 end
```

FIGURE 5.7: Translation template from actions to action-CAL actions.

based on the type of the assignment, it either assigns the value (evaluated from the right hand side) to the local variable or output variable for the immediate case, or a renamed variable indicating for the delayed case. Also, after the assignment the action should remember to set the up-to-date flag of the variable, so that other actions that are waiting on it can be enabled. The template of the translated act-CAL action is shown in Figure 5.7.

5.2.3 Additional Actions and FSM scheduling

Reaction to Absence

In order to complete the transformation, we still need some additional CAL actions. In particular, we need to complete the update of local variables. Previously we already introduced the act-CAL actions which actively update the local and output variables. However, still it is possible that during one macro-step, a local variable is never actively updated but still used by some other guarded actions for computation. In this case, the value of the local variable should be a default value, which in turn depends on the type of the variable (event or memorized). Therefore we need to implement CAL actions that assign the default values to the local variables for the case of reaction to absence. Another importance is that for the schedule of FSM related actions, which we will have detailed discussions in the next section. Note that the reaction to absence for value updates are only required by local variables, since both input and output variables are clocked variables, therefore they do not need a value when their clocks are false.

The job of the react-to-Abs actions are easy once we know the truth value of the guards of those guarded actions that may update the local variable, since by the synchronous semantics the variable is assigned a default value only all these guards evaluates to false. Therefore for each local variable, we generate a react-to-Abs action. The template of the action is shown in Figure 5.8, where the guard of the action is the conjunction of the test of the set of guards, in which each corresponds to a guarded action with an assignment that updates the variable.

```
RtAbs<ga_id>: action ==>
 guard (<grd_flag1_invalid>) and
       (<grd_flag2_invalid>) and
       ... and
       (<grd_flagn_invalid>)
 do
    <assign_default_to_local>;
    <set_loc_up_to_date>;
 end
```

FIGURE 5.8: Template for reaction to absence CAL actions.

For output variables, we also need react-to-Abs actions, but only for the update of the output flags and no default value is assigned. The output flag indicates that an output has been taken care of, i.e. either a token is produced or the output's clock is false. This information is important for the FSM scheduling later.

FSM scheduling

So far we have introduced the CAL actions that perform the testing of guards as well as execution of assignments. Are we done for the job? As we look back at the updates of flags and variables, we can find out at least the following tasks should be taken care of:

- reset of the input consumption flags as well as up-to-date flags of local and output variables
- update the local variables and output tokens for the delayed assignments.

These tasks should be taken care of before the next wave of macro-step transitions start, and should be performed after all the local variables and output ports are updated. We refer to the actions we introduced before together as *immediate actions* and these actions to the *delayed actions*, as shown in Figure 5.4. In order to schedule these tasks as planned, we utilize the *finite state machine scheduling* of CAL. As introduced in section 5.1, FSM scheduling assigns each action to a state, indicated by the prefix of the action's label. As long as the current state does not match the prefix of the action's label, the action is not considered for execution. Figure 5.9 shows the structure of the FSM we used for scheduling the immediate and delayed actions.

As shown in the FSM, we used three labels (**imm**, **to_delay**, **to_imm**) and two states (**imm**, **delay**) where the initial state is set as **imm**. All labels of immediate actions are now prefixed with `imm.`, so that initially these actions can be tested and executed. Also that any execution of an immediate action would still leads the next state to **imm**. The transition from state **imm** to state **delay** is performed by the action with label `to_delay`, which is shown in Figure 5.10. This action simply checks if all flags of output variables have been updated.

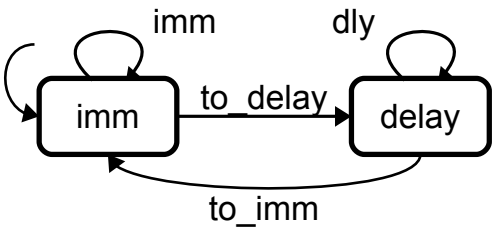


FIGURE 5.9: FSM scheduling.

```
to_delay: action ==>
 guard (<output1_flag_valid>) and
       (<output2_flag_valid>) and
       ... and
       (<outputn_flag_valid>)
 do

 end
```

FIGURE 5.10: Template for the to-delay CAL action.

Once in state **delay**, the delayed actions are executed. Figure 5.11 shows the template of the delayed actions. As shown in the figure, there are two types of delayed actions. The first type is a single action that takes care of unsetting all the flags, and it is executed immediately once the state is reached. The second type of actions is a set of actions for each variable that has a delayed assignment. Such an action checks if there has been a delayed assignment by checking the corresponding flag (by the if-statement), and if so it either updates the local variable or outputs a token, depending on the type of variable (in case of local variable, the output expression part would be empty). In the end of both type of actions, there is an assignment that sets the flag of the actions. As long as a delayed action is executed, the FSM's state remains still. The action `to_imm` monitors these flags, and once all set, this action brings back the state from **delay** to **imm** and starts the new wave of macro-step transitions. This action is also shown in Figure 5.11.

Correctness of Translation

It is not difficult to check that all the issues listed in the beginning of this section are tackled properly. The crucial part is of course to figure out when to transit from the immediate state to the delay state, as this in turn sets the boundary between different waves of macro-set transitions. The key to the correct decisions is Proposition 5.1, which ensures us by simply evaluation the approved guards we can also figure out the invalidation of other guards, and this process completely covers all guards. The rest of the translation are then technical details that are necessary to complete the preservation of synchronous semantics.

```
delay.flags: action ==>
guard true
do
   <unset_input_read>;
   <unset_loc_up_to_date>;
   <unset_output_up_to_date>;
   <set_twins>;
   <set_delay_flag>
end

delay.<output_id>: action ==> <output_channel_name>: [<output_var_name>]
guard true
do
   if <delay_flag_valid> {
       <assign_delayed_value>;
       <unset_delay_flag>;
   }
   <set_delay_output_id>;
end

to_imm: action ==>
guard (<delay_flag_set>) and
      (<delay_output1_flag_set>) and
      ... and
      (<delay_outputn_flag_set>)
do

end
```

FIGURE 5.11: Template for the delayed CAL action.

5.2.4 Case Study

As a simple and complete example showing the result of the transformation, Figure 5.12, Figure 5.13 and Figure 5.14 together shows the corresponding CAL actions of the synchronous component SITE.

5.3 Experimental Results

5.3.1 Translation to CAL Actors

We evaluated our method of translating endochronous components to CAL actors on the same set of examples as we used in section 4.3.3. Again, the whole set of experiments is carried out on a laptop with an 2.9 GHz Intel Core i7 processor and 8 GB DDR3 memory. Since all modules are middle to small sized, all transformations are successfully done within one second. For the measurement, we collect the size of the original `.aif` file of the Quartz programs' intermediate code and the size of the `.cal` files of the genearted CAL actors. We also collect the number of guarded actions of the Quartz intermediate codes as well as the number of CAL actions of the corresponding actors. Table 5.1

```
package mytest;

actor site () int(size=8) Input1, int(size=8) Input2, int(size=8) Input3
    ==> int(size=8) Output:

    // guard flags
    int g1 := 0;
    int g2 := 0;
    // input consumption flags
    bool ca := false;
    bool cb := false;
    bool cc := false;
    // output set flags
    bool cy := false;
    // local variable
    bool state := true;
    // local variable up-to-date flags
    bool cstate := false;
    // delay action flags
    bool delay_flags := false;
    bool delay_state_flag := false;

    // local temp vars for inputs
    int(size=8) ap;
    int(size=8) bp;
    int(size=8) cp;


    // immediate actions
    imm.grdr1: action Input1: [a], Input2: [b] ==>
    guard (g1=0) and (ca=false) and (cb=false) and (a=0)
    do
       ap:=a; bp:=b;
       ca:=true;cb:=true;
       g1:=1;g2:=-1;
    end

    imm.grdr2: action Input1: [a], Input3: [c] ==>
    guard (g2=0) and (ca=false) and (cc=false) and (a=1)
    do
       ap:=a; cp:=c;
       ca:=true;cc:=true;
       g1:=-1;g2:=1;
    end

    imm.actr1: action ==> Output: [y]
    guard (g1=1)
    var int(size=8) y
    do
       y:=bp;
       cy:=true;
    end
```

FIGURE 5.12: CAL actions for SITE (part 1).

```
imm.actr2: action ==> Output: [y]
guard g2=1
var int(size=8) y
do
   y:=cp;
   cy:=true;
end

// Reaction to Absence Actions
RtAbs_act_y: action ==>
guard (g1=-1) and (g2=-1)
do
   cy := true;
end

// immediate to delay action
imm.to_delay: action ==>
guard cy=true
do
end

// delay actions
delay.flags: action ==>
guard true
do
   ca := false;
   cb := false;
   cc := false;
   cy := false;
   state := true;

   g1 :=0; g2:=0;

   delay_flags = true;
end

delay.state: action ==>
guard true
do
   if (delay_state_set) {
       state:=state_delay;
   }
   delay_state_flag:=true;
end

// delay to immediate action
delay.to_imm: action ==>
guard delay_flags and delay_state_flag
do
end
```

FIGURE 5.13: CAL actions for SITE (continued part 2).

```
    // FSM scheduling
    schedule fsm imm :
    imm (imm) --> imm;
    imm (to_delay) --> delay;
    delay(delay) --> delay;
    delay(to_imm) --> imm;
    end

end
```

FIGURE 5.14: CAL actions for SITE (continued part 3).

shows the preliminary result of our model transformation, where the columns of SGA corresponds to the Quartz intermediate files and SGAs, and columns of CAL correspond to the `.cal` files and the CAL actions respectively.

TABLE 5.1: Experimental Results — Translation to CAL Actors

Example	**Size of File (KB)**		**No. of Actions**	
	SGA	CAL	SGA	CAL
Seq-Or	3	2	3	9
Seq-ITE	3	2	3	9
Gustave	8	11	11	42
OpDecode	26	28	23	81
Heating	32	38	51	174
FilterA	21	22	23	72
FilterB	43	53	42	146
FilterC	82	102	76	258

As shown in the table, the size of the `.cal` files are up to 30% larger than the `.aif` files, while the number of CAL actions are at basically triple the number of SGAs. This is because that the `.aif` file contains lots of abbreviations and assertions that are not used for the model transformation, but still took a lot of space of the file. Therefore the number of actions more precisely reflects the enlarge of the transformed models.

Translations of Synchronous Models

There are plenty of works previously done over the topic of translating synchronous models to different forms of systems. A survey [105] introduced a large set of techniques translating from different synchronous programs into distributed systems. The form of distributed systems includes Communication FSM [106] as well as distributed sequential programs with message passing communications [107]. Multi-threaded codes are later synthesized from polychronous models [97].

In the area of embedded software design, desynchronization methods can be dated

back to [108] where the main purpose is to distribute a synchronous program. A detailed survey is provided in [105]. In general, we can roughly derive two categories for previous methods: automaton-based (earlier works) and clock-based (later works). Automaton-based methods compile programs into automaton-like intermediate codes and then distribute these codes following the shared control flow distributed data flow principle [109]. They mostly concentrate on distributing uni-clocked synchronous programs, and seldom considered multi-clocked programs. Later works [96, 110] take different speed of system components–therefore multiclocked systems into consideration. In particular, for programs written in synchronous dataflow languages like Signal [79] and Lustre [29], it is possible to derive a *clock tree* describing clock relations between different system computations. Then a distribution based on clock relations can be derived, where least dependent components can be identified and distributed into different threads so that communication is optimized. This is much more difficult for imperative programs written in languages like Esterel [45] or Quartz [20], since clock relations are implicitly lying under program semantics. Except for [110], no other works on desynchronizing imperative programs are clock-based. This work is originally dssigned for Lustre, therefore assumes that all programs to be distributed should possess a base clock (that is the finest clock for all signals) and will reject a program otherwise. In works of [111–115] SGAs have been translated to OpenMP and MPI based multi-threaded codes as well as DPNs. However the works here basically still preserves the synchronous model of computation and did not consider desynchronization as a drastic model transformation as we did in this thesis. Indeed, communication and performance optimization is heavily considered and share some similarity with our desynchronization philosophy [115, 116].
Our work differentiates from all previous works in the sense that we deal with a multi-clocked synchronous model of computation, and we perform model transformation over such a synchronous model to a decoupled DPN – via desynchronization, with additional care of causality. In works of [81, 96] the theories did not consider causality, and the corresponding verification techniques are based on clock calculus. As we face the clocked SGAs, causality is an inherited problem that we can not avoid. Therefore we need to go from macro-step level deeper into micro-step level, and try to preserve the causal relations as we perform desynchronization. In the meantime, we do not to overly constraint the causal relations as works in [83, 85]. The distribution of Esterel into CFSMs [117] shared some similarity as it also considers the causality problem, however it fully preserves the synchronizations, therefore the derived distributed system would still be tightly coupled.

5.4 Summary

In this chapter we introduced the transformation from clocked SGAs to CAL actors in detail. We chose CAL mainly because of the similarity of its formalization to SGAs.

But we also benefit from CAL by its abstract modeling of DPNs as well as its capability to be synthesized to both software and hardware. For the transformation, we utilize proposition 5.1 derived from endochrony to make sure that all guards can be evaluated. This is crucial for the scheduling of control flow transitions as only after the actor makes sure that no more data-flow transitions can be executed, can the control flow move to the next wave of macro-step transitions. It is not difficult to check that our model transformation preserves the functional behavior of the original synchronous component. The preliminary experimental results shows the effectiveness as well as the efficiency of the transformation.

Chapter 6

Conclusion

6.1 Conclusion of the Thesis

This thesis faces two design difficulties of the development of distributed embedded systems:

- The difficulty in direct analysis of asynchronous models of computation,
- The inefficiency in classic synchronous models of computation.

and presents desynchronization as a solution. In particular, desynchronization starts from a synchronous model – a synchronous process network that consists of a synchronous composition of synchronous components, and transforms the synchronous network into an asynchronous network of components – a dataflow processing network.

The transformation's goal is to preserve the functional behavior of the original synchronous network in operational level. Therefore a correct transformation must preserve the causal relations of the synchronous network. Furthermore, the transformation tries to aggressively decouple the synchronous components from each other by removing absent signal transfers from communication. This aggressive communication optimization in turn is not working every time. In order to preserve the functional behavior of each component as well as the global network, we developed a theory of sufficient conditions – which includes a global condition and a local condition. The global condition is also the correctness requirement of a synchronous system in general, and therefore can be taken as an assumption. The difficulty lies in the local condition, which might not hold for all synchronous components in general. For practical verification of this local condition, we further introduced a sufficient condition – endochrony –that can be verified directly against the transition system of the synchronous component. The efficiency of the verification is validated via experiments. Finally, we implemented a model transformation method to transform endochronous components into CAL actors. The transformation utilizes the endochrony of the components and can be performed very efficiently. The

resulted code of CAL actors still maintain reasonably sizes, and can be further synthesized into multi-threaded software or hardware. This model transformation effectively validates our desynchronization theory.

6.2 Future Works

There are still future works to be done, as listed in the following directions:

- Explore different sufficient conditions for local confluence: in this thesis we applied endochrony as the chosen sufficient condition for the local condition of desynchronization criteria. There are also other possibilities like weak-endochrony. As stated in the thesis, although the author believes that weak-endochrony is more related to partitioning, it is still of independent interest for us to find out how weak-endochrony can be efficiently verified.
- Partitioning of synchronous systems: in our thesis the desynchronization design flow starts from a synchronous process network, which means that the synchronous system is already partitioned. In general we may face a synchronous system as a single entity, and may need to find out an "optimal partition" of the system for desynchronization. The partition's criteria may include: better synchronization reduction, physical location constraints and so on.
- Decoupling single-clocked synchronous systems: in this thesis our aggressive desynchronization utilises the multi-clock nature of synchronous process. If the given synchronous process is single-clocked, then we currently have no chance to reduce its communication. However it is possible to employ analysis methods like [90–92] to synthesize multi-clocks from single clocked systems, and perform desynchronization after. Further more, we may be able to solve the problem of synthesizing multi-clocked systems and ensuring the derived system fulfills the desynchronization criteria at the same time.
- Property preservation of desynchronization: in this thesis we already proved that once fulfilling the desynchronization criteria, the derived DPN is deadlock free and has bounded buffers. In general, we wonder that what other properties can be preserved after desynchronization. Some preliminary works has been done [118], but there are definitely more to explore.
- Practical issues: the major work of this thesis is a theory for correct desynchronization with the consideration of causal relations. The experimental results are developed mainly for the validation of the effectiveness of the theory. There are more practical optimization can be done in order to improve the efficiency of the desynchronized DPN. Also, we proposed a model-transformation that targets DPN as our target model. Other asynchronous models (e.g. asynchronous shared memory) can also be explored for synthesis of synchronous systems onto different architectures. Finally, more experiments need to be done for different target platforms to practice and improve the theory.

Bibliography

[1] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM (CACM)*, 21(8):666–677, 1978.

[2] R. Milner. *Communication and Concurrency*. Prentice Hall, London, UK, 1989.

[3] Wolfgang Reisig and Grzegorz Rozenberg, editors. *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets*, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-65306-6.

[4] N. Lynch. *Distributed Algorithms*. Data Management Systems. Morgan Kaufmann, 1997.

[5] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992. ISBN 0-201-54856-9.

[6] E.A. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83 (5):773–801, May 1995.

[7] M. Geilen and T. Basten. Requirements on the execution of Kahn process networks. In P. Degano, editor, *European Symposium on Programming (ESOP)*, volume 2618 of *LNCS*, pages 319–334, Warsaw, Poland, 2003. Springer.

[8] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science (TCS)*, 5(3):223–255, December 1977.

[9] E.W. Stark. On the relations computable by a class of concurrent automata. In *Principles of Programming Languages (POPL)*, pages 329–340, San Francisco, California, USA, 1990. ACM.

[10] E.W. Stark. Concurrent transition system semantics of process networks. In *Principles of Programming Languages (POPL)*, pages 199–210. ACM, 1987.

[11] A.A. Faustini. An operational semantics for pure data flow. In M. Nielsen and E.M. Schmidt, editors, *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 140 of *LNCS*, pages 212–224, Ã. . . rhus, Denmark, 1982. Springer.

[12] J.T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, University of California, Berkeley, California, USA, 1993. PhD.

[13] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, April 1983. ISSN 0004-5411.

[14] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *International Conference on Functional Programming (ICFP)*, pages 226–238, Philadelphia, Pennsylvania, USA, 1996. ACM.

[15] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: A declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM. ISBN 0-89791-215-2.

[16] Philip Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 45–52, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3.

[17] D.S. Scott. Outline of a mathematical theory of computation. Technical Monograph PRG-2, Oxford University, Computing Laboratory, Programming Research Group, Oxford, UK, November 1970.

[18] G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Information Processing*, pages 471–475, Stockholm, Sweden, 1974. North-Holland.

[19] J. Brock and W. Ackerman. Scenarios: A model of nondeterminate computation. In J. Diaz and I. Ramos, editors, *Formalization of Programming Concepts*, volume 107 of *LNCS*, pages 252–259, Peniscola, Spain, 1981. Springer.

[20] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

[21] Robin Milner. Models of lcf. Technical report, Stanford, CA, USA, 1973.

[22] R.M. Keller. Denotational models for parallel programs with indeterminate operators. In E.J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 337–366. North-Holland, 1978.

[23] J.N. Kok. Denotational semantics of nets with nondeterminism. In B. Robinet and R. Wilhelm, editors, *European Symposium on Programming (ESOP)*, volume 213 of *LNCS*, pages 237–249, Saarbruecken, Germany, 1986. Springer.

[24] P. Panangaden and E. Stark. Computations, residuals, and the power of indeterminacy. In T. Lepistoe and A. Salomaa, editors, *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 317 of *LNCS*, pages 439–454, Tampere, Finland, 1988. Springer.

[25] James R. Russell. Full abstraction for nondeterministic dataflow networks, June 1989.

[26] E.W. Stark. A simple generalization of Kahn's principle to indeterminate dataflow networks (extended abstract). In *Semantics for Concurrency*, pages 157–174. Springer, 1990.

[27] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[28] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, November 1992. ISSN 0167-6423.

[29] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[30] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, September 1991.

[31] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. ISBN 3540607617.

[32] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[33] G. Berry. The Esterel v5 language primer, July 2000.

[34] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logic and Models of Concurrent Systems*, pages 477–498. Springer, 1985.

[35] D. Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[36] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Readings in hardware/software co-design. chapter Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems, pages 527–543. Kluwer Academic Publishers, Norwell, MA, USA, 2002. ISBN 1-55860-702-1.

[37] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science (TCS)*, 94 (1):125–140, March 1992.

[38] Louis Mandel and Marc Pouzet. Reactiveml: A reactive extension to ml. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 82–93, New York, NY, USA, 2005. ACM. ISBN 1-59593-090-6.

[39] P. Caspi and M. Pouzet. The lucid synchrone distribution. URL http://www-spi.lip6.fr/~pouzet/lucid-synchrone/.

[40] B.A. Jose and S.K. Shukla. An alternative polychronous model and synthesis methodology for model-driven embedded software. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 13–18, Taipei, China, 2010. IEEE Computer Society.

[41] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1990. ISBN 020102988X.

[42] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In J.C.M. Baeten and S. Mauw, editors, *Concurrency Theory (CONCUR)*, volume 1664 of *LNCS*, pages 162–177, Eindhoven, The Netherlands, 1999. Springer.

[43] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, 163(1):125–171, 2000.

[44] David Harel, Amir Pnueli, Jeanette P. Schmidt, and Rivi Sherman. On the formal semantics of statecharts (extended abstract). In *Proceedings of the Symposium on Logic in Computer Science (LICS'87)*, pages 54–64, Ithaca, New York, USA, June 1987.

[45] G. Berry. The constructive semantics of pure Esterel, July 1999.

[46] Dumitru Potop-Butucaru and Benoit Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In *Proceedings of the Fifth International Conference on Application of Concurrency to System Design*, ACSD '05, pages 48–57, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2363-3.

[47] J. Brandt and K. Schneider. Separate compilation for synchronous programs. In H. Falk, editor, *Software and Compilers for Embedded Systems (SCOPES)*, volume 320 of *ACM International Conference Proceeding Series*, pages 1–10, Nice, France, 2009. ACM.

[48] David R. Coelho. *The VHDL Handbook*. Kluwer Academic Publishers, Norwell, MA, USA, 1989. ISBN 0792390318.

[49] G. Berry and E. Sentovich. Multiclock Esterel. In T. Margaria and T.F. Melham, editors, *Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*, pages 110–125, Livingston, Scotland, UK, 2001. Springer.

[50] M. Gemuende. *Clock Refinement in Imperative Synchronous Programs*. PhD thesis, Department of Computer Science, University of Kaiserslautern, Germany, Kaiserslautern, Germany, October 2013. PhD.

[51] B. Rajan and R.K. Shyamasundar. Multiclock ESTEREL: A reactive framework for asynchronous design. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 201–209, CancÃºn, Quintana Roo, Mexico, 2000. IEEE Computer Society.

[52] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer Publishing Company, Incorporated, 2nd edition, 2010. ISBN 1441941193, 9781441941190.

[53] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

[54] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 20(9):1059–1076, 2001.

[55] M. Singh and M. Theobald. Generalized latency-insensitive systems for single-clock and multi-clock architectures. In *Design, Automation and Test in Europe (DATE)*, pages 1008–1013, Paris, France, 2004. IEEE Computer Society.

[56] Sava Krstic, Jordi Cortadella, Mike Kishinevsky, and John O'Leary. Synchronous elastic networks. In *Proceedings of the Formal Methods in Computer Aided Design*, FMCAD '06, pages 19–30, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2707-8.

[57] J. Cortadella, M. Galceran-Oms, and M. Kishinevsky. Elastic systems. In L. Carloni and B. Jobstmann, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 149–158, Grenoble, France, 2010. IEEE Computer Society.

[58] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin. Elastic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 28(10):1437–1455, October 2009.

[59] Jean Berstel. *Transductions and Context-Free Languages*. Teubner Studienbücher, Stuttgart, 1979.

[60] D. Potop-Butucaru, R. de Simone, and Y. Sorel. Necessary and sufficient conditions for deterministic desynchronization. In C.M. Kirsch and R. Wilhelm, editors, *Embedded Software (EMSOFT)*, pages 124–133, Salzburg, Austria, 2007. ACM.

[61] Y. Bai and K. Schneider. Isochronous networks by construction. In *Design, Automation and Test in Europe (DATE)*, Dresden, Germany, 2014. IEEE Computer Society.

[62] Daniel Marcos Chapiro. *Globally-asynchronous Locally-synchronous Systems (Performance, Reliability, Digital)*. PhD thesis, Stanford, CA, USA, 1985. AAI8506166.

[63] Jens Muttersbach, Thomas Villiger, and Wolfgang Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, ASYNC '00, pages 52–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0586-4.

[64] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975. ISSN 0001-0782.

[65] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Austin, Texas, USA, May 1989.

[66] D.L. Dill. The Murphi verification system. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 390–393, New Brunswick, New Jersey, USA, 1996. Springer.

[67] H. Jaervinen and R. Kurki-Suonio. The DisCo language and temporal logic of actions. Technical Report 11, Tampere University of Technology, Software Systems Laboratory, 1990.

[68] J. Brandt, M. Gemünde, and K. Schneider. From synchronous guarded actions to SystemC. In M. Dietrich, editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 187–196, Dresden, Germany, 2010. Fraunhofer Verlag.

[69] J. Brandt and K. Schneider. Separate translation of synchronous programs to guarded actions. Internal Report 382/11, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, March 2011.

[70] J. Brandt, M. Gemünde, K. Schneider, S. Shukla, and J.-P. Talpin. Integrating system descriptions by clocked guarded actions. In A. Morawiec, J. Hinderscheit, and O. Ghenassia, editors, *Forum on Specification and Design Languages (FDL)*, pages 1–8, Oldenburg, Germany, 2011. IEEE Computer Society.

[71] J. Brandt, K. Schneider, and S.A. Edwards. Translating SHIM to guarded actions. Technical Report 387/12, University of Kaiserslautern, Kaiserslautern, Germany, February 2012.

[72] J. Brandt, M. Gemünde, K. Schneider, S.K. Shukla, and J.-P. Talpin. Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions. *Design Automation for Embedded Systems (DAEM)*, July 2012. DOI 10.1007/s10617-012-9087-9.

[73] J. Brandt, M. Gemünde, K. Schneider, S.K. Shukla, and J.-P. Talpin. Embedding polychrony into synchrony. *IEEE Transactions on Software Engineering (TSE)*, 39(7):917–929, July 2013.

[74] K. Schneider. *Verification of Reactive Systems – Formal Methods and Algorithms.* Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[75] J. Sifakis. A unified approach for studying the properties of transition systems. *Theoretical Computer Science (TCS)*, 18:227–258, 1982.

[76] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Logic in Computer Science (LICS)*, pages 1–33, Washington, District of Columbia, USA, 1990. IEEE Computer Society.

[77] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 72–83, London, UK, UK, 1997. Springer-Verlag. ISBN 3-540-63166-6.

[78] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Maximal causality analysis. In J. Desel and Y. Watanabe, editors, *Application of Concurrency to System Design (ACSD)*, pages 106–115, Saint-Malo, France, 2005. IEEE Computer Society.

[79] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal of Circuits, Systems, and Computers (JCSC)*, 12(3):261–304, June 2003.

[80] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. In *Application of Concurrency to System Design (ACSD)*, pages 67–76, Hamilton, Ontario, Canada, 2004. IEEE Computer Society.

[81] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design (FMSD)*, 28(2):111–130, 2006.

[82] Dumitru Potop-Butucaru and Benoit Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In *Proceedings of the Fifth International Conference on Application of Concurrency to System Design*, ACSD '05, pages 48–57, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2363-3.

[83] D. Potop-Butucaru, R. de Simone, Y. Sorel, and J.-P. Talpin. From concurrent multi-clock programs to deterministic asynchronous implementations. In S. Edwards, R. Lorenz, and W. Vogler, editors, *Application of Concurrency to System Design (ACSD)*, pages 42–51, Augsburg, Germany, 2009. IEEE Computer Society.

[84] D. Potop-Butucaru, Y. Sorel, R. de Simone, and J.-P. Talpin. Correct-by-construction asynchronous implementation of modular synchronous specifications. *Fundamenta Informaticae*, 108(1-2):91–118, 2011.

[85] J.-P. Talpin, C. Brunette, T. Gautier, and A. GamatiÃ©. Polychronous mode automata. In S.L. Min and W. Yi, editors, *Embedded Software (EMSOFT)*, pages 83–92, Seoul, South Korea, 2006. ACM.

[86] Y. Bai, K. Schneider, N. Bhardwaj, B. Katti, and T. Shazadi. From clock-driven to data-driven models. In *Formal Methods and Models for Codesign (MEMOCODE)*, Lausanne, Switzerland, 2014. IEEE Computer Society.

[87] J.C. King. Symbolic execution and program testing. *Communications of the ACM (CACM)*, 19(7):385–394, July 1976.

[88] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013. ISSN 0001-0782.

[89] Y. Bai. Dependency analysis of synchronous programming languages. Master's thesis, Department of Computer Science, University of Kaiserslautern, Germany, November 2010. Master.

[90] Y. Bai, J. Brandt, and K. Schneider. Data-flow analysis of extended finite state machines. In B. Caillaud, J. Carmona, and K. Hiraishi, editors, *Application of Concurrency to System Design (ACSD)*, pages 163–172, Newcastle Upon Tyne, England, UK, 2011. IEEE Computer Society.

[91] Y. Bai, J. Brandt, and K. Schneider. SMT-based optimization for synchronous programs. In S. Stuijk, editor, *Software and Compilers for Embedded Systems (SCOPES)*, pages 11–20, St. Goar, Germany, 2011. ACM.

[92] J. Brandt, K. Schneider, and Y. Bai. Passive code in synchronous programs. *Transactions on Embedded Computing Systems (TECS)*, 13(2):67, January 2014.

[93] The model checker nusmv. URL http://nusmv.fbk.eu/.

[94] The smt solver z3. URL http://z3.codeplex.com/.

[95] G. Huet. Formal structures for computation and deduction, May 1986.

[96] J.-P. Talpin, J. Ouy, T. Gautier, L. Besnard, and P. Le Guernic. Compositional design of isochronous systems. *Science of Computer Programming*, 77(2):113–128, February 2012.

[97] M. Nanjundappa, M. Kracht, J. Ouy, and S. Shukla. A new multi-threaded code synthesis methodology and tool for correct-by-construction synthesis from polychronous specifications. In J. Carmona Mihai, T. Lazarescu, and M. Pietkiewicz-Koutny, editors, *Application of Concurrency to System Design (ACSD)*, pages 21–30, Barcelona, Spain, 2013. IEEE Computer Society.

[98] B.A. Jose, S.K. Shukla, H.D. Patel, and J.-P. Talpin. On the deterministic multi-threaded software synthesis from polychronous specifications. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 129–138, Anaheim, California, USA, 2008. IEEE Computer Society.

[99] Dumitru Potop-Butucaru, Robert de Simone, Yves Sorel, and Jean-Pierre Talpin. Clock-driven distributed real-time implementation of endochronous synchronous programs. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 147–156, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-627-4.

[100] B.A. Jose, H.D. Patel, S.K. Shukla, and J.-P. Talpin. Generating multi-threaded code from polychronous specifications. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 238(1):57–69, June 2009.

[101] Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan. Software code generation for the rvc-cal language. *J. Signal Process. Syst.*, 63(2):203–213, May 2011. ISSN 1939-8018.

[102] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. URL http://ptolemy.org/books/Systems.

[103] Johan Eker and Jörn Janneck. Caltrop—language report (draft). Technical memorandum, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, 2002. http://www.gigascale.org/caltrop.

[104] Shuvra S. Bhattacharyya, Gordon Brebner, Jörn W. Janneck, Johan Eker, Carl von Platen, Marco Mattavelli, and Mickaël Raulet. Opendf: A dataflow toolset for reconfigurable hardware and multicore systems. *SIGARCH Comput. Archit. News*, 36(5):29–35, June 2009. ISSN 0163-5964. doi: 10.1145/1556444.1556449. URL http://doi.acm.org/10.1145/1556444.1556449.

[105] A. Girault. A survey of automatic distribution method for synchronous programs. In *Synchronous Languages, Applications, and Programming (SLAP)*, pages 1–20, Edinburgh, Scotland, UK, 2005. unpublished workshop proceedings.

[106] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara, editors. *Hardware-software Co-design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1997. ISBN 0-7923-9936-6.

[107] K. Sun, L. Besnard, and T. Gautier. Optimized distribution of synchronous programs via a polychronous model. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 42–51, Lausanne, Switzerland, 2014. IEEE Computer Society.

[108] P. Caspi and A. Girault. Execution of distributed reactive systems. In S. Haridi, K. Ali, and P. Magnusson, editors, *European Conference on Parallel Processing (EuroPar)*, volume 966 of *LNCS*, pages 15–26, Stockholm, Sweden, 1995. Springer.

[109] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2(2):151–169, 1988.

[110] A. Girault, X. Nicolin, and M. Pouzet. Automatic rate desynchronization of embedded reactive programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(3):687–717, August 2006.

[111] D. Baudisch, J. Brandt, and K. Schneider. Multithreaded code from synchronous programs: Extracting independent threads for OpenMP. In *Design, Automation and Test in Europe (DATE)*, pages 949–952, Dresden, Germany, 2010. EDA Consortium.

[112] D. Baudisch, J. Brandt, and K. Schneider. Multithreaded code from synchronous programs: Generating software pipelines for OpenMP. In M. Dietrich, editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 11–20, Dresden, Germany, 2010. Fraunhofer Verlag.

[113] D. Baudisch, J. Brandt, and K. Schneider. Translating synchronous systems to data-flow process networks. In S.-S. Yeo, B. Vaidya, and G.A. Papadopoulos, editors, *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 354–361, Gwangju, Korea, 2011. IEEE Computer Society.

[114] D. Baudisch. *Synthesis of Synchronous Programs to Parallel Software Architectures*. PhD thesis, Department of Computer Science, University of Kaiserslautern, November 2013. PhD.

[115] D. Baudisch, Y. Bai, and K. Schneider. Reducing the communication of message-passing systems synthesized from synchronous programs. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Turin, Italy, 2014. IEEE Computer Society.

[116] D. Baudisch and K. Schneider. Evaluation of speculation in out-of-order execution of synchronous data-flow networks. *International Journal of Parallel Programming (IJPP)*, 43(1):1–44, February 2015. http://dx.doi.org/10.1007/s10766-013-0277-2.

[117] G. Berry and E.M. Sentovich. An implementation of constructive synchronous programs in POLIS. *Formal Methods in System Design (FMSD)*, 17(2):135–161, October 2000.

[118] Y. Bai, J. Brandt, and K. Schneider. Preservation of LTL properties in desynchronized systems. In S. Shukla, L. Carloni, D. Kroening, and J. Brandt, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 53–63, Arlington, Virginia, USA, 2012. ACM.

Lebenslauf

Persönliches

Yu Bai
Chinesisch

Studium und Ausbildung

09/2002 – 07/2006	**Guilin University of Electronic Technology, China** **Bachelor of Science**
09/2006 – 01/2008	**Hebei University, China**
03/2008 – 10/2010	**TU Kaiserslautern,** **Master of Science**
Seit 12/2010	**TU Kaiserslautern, Eingebetteter Systeme Group,** **Doktorand, Informatik**