UNIVERSITY OF HERTFORDSHIRE

# Analysis and Coordination of Mixed-criticality Cyber-physical Systems

by

Simon Maurer

A thesis submitted to the University of Hertfordshire
in partial fulfillment of the requirements of the degree of

Doctor of Philosophy

March 2018

UNIVERSITY OF HERTFORDSHIRE

# *Abstract*

Centre for Computer Science and Informatics Research (CCSIR)
School of Computer Science

Doctor of Philosophy

by Simon Maurer

A Cyber-physical System (CPS) can be described as a network of interlinked, concurrent computational components that interact with the physical world. Such a system is usually of reactive nature and must satisfy strict timing requirements to guarantee a correct behaviour. The components can be of mixed-criticality which implies different progress models and communication models, depending whether the focus of a component lies on predictability or resource efficiency.

In this dissertation I present a novel approach that bridges the gap between stream processing models and Labelled Transition Systems (LTSs). The former offer powerful tools to describe concurrent systems of, usually simple, components while the latter allow to describe complex, reactive, components and their mutual interaction. In order to achieve the bridge between the two domains I introduce the novel LTS *Synchronous Interface Automaton (SIA)* that allows to model the interaction protocol of a process via its interface and to incrementally compose simple processes into more complex ones while preserving the system properties. Exploiting these properties I introduce an analysis to identify permanent blocking situations in a network of composed processes. SIAs are wrapped by the novel component-based coordination model *Process Network with Synchronous Communication (PNSC)* that allows to describe a network of concurrent processes where multiple communication models and the co-existence and interaction of heterogeneous processes is supported due to well defined interfaces.

The work presented in this dissertation follows a holistic approach which spans from the theory of the underlying model to an instantiation of the model as a novel coordination language, called *Streamix*. The language uses network operators to compose networks of concurrent processes in a structured and hierarchical way. The work is validated by a prototype implementation of a compiler and a Run-time System (RTS) that allows to compile a Streamix program and execute it on a platform with support for ISO C, POSIX threads, and a Linux operating system.

# *Acknowledgements*

First, my gratitude goes to the University of Hertfordshire for providing the funding and infrastructure for my PhD and for giving me the opportunity to start a career in academia.

I would like to thank my primary supervisor and research mentor Dr Raimund Kirner for the support of my PhD and related research. I am grateful for his commitment to provide me with guidance even late at night or on weekends. His broad knowledge and his patience were a huge help throughout the whole of the PhD.

Further, I would like to express my gratitude to Dr Olga Tveretina, my secondary supervisor, for her valuable inputs and her mental support. My thanks also go to Dr Prof Alex Shafarenko, leader of the research group Compiler Technology and Computer Architecture (CTCA), for his invaluable feedback.

I thank Giovina for her patience and her interest in abstract topics that are not hers. Her cheerful and supportive attitude were a big help throughout this project and I am ever grateful to her for leading me onto this path. Her hilarious personifications of certain aspects of my work, complete with pictures and all, is only one of many examples of her brilliant way of supporting me.

Although I do not know these people personally I want to give a shout out to Randall Munroe for his book "Thing Exaplainer - Complicated Stuff in Simple Words", showing that complicated things can be explained with only a 1000 words and Linda Liukas, for her book "Hello Ruby", advocating the fact that computer science is not as difficult as common belief suggests and that it can be taught to anyone.

Last but not least, I want to thank my family and friends from abroad for their continuous support and their willingness to pay us visit on a regular basis.

# Contents

# List of Figures

# List of Definitions

# Acronyms

**AST** Abstract Syntax Tree. 135

**BIP** Behaviour, Interaction, Priorities. 151, 154, 160

**CCI** Cross-criticality Interface. 7, 8, 20, 45, 63, 68, 89, 119, 149, 157, 159

**CFS** Completely Fair Scheduler. 145

**CPS** Cyber-physical System. i, vii, 1–9, 11, 13, 15, 19, 20, 23, 24, 44, 46, 85, 87, 90, 97, 108, 115, 118, 119, 145, 149, 150, 154, 156, 158, 159

**CSP** Communicating Sequential Processes. 4, 155

**DAG** Directed Acyclic Graph. 71, 81

**DFS** Depth-first Search. 81

**EDF** Earliest Deadline First. 145

**FIFO** First-in, First-out. viii, 41, 42, 46, 49–51, 53, 57, 63, 90, 91, 93, 112, 120, 122, 124, 126, 127, 142, 143, 150, 151, 155

**GML** Graph Modelling Language. 139, 141

**IA** Interface Automaton. 3, 16, 147, 148, 153, 157, 159

**IIOA** Interface Input/Output Automaton. 148

**IOA** Input/Output Automaton. 147, 148

**IOT** Internet of Things. 13, 15

**KPN** Kahn Process Network. 3, 21, 67, 150, 152

**LET** Logical Execution Time. 24, 152

**LTS** Labelled Transition System. i, vii, 3, 6, 8, 9, 38, 83

**MIMO** Multiple Input, Multiple Output. 26, 88

**MIOA** Modal Input/Output Automaton. 148, 153

**MIRT** Minimal Inter-release Time. 56–60, 132

**MoC** Model of Computation. 152

**MPI** Message Passing interface. 4

**NoC** Network on Chip. 67

**PBRT** Period-bounded Release Time. 56, 59, 60

**PNSC** Process Network with Synchronous Communication. i, vii, viii, 3, 5, 8–11, 13, 25–28, 33–37, 39, 41–47, 52–55, 60, 65–67, 70–77, 80, 83, 85, 88–90, 119, 120, 129, 131, 142, 143, 145, 148, 149, 151, 152, 154, 156–161

**RTS** Run-time System. i, 10, 11, 67, 93, 95, 120, 121, 123–125, 127–129, 131, 132, 141–146, 157, 158, 161

**SDF** Synchronous Data Flow. 21, 152

**SIA** Synchronous Interface Automaton. i, vii, viii, 2, 3, 6–11, 25, 28–43, 47, 48, 50, 51, 53, 67, 69–80, 82–84, 88–92, 94, 95, 120, 121, 123, 134, 135, 140–145, 148, 151, 156–161

**SISO** Single Input, Single Output. 150

**TTA** Time-triggered Architecture. 21, 52, 69, 149

**WCET** Worst-case Execution Time. 20, 52, 54, 67, 145, 160

**XML** Extensible Markup Language. 139

# Chapter 1

# Introduction

Nowadays, with microcontrollers getting smaller and more efficient, computing is becoming more and more pervasive in our everyday life. This is achieved by embedding computer devices in physical objects such that the computing devices interact with the physical world. Such systems are called Cyber-physical Systems (CPSs). A CPS is a reactive system that senses its environment (the physical world), performs a computation on a computational entity (this can be anything from a simple embedded device to a large scale distributed system), and then actuates on the environment according to the computations. The actuation on the environment causes the environment to change which, in turn, is detected by the sensors and the computation is performed with the new dataset. Typically, this reactive loop is time-critical and is executed as long as the system is running. In contrast to a traditional embedded system, i.e. a time-critical system, dedicated to a single hardware platform, a CPS tends to be an assembly of networked subsystems where some subsystems interact with the physical world and some may be purely computational.

Examples of CPSs can be found in the domain of automotive vehicles (e.g. anti-lock braking system, adaptive cruise control, electronic stability control, platooning), avionic vehicles (e.g. flight control systems, black box, pressure control), or smart spaces (e.g. intelligent highway control, building control), to name only a few.

Because of the direct interaction of CPSs with the physical world, it is often crucial to respect timing requirements imposed by the physical world to guarantee a correct behaviour of the system. In case of critical applications such as nuclear power plants, avionic systems, or cars, huge efforts are made to verify the correct behaviour of the application. The main difference between critical systems and best-effort systems is that critical systems are designed for the worst case whereas best-effort systems are designed for the average case. Henzinger and Sifakis argue that over time, this difference led to a gap between the models employed in the two domains and that the gap is continuing to widen [1]. Due to the difference of the employed models, critical and non-critical applications tend to be physically separated and run on dedicated hardware platforms. This is a problem because with the evolution of hardware towards multi/many-core architectures, there is the interest to integrate components with different criticality requirements on the same platform. Such systems are typically called mixed-criticality systems [2]. What adds further to the challenge of integrating CPSs on a multi/many-core architecture is that CPSs are often heterogeneous in the sense that several applications from different domains with different characteristics must coexist or interact with each other [3]. Eker et al. address this challenge by assembling multiple models, each suitable for its specific application domain, in one framework [3]. Others argue that one meta model, allowing to describe and compose heterogeneous systems, is more beneficial because it provides a better ground for a meaningful analysis of the system [1, 4, 5].

Another challenge is the inherent concurrency of CPSs [6]. Streaming networks are well-recognised for coping with concurrent systems [7]. They consist of processing nodes (often called filters) connected via communication channels where a channel is connected to a single producer node and a single consumer node. This property of streaming networks, combined with the implicit synchronisation of producers and consumers due to a blocking channel access, allows to tame the complexity of concurrent systems which makes them also an interesting paradigm to apply to CPSs. However, current streaming models (e.g. [8–10]) tend to rely on the possibility that a system can be decomposed into transformational components such that the behaviour of a component can be described as a pure function. Such a decomposition makes it easier to understand component dependencies and allows to analyse the system, e.g. for schedulability [11] or deadlocks [12]. However, given that CPSs are often systems of reactive nature, such a decomposition is difficult [13].

An interesting approach where no decomposition in transformational components is required are interface theories [14] which allow to describe components with arbitrary behaviour by their interfaces and build complex components out of simple ones by composition. Several interface models have been proposed to describe communication compatibility between components [15–17] as well as additional properties such as modalities [18], resource usage [19, 20], or timing constraints [21, 22]. However, these models do not target streaming networks and lack the capability of describing the blocking semantics of message passing in streaming networks.

In this dissertation I introduce a novel automata-based interface description model, called Synchronous Interface Automaton (SIA), that allows to describe the interaction protocol of processes with their environment. SIAs are suitable to describe the blocking semantics of Kahn Process Network (KPN)-based [23] streaming networks and thus serve as a powerful tool to bridge the gap between Labelled Transition System (LTS) and stream processing. An incremental composition operation allows to build complex processes out of simple ones. A novelty of the SIA model is that it allows to identify permanent blocking situations (e.g. deadlocks) in a composed network due to the blocking semantics of the model. The inspiration for the SIA model stems from Interface Automata (IAs) [15]. The blocking semantics of the here presented SIAs differs fundamentally from the blocking semantics of IAs. This enables SIAs to describe process networks where processes interact with synchronous communication, e.g. stream processing applications.

I further introduce a novel component-based model, called Process Network with Synchronous Communication (PNSC) that serves as a wrapper for SIAs and allows to model an assembly of reactive processes, i.e. processes capable of consuming and producing streams of infinite length. I extend the model to support the coexistence and interaction of processes with an event-triggered and a time-triggered execution scheme. In the former case, sporadically occurring events are causing a subsystem to perform its computation while in the latter case a fixed schedule imposes time instances when a subsystem is performing its computation. There are application domains where both models are required (e.g. automotive domain) but the two models are often used in limited ways to just co-exist but do not directly interact with each other (e.g. [24–26]). The problem is that interaction may cause interference from a subsystem of low criticality towards a subsystem of high criticality which must be avoided. A novelty of the extended PNSC model is that it allows not only the co-existence of the two triggering semantics in the same system but also allows interaction between time-triggered subsystems and event-triggered subsystems. The key point is to guarantee that the event-triggered subsystem, having a lower criticality level, is not interfering with the time-triggered subsystem of higher criticality. Further, the model allows to use the same mechanism to avoid interference from a lower critical event-triggered subsystem to a higher critical event-triggered subsystem. Two time-triggered subsystems do not interfere with each other out of construction [27], independent of their criticality level.

To provide control on communication bandwidth usage, the model allows to limit the communication rate of a process to an upper bound. This is achieved with a novel approach of controlling message passing between processes with different consequences depending on the message semantics.

A challenge related to the inherent concurrency of CPSs is the efficient execution of concurrent systems on multi-core hardware platforms. With the ever growing capabilities of integrated circuits due to the persistence of Moore's law, software engineers face the challenge to design, develop, and maintain complex software systems that exploit the available computation power. As a consequence of reaching the power wall through frequency scaling [28], parallel hardware architectures have been designed. Even in the domain of embarrassingly parallelisable applications where it is easy to split the computation load in a large number of independent computational chunks due to lack of data or code dependencies, parallelization remains not a simple task because of the tight coupling between hardware and software (e.g. efficient use of memory hierarchy) and the inherent difficulty of debugging parallel code. Nowadays, there are tools, libraries, and languages available that help to cope with some of the problems. In the domain of Big Data on large scale distributed systems examples are Hadoop[1], HPCC[2], and Hydra[3]. For shared memory multi-core architectures some examples are Cuda[4], OpenMP[5], and TBB[6].

However, multi- and many-core processor architectures have emerged to a broad variety of application fields, including CPSs, where it is hard to identify potential blocks that can exhibit parallelism due to dependencies between tasks and, especially, due to the reactive nature of CPSs. In their survey on programming solutions for multicore architectures in the domain of CPSs, Castrillon et al. note that even though achievements were made in academia, in industry, CPS software development for parallel architectures remains mostly manual [5]. A reason for this is that industry often relies on legacy code that may include system libraries or multiple layers of mixed languages which is often ignored by academic solutions (e.g. introducing a new language requires industry to rewrite lots of legacy source code in the new language which they may be hesitant to do).

To cope with concurrency in applications, programming languages either incorporate models (e.g. Actor model [29] in Scala, Communicating Sequential Processes (CSP) [30] in Ada) or libraries are provided (Open Message Passing interface (MPI) [31]) to simplify design, development, and maintainability of the application. While such models certainly help to structure the code and enforce good practices in parallel programming, it is still up to the programmer to separate between coordinational and computational aspects of the program. While an expert in a certain application domain - a *domain expert* - is very adept in solving problems and working with models related to this domain, it is rarely his or her expertise to cope with the inherent problems of concurrency and parallelization.

[1]`http://hadoop.apache.org/`
[2]`https://hpccsystems.com/`
[3]`https://github.com/addthis/hydra`
[4]`http://www.nvidia.com/object/cuda_home_new.html`
[5]`http://www.openmp.org/`
[6]`https://www.threadingbuildingblocks.org/`

An interesting approach to solve this problem is to separate the different concerns of an application, as proposed by Gelernter and Carriero in [32]. The main idea is to separate the concerns of computation and coordination by using a coordination language in addition to a programming language. This allows the domain expert to choose a language that suits his needs to program computational blocks which are then linked together, potentially by another person, using a coordination language based on a model that is suitable to cope with concurrency. This clear separation of concerns is an intriguing concept, especially for interdisciplinary application fields where experts from different domains work together. As CPSs tend to describe applications in interdisciplinary fields, a clear separation of concerns is desirable.

In this dissertation I introduce a new coordination language, called Streamix, that is based on the PNSC model, also introduced in this thesis. While Streamix is an instantiation of the PNSC model, the language provides more than just concrete syntax for the model: Streamix allows to describe a network of processes in a structured and hierarchical manner due to its usage of network composition operators. This is inspired by the coordination language S-Net [10] which allows to describe networks of pure components. The novelty of Streamix is that while it is based on the stream processing paradigm it retains the capability of describing CPSs where components are hard to decompose due to the reactive nature of CPS. Streamix is an exogenous coordination language where the coordinated components are unaware of the coordination exert on them [33].

This dissertation describes a holistic approach, reaching from the underlying theory of the coordination model to an instantiation of the model in the form of a language and a prototype toolchain that allows to compile Streamix code into an executable C program, check the program for permanent blocking, and link it with C implementations of PNSC processes. The resulting application can be executed on a platform with support for ISO C, POSIX threads, and a Linux operating system.

## 1.1 Thesis and Research Questions

In this section I present the thesis this dissertation aims to maintain and several research questions that guided me through my research. As described in the beginning of this chapter, my work aims at using models from different fields, namely stream processing and interface theory, and adapt the models in such a way that they are applicable for CPSs. Consequently, I formulate my thesis as follows:

*It is possible to bridge the gap between stream processing and Labelled Transition Systems (LTSs) for complex components.*

In the following I will describe several research questions that focus on sub-aspects of the thesis and help to dissect each aspect independently. I first ask a general question about coordination models to then refine it with three more precise questions:

**How to coordinate mixed-criticality CPSs?**

To answer this question, as a first step, it is crucial to identify properties of CPSs and understand how they relate to coordination aspects. A mixed-criticality CPS is an assembly of networked subsystems with strong and weak coupling between them. Some of the subsystems are critical systems, i.e. of high criticality, some are best-effort systems, i.e. of low criticality. The coupling between the different subsystems is of various degrees. Critical subsystems tend to have a weak coupling with other subsystems to prevent mutual interference while best-effort systems tend to be strongly coupled. A coordination model for a mixed-criticality CPS must be able to model these various degrees of coupling between subsystems while providing guarantees of correctness of the overall behaviour of the system.

To refine the research question I ask two follow-up questions centring around the coupling of subsystems. A third follow-up question focuses on coordination languages and the implication of reactive components on a language.

**What are suitable interfaces for reactive components with strong coupling to ensure correct behaviour of a system of such components assembled in a network?**

A main property of a component that supports reactive behaviour is that it must be able to cope with infinite streams, i.e. potentially run infinitely. Due to this property, a component must support to read an input as a reaction to writing an output. This is because of a potential coupling between the output and the input through the environment. This must be reflected in the interface describing the component. In this dissertation I introduce the automata-based model SIA that allows to describe the interaction of a component with its environment. SIAs are based on a strict blocking semantics, modelling synchronous communication, which allows to describe streaming applications and their inherent strong coupling between components. The SIA model allows the composition of simple components into more complex ones while preserving the blocking semantics. I further introduce an analytic method, based on the SIA model, to detect situations where components are blocking indefinitely, e.g. deadlock situations.

**What are suitable interfaces for CPSs to integrate subsystems with different criticality levels?**

In contrast to a strong coupling, addressed in the previous question, a weak coupling allows to ensure the system correctness based on the component correctness. The focus of this question lies on the interaction of subsystems with different degrees of coupling. A main challenge of a mixed-criticality system is to allow multiple subsystems with different criticality levels to co-exist on the same platform or even to interact with each other. The challenge of such a configuration is to assure that the subsystems with lower criticality levels do not interfere with subsystems of higher criticality levels. In this dissertation I introduce Cross-criticality Interfaces (CCIs) that allow to selectively implement a weak coupling between two components and prevent such an interference.

**What are the implications of reactive components on an exogenous coordination language?**

The interaction of subsystems in a CPS is often based on reactive data processing where the environment imposes a link between the inputs and outputs of the component. A reactive component is hard to decompose because reactive components tend to rely on persistent state and internal synchronisation points. Due to this, simple and intuitive language primitives are required to describe reactive communication patterns in a structured way, such as chained components with mutual bi-directional interaction, without loosing in terms of locality. I adopt the notion of network operators to describe a network of reactive components. Different types of operators allow to describe a network in a structured and hierarchical manner by keeping information local.

## 1.2 Contributions

To illustrate the contribution of this dissertation, Figure 1.1 provides a simplistic overview of related work and the gap between research fields that this dissertation aims to bridge. In the figure, three circles represent three aspects that tend to be part of a CPS. Each circle is associated with a model that allows to describe particular properties of each aspect and tackle the challenge they pose for modelling CPSs.

- CPSs tend to be *concurrent* due to the fact that they interact with the physical world which is inherently concurrent [6]. In this work, concurrency aspects are tackled with a stream processing model which provides inherent synchronisation between interacting components.

- CPSs tend to be *heterogeneous* with respect to multiple aspects such as mixed-criticality [2], different timing semantics [34], or different underlying theoretical models [3]. In this dissertation I focus on mixed-criticality aspects and different timing semantics and use well defined communication interfaces, called CCIs, to control the interaction between components.
- CPSs tend to be *reactive* systems with complex interacting components which are hard to decompose [13]. In order to support complex components in a stream processing model I introduce the novel SIA model, an analysable component abstraction.

Further, the figure shows a few examples of related work, prominent representatives of their domain, that are placed in the circles or the intersection of two circles to illustrate which aspects of CPSs are covered by their respective model. The symbolic dial, surrounding the three circles, represents the time-criticality aspect of CPSs. Of the represented models only the names that are not greyed-out support some sort of control over timing behaviour. The figure illustrates a clear gap between reactive systems with support for complex components and stream processing models. This gap is filled by the work presented in this dissertation, namely the PNSC model and the extension of the model.

The work described in this dissertation provides the following novelties and contributions:

- introduction of a novel exogenous, component-based coordination model, called PNSC, that allows to describe networks of processes, communicating through sporadic message passing. The novelty of the model is an LTS, called SIA, first introduced in this dissertation, that allows to capture the blocking semantics of interacting processes. The model allows to describe processes with persistent state and internal synchronisation points. These are properties that fit well with the requirements of CPSs which tend to be networks of reactive components. A composition operator allows to compose processes while preserving the properties of blocking communication.
- extension of the novel PNSC model. The extensions have no influence on the behavioural aspect of a process which preserves the exogenous coordination property of the PNSC model. The extensions provide support for

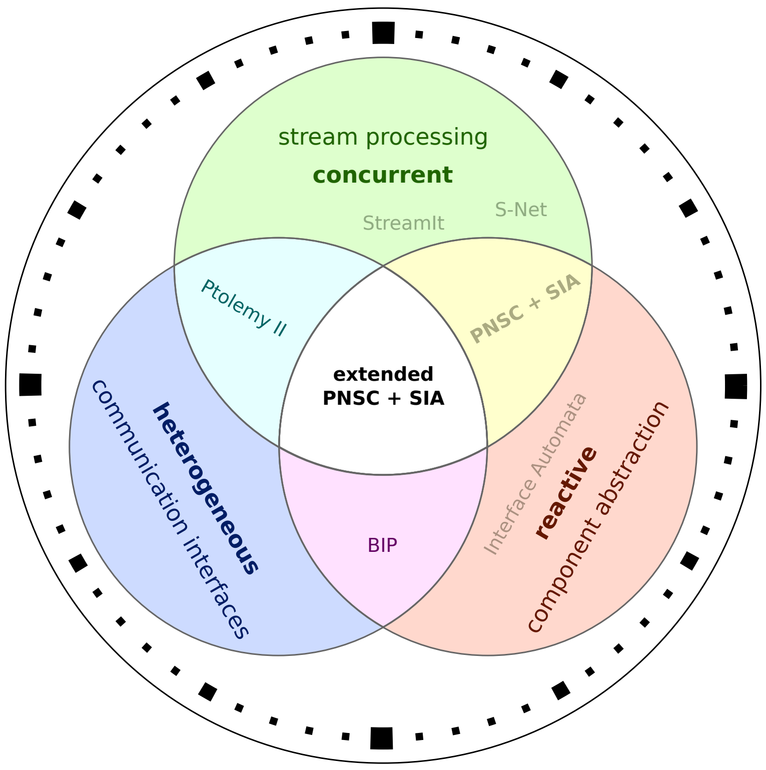


FIGURE 1.1: A simple schematic representation of properties of CPSs and how the novel PNSC model bridges the gap between stream processing and LTS for complex componets.

  - mixed-criticality systems through selective communication decoupling to prevent unwanted interference between processes. The extension integrates seamlessly with SIAs which are used to describe the communication decoupling mechanism.
  - multiple process execution schemes in a process network. This is achieved by using the communication decoupling mechanism in conjunction with clock signals to enforce a time-triggered process execution on a subset of processes. The model allows to enforce time-triggered communication on individual processes or on a network of processes.
  - rate-controlled event-triggered communication. This is also achieved through selective communication decoupling. Two types of protocols are proposed, each applicable in a buffered and non-buffered situation, to enforce a limit on the communication rate.

- proposition of the novel message type *semi-state* which complements the well known message semantics of state messages and event messages in the context of stream processing.

- development of a novel static analysis, based on the interface theory with SIA, that allows to detect permanent blocking situations in the network. The analysis distinguishes between deadlock and lonely blocking situations.
- introduction of the novel exogenous coordination language Streamix. Streamix is an instantiation of the PNSC model. It allows to describe a network of complex reactive processes in a structured and hierarchical manner.
- development of a toolchain including a Run-time System (RTS), compiler, and permanent blocking checker that allows to produce executable applications by describing a system network by a Streamix program and linking it to individual C implementations and SIA descriptions of PNSC processes. The resulting application can be executed on a platform with support for ISO C, POSIX threads, and a Linux operating system.

### 1.2.1 Publications

The following conference and workshop publications resulted from my research and have been published:

- Simon Maurer and Raimund Kirner. Coordination with Structured Composition for Cyber-physical Systems. In *Parallel Computing: On the Road to Exascale*, volume 27 of *Advances in Parallel Computing*, pages 615 – 624, Edinburgh, UK, September 2015. IOS Press. ISBN 978-1-61499-620-0. doi: 10.3233/978-1-61499-621-7-615
- Simon Maurer and Raimund Kirner. Cross-criticality Interfaces for Cyber-physical Systems. In *Proc. 1st IEEE Int'l Conference on Event-Based Control, Communication, and Signal Processing*, pages 1–8, Krakow, Poland, June 2015. IEEE. doi: 10.1109/EBCCSP.2015.7300670
- Raimund Kirner and Simon Maurer. On the Specification of Real-time Properties of Streaming Networks. In *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, Kärnten, Austria, October 2015

The following journal publication is ready for submission:

- Simon Maurer, Raimund Kirner, and Olga Tveretina. Static Deadlock Analysis of Process Networks with Synchronous Interface Automata. *Ready for Submission*, 2017

## 1.3 Structure of this Dissertation

The remainder of this dissertation is structured as follows:

**Chapter 2** provides background information on concepts I use throughout the dissertation and discusses terminology. The related topics are component-based design and more specifically interface theory, CPSs and their challenges, different communication models in CPSs, and coordination languages.

**Chapter 3** introduces the PNSC model that allows to describe networks of processes. The interaction of a process with its environment follows a clearly defined blocking semantics. In this chapter I further introduce the automata-based SIA model that allows to describe the interaction protocol of a process as its interface with its environment and I define a composition operator that allows to compose PNSC processes in arbitrary order.

**Chapter 4** describes an extension to the PNSC model that allows to punctually loosen the communication coupling between interacting processes. This can be used to model mixed-criticality systems, based on a sporadic communication scheme to prevent undesired interference. The chapter further describes that, in conjunction with clock signals, the decoupling elements allow to construct temporal firewalls which can be used to change the sporadic communication model of a subset of processes to a time-triggered communication model. In this chapter I further introduce rate-control mechanisms to bound communication rates of processes to a maximum limit.

**Chapter 5** introduces a permanent blocking analysis that allows to identify permanent blocking situations in a PNSC. Further, the chapter introduces the distinction between deadlock situations and lonely blocking situations.

**Chapter 6** describes the coordination language Streamix which represents an instance of the extended PNSC model. Streamix is an exogenous coordination language that allows to compose reactive components in a structured and hierarchical manner to form a network of processes.

**Chapter 7** describes a prototype of a toolchain for the coordination language Streamix. It includes the compiler for the Streamix language, the RTS preprocessor, the RTS, and the SIA model checker. Together, the tools allow to build an application that can be executed on a platform with support for ISO C, POSIX threads, and a Linux operating system.

**Chapter 8** compares the different aspects of my work with the state of the art. This includes interface theory, mixed-criticality models, coordination languages and models, and methods to detect or prevent permanent blocking situations.

**Chapter 9** finally concludes the dissertation and discusses the results and contributions of the thesis. It also includes directions for future research on this topic.

# Chapter 2

# Background

In this chapter I will discuss terminology and give some background on topics relevant for this dissertation.

The chapter is structured as follows: Section 2.1 discusses the term *real-time* and its ambiguous meaning in different research areas and relates embedded systems to Cyber-physical Systems (CPSs) and Internet of Things (IOT). Section 2.2 describes the basic idea of component-based design, the approach used as a corner stone for the Process Network with Synchronous Communication (PNSC) model proposed in this dissertation. I then describe properties of CPSs and discuss their relevance with respect to the design approach of CPSs in Section 2.4. Section 2.5 focuses on communication aspects such as the triggering semantics, i.e. time-triggered or event-triggered communication and communication coupling. Section 2.6 provides a short history on coordination languages and discusses classification aspects of coordination languages.

## 2.1 A Side-note on Terminology

Traditionally, a hard real-time system describes a system where the consequence of missing a deadline may result in a catastrophic event. Hence, the correctness criteria of a piece of software not only rely on the correctness of the result but also on the time of availability of the result. If the software provides a correct result but misses the deadline by doing so, the correctness criteria are not met. One tends to distinguish between hard real-time systems and soft real-time systems where the latter also imposes a deadline but the consequences of missing a soft deadline are less severe. By missing a soft deadline, the quality of the result only decreases but does not become useless.

There are two general points I want to make concerning the concepts of real-time systems. Firstly, I find that the distinction between best effort systems and soft real-time systems is only marginal and that a soft real-time system has much more in common with a best-effort application than with a hard real-time system. Ultimately, every system has a soft deadline because if a computation is never producing a result, the computation is hardly useful. Hence, technically, every system is a real-time system. However, there is a difference in terms of the usefulness of a result depending on when it is available. In the case of a best effort application, the expectation is that eventually a result will be available (before a non-specified deadline $< \infty$) and the sooner it is available the better. With soft real-time systems, the expectation is put into numbers, meaning that the application is expected to deliver a result within a specified deadline and there is no immediate benefit if the result is available earlier. In contrast to a hard real-time system where the focus lies on giving *guarantees* to meet deadlines, in soft real-time systems no guarantees are given that a deadline is met. Rather, methods are used to decrease the possibility for deadlines to be missed. For example, in the case of a video stream application, buffers are used to store frames that resulted before the deadline in order to compensate during times when the deadline is missed.

The second point concerns the use of the term *real-time* which has an ambiguous meaning depending on the research community. Most commonly, in research, the term real-time describes the fact that an application has to respect a deadline, i.e. a result must be made available before a specified time limit has passed. However, I found that a lot of people, not necessarily researchers, associate the term real-time with simultaneously performing a computation as a direct reaction on events happening in the world, e.g. a football live stream, the logging of events while they are happening, the capturing and visualising of human motions while the human is performing the motions, etc. All these problems are probably designed and programmed, using in one way or another the notion of a deadline but the term real-time does not reflect that. Rather it reflects liveness of an application which creates, in my opinion, unnecessary confusion when explaining such problems to non-experts. Following the argument I made above that soft real-time and best-effort applications are more closely related than hard real-time problems are related to soft real-time problems, I advocate to use the term *time-critical* when talking about hard real-time problems. It immediately carries the message that time plays a critical part in such an application and avoids the confusion with live systems. Hence, throughout this dissertation I will use the term *time-critical* system when talking about a hard real-time system.

Time-critical systems are tightly coupled to the hardware they are running on. This is because the execution time of an application is dependent on the hardware architecture and to give guarantees that an application will meet the specified deadlines, it is important to fully understand the associated hardware architecture. Due to this, in the past, target hardware platforms for time-critical systems were often specialised boards fulfilling the exact requirements to provide the resource demands of the software application executed on the platform. As components on such boards were directly soldered on, such hardware boards were called *embedded systems* and the term became synonymous for *hard real-time systems* (or time-critical systems as I call them), including software and hardware components. With the evolution of hardware architectures, computational units were and still are becoming increasingly efficient and performant. Combined with the increasing demand for smarter, time-critical applications it often became necessary to build networked systems where not one single platform was used. Also, a lot of devices of today are embedded on a single board without necessarily hosting a time-critical system. Literally, the term *embedded system* only describes the technique of how a hardware board is assembled which is no longer an indication that the executed application is actually a time-critical system. Despite the potential confusion, the term *embedded system* is still used to describe a single platform, time-critical system. However, more complex systems, including multiple networked time-critical platforms, i.e. embedded systems, are referred to as *Cyber-physical Systems (CPSs).* The focus of the term CPS is put on the interaction between the physical and the cyber world through sensors and actuators.

Another term that relates to similar concepts is *Internet of Things (IOT).* An IOT is an instance of the more general term CPS. It describes largely distributed applications where each "thing" represents a node in a network of interacting nodes.

Note that neither embedded systems, nor CPSs, nor the IOT must necessarily describe time-critical applications but as they interact with the physical world they generally do have timing constraints.

## 2.2 Component-based Design

Component-based design aims at separating concerns by dividing large systems into loosely coupled independent, concurrent components [39, 40]. This concept is a corner stone of a multitude of models, for example, in the area of stream processing [7] or coordination languages [41]. Typically, these models describe a composition of computational components that form a network. Linking structures, such as channels or shared memory locations, establishes connections between components. Such networks are usually explicitly constructed with the help of either a language (e.g. StreamIt [8] or S-Net [10] where networks are constructed by applying binary wiring operators on components) or a library (e.g. Open MPI [31] which provides an API to spawn channels linking components together). The question of compatibility between components is either inherently solved by construction of the language (S-Net only allows functional components and uses component signatures to check for compatibility) or is delegated to the programmer (Open MPI).

In component-based design, implementation details of components are often ignored and a component is represented by an abstraction. The key is to choose an abstraction that allows to describe the component accurately enough to expose certain properties while keeping the abstraction as simple as possible [39]. By composing simple components to build complexer ones, the properties exposed by the abstraction are used to check for compatibility of the components. Ideally, such a model supports heterogeneous systems and unifies the compatibility problem within one model [40]. In order for the programmer to be able to cope with concurrent systems, models have been proposed to check interoperability of components either by using an automata-based interface description of components such as Interface Automata (IAs) [15], causality interfaces [12], or transfer functions [20], to name only a few.

A specific aspect of compatibility of components is the liveness property of a system composed of components. A system is not guaranteed to be alive if a subset of the complete system or a subset of involved components can permanently block. A widely known case of permanent blocking is a deadlock situation [42].

In this dissertation I am interested in an automata-based approach to describe interfaces of components and use it to check for freedom of permanent blocking in the system. My approach is inspired by IAs but uses a different blocking semantics to make it suitable in the context of stream processing.

## 2.3 Permanent Blocking and Deadlocks

In this dissertation I use the term *liveness* to describe whether a system is void of any blocking subsystems. I use the term *permanent blocking* as the opposite of *liveness*, i.e. to describe a system that has at least one blocking subsystem:

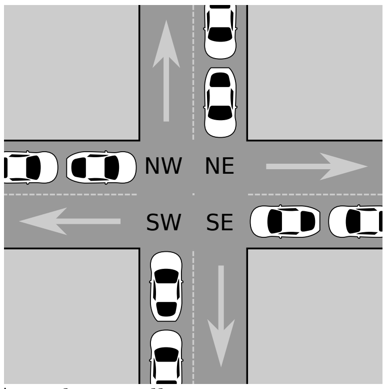
$$permanent\_blocking = \neg(liveness) \tag{2.1}$$

Permanent blocking must not necessarily be a deadlock. Coffman et al. identified four conditions that must hold simultaneously for a deadlock to occur [42]. These conditions are listed in Definition 2.1.

**Definition 2.1** (Four Deadlock Conditions)**.** *A system is in a deadlock situation if the following four conditions hold simultaneously:*

1. Mutual exclusion: *a task has exclusive control over a resource.*
2. No pre-emption: *a resource can only be released voluntarily by the task holding it.*
3. Hold and wait: *a task is holding at least one resource and is requesting at least another.*
4. Circular wait: *a task $T_1$ is holding a resource $x$ and requests a resource $y$ while a task $T_2$ is holding resource $y$ and requests resource $x$. A circular wait is not necessarily limited to only two participants and can span over multiple parties.*

A practical example of a deadlock, or more specifically a gridlock, is illustrated in Figure 2.1. It depicts an intersection of two roads where cars are assumed to only drive straight ahead without turning. For a car, e.g. arriving from the West, to be able to cross the intersection, two spaces, e.g. *NW* and *NE*, have to be allocated. If traffic control allows to allocate spaces separately for cars arriving from each direction, the situation in Figure 2.1b can occur, where all spaces are occupied by an individual car such that no progression is possible for any car.



(A) A car has to allocate two spaces in order to advance.

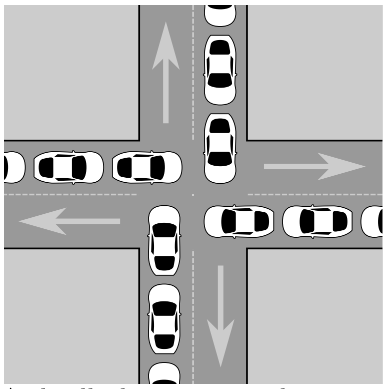

(B) A deadlock situation where no car is able to advance.

Figure 2.1: An example of gridlock on a crossing (no turning)

It is possible that in a system of multiple interacting processes only one process is permanently blocked while the rest of the system is processing without permanently blocking. If a process is blocking alone and no circular wait, as defined in Definition 2.1.4, is involved I call this process a *lonely blocker*. Hence, I distinguish between *deadlock* where multiple components block each other due to a circular wait and *lonely blocking* where one component is blocked by other components but is itself not causing other processes to block.

Therefore I conclude that

$$permanent\_blocking \neq deadlock$$

and consequently with Equation 2.1 I conclude

$$deadlock \neq \neg(liveness)$$

However, deadlock implies permanent blocking, hence

$$deadlock \rightarrow permanent\_blocking$$

but permanent blocking does not necessarily imply deadlock:

$$\neg(permanent\_blocking \rightarrow deadlock)$$

Note that a permanent blocking process is either a lonely blocker or involved in a deadlock but not both.

Using again the example of a road intersection I illustrate a permanently blocking system that is not in a deadlock situation but rather a lonely blocking situation in Figure 2.2. While cars from the East are able to progress to the West and vice versa, cars in the

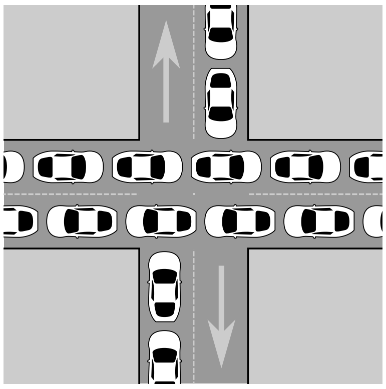


FIGURE 2.2: A lonely blocking situation where progress is only possible from the East to the West and vice versa.

North and in the South are blocked and cannot progress. Assuming an infinite stream of cars and no priority rules, the cars from the North and the South will be blocked infinitely. Note that this blocking is not due to a circular wait condition, as defined in Definition 2.1.4, because the cars moving from East to West and vice versa are not blocked. Consequently, given that the system is in a permanent blocking situation without any deadlocks, the system is lonely blocking.

## 2.4 Cyber-physical Systems

CPSs are systems that interact with the physical world through sensors and actuators. Because the actuation is usually a reaction to the sensing of the environment, CPSs are generally *reactive* and must often satisfy timing requirements.

Harel and Pnueli introduced the terms *transformational* and *reactive* to distinguish systems that are "relatively easy to develop from those that are not" [13]. In contrast to a transformational system that takes inputs, performs computation, and produces outputs, CPSs are often reactive systems where inputs are coupled to outputs via the environment. The output *out* of a transformational system, described by the function $f$, is dependent on the state *state* of the system and the input *in* to the system:

$$out = f(state, in)$$

Reactive systems, on the other hand, additionally, have a relation where the input *in* of the system is not a variable but defined by a function $f_{env}$ which is dependent on the output of the system and an unknown variable *env*, imposed by the environment:

$$out = f(state, in) \ \wedge \ in = f_{env}(env, out)$$

Furthermore, a reactive system constantly reacts on changes in the environment while a transformational system is active on its own behalf. These properties make it harder to decompose a reactive system in subcomponents which is less the case for transformational systems [13]. Kopetz distinguishes between *simple* tasks (S-tasks) and *complex* tasks (C-tasks) [43]. A C-task contains blocking synchronisation points (e.g. the task is blocking because it has to wait for the environment to provide it with an event) while an S-task does not. A reactive system component tends to fit better with the C-task model and it can be hard to avoid this. Consequently, reactive system components are often required to maintain a persistent state and to support bidirectional communication between components to allow an intuitive description of the reactive system. While allowing persistent state in a system component makes it harder to exploit concurrency on parallel architectures, it has the benefit of making a model more accessible for legacy code.

The design and development of a CPS often involves a team of interdisciplinary experts to assemble the required knowledge to correctly model the interaction of the physical world with the cyber world. As a consequence, different models from different application domains need to be brought together. To cope with this Eker et al. [3] proposed a unifying framework that allows to choose suitable models for each subsystem and assemble them through the framework. More recent work proposes to focus on single meta-models with support for heterogeneity to increase the analysability of the model [1, 4, 5]. The approach discussed in this dissertation aims towards this goal and is based on a component-based design approach (see Section 2.2).

To satisfy timing requirements of a CPS, the software components must be executed on a hardware platform that is able to provide the required resources. For time-critical systems, guarantees must be provided that timing requirements are met, i.e. such systems are designed for the worst case. Guarantees are given by computing the Worst-case Execution Time (WCET) of the software components, executed on a target platform. Generally there are three different approaches to compute the WCET: By the conjunction of code analysis and a precise hardware model, by simulation, or by measurement through extensive testing. The most commonly used technique in industry is the approach where extensive testing is performed in order to verify that requirements are satisfied [6]. The WCET problem is nothing I will address in this dissertation but the interested reader might want to refer to the survey of Wilhelm et al. [44] for an overview of methods and available tools.

Another aspect of CPSs is *mixed-criticality.* With the increasing capability of hardware architectures, the need arises to execute multiple applications on the same hardware platform. Often, there is a difference in terms of criticality of the applications intended for the same platform (e.g. in a car, the multimedia system is of lower criticality than the parking assistance system). In order to prevent interference from the low critical system to the high critical system, Vestal proposed a mixed-criticality scheduler [45]. Burns and Davis provide an extensive review paper which they constantly update with new findings on the topic [2]. The focus of the research is almost exclusively on providing a scheduler, capable of scheduling a mixed-criticality application on single or multi-core architectures. I am, however, interested in Cross-criticality Interface (CCI), i.e. an abstract model that allows components of different criticality levels to interact without interfering.

## 2.5 Communication in Cyber-physical Systems

Human interaction with the physical world tends to be event-driven [46]: We react to sporadic changes in our environment and perform actions to adapt to the changes (turn the head upon registering a movement, keeping the balance on a ship). The corresponding communication model for CPSs is the *sporadic* or *event-triggered* communication model. In such a communication model, the transmission of information is triggered by the occurrence of events. In a producer component, i.e. a sensor, an event trigger occurs when a significant change of the state occurs which is then communicated to a consumer component. The occurrence of events is sporadic, hence the time instant of the next event occurrence is unknown. Due to this, it is hard to predict the required communication bandwidth and maximum load assumptions are needed in order to fulfil temporal constraints. Local changes, like adding new components or changing the behaviour of an existing component, can invalidate the temporal behaviour of the system. An event based communication scheme is useful for open systems and for systems with sporadic data. Typical examples of deterministic models that are based on event-triggered communication are Kahn Process Networks (KPNs) [23] or the more restrictive Synchronous Data Flow (SDF) model [11].

However, for critical applications where predictability and fault-tolerance are key, Kopetz et al. proposed the Time-triggered Architecture (TTA) [47]. In a *time-triggered* communication scheme the participants use a common time basis and communicate at defined time instants. The communication instants are often periodic due to simplicity but this is not a necessity. Within a time-triggered communication system, local changes cannot invalidate the temporal behaviour of the system and the system load is independent of the number of message transmissions. This is achieved through communication decoupling with temporal firewalls [27]. Due to the stability and predictability with respect to load and time behaviour, a time-triggered communication scheme is useful for fault tolerant systems.

Loosely coupled communication has also been studied in the domain of large distributed systems and three dimensions of coupling have been identified: time, space, and synchronisation ([48, 49]). The three dimensions are orthogonal to each other. In the following, a short description of each dimension is provided.

### 2.5.1 Communication Coupling in Time

An interaction is time decoupled if the participating components are not required to operate at the same time in order to transmit messages. In contrast, a time coupled interaction requires participants to operate at the same time for being able to communicate. To achieve decoupling in time, a storage is necessary where messages can be placed by the sender until they are retrieved by the receiver. Time decoupling allows the receiver to be disconnected at the time of the transmission by the sender and it allows the sender to be disconnected at the time of reception by the receiver. The notion of time decoupling is binary. Communication is either coupled or decoupled in time, there is no middle ground.

### 2.5.2 Communication Coupling in Space

Communication is space decoupled if the involved parties do not know each other's address. Contrary, in a space coupled interaction the sender uses a direct address of the receiver to transmit a message. Space decoupling can be achieved by introducing an intermediate medium of communication. This may be a shared storage, a channel for point-to-point communication, or a complex broker infrastructure taking care of the communication process. Decoupling communication in space allows replacement and maintenance of components at runtime and is a prerequisite for open systems.

Decoupling in space is based on the notion of binding an address to a component. If said address is known by the other communicating party the communication is coupled in space, otherwise, it is decoupled in space. However, let's consider the example of a network component, addressed by its IP-address. If this component is replaced with a new component using the exact same configuration, including the IP-address, the communication would work without any need for updating addresses in the participating communication parties. Although, the recipient address must be known by the sender, this communication is space decoupled. That is because the IP address is not an identification of the router (a MAC address would be tighter coupled but is still configurable) but a binding of an alias to a service. It is not the component that the other communication parties are interested in, it is the service provided by that component (or any other component able to provide it). Space decoupling is about the stability of an alias that is always providing an identical service, where the notion of identity means identical state and behaviour.

### 2.5.3 Communication Coupling in Synchronisation

Decoupling in synchronisation depends upon the read and write operation of the sender and receiver respectively. Blocking operations imply a synchronous communication while non-blocking operations allow asynchronous transmission of messages. To have a fully asynchronous interaction, both, writing and reading must be non-blocking. Because coupling in synchronisation is an independent attribute of read and write operations, communication can be fully decoupled, partly decoupled or coupled in synchronisation.

In this dissertation the synchronisation decoupling is linked to forward and backward progress control. Forward progress control describes the triggering semantics of the receiving component by a blocking read and backward progress control describes the back-pressure exert on the sending component by a blocking write.

## 2.6 Coordination Languages

The increasing system complexity of CPSs drives a high pressure on software engineering methodologies to assure an effective and predictable control of resources. Coordination languages have been proposed as a solution to tackle this problem by decomposing application software into coordination and algorithmic programming [6]. The first coordination language, called Linda, was introduced in 1992 by Gelrenter et al. [32]. The concept of Linda is based on a shared memory space, called tuple space, where sets of data elements, called tuples, are placed and retrieved by processes with the help of specific primitives. Over the years, several coordination languages, based on the tuple space paradigm, have been proposed [50, 51]. In their survey on coordination languages [41], Papadopoulos and Arbab distinguish between data-driven and control-driven coordination languages where Linda and its derivates fall in the former class. Data-driven coordination languages tend to provide some primitives that are used in the purely computational part to coordinate the exchange of information between processes. This concept allows to build hierarchies and to broadcast information, two properties that are well suited for structured programming. There is, however, not a strict separation between coordination and computation imposed by the model but the task of separating these concerns is left to the programmer.

Control-driven coordination languages, on the other hand, tend to enforce a clear separation of concerns because coordination elements are not part of the computational components. This is usually achieved by linking computational components by channels or more complex coordination constructs [52, 53]. While data-driven models tend to coordinate data, control-driven models coordinate entities. This model-based approach fits well with the world of CPSs where reactive components are common [1, 4, 5]. A draw-back of the control-driven approach is that models often rely on point-to-point connections to describe the communication channels connecting computational components which lacks a clear structure. However, Grelck et al. show with the coordination language S-Net that by using network operators, structured programming can also be achieved with a control-driven coordination model [10].

Later, Arbab extended the classification of coordination languages and he introduced the terms *endogenous* and *exogenous* coordination, describing whether coordination is done from within a behavioural component or from the outside, respectively [33]. An exogenous coordination model assures a clear separation between coordination and behaviour by leaving the behavioural components oblivious of the coordination constructs that exert coordination upon them. An endogenous coordination model, on the other hand, has no such separation and the coordination is done from within a behavioural component which makes the component aware of the coordination exert on it. A clear separation of concerns is important to simplify development, integration, and verification of applications. In the domain of CPSs where systems are often safety critical, this is a property we need to enforce.

Lee suggested that the use of coordination models not only allows to separate behaviour and coordination but also provides an abstraction of the real-time behaviour from the underlying hardware platform [6]. An example of such an approach is the Ptides [54] model (which is part of the Ptolemy project [55]) that allows to model timing behaviour of an event-triggered communication model. Another example is Giotto which is a coordination language based on the time-triggered communication model [56]. With the concept of Logical Execution Time (LET), Giotto provides a well defined input/output timing behaviour of interacting components that allow to loosen the coupling between the modelled timing specification and the timing behaviour of the application executed on a specific hardware platform.

# Chapter 3

# PNSC with SIA - An Analysable Event-based Component Model

In this chapter I present a model that allows to describe the interaction of concurrent components. Each component is modelled as a, possibly stateful, process that is interacting with its environment by reacting to input messages and producing output messages. A process follows the sporadic communication semantics, as described in Subsection 2.5.

The chapter is structured as follows: Section 3.1 defines the model, called Process Network with Synchronous Communication (PNSC), where concurrent processes build a network by interacting over synchronous communication channels. The synchronous communication model imposes strict blocking rules on the communication which allows to gain a deeper understanding of the interaction. In Section 3.2 I introduce Synchronous Interface Automata (SIAs), a model to describe the interaction protocol of processes. It serves to describe how a process interacts with its environment and whether potential problems, such as permanent blocking situations, can occur. In Section 3.3 I describe how the PNSC model can be used to model an example of a traffic situation on an intersection. Further, I extend the example by modelling an assembly of intersections as streaming network with buffered communication. Finally, in Section 3.4 I summarise the chapter.

## 3.1 Process Networks with Synchronous Communication (PNSC)

The communication model of process networks described in this chapter is based on synchronous communication. For further reference I use the name *Process Network with*

*Synchronous Communication (PNSC).* A PNSC consists of a set of processes $PN$. A process in a PNSC interacts with other processes via input and output ports.

**Definition 3.1** (PNSC Process). *Formally, a process $N$ is defined as a tuple*

$$N = \langle \mathcal{P}_N^I, \mathcal{P}_N^O \rangle$$

*where*

- $\mathcal{P}_N^I$ *is the finite set of input ports of process $N$.*
- $\mathcal{P}_N^O$ *is the finite set of output ports of process $N$.*
- $\mathcal{P}_N$ *is the* signature *of process $N$ and is defined as: $\mathcal{P}_N = \mathcal{P}_N^I \cup \mathcal{P}_N^O$. The ports of these two port sets have to be mutually distinctive: $\mathcal{P}_N^I \cap \mathcal{P}_N^O = \emptyset$.*

A process is of type *Multiple Input, Multiple Output (MIMO)*, hence it holds that

$$|\mathcal{P}_N^I| \geq 0 \wedge |\mathcal{P}_N^O| \geq 0$$

Note, that a process can have a persistent state and thus, is not necessarily functional.

Two processes $M$ and $N$ are connected through synchronous channels if they have *shared* ports. One end of the channel connects to an input port of one process and the other end connects to an output port with the same name of the other process. The shared ports of processes $M$ and $N$ are defined in Equation 3.1.

$$shared_{\mathcal{P}}(M, N) = (\mathcal{P}_M^I \cap \mathcal{P}_N^O) \cup (\mathcal{P}_M^O \cap \mathcal{P}_N^I) \tag{3.1}$$

All ports in a set of $n$ processes $PN = \{N_0, \ldots, N_n\}$ describing a PNSC must be non-conflicting:

$$\bigcap_{i=0}^{n} \mathcal{P}_{N_i}^I = \emptyset \tag{3.2}$$

$$\bigcap_{i=0}^{n} \mathcal{P}_{N_i}^O = \emptyset \tag{3.3}$$

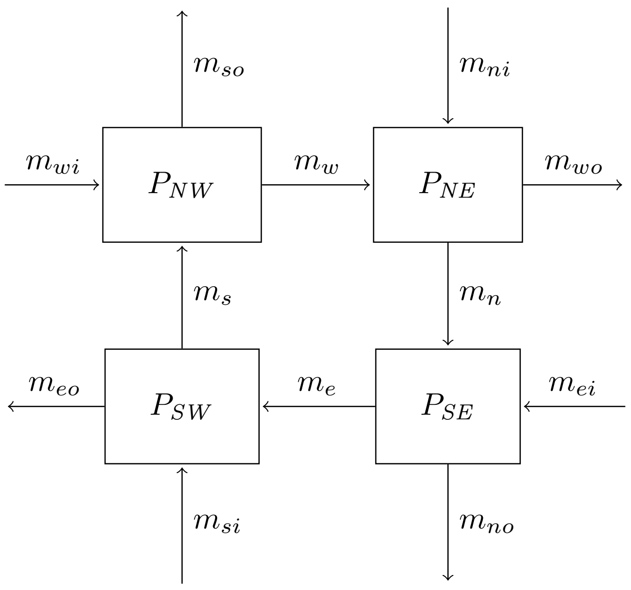
I distinguish between two types of processes, an *atomic process* and a *composed process.* An atomic process is a black box where the implementation of the process is unknown to the model. A composed process is an abstraction of a PNSC in the sense that the processes of the PNSC are composed into a single process. The signature of a composed process is formed out of non-connected ports of each process in the set of the PNSC abstracted by the composed process as defined by Definition 3.2. This composition can be applied incrementally on a set of processes in any order if Equation 3.2 and Equation 3.3 are satisfied.

**Definition 3.2** (Composed PNSC Process)**.** *Let $M, N$ be the set of processes of a PNSC. Formally, the signature $\mathcal{P}_{MN}$ of the composed process $MN$ is defined as follows:*

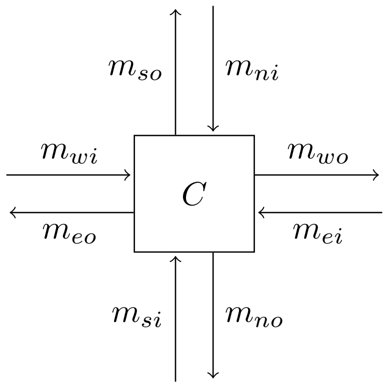
$$\mathcal{P}^I_{MN} = (\mathcal{P}^I_M \cup \mathcal{P}^I_N) \setminus shared_{\mathcal{P}}(M, N)$$

$$\mathcal{P}^O_{MN} = (\mathcal{P}^O_M \cup \mathcal{P}^O_N) \setminus shared_{\mathcal{P}}(M, N)$$

Figure 3.1 depicts the process model of the crossroad example introduced in Section 2.2. The four processes $P_{NW}$, $P_{NE}$, $P_{SE}$, and $P_{SW}$ in Figure 3.1a represent the guards for the mutually exclusive allocation of the four critical sections $NW$, $NE$, $SE$, and $SW$. Processing the messages $m_{wi}$, $m_{ni}$, $m_{ei}$, and $m_{si}$ represent the advancing of a car from the corresponding direction onto the respective spaces $NW$, $NE$, $SE$, and $SW$. Similarly, processing the messages $m_w$, $m_n$, $m_e$, and $m_s$ represent the advancing of a respective car by one space: $NW \rightarrow NE$, $NE \rightarrow SW$, $SE \rightarrow SW$, and $SW \rightarrow NW$. Producing the messages $m_{wo}$, $m_{no}$, $m_{eo}$, and $m_{so}$ represents freeing the respective spaces $(NW, NE)$, $(NE, SE)$, $(SE, SW)$, and $(SW, NW)$.



(A) The PNSC network of atomic processes.



(B) The PNSC network as composed process where pairs of atomic processes are composed incrementally.

Figure 3.1: The PNSC network modelling the crossroad example depicted in Figure 2.1.

Due to the synchronous communication semantics of PNSCs, in order to communicate a message from one process to another, both, the sender and receiver process must be ready at the same time for the transaction. As processes in PNSCs can be stateful, in order for a process to be able to send a message, the receiver process must be in a state where it is ready to receive the message. Otherwise, the sending process is temporarily blocking in its current send state. Similarly, for a process to be able to receive a message, the sending process must be in a state where it is able to send the message. Otherwise, the receiving process is temporarily blocking in its current receive state.

If a process resides in a blocking state indefinitely, the PNSC (or a subset of the processes in the PNSC) is in a permanent blocking state. The question that arises is how to detect such permanent blocking states and how to decide if a PNSC is free of any possibility to enter a permanent blocking state. To answer this question I first need to describe the abstract behaviour of processes in order to understand how they interact with each other. To do this I introduce Synchronous Interface Automata (SIAs) in the next section.

## 3.2 Synchronous Interface Automata (SIA)

A *Synchronous Interface Automaton (SIA)* $\widetilde{N}$ is a finite state automaton that describes the interaction protocol of a process $N$ with its environment.

### 3.2.1 Definition of SIAs

The alphabet of a SIA is a set of *actions* where each action describes the label of a transition from one protocol state to another. The states of a SIA $\widetilde{N}$ do not necessarily represent the internal states of a process $N$ but only the states of the interaction protocol of process $N$.

**Definition 3.3** (SIA)**.** *Formally, a Synchronous Interface Automaton (SIA) $\widetilde{N}$ of a process $N$ (defined in Definition 3.1) is defined as a tuple*

$$\widetilde{N} = \langle S_{\widetilde{N}}, s_{\widetilde{N}}, \mathcal{A}^I_{\widetilde{N}}, \mathcal{A}^O_{\widetilde{N}}, \mathcal{A}^H_{\widetilde{N}}, \delta_{\widetilde{N}} \rangle$$

*where*

- $S_{\widetilde{N}}$ *is the finite set of interface states of SIA* $\widetilde{N}$*.*
- $s_{\widetilde{N}} \in S_{\widetilde{N}}$ *is the unique initial interface state of SIA* $\widetilde{N}$*.*
- $\mathcal{A}^I_{\widetilde{N}} \subseteq \mathcal{P}^I_N$ *is the finite set of input actions of SIA* $\widetilde{N}$*.*
- $\mathcal{A}^O_{\widetilde{N}} \subseteq \mathcal{P}^O_N$ *is the finite set of output actions of SIA* $\widetilde{N}$*.*

- $\mathcal{A}^H_{\widetilde{N}}$ *is the finite set of internal (i.e., hidden) actions of SIA* $\widetilde{N}$.
- $\mathcal{A}_{\widetilde{N}}$ *is the total finite set of actions.* $\mathcal{A}_{\widetilde{N}}$ *is defined as:* $\mathcal{A}_{\widetilde{N}} = \mathcal{A}^I_{\widetilde{N}} \cup \mathcal{A}^O_{\widetilde{N}} \cup \mathcal{A}^H_{\widetilde{N}}$. *The actions of these three action sets have to be mutually distinctive:* $\mathcal{A}^I_{\widetilde{N}} \cap \mathcal{A}^O_{\widetilde{N}} \cap \mathcal{A}^H_{\widetilde{N}} = \emptyset$.
- $\delta_{\widetilde{N}}$ *is the finite set of interface state transitions of SIA* $\widetilde{N}$: $\delta_{\widetilde{N}} \subseteq S_{\widetilde{N}} \times \mathcal{A}_{\widetilde{N}} \times S_{\widetilde{N}}$. *Note that an action* $a \in \mathcal{A}_{\widetilde{N}}$ *is not limited to a single state transition but can be part of multiple state transitions in SIA* $\widetilde{N}$.

The actions in $\mathcal{A}^I$ and $\mathcal{A}^O$ are *blocking* and *non-autonomous*, while actions in $\mathcal{A}^H$ are *non-blocking* and *autonomous*. The progress of *non-autonomous* actions depends on the environment, while the progress of *autonomous* actions is independent of the environment. Hence, autonomous actions are controlled by the process while non-autonomous actions are controlled by the environment. Blocking actions will wait until the environment provides the corresponding action, while non-blocking actions can be processed immediately.

Note that SIAs are input and output deterministic (hidden actions do not require determinism), which is formally defined as:

$$\forall \langle s_i, a, s_j \rangle \in \delta \ . \ \langle s_i, a, s_k \rangle \in \delta \ . \ a \in (\mathcal{A}^I \cup \mathcal{A}^O) \implies s_j = s_k$$

Above condition means that from one interface state any action $a \in (\mathcal{A}^I \cup \mathcal{A}^O)$ can be part of at most one outgoing transition. An action $a \in \mathcal{A}$ is called *enabled* in a state $s \in S$ if and only if the condition defined in Equation 3.4 is satisfied.

$$\exists s' \in S \ . \ \langle s, a, s' \rangle \in \delta \tag{3.4}$$

It might be desirable for a protocol to reach a certain state in which it will reside indefinitely. Hence, I define a set of *end states* $S^{end}$ of a SIA in Equation 3.5.

$$S^{end} = \Big\{ s \in S \mid \forall s' \in S, \forall a \in \mathcal{A} \ . \ \langle s, a, s' \rangle \notin \delta \Big\} \tag{3.5}$$

Informally, an end state is a state where no further transition, triggered by any action, is possible. Note that this holds for any action, i.e. no distinction is made between different action types.

As two processes can share ports, their corresponding SIAs can share actions. In fact, for two processes $M$ and $N$ to interact, their corresponding SIAs $\widetilde{M}$ and $\widetilde{N}$ must have *shared* actions. Shared actions are defined in Equation 3.6 which is similar to the definition of shared ports of Equation 3.1.

$$shared_{A}(\widetilde{M}, \widetilde{N}) = (\mathcal{A}^I_{\widetilde{M}} \cap \mathcal{A}^O_{\widetilde{N}}) \cup (\mathcal{A}^O_{\widetilde{M}} \cap \mathcal{A}^I_{\widetilde{N}}) \tag{3.6}$$

From Equation 3.1, Equation 3.11, and Equation 3.12 it follows that $shared_{\mathcal{A}}(\widetilde{M}, \widetilde{N}) \subseteq shared_{\mathcal{P}}(M, N)$, which means that not all ports of the process interface have to be used by its SIA. The set of actions that is excluded from the set of shared actions is called *ignored* actions and is defined in Equation 3.7.

$$ignored_{\mathcal{A}}(\widetilde{M}, \widetilde{N}) = shared_{\mathcal{P}}(M, N) \setminus shared_{\mathcal{A}}(\widetilde{M}, \widetilde{N}) \tag{3.7}$$

All input and output actions of the SIAs $\widetilde{M}$ and $\widetilde{N}$ that do not correspond to the set of shared ports of the processes $M$ and $N$ are called *open* actions. Open actions are defined in Equation 3.8.

$$open_{\mathcal{A}}(\widetilde{M}, \widetilde{N}) = (\mathcal{A}^I_{\widetilde{M}} \cup \mathcal{A}^I_{\widetilde{N}} \cup \mathcal{A}^O_{\widetilde{M}} \cup \mathcal{A}^O_{\widetilde{N}}) \setminus shared_{\mathcal{P}}(M, N) \tag{3.8}$$

### 3.2.2 Composition of SIAs

Two SIAs $\widetilde{M}$ and $\widetilde{N}$ can be composed into a combined SIA $\widetilde{MN}$ if their actions are non-conflicting as defined in Equation 3.9.

$$\begin{aligned}
\mathcal{A}^I_{\widetilde{M}} \cap \mathcal{A}^I_{\widetilde{N}} &= \emptyset \\
\mathcal{A}^O_{\widetilde{M}} \cap \mathcal{A}^O_{\widetilde{N}} &= \emptyset \\
\mathcal{A}_{\widetilde{M}} \cap \mathcal{A}^H_{\widetilde{N}} &= \emptyset \\
\mathcal{A}^H_{\widetilde{M}} \cap \mathcal{A}_{\widetilde{N}} &= \emptyset
\end{aligned} \tag{3.9}$$

Note that actions can be renamed to solve such conflicts.

**Definition 3.4** (SIA composition operator $\otimes$)**.** *Formally, the composition of two SIA $\widetilde{M}$ and $\widetilde{N}$ into a SIA $\widetilde{MN} = \widetilde{M} \otimes \widetilde{N}$ is defined as a tuple*

$$\widetilde{MN} = \widetilde{M} \otimes \widetilde{N} = \langle S_{\widetilde{MN}}, s_{\widetilde{MN}}, \mathcal{A}^I_{\widetilde{MN}}, \mathcal{A}^O_{\widetilde{MN}}, \mathcal{A}^H_{\widetilde{MN}}, \delta_{\widetilde{MN}} \rangle$$

*where*

$$\begin{aligned}
S_{\widetilde{MN}} &= S_{\widetilde{M}} \times S_{\widetilde{N}} \\
s_{\widetilde{MN}} &= \langle s^0_{\widetilde{M}}, s^0_{\widetilde{N}} \rangle \\
\mathcal{A}^I_{\widetilde{MN}} &= (\mathcal{A}^I_{\widetilde{M}} \cup \mathcal{A}^I_{\widetilde{N}}) \setminus (shared_{\mathcal{A}}(\widetilde{M}, \widetilde{N}) \cup ignored_{\mathcal{A}}(\widetilde{M}, \widetilde{N})) \\
\mathcal{A}^O_{\widetilde{MN}} &= (\mathcal{A}^O_{\widetilde{M}} \cup \mathcal{A}^O_{\widetilde{N}}) \setminus (shared_{\mathcal{A}}(\widetilde{M}, \widetilde{N}) \cup ignored_{\mathcal{A}}(\widetilde{M}, \widetilde{N})) \\
\mathcal{A}^H_{\widetilde{MN}} &= \mathcal{A}^H_{\widetilde{M}} \cup \mathcal{A}^H_{\widetilde{N}} \cup shared_{\mathcal{A}}(\widetilde{M}, \widetilde{N})
\end{aligned}$$

*with $shared_{\mathcal{A}}(\widetilde{M}, \widetilde{N})$ and $ignored_{\mathcal{A}}(\widetilde{M}, \widetilde{N})$ defined in Equation 3.6 and Equation 3.7, respectively.*

*The set of transitions $\delta_{\widetilde{MN}}$ of the composed SIA $\widetilde{M} \otimes \widetilde{N}$ is defined in Equation 3.10.*

$$\begin{aligned}
&\langle \langle s_m, s_n \rangle, a, \langle s'_m, s'_n \rangle \rangle \in \delta_{\widetilde{MN}} \text{ iff} \\
&\qquad \vee \, (a \in shared_{\mathcal{A}}(\widetilde{M}, \widetilde{N}) \wedge \langle s_m, a, s'_m \rangle \in \delta_{\widetilde{M}} \wedge \langle s_n, a, s'_n \rangle \in \delta_{\widetilde{N}}) \\
&\qquad\quad \vee \, (a \notin shared_{\mathcal{A}}(\widetilde{M}, \widetilde{N}) \wedge \langle s_m, a, s'_m \rangle \in \delta_{\widetilde{M}} \wedge s_n = s'_n) \\
&\qquad\qquad \vee \, (a \notin shared_{\mathcal{A}}(\widetilde{M}, \widetilde{N}) \wedge s_m = s'_m \wedge \langle s_n, a, s'_n \rangle \in \delta_{\widetilde{N}})
\end{aligned} \tag{3.10}$$

By composing two SIAs $\widetilde{M}$ and $\widetilde{N}$ into a resulting SIA $\widetilde{MN}$, shared actions of the SIAs $\widetilde{M}$ and $\widetilde{N}$ become internal actions in the resulting SIA $\widetilde{MN}$. Together with the internal actions of each SIA they form the set of internal actions of the resulting SIA $\widetilde{MN}$:

$$\mathcal{A}^H_{\widetilde{MN}} = \mathcal{A}^H_{\widetilde{M}} \cup \mathcal{A}^H_{\widetilde{N}} \cup shared_{\mathcal{A}}(\widetilde{M}, \widetilde{N})$$

Note that ignored actions of the SIAs $\widetilde{M}$ and $\widetilde{N}$ are not propagated to the resulting SIA $\widetilde{MN}$.

The composition of two SIAs as defined by Definition 3.4 crates a composed process from the corresponding processes as defined by Definition 3.2.

### 3.2.3 Relation of a SIA to its Process

A SIA describes the protocol behaviour of a process, hence there is a direct relation between the set of ports $\mathcal{P}_N$ of a process $N$ and the set of actions $\mathcal{A}_{\widetilde{N}}$ of its SIA $\widetilde{N}$.

An output action $a_i \in \mathcal{A}^O_{\widetilde{N}}$ and an input action $a_j \in \mathcal{A}^I_{\widetilde{N}}$, each labelling a transition of SIA $\widetilde{N}$, represent the ability of the protocol of a process $N$ to write a message to the environment via an output port $a_i \in \mathcal{P}^O_N$ or read a message from the environment via an input port $a_j \in \mathcal{P}^I_N$, respectively. Note that input and output actions only represent the ability of the protocol to perform the action, whether it is possible to do so depends on the environment. This is due to the semantics of synchronous communication where both communication partners need to be ready for a transmission in order to transmit a message. Every input action $a_i \in \mathcal{A}^I_{\widetilde{N}}$ (output action $a_j \in \mathcal{A}^O_{\widetilde{N}}$) of the set of actions $\mathcal{A}_{\widetilde{N}}$ of SIA $\widetilde{N}$ has a corresponding input port $a_i \in \mathcal{P}^I_N$ (output port $a_j \in \mathcal{P}^O_N$) in the signature $\mathcal{P}_N$ of process $N$. However, not every port is necessarily used by a specific protocol description. Hence,

$$\mathcal{A}^I_{\widetilde{N}} \subseteq \mathcal{P}^I_N \tag{3.11}$$

$$\mathcal{A}^O_{\widetilde{N}} \subseteq \mathcal{P}^O_N \tag{3.12}$$

Internal actions are not related to the ports of a process as they happen independently of the environment the process is placed in. From Equation 3.1, Equation 3.11, and Equation 3.12 it follows that

$$shared_{\mathcal{A}}(\widetilde{M}, \widetilde{N}) \subseteq shared_{\mathcal{P}}(M, N)$$

This means that even though two processes $M$ and $N$ might share ports, their corresponding SIAs $\widetilde{M}$ and $\widetilde{N}$ need not necessarily describe this interaction, i.e. the process implementation does not rely on ignored ports.

Figure 3.2 depicts a process $N_1$ where the interaction protocol is described by a SIA $\widetilde{N}_1$. States are represented as nodes, transitions as directed edges where the label of an edge is the name of an action. Input actions are marked with a question mark '?', output actions are marked with an exclamation mark '!', and internal actions are marked with a semicolon ';'. Note that in the example of Figure 3.2 port $a2 \in \mathcal{P}_{N_1}$ has no corresponding action in SIA $\widetilde{N}_1$.

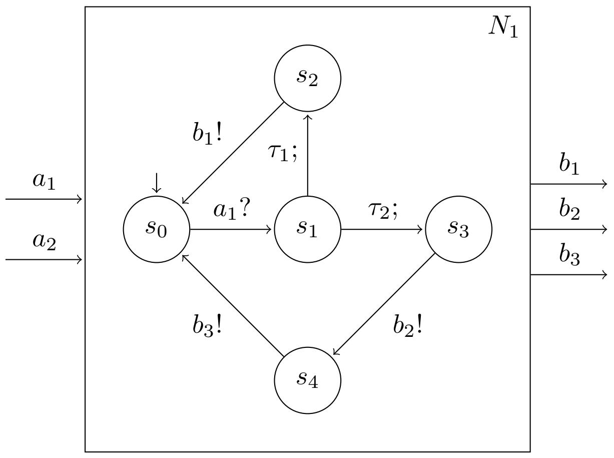


FIGURE 3.2: An example of a process $N_1$ where SIA $\widetilde{N}_1$ describes the interaction protocol of $N_1$.

Now, with a model to describe the interaction protocol of a process in a PNSC I need to take a closer lock on how SIAs interact with each other.

### 3.2.4 Interaction of SIAs

I use SIAs to describe the interaction protocol of processes in order to understand how processes interact with each other. The idea is to perform a binary composition of two SIAs, each describing the protocol of a process. The formal method of the composition is described later in Section 3.2.2. In this section I describe how two SIAs $\widetilde{M}$ and $\widetilde{N}$ interact with each other and in particular how the actions of each particular SIA are controlled.

Input and output ports of a process define the interface of the process to its environment. Input and output actions of a SIA, describing the protocol of this process, define how the process behaviour is interacting with the environment. Shared actions represent the actual transmissions of messages from one process to another while shared ports represent the synchronous channels, spawned between processes. Open actions represent the ability of a process to communicate with its environment via ports where the corresponding communication partner is not (yet) known.

In order to understand how a SIA reaches a state that is permanently blocking we have to understand how actions are controlled in more detail. To do this I will first discuss the control of shared input and output actions. For this purpose let's consider the example in Figure 3.3. The two processes $M_1$ and $N_2$ share the ports $a$ and $b$ and their corresponding SIAs $\widetilde{M}_1$ and $\widetilde{N}_2$ share the action $a$. The output action $b$ of SIA $\widetilde{N}_2$ is ignored.

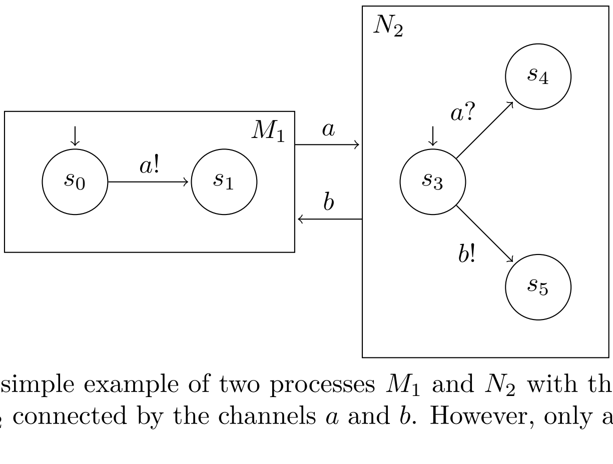


FIGURE 3.3: A simple example of two processes $M_1$ and $N_2$ with their corresponding SIAs $\widetilde{M}_1$ and $\widetilde{N}_2$ connected by the channels $a$ and $b$. However, only action $a$ is shared.

Initially, both SIAs $\widetilde{M}_1$ and $\widetilde{N}_2$ are in their initial state $s_0$ and $s_3$, respectively. From both of these states action $a$ is enabled. Additionally, action $b$ is enabled in state $s_3 \in S_{\widetilde{N}_2}$. Hence, in state $s_3$ two transitions are possible. However, as the output action

$b$ will never be able to be served by $N_2$'s environment (SIA $\widetilde{M_1}$ in this case) because no matching input action is available, the transition $\langle s_3, b, s_5\rangle$ will never be possible with the given environment $\widetilde{M_1}$. Therefore, from their respective initial state $s_0$ and $s_3$, SIAs $\widetilde{M_1}$ and $\widetilde{N_2}$ transition to state $s_1$ and $s_4$, synchronized by the label $a$. The decision in state $s_3 \in S_{\widetilde{N_2}}$ is imposed by the environment $\widetilde{M_1}$.

Let's now study the control of open actions. Open actions are either of direction input or output, hence, blocking and non-autonomous. However, in contrast to shared actions, with open actions the environment is not (yet) known. Hence, an assumption has to be made whether the environment will eventually provide the corresponding counter parts of the open actions. The goal of describing the protocol behaviour of processes with the help of SIA is to check all possible interaction behaviour in order to guarantee liveness. Hence, I will assume a *helpful* environment that is always providing open actions. This guarantees that potential permanent blocking states, that are only reachable through open actions, will still be considered. During the incremental composition of SIAs, open actions will eventually become shared actions and will then be considered where the environment is known. Therefore, potential permanent blocking states caused by such actions will be detected at this later stage.

Note that open actions are, like shared actions, blocking and non-autonomous but because a helpful environment is assumed they will never block and will always be served. Hence, open actions will always trigger the corresponding transitions in a SIA. Such an example is illustrated in Figure 3.4 where two processes $M_1$ and $N_3$ are connected by the shared ports $a$ and $b$. The interaction protocol of process $M_1$ is described by SIA $\widetilde{M_1}$. Process $N_3$ has additional ports $d$ and $e$ which are not connected and its interaction protocol is described by SIA $\widetilde{N_3}$. SIAs $\widetilde{M_1}$ and $\widetilde{N_3}$ share the action $a$.

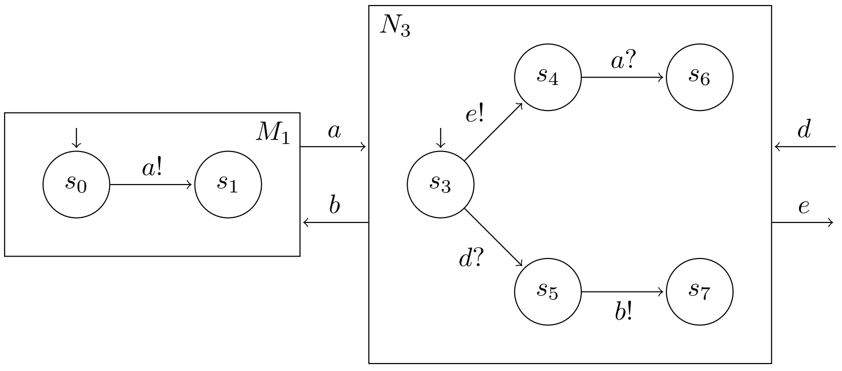


FIGURE 3.4: An example of a PNSC where the SIA $\widetilde{N_3}$ of process $N_3$ has the open actions $d$ and $e$.

While SIA $\widetilde{M_1}$ resides in its initial state $s_0$, SIA $\widetilde{N_3}$ starts in state $s_3$ and can transition to either state $s_4$ or $s_5$ triggered by the open action $e$ or $d$, respectively. As open actions are assumed to always be served by the environment, $\widetilde{N_3}$ can reach either of the states

$s_4$ and $s_5$ while SIA $\widetilde{M}_1$ is blocked in state $s_0$. With SIA $\widetilde{N}_3$ in state $s_4$, $\widetilde{M}_1$ and $\widetilde{N}_3$ synchronise on the shared action $a$ and transition to their respective state $s_1$ and $s_6$. Because the action $b \in \mathcal{A}_{\widetilde{N}_3}$ is ignored, $\widetilde{N}_3$ cannot perform any further transition from state $s_5$ to state $s_7$. Note that the decision at state $s_3 \in S_{\widetilde{N}_3}$ depends on a currently unknown environment. At a later stage, a new process, interaction with its environment via ports $d$ and $e$, might be added to the PNSC. By this, the system will gain knowledge of the environment, with respect to ports $d$ and $e$ and consequently, the corresponding actions of the respective SIA will be turned into shared actions.

As established in the previous section, internal actions are controlled only by the process itself and not by the environment. Hence, internal actions are independent of interactions and do trigger transitions autonomously. The PNSC illustrated in Figure 3.5 shows the two interacting processes $M_1$ and $N_4$. The interaction protocol of process $N_4$ includes transitions that are triggered by the internal actions $\tau_1$ and $\tau_2$.

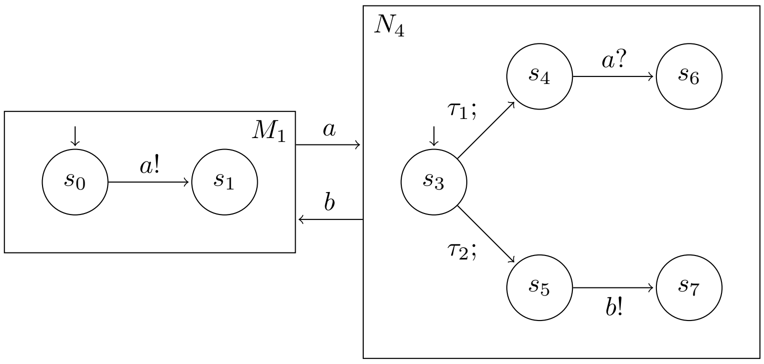


FIGURE 3.5: An example of a PNSC where the SIA $\widetilde{N}_4$ of process $N_4$ has the internal actions $\tau_1$ and $\tau_2$.

While SIA $\widetilde{M}_1$ temporarily blocks in its initial state $s_0$, SIA $\widetilde{N}_4$ starts in state $s_3$ and can transition to either state $s_4$ or $s_5$. The transitions are triggered by the internal actions $\tau_1$ and $\tau_2$, respectively. In contrast to the example of Figure 3.4 where open actions are triggering the transitions, here the transitions are independent of the environment. The choice at state $s_3 \in S_{\widetilde{N}_4}$ is unknown to the system because internal actions are hidden.

In Figure 3.5, the states $s_5 \in S_{\widetilde{N}_4}$ and $s_0 \in S_{\widetilde{M}_1}$ are permanent blocking states: With both SIAs $\widetilde{M}_1$ and $\widetilde{N}_4$ starting in their respective initial state $s_0$ and $s_3$, an autonomous choice of process $N_4$ may lead to an internal transition of $\widetilde{N}_4$ to state $s_5$. Because action $b \in \mathcal{A}^O_{\widetilde{N}_4}$ is ignored, a further transition from state $s_5$ will never be possible. Hence, SIA $\widetilde{N}_4$ is blocking indefinitely in state $s_5$. At the same time SIA $\widetilde{M}_1$ is still blocking in state $s_0$ and waits for SIA $\widetilde{N}_4$ to reach state $s_4$ in order to synchronize on shared action $a$. In the current situation, this can never happen because SIA $\widetilde{N}_4$ is indefinitely blocking in state $s_5$ and therefore SIA $\widetilde{M}_1$ is indefinitely blocking in state $s_0$.

In contrast to the example with internal action (as depicted in Figure 3.5), the example with open actions (as depicted in Figure 3.4) has permanent blocking states, only if the environment provides an action $d$. This is because in Figure 3.4 the environment with respect to actions $d$ and $e$ might become known at a later stage and action $d$ might be ignored, the transition $\langle s_3, d, s_5\rangle$ would not be possible and hence the permanent blocking state $s_5$ would never be reachable. Consequently, SIA $\widetilde{M_1}$ might never be permanently blocked in state $s_0$.

Up to this point, this chapter gave an informal understanding what permanent blocking means. Chapter 5 will provide a formal definition of permanent blocking states by analysing a composed system, as defined in Section 3.2.2.

As an example of a composition, let's consider the PNSC depicted in Figure 3.6 where two processes $M_2$ and $N_4$ share the ports $a$ and $b$ and process $M_2$ has an unconnected port $c$. By folding their corresponding SIAs $\widetilde{M_2}$ and $\widetilde{N_4}$ into the resulting SIA $\widetilde{M_2N_4}$, a

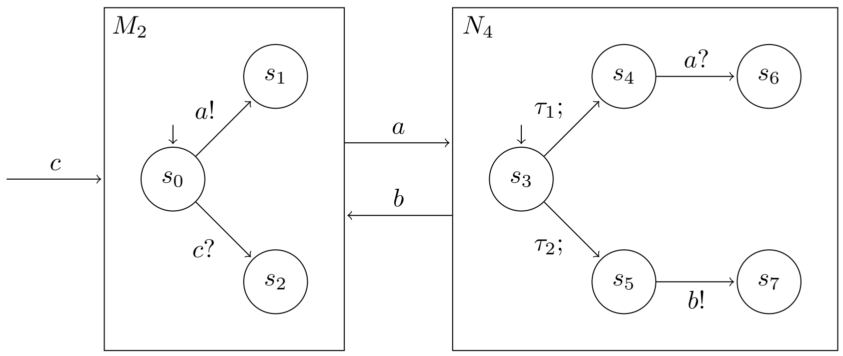


FIGURE 3.6: An example of a PNSC with SIAs containing open actions, shared actions, ignored actions and internal actions.

new composed process, I call it $M_2N_4$ in this example, is created. The composed process $M_2N_4$ with its corresponding SIA $\widetilde{M_2N_4}$ is depicted in Figure 3.7. All unreachable states of SIA $\widetilde{M_2N_4}$ have been removed for the sake of readability.

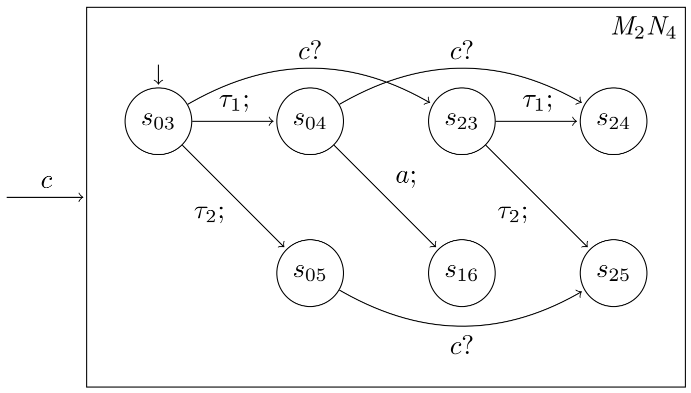


FIGURE 3.7: The composed process $M_2N_4$ with its composed SIA $\widetilde{M_2N_4}$ as a result of the composition of the system depicted in Figure 3.6.

The only remaining port of $M_2N_4$ is the unconnected port $c$ which relates to the open action $c \in \mathcal{A}^I_{\widetilde{M_2N_4}}$, triggering the transitions $\langle s_{03}, c, s_{23}\rangle$, $\langle s_{04}, c, s_{24}\rangle$, and $\langle s_{05}, c, s_{25}\rangle$.

Note that all transitions triggered by action $c \in \mathcal{A}^{I}_{\widetilde{M_2 N_4}}$ change only the part of the state corresponding to the SIA $\widetilde{M}_2$. The reason for this is that it is not possible to reach $s_{06}$ because $\widetilde{M}_2$ must transition to state $s_1$ in order to synchronize on action $a$. State $s_{07}$ cannot be reached because action $b \in \mathcal{A}^{O}_{\widetilde{N}_4}$ is ignored. With similar reasoning, I identify the internal actions $\tau_1$ and $\tau_2$ that originate from SIA $\widetilde{N}_4$. The shared actions $a$ of the SIAs $\widetilde{M}_2$ and $\widetilde{N}_4$ are turned into the internal action $a \in \mathcal{A}^{H}_{\widetilde{M_2 N_4}}$, triggering the transition $\langle s_{04}, a, s_{16} \rangle$.

## 3.3 Modelling a Crossroad with SIAs

In this section I will apply the SIA model on the crossroad example described in Figure 3.1. The interaction behaviour of the processes $P_{NW}$, $P_{NE}$, $P_{SE}$, and $P_{SW}$ are modelled with the SIAs $\widetilde{P}_{NW}$, $\widetilde{P}_{NE}$, $\widetilde{P}_{SE}$, and $\widetilde{P}_{SW}$, respectively, as depicted in Figure 3.8. I will only explain the rationale behind the system behaviour with respect to

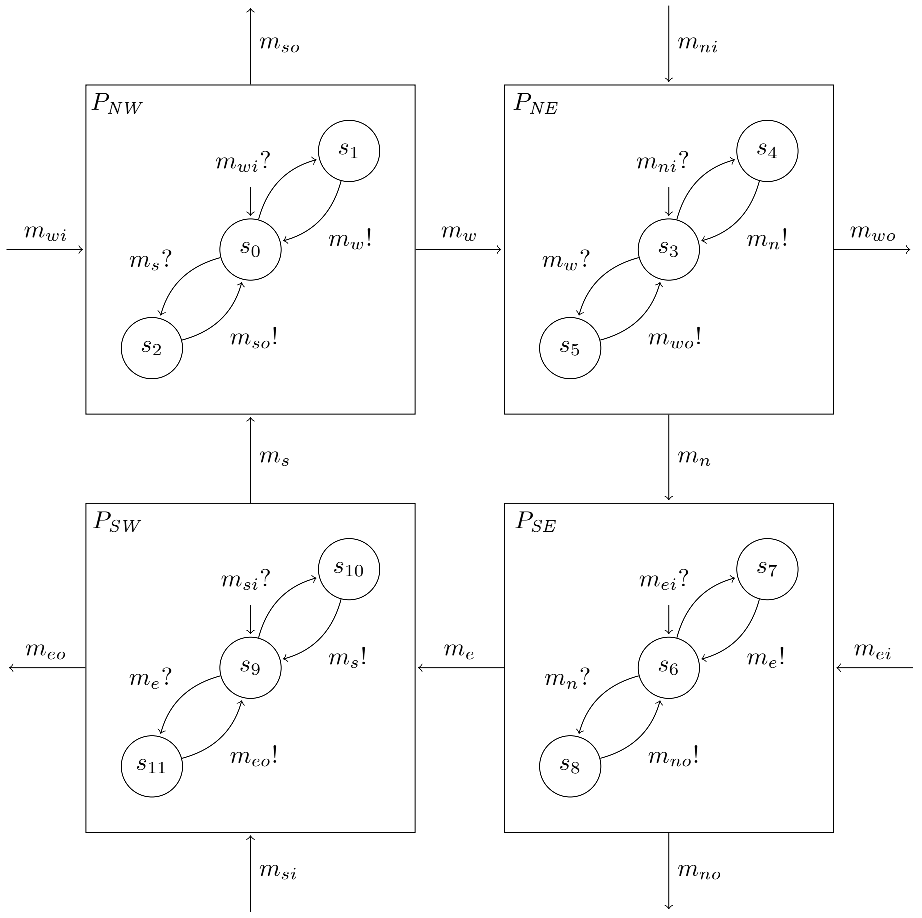


FIGURE 3.8: A PNSC model of Figure 3.1 extended by the corresponding SIAs of each process. Due to the symmetry of the model, the system can potentially reach a permanent blocking state — a deadlock, involving all four processes.

the lane arriving from the West and going to the East ($m_{wi}$, $m_w$, $m_{wo}$) but as each SIA in this example is modelled in a similar fashion the explanation is applicable for all of the lanes. At its initial state $s_0$ SIA $\widetilde{P}_{NW}$ is either triggered by the input action $m_{wi}$ or $m_s$. Those actions correspond to allocating a space on the intersection for a car as explained in Section 3.1. By triggering transition $\langle s_0, m_{wi}, s_1\rangle$ the space *NW* is allocated for the car arriving from the West. The request for space *NE* corresponds to the transition $\langle s_1, m_w, s_0\rangle$. The action $m_w$ is shared with SIA $\widetilde{P}_{NE}$, hence, the transition $\langle s_1, m_w, s_0\rangle$ can only be triggered if $\widetilde{P}_{NE}$ is in its initial state $s_3$. Triggered by the input action $m_w$, SIA $\widetilde{P}_{NE}$ performs the transition $\langle s_3, m_w, s_5\rangle$ to allocate the space *NE* to the car arriving from the West. By performing transition $\langle s_5, m_{wo}, s_0\rangle$, triggered by the output action $m_{wo}$, the car crosses the intersection and the spaces *NW* and *NE* are released.

The folding operation, described in Section 3.2.2, is applied incrementally on the model such that the resulting SIA $\widetilde{P}_{res} = \widetilde{P}_{NW} \otimes \widetilde{P}_{NE} \otimes \widetilde{P}_{SE} \otimes \widetilde{P}_{SW}$ is produced. By doing this we observe that the model is not free of permanent blocking. The SIA $\widetilde{P}_{res}$ is depicted in Figure 3.9[1] where a node, depicted as a black square, represents a state where no further progress is possible. The resulting SIA $\widetilde{P}_{res}$ of this system is already

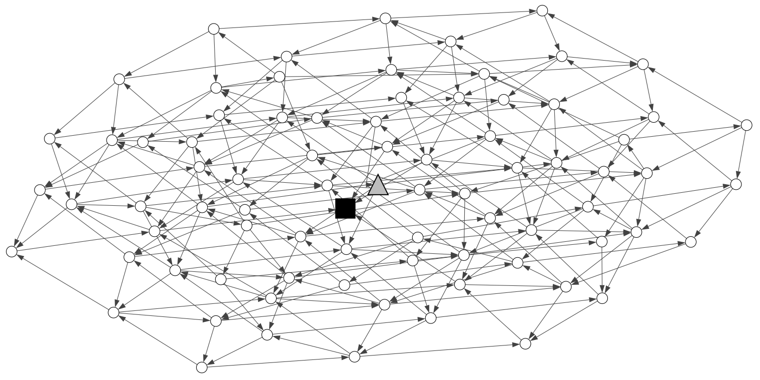


FIGURE 3.9: The graph representing the SIA $\widetilde{P}_{res}$. It consists of 80 states and 248 transitions. The black square node represents a state where no further transitions are possible and the grey triangle node represents the initial state.

quite complex. On the one hand this is due to the exponential growth of the state space. There exist solutions to prevent state space explosion in Labelled Transition Systems (LTSs) (e.g. [57]) but I will not discuss this further in this dissertation and postpone the analysis of how such a method can be applied to the here presented model to future work. On the other hand, the crossroad example has several open ports which causes the SIA $\widetilde{P}_{res}$ to describe every possible combination of cars, arriving and departing in a different order. Connecting producer processes to the open ports can serve to impose an order on how cars arrive and depart from the intersection and reduce the system complexity.

As mentioned in Section 2.2, this crossroad situation can lead to a deadlock. This is indicated by the state represented as a black square in Figure 3.9 where no further transitions are possible. While an imposed order of departing cars can always cause

[1]Produced with https://github.com/moiri/streamix-ifa

potential permanent blocking by placing the system in a hostile environment (e.g. a congestion on a lane after the intersection which prevents cars from freeing space on the intersection), the problem with this system is that the system can be stuck in a permanent blocking state even under the assumption that the open output actions $m_{wo}$, $m_{no}$, $m_{eo}$, and $m_{so}$ can always be served. A simple example where a deadlock situation occurs is when allocating one space for each car on their respective lane. This is illustrated by Figure 2.1b. The system is in a deadlock state because each SIA $\widetilde{P}_{NW}$, $\widetilde{P}_{NE}$, $\widetilde{P}_{SE}$, and $\widetilde{P}_{SW}$ synchronized on its respective input action $m_{wi}$, $m_{ni}$, $m_{ei}$, and $m_{si}$ and is permanently blocked in its respective state $s_1$, $s_4$, $s_7$, and $s_{10}$.

The fact that the model is not free of permanent blocking only means that there is the possibility of permanent blocking. For example, let's assume that on each lane a car arrives and departs before another car arrives on any other lane. This is modelled

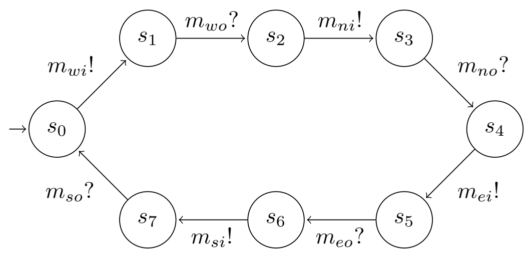


FIGURE 3.10: An example of an environment $\widetilde{P}_{env}$ for the PNSC of Figure 3.8.

by a SIA (representing the environment) $\widetilde{P}_{env}$ as depicted in Figure 3.10. Applying this environment to the system a trivial SIA $\widetilde{P}_{res} \otimes \widetilde{P}_{env}$ results, where in each state a transition is possible. It is a single circle as depicted in Figure 3.11.

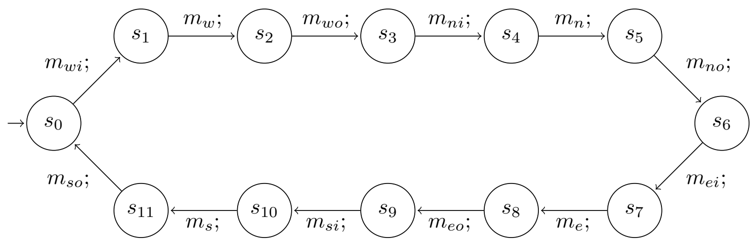


FIGURE 3.11: The resulting SIA $\widetilde{P}_{res} \otimes \widetilde{P}_{env}$ when applying the environment $\widetilde{P}_{env}$ of Figure 3.10 on the PNSC $\widetilde{P}_{res}$ of Figure 3.8.

In order to avoid the possibility of permanent blocking in the system of Figure 3.8, I have to change the topology of the PNSC and, as a consequence, adapt the SIAs according to the changes. The problem of the intersection is equivalent to the *dining philosophers problem* with four philosophers and four forks. Instead of sharing forks, as the philosophers are forced to do, the streams of cars have to share spaces. The deadlock of the dining philosophers problem can be solved by imposing the rule that one philosopher starts by picking up the right fork while all others start by picking up the left fork. Similarly, I break the circular wait in the crossroad example by letting process $P'_{NW}$ control the input flow from the West and the South and letting process $P'_{SW}$

control the output flow of the East and the South. Consequently, the directions of the ports $m_s$ are changed such that process $P'_{NW}$ produces $m_s$ and $P'_{SW}$ consumes it (while before process $P_{SW}$ was producing $m_s$ and $P_{NW}$ consuming it). The corresponding system is depicted in Figure 3.12. Note that only the processes $P'_{NW}$ and $P'_{SW}$ and their corresponding SIAs $\widetilde{P}'_{NW}$ and $\widetilde{P}'_{SW}$ are changed. Processes $P_{NE}$ and $P_{SE}$ and their corresponding SIAs $\widetilde{P}_{NE}$ and $\widetilde{P}_{SE}$ stay the same as in Figure 3.8.

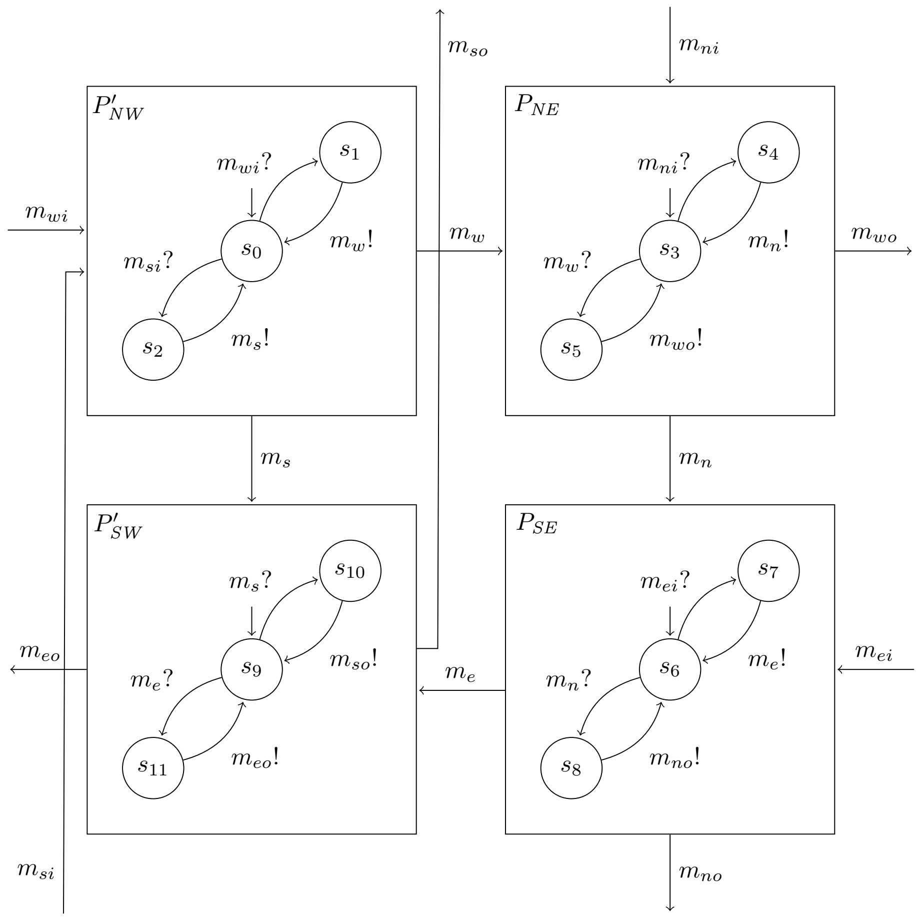


Figure 3.12: An adapted version of the crossroad model of Figure 3.8 that breaks the symmetry and resolves the problem of permanent blocking.

With this model the situation of Figure 2.1b cannot occur because the actions $m_{si}$ and $m_{wi}$ are both enabled in the same state $s_0$ of SIA $\widetilde{P}'_{NW}$ and only one of them can trigger a transition at a time. By applying the folding operation $\otimes$ incrementally on the subsystems the resulting SIA $\widetilde{P}'_{res} = \widetilde{P}'_{NW} \otimes \widetilde{P}_{NE} \otimes \widetilde{P}_{SE} \otimes \widetilde{P}'_{SW}$, as depicted in Figure 3.13, is computed. Here, from all states progress is possible. As with the previous example (Figure 3.8), because the system is open it can be placed in a hostile environment where back pressure generated by a congestion would prevent cars from freeing occupied places and hence lead to a permanent blocking. However, in contrast to the previous example, the redesigned system can cope with any input and will never block permanently.

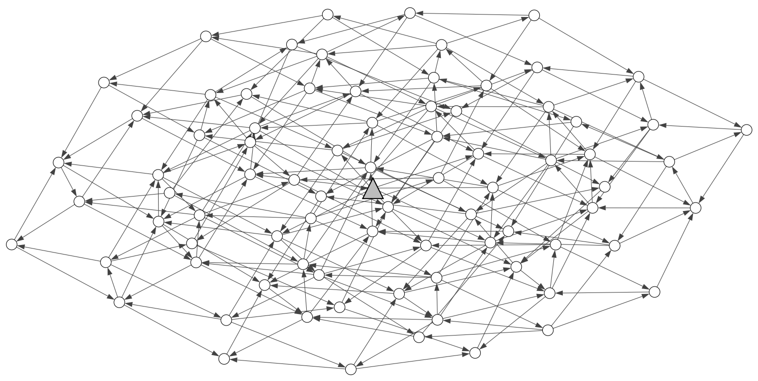


FIGURE 3.13: The graph representing the SIA $\widetilde{P}'_{res}$. It consists of 81 states and 252 transitions. The grey triangle node represents the initial state of the system.

### 3.3.1 Streaming Network with Buffered Communication

Up to this point I used a communication model based on synchronous communication. In this section I describe how to model a streaming network with buffered communication which is asynchronous.

In general, a streaming network is a composition of computational components and directed channels. Components consume and produce message tokens via input ports and output ports, respectively, at their interface. The streaming network is spawned by a composition of components that are interlinked by channels. A channel connects an input port with an output port and transmits message tokens via a *First-in, First-out (FIFO)* buffer. Message tokens are produced sporadically by components while components are executed. The execution of a component is triggered by message tokens arriving at the input ports of the component. Every input port is an independent trigger of a component. However, triggers become only active once the component is in the correct state to consume the message tokens at the corresponding input port.

If I map computational components and FIFO buffers to processes of a PNSC, a streaming network can be described by the interaction model presented in this chapter. The same as the interaction protocol of a process is described by a SIA, the order in which message tokens are consumed and produced by a computational component is described by a SIA.

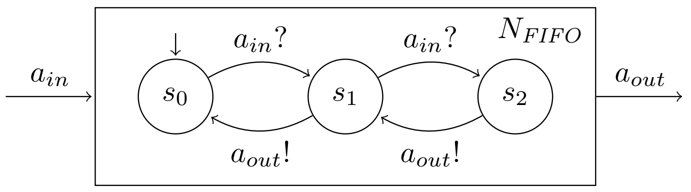


FIGURE 3.14: An example of a process $N_{FIFO}$ and its corresponding SIA $\widetilde{N}_{FIFO}$, modelling a FIFO buffer of length two with input $a_{in}$ and output $a_{out}$.

A FIFO buffer is modelled as a process with one input port and one output port. The interaction protocol of a FIFO buffer is described by a SIA with an input and an output action that trigger one or more transitions. Figure 3.14 depicts a process $P_{FIFO}$ and its corresponding SIA $\widetilde{P}_{FIFO}$, describing a FIFO buffer with two memory spaces. In order to model a FIFO buffer with more memory spaces, the chain of states and transitions in the SIA is increased accordingly.

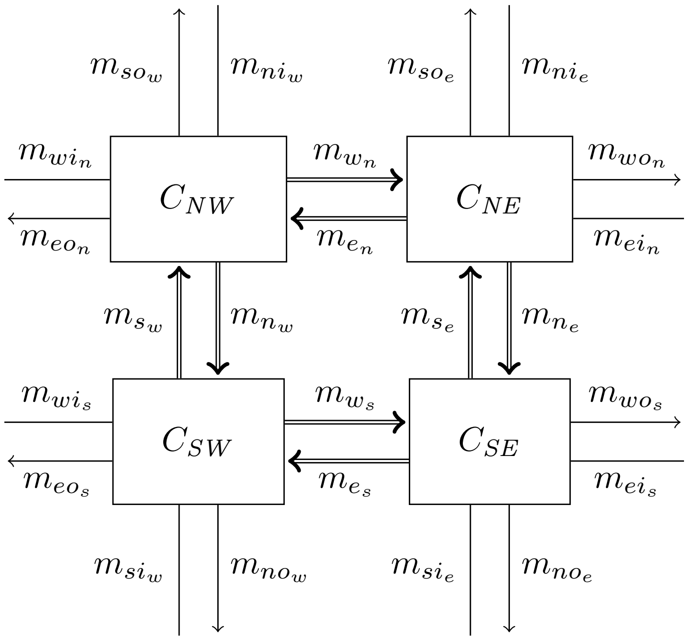


FIGURE 3.15: A PNSC with four composed processes, each modelling an intersection as depicted in Figure 3.12, interconnected by FIFO buffers (double-line arrows) forming a streaming network.

Applied on the crossroad example, multiple crossings can be connected as depicted in Figure 3.15. The four PNSCs $C_{NW}$, $C_{NE}$, $C_{SE}$, and $C_{SW}$ each represent the assembly of processes modelling the deadlock-free crossroad behaviour (depicted in Figure 3.12). The composed SIA of Figure 3.13 describes the interaction protocol of each of the computational components $C_{NW}$, $C_{NE}$, $C_{SE}$, and $C_{SW}$. The double-line arrows represent FIFO channels that are connected to the corresponding input and output ports of the computational components. The FIFO channels are modelled with processes that are similar to the process depicted in Figure 3.14, but not necessarily of the same length.

Even though the system is composed of subsystems that are guaranteed to be free of permanent blocking, its is not guaranteed that the composed system is free of permanent blocking. This is because of the back-pressure imposed on cars attempting to leave an intersection, caused by cars that are blocked in an intersection ahead. The situation depicted in Figure 3.16 is one example of a potential permanent blocking situation that can occur.

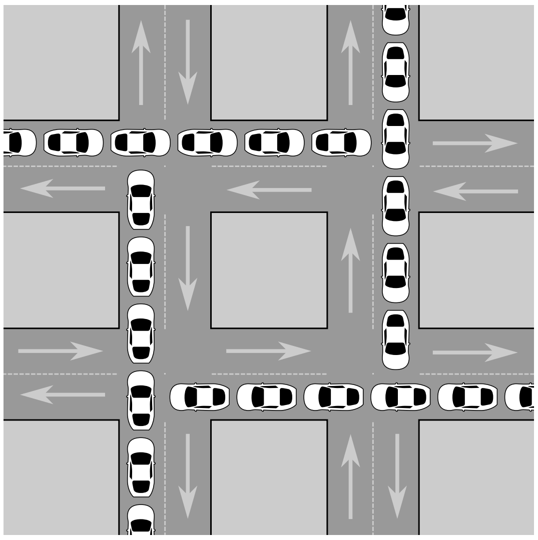


FIGURE 3.16: A gridlock with four intersections. This system is modelled by a PNSC in Figure 3.15.

## 3.4 Chapter Summary

In this chapter I introduced the PNSC model. It allows to describe networks of processes that are connected through synchronous channels. In order to describe the interaction of a process with its environment I introduced SIAs. A SIA is an automata-based model that describes the interaction protocol of a process where actions trigger transitions from one protocol state to another. The model allows to compose processes and their corresponding SIAs in order to provide an abstract representation of the network. This allows to build hierarchical networks and to reuse sub-networks as composed processes. Due to the synchronous communication semantics of the channels, SIAs provide a rigorous language to describe the blocking behaviour of processes.

I used an example of a crossroad to illustrate the power of the model to analyse the interaction of processes. I used the same example in a bigger context where each crossroad, as a composed process, is part of bigger system, modelled as a streaming application.

Chapter 4

# Mixed-criticality PNSCs and Time-based Processes

A property of systems in the world of Cyber-physical Systems (CPSs) is the diversity of components with respect to their criticality and their underlying communication paradigms. Such systems are called mixed-criticality CPSs. They are composed out of components where the consequence of a failure depends on the criticality level of the component. For example, in a car, a failure of the anti-lock braking system is more critical than the failure of the navigation system. One challenge of the design and implementation of a mixed-criticality CPS lies in preventing interference between components of different criticality levels. This is especially challenging if components with different criticality levels need to interact with each other.

Another challenge in the design of CPSs is the time-criticality of an application. Traditional software development is based on the best-effort principle where the aspect of time is removed from the control of the programmer. This lies in contrast to the requirements of a time-critical application where guarantees must be given that a certain piece of code must execute and terminate before a given deadline. Such deadlines are imposed by the physical world due to physical properties tied to the problem the application aims to solve.

In this chapter I am going to describe how the Process Network with Synchronous Communication (PNSC) model (Chapter 3) is extended to support mixed-criticality aspects in the event-triggered domain by communication decoupling. I further introduce timing aspects to the model in order to control communication and triggering rates of processes. This serves to enable the modelling of safety-critical systems as such systems profit form predictability and stability of communication rates. The reminder of the chapter is structured as follows: In Section 4.1 I discuss the different coupling aspects of communication and describe how communication decoupling in PNSCs is achieved in different dimensions. I use the decoupling mechanisms in conjunction with timers to control communication and execution rates of processes. This is described in Section 4.2 where two aspects of rate-control are discussed: First, control on the execution-rate of processes is achieved by removing the event-triggering semantics of a process with the help of communication decoupling and replacing it with a time-triggering semantics where a clock triggers the process execution periodically (Subsection 4.2.1). Second, the communication rate on individual input ports of processes can be bound to a upper limit to gain control over the used communication bandwidth. I describe several strategies of how this is achieved in Subsection 4.2.2. In Section 4.3 I define three different message types and describe their properties, application fields, and how this relates to communication coupling. In Section 4.4 I describe Cross-criticality Interfaces (CCIs) which summarise the different interaction patterns between processes, based on the triggering semantics of the processes and the types of message tokens. I also provide an example of a mixed-criticality system from the video-processing domain and use the extended PNSC model to describe the system. Finally, in Section 4.5 I provide a summary of the chapter, discuss the limitations of the PNSC model with respect to time-critical systems, and propose future work related to the topics discussed in this chapter.

## 4.1 Communication Decoupling of PNSCs

One research goal is to provide a platform, suitable for safety-critical services, that is able to perform controlled changes at runtime which is in contrast to the current practice of allowing no changes in safety-critical systems. Such a platform needs to support the coexistence and segregation of closed subsystems and open subsystems with *CCIs* between them. I consider a subsystem as *closed* if it doesn't have information flow to or from another subsystem that is essential for the correct functioning of the closed subsystem. These CCIs will ensure controlled information flow between closed subsystems and open subsystems such that the closed subsystem cannot be compromised in its correctness in case of misbehaviour by an open subsystem. CCIs avert unwanted coupling of behaviour through the interfaces.

In Section 2.5 I described communication decoupling in three different dimensions: time, space, and synchronization. For mixed-criticality systems it is crucial to achieve communication decoupling in order to prevent interference of the lower critical component to the higher critical component. A PNSC, as described in this chapter, relies heavily on synchronous communication. Synchronous communication is coupled in time because both communication partners need to operate at the same time in order to communicate. Further, it is also coupled in synchronization because both, the read and the write operation, is blocking. However, the PNSC model achieves decoupling in space due to the presence of channels: processes read from ports (write to ports) without knowing who the producer (the consumer) is. Communication dependencies are automatically defined through channels that connect to ports with each end.

To make the PNSC model suitable for mixed-criticality CPSs the model needs to be enriched with the possibility to decouple communication in time and synchronisation.

### 4.1.1 Decoupling PNSCs in Time

To decouple communication in time, a process $N_1$ needs to be able to send a message token to another process $N_2$ without having to wait for $N_2$ to be ready to receive the message token and vice versa. This can be achieved by introducing a buffer that is independent of the communication partners: a producer can write to the buffer independent of the consumer and the consumer can read from the buffer independent of the producer. Instead of using a synchronous channel to connect two ports of two processes, a buffering element can be used. As shown in Section 3.3.1, a streaming network with buffered communication can easily be built with the PNSC model if each First-in, First-out (FIFO) buffer (with predefined buffer size) is simply modelled by a process.

Consequently, communication decoupling in time is achieved by introducing an abstraction layer where atomic processes are no longer connected by synchronous channels but by FIFO channels (each modelled as an atomic PNSC process). An example of such an abstraction is illustrated in Figure 3.15 where each double arrow represents an abstraction for a FIFO buffer. An example of a FIFO buffer with two memory spaces is depicted in Figure 3.14. I use the notation $\xRightarrow{a[l]}$ to describe a FIFO buffer $a$ of length $l$. If the length $l$ is omitted, a default of $l = 1$ is assumed.

### 4.1.2 Decoupling PNSCs in Synchronisation

In an event-triggered communication model, such as the PNSC model, the process trigger semantics controls the forward progress of the computation. Since a read operation on a port is blocking, the process will be triggered by the arrival of a message token, i.e. the arrival of a message token initiates the progress in the flow direction of the data. Due to the blocking write of a PNSC process the progression of computation is controlled against the flow direction of the data. I call this back-pressure.

The blocking communication semantics in a PNSC is crucial to ensure event safety in the communication: It is not possible to unintentionally duplicate or lose message tokens as processes are blocked as long as the communication partner is not available to exchange information. The cost, however, is that blocking creates mutual interference between components. In case of a mixed-criticality system, it must be possible to punctually remove the blocking semantics of read and/or write operations of a process (decoupling in synchronisation) to prevent interference.

Note that decoupling communication in time does not automatically achieve decoupling in synchronisation. This is because blocking is possible if no space is available in the buffer for a producer to write to or the buffer is empty and can offer nothing to a consumer.

The blocking semantics of an action $a \in \mathcal{A}^I_{\widetilde{N}} \cup \mathcal{A}^O_{\widetilde{N}}$ in a Synchronous Interface Automaton (SIA) $\widetilde{N}$ can be removed by adding a self-loop transition, triggered by action $a$, to each state of the SIA where action $a$ is not enabled. This means that the implementation of process $N$ needs to be able to serve the action $a$ in each state of its corresponding SIA $\widetilde{N}$. This delegates the problem of decoupling to the implementation of the process which is undesirable because it violates the concept of exogenous coordination.

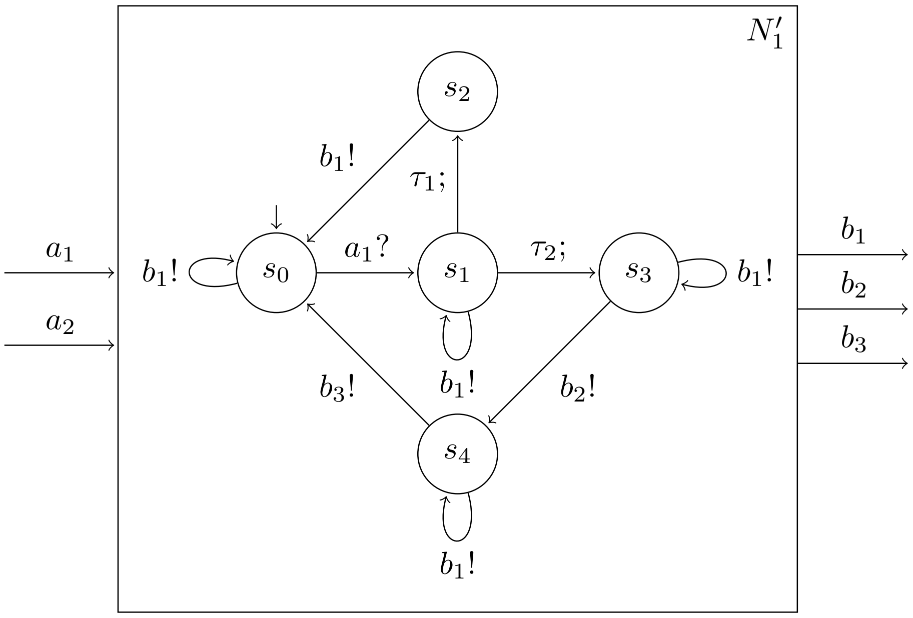


FIGURE 4.1: An example of a process $N'_1$ where SIA $\widetilde{N}'_1$ describes the interaction protocol of $N'_1$ with port $b_1$ decoupled in synchronisation.

As an example let's consider the process $N_1$ and its corresponding SIA $\widetilde{N}_1$ as depicted in Figure 3.2 where no port is decoupled in synchronisation. Figure 4.1 depicts the same situation expect that here, port $b_1$ is decoupled in synchronisation. Due to this decoupling, the SIA $\widetilde{N}'_1$ has a self-loop transition, triggered by action $b_1 \in \mathcal{A}^O_{\widetilde{N}'_1}$, on each state except state $s_2$ where action $b_1$ is enabled.

This modification of a process and its SIA is not very convenient because it requires the necessity of different process implementations depending on the context a process is used in. To avoid this, I introduce a decoupling process $N_D$ with SIA $\widetilde{N}_D$ as depicted in Figure 4.2. The SIA $\widetilde{N}_D$ of the decoupling process has only one state $s_5$ where it

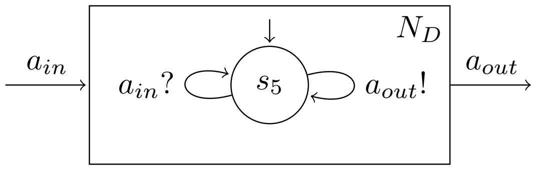


FIGURE 4.2: An example of a decoupling process $N_D$ and its SIA $\widetilde{N}_D$ with input $a_{in}$ and output $a_{out}$.

accepts an action $a_{in} \in \mathcal{A}^I_{\widetilde{N}_D}$ and an action $a_{out} \in \mathcal{A}_{\widetilde{N}_D}$. The decoupling process can never block because each transition in its SIA is immediately returning to the initial state which allows the process to always consume an input when one is provided and always produce an output when one is required. In order for a decoupling process to provide an output before it received an input, an initial value must be defined.

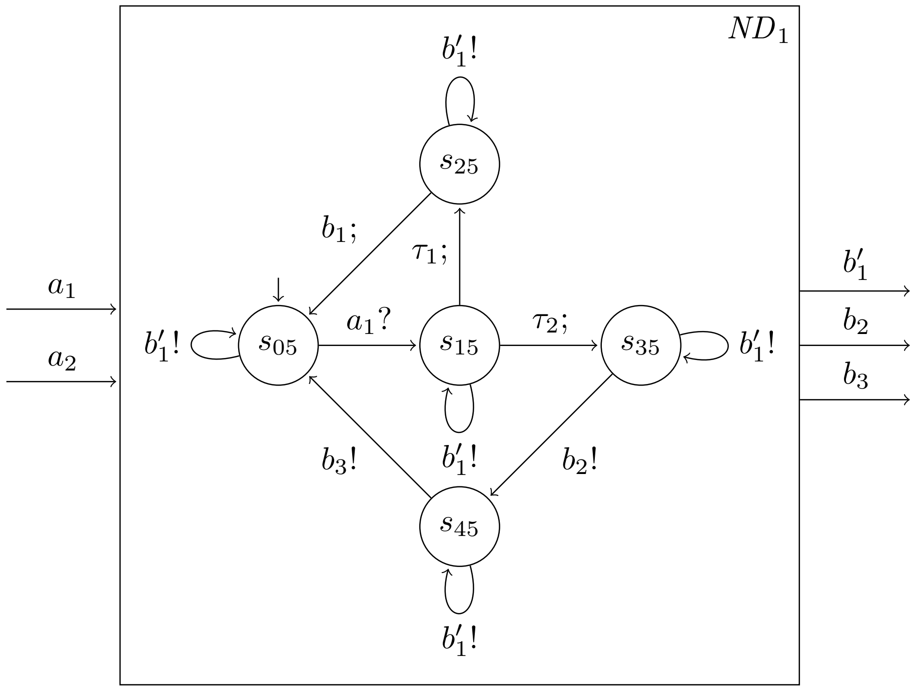


FIGURE 4.3: The resulting abstract process $ND_1$ of the composition $N_1 \otimes N_D$ where SIA $\widetilde{ND_1} = \widetilde{N}_1 \otimes \widetilde{N_D}$. SIA $\widetilde{ND_1}$ is syntactically equivalent to SIA $\widetilde{N}'_1$, depicted in Figure 4.1.

Note that a decoupling process is symmetric. This means that if a decoupling process $N_D(b_1)$ is used to decouple the output port $b_1$ of process $N_1$, not only does it make the write access of process $N_1$ to port $b_1$ non-blocking but also any read access from the environment to the decoupling process $N_D(b_1)$ becomes non-blocking. This is illustrated by Figure 4.3 where a decoupling process on channel $b_1$ is composed with process $N_1$ (in order to avoid naming conflicts, the output port of the decoupling process is renamed to $b'_1$). The resulting process is named $ND_1$ and its corresponding SIA is defined as

$\widetilde{N}_1 \otimes \widetilde{N}_D(b_1)$. We observe, that the transition $\langle s_{25}, b_1, s_{05}\rangle$ is now triggered by the internal action $b_1$. Further, we observe that on each state a transition, triggered by the output action $b_1'$, is available which means that the process $ND_1$ is able to serve action $b_1'$ in any state to the environment i.e. any write access to the environment and any read access from the environment is decoupled. Note that Figure 4.1 and Figure 4.3 are syntactically equivalent.

I conclude that by using a decoupling process on a synchronous channel, the blocking semantics of the channel is removed for the read and the write operation. Consequently, on a synchronous channel it is only possible to decouple read and write operations simultaneously. If read and write access must be decoupled individually, the decoupling in synchronisation must be combined with decoupling in time. I discuss this in detail in Subsection 4.1.3 of this section. I use the notation $\rightarrowtail$ to describe a channel where the write access is non-blocking, $\twoheadrightarrow$ to describe a channel where the read access is non-blocking, and $\twoheadrightarrowtail$ to describe a channel where the write and the read access is non-blocking. As discussed above, a decoupling process, as depicted in Figure 4.2, enforces decoupling in synchronisation for read and write access and is, hence, represented as $\twoheadrightarrowtail$. Note that $\twoheadrightarrowtail$ is only a theoretical construct and in practice decoupling a synchronous channel in synchronisation is equivalent to removing the channel. This is because the producer and the consumer would need to access the channel at the exact same time and use the physical wire as a buffer to transmit a message token. In combination with decoupling in time, however, the decoupling process becomes a powerful tool to punctually prevent interference between interacting processes.

### 4.1.3 Decoupling PNSCs in Time and Synchronisation

Achieving decoupling in synchronisation of a process requires either a modification of the process implementation or an additional decoupling process (as depicted in Figure 4.2). However, by combining decoupling in synchronisation with decoupling in time the modification can be applied to the buffer instead of the process. This allows to connect a process to different types of buffers in order to achieve either input or output decoupling (or both) in synchronization and leave the process implementation unchanged. In this section, ports are always decoupled in time and space. When, additionally, a port is decoupled in synchronisation I will simply describe it as *decoupled.*

To achieve this I use the same abstraction as with decoupling in time: synchronous channels are replaced by buffers. The buffers are not simple FIFO buffers but have internal logic. The type of buffer depends on the decoupling semantics of the process ports. In the following I will describe the different buffer types.

Decoupling an output of a process means that no back-pressure is exert on this output and the process cannot be blocked due to a congestion on the connecting channel. This is achieved by allowing the process to write to the connecting FIFO channel, independent of whether there is available space or not. If the FIFO buffer is full, the message token in the last buffer space is overwritten by the new message token. This leads to the potential loss of message tokens. The loss of message tokens can be tolerable in certain circumstances. This will be discussed in more detail in Section 4.3. Figure 4.4 depicts

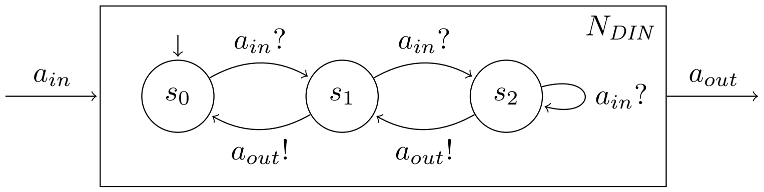


FIGURE 4.4: An example of a SIA, modelling a FIFO buffer of length 2 with a decoupled input $a_{in}$ and an output $a_{out}$.

a FIFO buffer $N_{DIN}$ of length two that is attached to a decoupled output port $a_{in}$ of a process and to a blocking input port $a_{out}$ of another process. In state $s_0$ the buffer waits for an input action $a_{in}$ to trigger the transition $\langle s_0, a_{in}, s_1 \rangle$. Once the input is served, in state $s_1$, two transitions are possible. Either the transition $\langle s_1, a_{out}, s_0 \rangle$ if a consumer is reading from the buffer or the transition $\langle s_1, a_{in}, s_2 \rangle$ if another message is written to the buffer. In state $s_2$ the buffer is ready to receive any number of message tokens due to the transition $\langle s_2, a_{in}, s_2 \rangle$ which models the non-blocking behaviour of the buffer. I use the notation $\rightarrowtail\xrightarrow{a[l]}$ to describe a decoupled FIFO buffer $a$ of length $l$ where the arch symbol at the source of the arrow indicates the fact that the input of the FIFO is decoupled in synchronisation and, consequently, message tokens can potentially be overwritten and discarded due to the decoupling.

By decoupling an input of a process, this input is excluded from the triggering semantics of the process: a read operation on this port will never block and always return the last message token stored in the connected buffer. Due to this, decoupling of an input port is only possible for processes with multiple input ports where at least one other input port is triggering the component. This prevents a component from constantly consuming the same message token without ever being blocked. Reading from a decoupled input can lead to the duplication of message tokens. The same as the loss of message tokens, the duplication of message tokens can be tolerated in particular circumstance. Refer to Section 4.3 for more information on the subject. An example of a decoupled FIFO buffer $N_{DOUT}$ of length two is depicted in Figure 4.5. When connected between a producer and a consumer process, such a FIFO buffer models a decoupled read access of the consumer process to port $a_{out}$ of the FIFO buffer while the write access of the producer

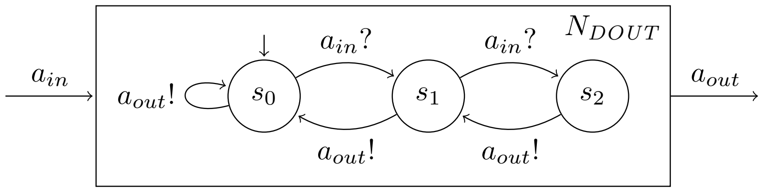


FIGURE 4.5: An example of a SIA, modelling a FIFO buffer of length 2 with input $a_{in}$ and a decoupled output $a_{out}$.

process to the FIFO buffer (via port $a_{in}$) is blocking. The initial state $s_0$ allows two transitions: Either the transition $\langle s_0, a_{in}, s_1\rangle$ if a message token is written to the buffer or the transition $\langle s_0, a_{out}, s_0\rangle$ if a consumer is reading from the buffer. If the latter is the case, the last message token that was written to the buffer is duplicated and the duplicate message token is read by the consumer. Duplication of the message token ensures that any modification done to the message token by the consumer process does not change the original message token residing in the buffer.

Note that if a message token has never been written to the buffer an empty message token is returned. I use the notation $\xrightarrow{a[l]}\!\!\!\twoheadrightarrow$ to describe a decoupled FIFO buffer $a$ of length $l$ where the double arrow indicates the fact that the output of the FIFO is decoupled and, consequently, message tokens can potentially be duplicated.

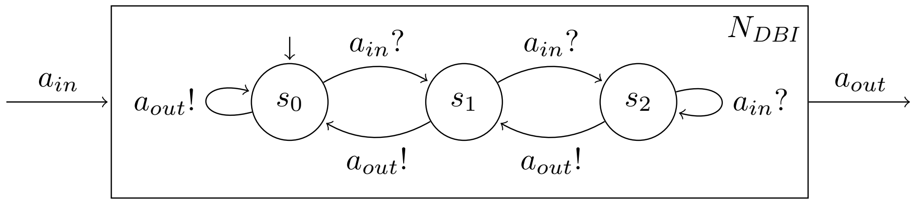


FIGURE 4.6: An example of a SIA, modelling a FIFO buffer of length 2 with a decoupled input $a_{in}$ and a decoupled output $a_{out}$.

Figure 4.6 shows an example of a decoupled FIFO buffer $N_{DBI}$ of length two that is a combination of the buffers depicted in Figure 4.4 and Figure 4.5. Such a buffer is used when a decoupled output port of one process is connected to a decoupled input port of another process. I use the notation $\rightarrowtail\!\!\xrightarrow{a[l]}\!\!\!\twoheadrightarrow$ to describe a FIFO buffer $a$ of length $l$ where the input and the output is decoupled.

## 4.2 Time-based Component Model of PNSCs

In order to give guarantees that a piece of code produces a result before a deadline has passed, one has to know the exact execution time of this piece of code, running on a specific piece of hardware. However, the execution time of an application, running on a particular hardware platform is not constant because of memory hierarchies, branch predictions, and multi-core designs that are commonly found in modern hardware architectures. In order to circumvent these problems, huge efforts have been made to compute the Worst-case Execution Time (WCET) of an application, running on a specific hardware platform [44].

In order to determine the WCET of a process, without analysing the behaviour of the environment this process is placed in, a clear separation between communication and computation is necessary [58]. This lies in stark contrast to an event-triggered component model, as the one described in Chapter 3, where the activation of a process is influenced by the interaction with other processes due to the blocking nature of the communication. As an alternative, Kopetz proposed the Time-triggered Architecture (TTA) where a system is decomposed into an assembly of components where each component is triggered according to a fixed schedule. In Section 2.5 I gave a short introduction to time-triggered communication which describes one aspect of a TTA. In his book [34], Kopetz discusses time-critical systems and provides an extensive description of TTAs.

In the following I will discuss two methods to control the execution rate of processes in a PNSC by using communication decoupling mechanisms, introduced in Section 4.1, and clock signals. One method is a PNSC version of a time-triggered architecture where the rate of production and consumption of message tokens in a PNSC is controlled by an element which I call *temporal firewall*. The other method consists of limiting the production rate of message tokens to bound the communication rate to an upper limit. The former method is described in Subsection 4.2.1 and the latter in Subsection 4.2.2.

### 4.2.1 Time-triggered Processes in a PNSC

In order to change the triggering semantics of a PNSC, I introduce the concept of a temporal firewall that allows to enforce a time-triggered semantics on a process or a PNSC instead of the sporadic triggering behaviour inherent to PNSCs. A temporal firewall $N_{FW}$ is a PNSC process that is connected to a clock signal through port $p_{clk} \in \mathcal{P}_{N_{FW}}$. The clock signal is oscillating at a constant rate and is causing the temporal firewall to trigger at the rate of the clock signal. A temporal firewall has another input and a matching output port which are both decoupled in synchronization. This means that the arrival of a message token at the input port is not triggering the execution of the process, a read access on the input port is non-blocking, and a write access on the output port is non-blocking (see Section 4.1.2). Upon the trigger event, caused by the connected clock signal, a temporal firewall consumes a message token from its input port and produces a message token at its output port. As the input and the output is decoupled, the production and consumption of message tokens is independent of the environment of the temporal firewall and it can never be blocked. An example of a behaviour of a temporal firewall is illustrated in Figure 4.7. The upper time line

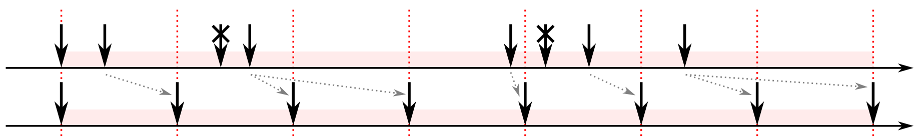


FIGURE 4.7: An example of the behaviour of a temporal firewall where the upper timeline represents the arrival instances of message tokens and the lower timeline represents the release instances of message tokens.

represents the arrival instances of message tokens at the input of a temporal firewall and the lower timeline represents the release instances of message tokens from the output of the temporal firewall. The vertical dotted lines indicate the rate of the clock that triggers the temporal firewall. A crossed-out arrow represents a message token that is overwritten by a newer message token (e.g. the third message token is overwritten by the fourth and the sixth message token is overwritten by the seventh).

Multiple temporal firewalls can be synchronized by synchronizing the clock signals triggering the temporal firewall. This is only possible if the clock rate of the synchronized temporal firewalls is equal.

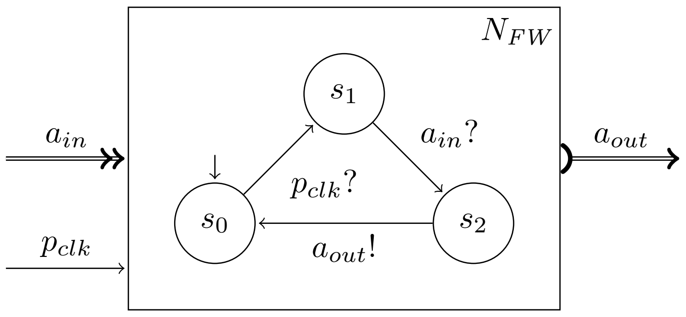


FIGURE 4.8: A temporal firewall with a decoupled port pair $\langle a_{in}, a_{out}\rangle$ and a port $p_{clk}$ which is connected to a clock signal.

A temporal firewall $N_{FW}$ with its corresponding SIA $\widetilde{N}_{FW}$ is depicted in Figure 4.8. The ports $a_{in}$ and $a_{out}$ are decoupled in synchronisation. A clock signal is connected to the port $p_{clk}$. For simplicity I will use the notation $\overset{a(p_{clk})}{\rightsquigarrow}$ to describe a temporal firewall on channel $a$, triggered by the clock signal $p_{clk}$, as depicted in Figure 4.8. Note that decoupled FIFO channels of length one instead of synchronous channels are connected to the input and output port of a temporal firewall. This is because of the following: While the input and output of a temporal firewall is decoupled in synchronisation by default, the output of the producing process and the input of the consuming process, connected to the temporal firewall, must not necessarily be decoupled in synchronisation. In order to achieve this separate control of decoupling in synchronisation, decoupling in time is required, hence, buffered channels are used by default.

By connecting all input and output ports of a PNSC (or a single process) to synchronized temporal firewalls, a time-triggered behaviour is imposed onto the PNSC. I call such a

PNSC a time-triggered PNSC. The time-triggered behaviour is enforced because message tokens can only reach the time-triggered PNSC through temporal firewalls and the temporal firewalls produce message tokens at the specified rate which causes the time-triggered PNSC to trigger at the specified rate. Temporal firewalls can be connected to a single process or a PNSC. When connected to a single process, no sporadic triggering is possible because the time-triggered process is completely decoupled from the rest of the network through temporal firewalls. The WCET of the process must be guaranteed to be smaller than the triggering rate imposed by the temporal firewalls. When connected to a PNSC, only processes that communicate with the environment of the time-triggered PNSC are decoupled by temporal firewalls. Other processes inside the time-triggered PNSC may still be triggered through sporadic message token transmissions.

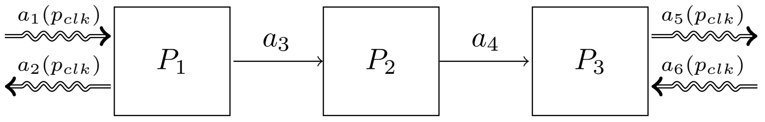


FIGURE 4.9: An example of a time-triggered PNSC where the PNSC is decoupled through temporal firewalls while processes inside the time-triggered PNSC trigger sporadically (e.g. process $P_2$).

As an example let's consider the time-triggered PNSC depicted in Figure 4.9. The process $P_1$ interacts through the temporal firewalls $a_1(p_{clk})$ and $a_2(p_{clk})$ with its environment to the left and process $P_3$ interacts through $a_5(p_{clk})$ and $a_6(p_{clk})$ with its environment to the right. Consequently, the time-triggered PNSC, framed by the four temporal firewalls $a_1$, $a_2$, $a_5$, and $a_6$, is decoupled from its environment through the temporal firewalls. Process $P_2$, however, which is placed inside the time-triggered PNSC is not connected to any temporal firewall and is triggering sporadically depending on the arrival of message tokens from process $P_1$. Process $P_3$ is triggered through message tokens arriving at the input $a_6$, produced at a rate $p_{clk}$ from a temporal firewall, and through message tokens arriving through input port $a_4$, sporadically produced by process $P_2$. A time-triggered PNSC $PN$, framed by synchronized temporal firewalls, must satisfy Property 4.1 and Property 4.2.

**Property 4.1** (Time-triggered Input)**.** *At the start of each round $r$, a process $P \in PN$ connected to a temporal firewall via an input port consumes at least one message token.*

**Property 4.2** (Time-triggered Output)**.** *Before the start of the next round $r+1$, a process $P \in PN$ connected to a temporal firewall via an output port produces at least one message token.*

Note that if more than one message token is consumed or produced by a process $P \in PN$, connected to a temporal firewall, the corresponding port of the process must be decoupled in order to not block the process, given that a temporal firewall produces and consumes exactly one message token.

To tighten the control on the rate of message tokens passing in the time-triggered PNSC $PN$, each process $P \in PN$ must be framed by synchronized temporal firewalls. This is illustrated in Figure 4.10 which depicts the same example as Figure 4.9 but without any sporadic triggering of processes. Here, in each clock cycle message tokens are passed from

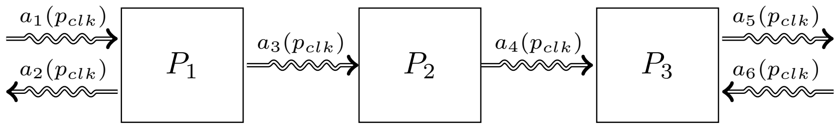


FIGURE 4.10: An example of a time-triggered PNSC where each process in the PNSC is decoupled through temporal firewalls.

one process to the next via temporal firewalls and no sporadic triggering is occurring in the time-triggered PNSC. However, this is only the case if temporal firewalls are synchronized which allows a concatenation of two temporal firewalls, triggered by the same clock rate, to be merged into one (as depicted in Figure 4.10).

If the temporal firewalls, framing the processes are not synchronized, i.e. their clock rate differs, temporal firewalls cannot be merged and the time instant of when one temporal firewall produces a message token is not necessarily synchronized with the time instant the neighbouring temporal firewall is consuming a message token. Consequently, the time instance of passing a message from one process to another is not coordinated. An example of such a case is depicted in Figure 4.11 where the processes $P_1$ and $P_2$ are framed by temporal firewalls, each triggered by a different clock rate $clk_1$ and $clk_2$, respectively.

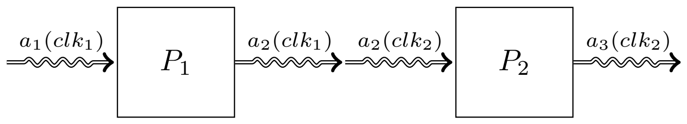


FIGURE 4.11: An example of a time-triggered PNSC where each process in the PNSC is decoupled through a temporal firewall with a different clock rate each.

### 4.2.2 Rate-bounded Communication

To control the bandwidth usage of a communication channel I introduce the concept of rate-bounded communication. The idea is to introduce a timed guard on a communication channel in order to limit the production rate of message tokens in the channel to a specified upper bound. This bound is specified by the consumer process. This allows a better prediction of the maximum communication bandwidth demand. A time interval, called *inter-arrival time*, is used to measure and control the communication rate.

The inter-arrival time describes the elapsed time between the arrival of two consecutive message tokens. If the inter-arrival time of a message token is lower than a specified bound, an action is performed to either delay the message token or to discard it. Rate-bounded communication is orthogonal to communication coupling and can be applied on all types of coupling configurations of two interacting processes. However, depending on whether the output of a producer process is decoupled in synchronisation or not, a different method is used to achieve rate-bounded communication.

If the output of the producer process is coupled in synchronisation, i.e. the write access to the channel is blocking, back-pressure is used to achieve rate-bounded communication: The write access not only blocks on write operations if the communication partner is not ready but also if the inter-arrival time of message tokens is lower than a specified bound. Due to the blocking semantics of this rate-control the producing process is not only influenced by the rate of the consuming process but also by the imposed rate-bound on the communication channel. Let $t_{mia}(p_i)$ describe the minimal inter-arrival time specified on an input port $p_i \in \mathcal{P}^I(N)$ of a process $N$ then the maximum rate $r_{max}(p_i)$ of arriving message tokens on port $p_i$ is defined in Equation 4.1.

$$r_{max}(p_i) = \frac{1}{t_{mia}(p_i)} \tag{4.1}$$

I use the notation $\overset{p_i(t_{mia})}{\dashrightarrow}$ to describe a channel $p_i$ bound by the rate $r_{max}(p_i)$ as defined in Equation 4.1.

If, on the other hand, the output port of the producer is decoupled in synchronisation, i.e. the write access to the channel is non-blocking, discarding of message tokens is used to limit the communication rate. Hence, in addition to discarding message tokens if the communication partner is not ready, the corresponding channel also discards message tokens if the communication rate is exceeded. Due to the discarding of message tokens the rate-bound on the communication channel does not impose back-pressure onto the producer process.

In the following I will propose two types of protocols that enforce rate-control without interfering with the producer by discarding message tokens:

1. The *Minimal Inter-release Time (MIRT)* protocol guarantees that the communication rate at a port $p_i$ never exceeds the maximum rate $r_{max}(p_i)$ as defined in Equation 4.1. This is ensured by enforcing a minimal time of $t_{mia}$ between the release of two consecutive message tokens.

2. The *Period-bounded Release Time (PBRT)* protocol allows to release at maximum one message token per period $t_{max}(p_i)$ at a port $p_i$ but does not enforce a minimal

time between two consecutive message tokens. This allows small peak loads where the time between two consecutive message tokens releases is smaller than $t_{mia}$ but ensures that on average the rate of $\frac{1}{t_{mia}(p_i)}$ is not exceeded.

Both types of protocols can either employ a buffer, i.e. decouple the communication in time, in order to delay messages and reduce the number of discarded message tokens or use a more aggressive strategy and discard message tokens straight away if they do not respect the imposed timed guard. A buffered guard must be an active process because it must be able to release a message token at a later time without blocking the producer process. Also, a buffered guard increases the latency of a single message token due to the imposed delay. An unbuffered guard is much simpler because message tokens must not be delayed but only discarded if the minimal inter-arrival time is not respected. This leads to more losses of message tokens but the latency of a single message token is not changed.

In the following subsection I will describe each protocol in more detail and give examples. I use the following notations to identify time values associated to guards:

$t_a(m)$ denotes the arrival time of a message token $m$ at the guard.

$t_r(m)$ denotes the time of the release of a message token $m$ from the guard to its connected FIFO buffer.
If a message token $m$ is discarded $t_r(m) = \infty$.
For the initial message token $m_0$ I define $t_r(m_0) = t_a(m_0)$.

$t_{nr}(m)$ denotes the minimum time to the next release after the arrival of a message token $m$ at the guard.
For the initial message token $m_0$ I define $t_{nr}(m_0) = t_r(m_0) + t_{mia}$.

$t_{mia}$ denotes the minimum inter-arrival time of two consecutive message tokens.

#### 4.2.2.1 Rate-control with the MIRT protocol

The MIRT protocol accepts message tokens only if the minimal inter-arrival time is respected. Once a message token arrives at the guard, all consecutive message tokens arriving within the minimal inter-arrival time interval are discarded. If the minimal inter-arrival time is respected, message tokens are released immediately. Let $(m_{i-1}, m_i)$ be a sequence of message tokens arriving at the guard, then the release time $t_r(m_i)$ of message token $m_i$ is defined in Equation 4.2

$$t_r(m_i) = \begin{cases} t_a(m_i) & \text{if } t_a(m_i) \geq t_{nr}(m_{i-1}) \\ \infty & \text{otherwise} \end{cases} \tag{4.2}$$

and the minimum time of the next release $t_{nr}(m_i)$ after the arrival of message token $m_i$ is defined in Equation 6.1

$$t_{nr}(m_i) = \begin{cases} t_r(m_i) + t_{mia} & \text{if } t_r(m_i) < \infty \\ t_r(m_{i-1}) & \text{otherwise} \end{cases} \tag{4.3}$$

This is illustrated in the example depicted in Figure 4.12a. The arrows on the upper time-line depict the arrival times of the message tokens. Crossed-out arrows represent message tokens that are discarded. The minimal inter-arrival time interval is represented as the lightly coloured area, framed by two dotted lines. The arrows on the lower time-line represent the release times of message tokens. The timer is activated upon releasing a message token as depicted by the dotted lines in Figure 4.12a.

#### 4.2.2.2 Rate-control with the buffered MIRT protocol

The buffered MIRT protocol buffers an arriving message token to delay its progress. Once the specified minimal inter-arrival time is respected, the delayed message token is released. Message tokens are discarded if multiple message tokens arrive within one minimal inter-arrival time interval. Let $(m_{i-1}, m_i, m_{i+1})$ be a sequence of message tokens arriving at the guard, then the release time $t_r(m_i)$ of message token $m_i$ is defined in Equation 4.4

$$t_r(m_i) = \begin{cases} t_a(m_i) & \text{if } t_a(m_i) \geq t_{nr}(m_{i-1}) \\ t_{nr}(m_{i-1}) & \text{if } t_a(m_{i+1}) \geq t_{nr}(m_{i-1}) \\ \infty & \text{otherwise} \end{cases} \tag{4.4}$$

The minimum time of the next release $t_{nr}(m_i)$ after the arrival of message token $m_i$ is equivalent to the unbuffered MIRT protocol as defined in Equation 6.1.

Figure 4.12b represents an example where the same message token arrival times are used as in the previous example but here, a buffered version of the MIRT protocol is used to release the message tokens. As a consequence, fewer message tokens are discarded at the cost of increasing the latency of individual message tokens. If multiple message tokens arrive within a minimal inter-arrival time interval, the newest message token is kept and the older ones are discarded (e.g. in Figure 4.12b the fourth message token replaces the third one and the seventh message token replaces the sixth one). This lies in contrast to the unbuffered MIRT protocol where the newer message tokens are discarded (see Figure 4.12a). This difference is solely due to the presence of the buffer which allows to store the newer message which is not possible in the unbuffered case. As with the unbuffered MIRT protocol, also here a timer is activated upon release of a message token.

#### 4.2.2.3 Rate-control with the PBRT protocol

The PBRT protocol is based on a periodic timer with period $t_{mia}$. The policy is to allow the release of at maximum one message token per period. This ensures that an average rate of $\frac{1}{t_{mia}}$ is not exceeded. However, it does not specify a maximal rate and allows bursts during a short period, i.e. the time between two consecutive message tokens can be smaller that the specified period but the average rate can never be exceeded. Let $(m_{i-1}, m_i)$ be a sequence of message tokens arriving at the guard, then the release time $t_r(m_i)$ of message token $m_i$ is equivalent to the unbuffered MIRT protocol as defined in Equation 4.2. The minimum time of the next release $t_{nr}(m_i)$ after the arrival of message token $m_i$ is defined in Equation 4.5

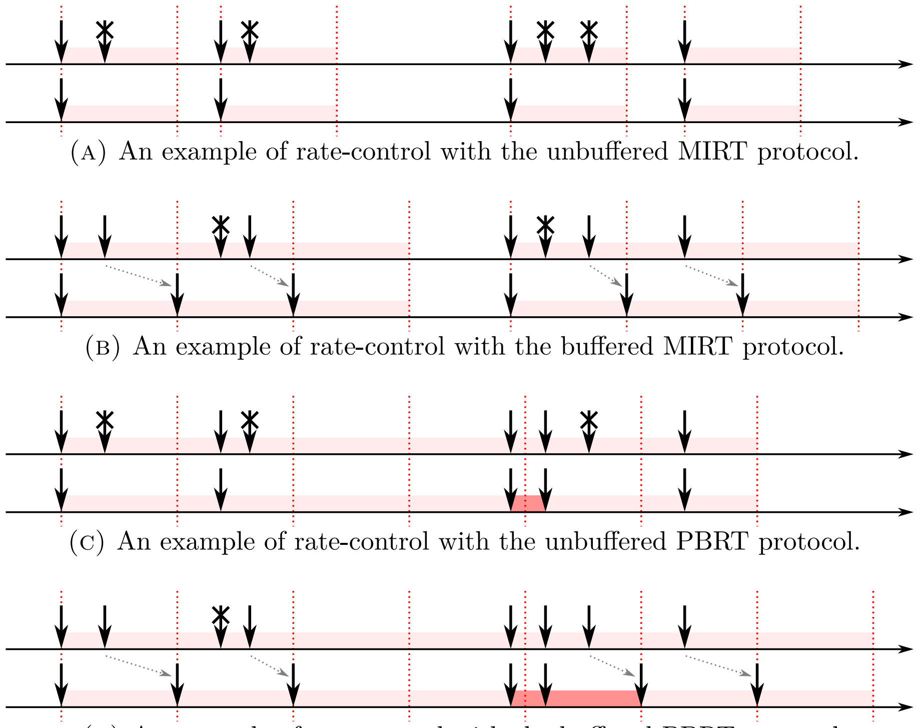
$$t_{nr}(m_i) = (k+1)t_{mia} \text{ with } k \in \mathbb{Z} \ . \ kt_{mia} \leq t_r(m_i) < (k+1)t_{mia} \tag{4.5}$$

Figure 4.12c depicts an example of a sequence of message tokens, released according to the PBRT protocol. Again, the same arrival times of message tokens as in the previous examples are used. In contrast to the MIRT protocol, the timer is activated periodically, as depicted by the dotted lines. During a short time the minimal inter-arrival time is violated to compensate for a short burst of message tokens. This allows to reduce the number of discarded message tokens. The violation of the minimal inter-arrival time is highlighted in Figure 4.12c.

#### 4.2.2.4 Rate-control with the buffered PBRT protocol

The buffered PBRT protocol reduces the number of discarded message tokens by using a buffer to delay message tokens in order to satisfy the rate requirements. As with the unbuffered PBRT protocol, the timer is periodic with period $t_{mia}$. An arriving message token can still be discarded if the buffer is already used to delay a message token that arrived before. In this case the new message token overwrites the old one in the buffer. Let $(m_{i-1}, m_i, m_{i+1})$ be a sequence of message tokens arriving at the guard, then the release time $t_r(m_i)$ of message token $m_i$ is equivalent to the buffered MIRT protocol defined in Equation 4.4. The minimum time of the next release $t_{nr}(m_i)$ after the arrival of message token $m_i$ is equivalent to the unbuffered PBRT protocol as defined in Equation 4.5.

An example of the buffered PBRT protocol, applied on a sequence of message tokens is depicted in Figure 4.12d. Also here, the arrival times of the message tokens are the same as in the previous examples to allow a comparison between the different approaches. With this protocol only one message token is discarded. Again, in contrast to the unbuffered version, the newer message token overwrites the older one and, consequently, the older one is discarded. This comes at the cost of increasing the latency of individual message tokens and allowing a violation of the minimal inter-arrival time. In Figure 4.12d, the interval where a burst of message tokens is accepted is highlighted.


(A) An example of rate-control with the unbuffered MIRT protocol.

(B) An example of rate-control with the buffered MIRT protocol.

(C) An example of rate-control with the unbuffered PBRT protocol.

(D) An example of rate-control with the buffered PBRT protocol.

FIGURE 4.12: Examples of rate-control protocols. In each figure the upper time-line represents the arrival instances of message tokens while the lower time-line represents the release instances of message tokens.

## 4.3 Message Semantics

In PNSCs, communication decoupling in synchronization, as described in Section 4.1.2, can lead to loss or duplication of message tokens. Depending on the content the message token carries, this can be more or less problematic. The level of criticality of the transmitted information is an aspect that dictates whether loss or duplication of the information is problematic. For example, message tokens that carry information to perform a monetary transaction have a high criticality and must not be lost. The loss of a signal of a TV remote control, on the other hand, is inconvenient at most as it can simply be repeated until the transmission is successful.

Apart from the criticality, also the type of the transmitted information plays a crucial role and can alleviate the criticality due to an inherent resilience. Traditionally, one distinguishes between message types where the content can have information of how to update a subsystem (aka event message) or information about the state of a subsystem (aka state message). As Kopetz [59] points out, state messages are well suited for time-triggered systems and event messages for sporadic (event-triggered) systems. However, as Powell [60] points out, it is not necessary that a time-triggered system operates solely with state messages nor is it necessary for a sporadic system to operate with event messages. In the following I will define each message type with respect to message loss and duplication and provide typical application examples for different message types. I further propose a novel message type, called semi-state message, that has some properties from state messages and some from event messages. I define a semi-state message in Definition 4.1, a state message in Definition 4.2, and an event message in Definition 4.3.

**Definition 4.1** (Semi-state Message)**.** *A message in a system is called a* semi-state message *if it enables a correct system state after a previous message of the same type has been lost and the repeated processing of a single message is tolerated by the system.*

**Definition 4.2** (State Message)**.** *A message in a system is called a* state message *if it is a* semi-state message *and its arrival makes the processing of non-processed previous messages of the same type obsolete.*

**Definition 4.3** (Event Message)**.** *Every message in a system that is not a semi-state message (and consequently also not a state message) is an* event message*. In case it is not known whether a message is an event or (semi-) state message, it is safe to assume it to be an* event message*.*

The loss or duplication of a message token is tolerated by a system if the message token is either of type state message or semi-state message. In a state message based communication scheme, the producer observes the value of the state variable at a particular time instant and transmits a state message. An observation is expressed by the triple $< name_{obseravtion}, value, t_{observation} >$. The transmitted information is useful on its own and no knowledge of a previous transmission is required by the consumer. A consumer of state messages is only interested in the newest state information of the sender. Hence, a state message already residing in the communication buffer can be overwritten by a new message without losing any information. Also, messages can be duplicated without problem as a consumer can perform a non-destructive read to keep the information available for possible further reading operations before a new state message arrives (read access is decoupled in space, time, and synchronisation). These properties make state messages useful for fault tolerant systems and hence for safety-critical systems.

In the case of a semi-state message, while the loss or duplication of a message token is tolerated, it is not desirable because the quality of the service might degrade. Hence, a semi-state message has similar properties as a state message but a system benefits from keeping a history of non-consumed message tokens, i.e. a message token is not made obsolete with the arrival of a newer message token.

There are cases when (semi-)state messages can not be used or only at high costs. If, for example, the state variable of the producer is not easily observable or needs to be accessed concurrently (e.g. concurrent bank transactions), an infrastructure must be put in place that allows concurrent access. A message is classified as event message, if the properties of a (semi-)state message, as defined above, are not sufficient to ensure a correct behaviour. In an event message based communication scheme, an event causes the state variable to change its value. The producer then transmits the change from the old to the new state variable that was caused by the event. An event can be expressed by the triple $< name_{event}, \Delta value, t_{event} >$. Generally, it is important that no event message is lost or duplicated and that the order of the messages is preserved. Event message semantics preserves event safety, i.e. prevents losses and duplication of message tokens, and is the default assumption of a communication channel, as defined in Definition 4.4.

**Definition 4.4** (Message Semantics of Communication Channel)**.** *A communication channel that carries only* state messages *has a state-message semantics. A communication channel that carries only* state messages *and* semi-state messages *has a semi-state-message semantics. A communication channel that carries* event messages *has an event-message semantics.*

It is a design decision of whether the message semantics of a communication channel can be explicitly declared to be of (semi-) state message semantics. Having a state message or a semi-state message semantics typically depends on the application context. For example, the stream of video frames of a surveillance camera does have *state semantics* if only life view is required: Only the most recent picture is relevant but it is important that the most recent picture reflects the current situation of the real world. But the same video stream would have *semi-state semantics* if it is provided as an online stream that can be consumed over great distances and lossy channels, i.e. a good trade-off between a minimum acceptable loss of frames and a minimum acceptable transmission delay is required. However, if the video is recorded and a later frame-by-frame analysis is required then the video stream would have *event semantics.*

## 4.4 Cross-criticality Interfaces

When interconnecting processes with different criticality levels it is important to provide suitable interfaces that allow communication without interference from processes with a lower criticality level towards processes with a higher criticality level. Further, such interfaces must allow the coexistence of processes with different timing behaviours and message semantics. I call such interfaces CCI because they allow interaction between processes of different criticality levels.

Throughout this chapter I introduced communication mechanisms that allow to decouple interactions of processes in space, time, and synchronisation and proposed communication channels that allow to control the communication and execution rate of processes. In the following list I provide an overview of the different communication channels introduced throughout this chapter and propose new composed channels where multiple properties are combined. Only combinations are listed that have sensible semantics and a potential use in real applications:

- $\xrightarrow{a}$ depicts a synchronous channel $a$. Decoupling of such a channel is not practical (see Section 4.1.2).
- $\xRightarrow{a[l]}$ depicts a FIFO channel $a$ of length $l$. Such a channel can be decoupled in synchronisation on its input $\succ\!\xRightarrow{a[l]}$ , its output $\overset{a[l]}{\Longrightarrow}\!\!\gg$ , or both $\succ\!\overset{a[l]}{\Longrightarrow}\!\!\gg$ .
- $\overset{a(t)}{\dashrightarrow}$ depicts a synchronous channel $a$ that is bound by a rate defined by $t$. Decoupling of such a channel is not practical (see Section 4.1.2).
- $\overset{a(t)[l]}{=\,=\,=\,\Rightarrow}$ depicts a FIFO channel $a$ of length $l$ that is bound by a rate defined by $t$. Such a channel can be decoupled in synchronisation on its input $\succ\!\overset{a(t)[l]}{=\,=\,=\,\Rightarrow}$ , its output $\overset{a(t)[l]}{=\,=\,=\,\Rightarrow}\!\!\gg$ , or both $\succ\!\overset{a(t)[l]}{=\,=\,=\,\Rightarrow}\!\!\gg$ .
- $\overset{a(clk)}{\rightsquigarrow}$ depicts a channel $a$ with an incorporated temporal firewall that is triggered at a rate $clk$. Such a channel can be decoupled in synchronisation on its input $\succ\!\overset{a(clk)}{\rightsquigarrow}$ , its output $\overset{a(clk)}{\rightsquigarrow}\!\!\gg$ , or both $\succ\!\overset{a(clk)}{\rightsquigarrow}\!\!\gg$ .

Table 4.1 lists the same communication channels and provides an overview of properties of each channel. The following properties are described:

**write** indicates whether the write access to the channel is blocking (block) or non-blocking (¬block), i.e. whether the write access is coupled or decoupled in synchronisation.

**read** indicates whether the read access to the channel is blocking (block) or non-blocking (¬block), i.e. whether the read access is coupled or decoupled in synchronisation.

**lossy** indicates whether message tokens can potentially be lost (yes) or not (no).

**copy** indicates whether message tokens can potentially be duplicated (yes) or not (no).

**rate** describes the communication rate of transmitted message tokens.

**buffer** indicates whether the channel is buffered (yes) or unbuffered (no), i.e. whether the channel is decoupled or coupled in time.

**event** indicates if the channel supports event messages (yes) or if (semi-)state messages are better suited (no).

The table shows that imposing a fixed communication rate with a temporal firewall, as described in Section 4.2.1, is only suitable for (semi-)state messages because loss or duplication of message tokens can happen independently of communication coupling. The decision of whether a read or write access on a temporal firewall should be blocking or non-blocking depends solely on the intended progression control of the process: It might be desirable to block a producer process, connected to a temporal firewall, until the produced message token is consumed by the temporal firewall in order to avoid unnecessary computation efforts of the producer process. On the other hand it might be desirable to allow the producer process to refine its result and overwrite the previous output. The latter requires decoupling in synchronisation to make the write access to the channel non-blocking.

The attributes of *sporadic* communication channels are equivalent to the attributes of channels with a *bounded* communication rate when considering the same coupling configuration (with the exception of the *rate* attribute). This means that bounding the communication rate, as described in Section 4.2.2, only influences the communication rate but does not change the communication coupling behaviour of a channel.

### 4.4.1 Mixed-criticality Network with CCIs

In this section I describe a mixed-criticality video processing application that relies on different communication channels, introduced in this chapter. The basic principle of the application is to record video images with a camera, perform two independent video processing tasks on the video frames, and display the result on a screen. One of the two video processing tasks is of high criticality and must be guaranteed to perform correctly while the other step is of low criticality as it only improves the result but is not absolutely

TABLE 4.1: A list of the properties of communication channels, formed from different combinations of communication patterns.

| | write | read | lossy | copy | rate | buffer | event |
|---|---|---|---|---|---|---|---|
| $\longrightarrow$ | block | block | no | no | sporadic | no | yes |
| $\Longrightarrow$ | block | block | no | no | sporadic | yes | yes |
| $\Longrightarrow\!\!\!\!\!\twoheadrightarrow$ | block | ¬block | no | yes | sporadic | yes | no |
| $\succ\!\!\!\Longrightarrow$ | ¬block | block | yes | no | sporadic | yes | no |
| $\succ\!\!\!\Longrightarrow\!\!\!\!\!\twoheadrightarrow$ | ¬block | ¬block | yes | yes | sporadic | yes | no |
| $\dashrightarrow$ | block | block | no | no | bounded | no | yes |
| $=\,=\,=\,\Rightarrow$ | block | block | no | no | bounded | yes | yes |
| $=\,=\,=\,\twoheadrightarrow$ | block | ¬block | no | yes | bounded | yes | no |
| $\succ=\,=\,=\,\Rightarrow$ | ¬block | block | yes | no | bounded | yes | no |
| $\succ=\,=\,=\,\twoheadrightarrow$ | ¬block | ¬block | yes | yes | bounded | yes | no |
| $=\!\rightsquigarrow$ | block | block | yes | yes | fixed | yes | no |
| $=\!\rightsquigarrow\!\!\twoheadrightarrow$ | block | ¬block | yes | yes | fixed | yes | no |
| $\succ\!\rightsquigarrow$ | ¬block | block | yes | yes | fixed | yes | no |
| $\succ\!\rightsquigarrow\!\!\twoheadrightarrow$ | ¬block | ¬block | yes | yes | fixed | yes | no |

necessary. The challenge of this application is that the lower critical processing task is more complicated and requires more resources than the high criticality task and it must be ensured that the low criticality task does not interfere with the high criticality task.

The example depicted in Figure 4.13 shows a PNSC that models such an image processing application. The process $P_{cam}$ produces message tokens containing video frames and communicates them to the process $P_{copy}$. The process $P_{copy}$ copies the arriving message tokens to each of two connected channels at its output ports. The upper channel connects to the process $P_{filter}$ where the video frames, consumed via its input port, are processed and then sent to the output port. The lower channel connects to the process $P_{fd}$ where a complex feature detection algorithm is executed on the video frames. The process $P_{merge}$ consumes message tokens from the processes $P_{filter}$ and $P_{fd}$ and merges the two video frames into one. Message tokens with the merged video frames are then transmitted to the process $P_{screen}$ where the video is displayed.

All communication channels are buffered to allow a variation in the production rate of the processes. The channel *cam*, connecting process $P_{cam}$ with process $P_{copy}$, is bounded to a maximum rate of $\frac{1}{24}$ which is sufficient for this particular video processing application. The process $P_{filter}$ is of high criticality (the box with the thick border in Figure 4.13) and process $P_{fd}$ is of low criticality, hence, it must be ensured that $P_{fd}$ is not interfering with $P_{filter}$. To achieve this, first, the output $f_2$ of process $P_{copy}$ is

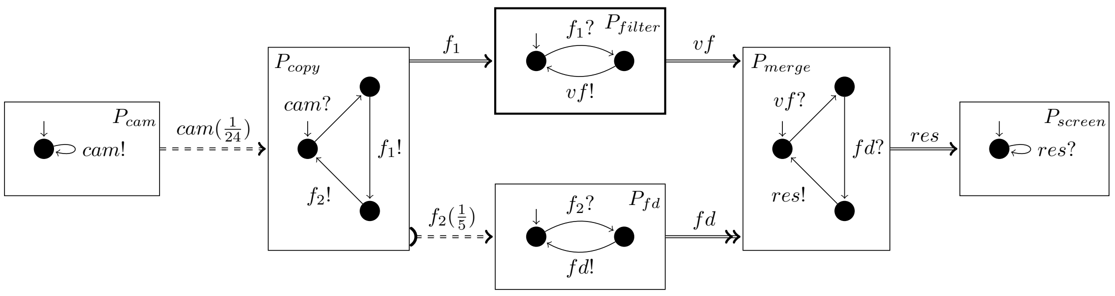


FIGURE 4.13: An example of a mixed-criticality video processing application where ports are decoupled in synchronisation to prevent interference from the low critical process $P_{fd}$ towards the high criticality process $P_{filter}$.

decoupled in synchronisation to prevent back-pressure from the low-criticality process $P_{fd}$. Consequently, process $P_{copy}$ cannot be blocked by process $P_{fd}$ which prevents indirect interference from process $P_{fd}$ towards process $P_{filter}$. Second, the input $fd$ of process $P_{merge}$ is decoupled in synchronisation to prevent a blocking read access from process $P_{merge}$ to this channel which prevents any indirect interference from process $P_{fd}$ towards process $P_{filter}$ via process $P_{merge}$.

Additionally, the channel $f_2$ is bounded to a maximum rate of $\frac{1}{5}$ to prevent the execution of process $P_{fd}$ at a faster rate than is useful for this particular application.

## 4.5 Discussion

A main contribution of this chapter is the modular approach to punctually remove communication coupling to prevent interference. It is especially useful that this is achieved without changing the specification of a process but solely by changing the communication channels the process interacts with. This modularity allows to reuse a process in different contexts without any need for the process to be aware of the context it is placed in.

For future work, it might be interesting to investigate if in a network of processes, channel types can be chosen automatically to prevent interference where necessary, depending on the criticality level of individual processes.

With respect to time-triggered PNSCs, as temporal firewalls can be connected independently to processes, it is possible to enforce a time-triggered semantics on individual processes in a PNSC or on a PNSC as a whole. In the former case each message token in the PNSC is transmitted according to the time-triggered communication semantics. In the latter case, processes inside the time-triggered PNSC communicate sporadically whereas a process interacting with the environment of the PNSC follows a time-triggered communication scheme. Such time-triggered PNSCs can be useful when designing an application for a many-core architecture, e.g. 256 cores, where several clusters of cores, e.g. 16 clusters, communicate via shared memory (use a sporadic communication scheme) while the clusters are interlinked by a Network on Chip (NoC) (use the time-triggered communication scheme)[1]. De Dinechin et al. propose a Run-time System (RTS) for such an architecture that supports i.a. a time-triggered and a Kahn Process Network (KPN) execution model [61].

In Section 4.2.1 I mentioned that a time-triggered PNSC must satisfy Property 4.1 and Property 4.2 in order to be valid. It is rather complex to provide sufficient requirements to guarantee these properties and an extensive investigation of this topic is required. The bulk of this work is postponed to future work. I will, however, give an overview of the work that needs to be done to achieve this goal: Both properties rely on the requirement that within one round each process of the time-triggered PNSC has completed its execution. This can only be guaranteed if the WCET of the PNSC is known and is guaranteed to be smaller than the specified time to complete one round. The WCET of the PNSC depends on the network topology of the PNSC and the WCET of each process in the PNSC. Further, the WCET of a process depends on the decisions made within the process and how it interacts with its environment. The SIA of a PNSC process describes the interaction protocol with its environment. By extending a SIA with WCET annotations on each action and using the network description provided by the PNSC model, it should be possible to check whether a time-triggered PNSC satisfies Property 4.1 and Property 4.2.

## 4.6 Chapter Summary

In this chapter I extended the PNSC model with communication channels that allow to change the communication coupling and triggering semantics of connected processes. This enables the PNSC model to describe mixed-criticality systems where processes with different criticality levels can interact without interference. While the example in Figure 4.13 describes a system with two criticality levels, the model is not limited to two criticality levels. The decoupling element is used to prevent interference between any two criticality levels and, thus, can be applied on multiple levels.

I further extended the PNSC model with two rate-controlled channels that allow to model time-critical systems: First, a fixed-rate communication channel, called a temporal firewall, can be used to enforce time-triggered semantics of a process. Second, a

[1]The Kalray MPPA®-256 (Andey) is an example of such a processor.

rate-bound communication channel serves to limit the communication rate to an upper bound. This increases the predictability of the required bandwidth and allows to distribute the available bandwidth where it is needed.

Finally, I proposed CCIs where combinations of these concepts are put together and I discussed the properties of the CCIs, in particular with respect to different message semantics. In this context I introduced a novel message type, namely *semi-state* messages which which has some properties of state messages and some properties of event messages.

# Chapter 5

# Permanent Blocking Analysis of PNSCs with SIAs

The interaction of concurrent systems in general is fragile in the sense that unwanted blocking can occur. Especially in the domain of critical systems it is important to ensure that a system is free of permanent blocking. In Chapter 4 I introduced a mechanism that allows to punctually remove communication coupling in synchronisation. In a system where all coupling in synchronisation is removed, no possibility of permanent blocking can occur. An example of this is a system which follows the Time-triggered Architecture (TTA) [47] design principles. However, as outlined in the previous chapters, in order to allow event-triggered systems (i.e. systems with communication coupling) and time-triggered systems to coexist or interact with each other, it must be possible to check that the processes, relying on an event-triggered communication scheme, are free of permanent blocking situations.

In this chapter, I present an analysis to detect permanent blocking in systems of concurrent processes where the interaction is based on synchronous communication, i.e. communication with synchronisation coupling. Based on the semantics of Synchronous Interface Automata (SIAs) I define the meaning of permanent blocking between two or more concurrent processes. I categorise two possible types of permanent blocking as either *lonely blocking* or *deadlocks* and present an analysis that is able to identify both types of permanent blocking between two or more concurrent processes.

The chapter is structured as follows: In Section 5.1 I identify permanent blocking situations in SIAs and propose an analysis to detect permanent blocking states in Section 5.1.1. In Section 5.1.2 I discuss how a subtype of permanent blocking, namely deadlocks, can be identified and propose an analysis to achieve this. For reasons of simplicity and readability I first focus on permanent blocking and deadlock situations in systems of two concurrent processes. Later, the findings are extended to hold for the general case. In Section 5.2 I propose an algorithm that allows to compute permanent blocking states and an algorithm to distinguish between permanent blocking and deadlock situations. Section 5.3 summarises the chapter.

## 5.1 Permanent Blocking of SIAs

In this section I will discuss how a process $M$ can become permanently blocked in a Process Network with Synchronous Communication (PNSC) $MN$ by analysing the SIA $\widetilde{M}$ of process $M$ in the context of SIA $\widetilde{MN}$ of PNSC $MN$. I call a PNSC $MN$ the *context* of a process $M$ if Definition 5.1 holds and a SIA $\widetilde{MN}$ the *context* of a SIA $\widetilde{N}$ if Definition 5.2 holds.

**Definition 5.1** (Context of a process)**.** *Let $S_{MN}$ be the set of all processes in a PNSC $MN$. Then $MN$ is the context of a process $M$ iff $M \in S_{MN}$.*

**Definition 5.2** (Context of a SIA)**.** *SIA $\widetilde{MN}$ is the context of SIA $\widetilde{M}$ iff the PNSC $MN$ is the context of process $M$ where $\widetilde{MN}$ is the SIA of $MN$ and $\widetilde{M}$ is the SIA of $M$.*

In order to talk about permanent blocking of a PNSC we need to establish an understanding of what permanent blocking means. As defined in Equation 2.1, we use the term *permanent blocking* as the opposite of *liveness*. Liveness of a PNSC is defined by Definition 5.3, 5.4, 5.5, and 5.6.

**Definition 5.3** (Liveness of a SIA state)**.** *A SIA $\widetilde{M}$ with context $\widetilde{MN}$ is alive in a state $s \in S_{\widetilde{MN}}$ with $s = \langle s_m, s_n \rangle$ and $s_m \in S_{\widetilde{M}}$, if either*

*a) $s_m$ is an end state of $\widetilde{M}$ as defined in Equation 3.5, or*

*b) progress is possible from $s$ into a state $s' = \langle s'_m, s'_n \rangle$ with $s'_m \in S_{\widetilde{M}}$ and $s_m \neq s'_m$.*

**Definition 5.4** (Liveness of a SIA)**.** *A SIA $\widetilde{M}$ is alive in context $\widetilde{MN}$, iff it is alive in all reachable states $s \in S_{\widetilde{MN}}$.*

**Definition 5.5** (Liveness of a PNSC process)**.** *A process $N$ is alive in the context $MN$, iff $\widetilde{M}$ is alive in context $\widetilde{MN}$ where $\widetilde{M}$ is the SIA of $M$ and $\widetilde{MN}$ is the SIA of $MN$.*

**Definition 5.6** (Liveness of a PNSC)**.** *A PNSC is alive iff all of its processes are alive.*

### 5.1.1 Permanent Blocking Analysis

To perform a permanent blocking analysis I use the composition operation $\otimes$, as defined in Section 3.2.2, and provide a formal definition of permanent blocking states in a composed system. In this dissertation I distinguish between cyclic and acyclic SIAs. An acyclic SIA can be represented by a Directed Acyclic Graph (DAG) whereas a cyclic SIA cannot. I first start with a simple analysis where I only consider acyclic SIAs. In a second step I extend the analysis by also taking cyclic SIAs into account.

#### 5.1.1.1 Permanent Blocking Analysis with Acyclic SIAs

In order to verify the liveness conditions of a SIA $\widetilde{M}$ in its context $\widetilde{MN}$, as defined in Definition 5.3, I can exploit a property of acyclic SIAs: In an acyclic subsystem $\widetilde{M}$ each state with no enabled action is an end state by definition. As a consequence, when composing two such subsystems with the composition operator $\otimes$, as defined in Definition 3.4, all states of the composed SIA where no actions are enabled must be end states in each subsystem. It follows that in order to check for possible progress (Definition 5.3.b)) of a subsystem $\widetilde{M}$ in a state $s \in S_{\widetilde{MN}}$ of its context, it is sufficient to check whether state $s$ has enabled actions. This holds under the assumption that all subsystems in context $\widetilde{MN}$ are acyclic. Finally, I conclude that a state $s \in S_{\widetilde{MN}}$ is permanent blocking if no actions are enabled in state $s$ and at least one of the two subsystems is not in an end state.

The above conclusion only holds if all involved SIAs have an end state, which is not necessarily the case for SIAs with cycles. Hence, I give a formal definition in Equation 5.1 of the set of permanent blocking states $S_{\widetilde{MN}}^{block-ac}$ of a PNSC with two processes $M$ and $N$, assuming both SIAs $\widetilde{M}$ and $\widetilde{N}$ are described by acyclic graphs (this is denoted by the *-ac* postfix). The set of end states of a SIA is defined in Equation 3.5.

$$S_{\widetilde{MN}}^{block-ac} = \left\{ \langle s_m, s_n \rangle \in S_{\widetilde{MN}} \mid \left( s_m \notin S_{\widetilde{M}}^{end} \vee s_n \notin S_{\widetilde{N}}^{end} \right) \wedge \langle s_m, s_n \rangle \in S_{\widetilde{MN}}^{end} \right\} \tag{5.1}$$

The SIA $\widetilde{MN}_1$, as depicted in Figure 3.7, is an example of the result of the composition of two acyclic subsystems $\widetilde{M}_2$ and $\widetilde{N}_4$ (see Figure 3.6). The composed SIA $\widetilde{MN}_1$ has three states with no enabled actions, namely the states $s_{16}$, $s_{24}$, and $s_{25}$. The question is whether these states are permanent blocking states or not. Given that all three states have no enabled actions, it holds that no progress for either subsystem is possible in these states, i.e. the liveness condition Definition 5.3.b) is not satisfied. Further, I have to check whether the states of both subsystems are end states (Definition 5.3.a)). In state $s_{16} \in S_{\widetilde{MN}_1}$ both states in the subsystems, $s_1 \in S_{\widetilde{M}_2}$ and $s_6 \in S_{\widetilde{N}_4}$, are end states as defined in Equation 3.5, hence, $s_{16}$ is not a permanent blocking state. States $s_{24}$ and $s_{25}$, however, are permanent blocking states because, while state $s_2 \in S_{\widetilde{M}_2}$ is an end state, Equation 3.5 does not hold for states $s_4 \in S_{\widetilde{N}_4}$ and $s_5 \in S_{\widetilde{N}_4}$. The two states $s_{24}$ and $s_{25}$ represent the situation of the PNSC of Figure 3.6 where process $M_2$ has concluded its processing (in state $s_2 \in S_{\widetilde{M}_2}$) while process $N_4$ is blocked (either in state $s_4 \in S_{\widetilde{N}_4}$ because the shared action $a$ is never served or in state $s_5 \in S_{\widetilde{N}_4}$ because action $b$ is ignored).

The rationale behind end states is that a process is either in a state where its job is accepted as accomplished or it still has to perform actions in order to accomplish its job and reach an end state. A process that performs periodic work, which would be represented by a cyclic SIA, will never reach an end state. If processes with cyclic SIAs are composed, it can happen that only one process is performing actions indefinitely (in a cycle) while the other processes are blocked. There are two different cases where a permanent blocking analysis based on the definition of Equation 5.1 falls short for systems with cyclic SIAs. I illustrate this with the help of the PNSC depicted in Figure 5.1.

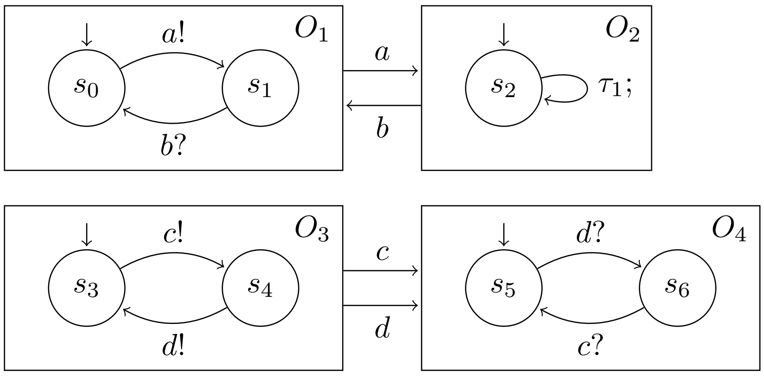


FIGURE 5.1: A PNSC formed of four processes, each described by a SIA with a cycle.

The first case is the problem of detecting a lonely blocker when cycles are involved. This becomes apparent when observing the composed SIA $\widetilde{O}_{12} = \widetilde{O}_1 \otimes \widetilde{O}_2$: With the actions $\{a, b\} \in \mathcal{A}_{\widetilde{O}_1}$ ignored, the composed SIA $\widetilde{O}_{12}$ is formed out of the singe initial state $s_{02}$ with a self looping edge, triggered by the internal action $\tau_1 \in \mathcal{A}^H_{\widetilde{O}_2}$. In this case the set of permanent blocking states defined in Equation 5.1 is empty because the composed SIA $\widetilde{O}_{12}$ has no end state. But clearly, SIA $O_1$ is permanently blocking in state $s_0$ because the actions $\{a, b\} \in \mathcal{A}_{\widetilde{O}_1}$ are ignored.

The second case is the problem of missing local permanent blocking situations by incrementally folding subsystems of a PNSC, depending on the order in which processes are composed. In the example of Figure 5.1, the SIAs $\widetilde{O}_3$ and $\widetilde{O}_4$ are both in a permanent blocking state $s_3 \in S_{\widetilde{O}_3}$ and $s_5 \in S_{\widetilde{O}_5}$, respectively. This is detectable when performing the composition operation $\widetilde{O}_{34} = \widetilde{O}_3 \otimes \widetilde{O}_4$ and computing the set of permanently blocking states as defined in Equation 5.1. The composition yields the single initial state $s_{35}$ with no transitions. Hence, the set of permanent blocking states is defined as $S^{block-ac}_{\widetilde{O}_{34}} = \{s_{35}\}$, because neither $s_3 \in S_{\widetilde{O}_3}$ nor $s_5 \in S_{\widetilde{O}_4}$ is an end state. However, when folding the system incrementally, e.g. $\widetilde{O}_{1234} = ((\widetilde{O}_1 \otimes \widetilde{O}_2) \otimes \widetilde{O}_3) \otimes \widetilde{O}_4$, the composed SIA $\widetilde{O}_{1234}$ has no longer an end state and hence, $S^{block-ac}_{\widetilde{O}_{1234}} = \emptyset$.

#### 5.1.1.2 Permanent Blocking Analysis with Cyclic SIAs

In the following I extend the previous analysis to work also for cyclic SIAs. As defined in Definition 5.3, in order to decide whether a SIA $\widetilde{M}$ with context $\widetilde{MN}$ is alive in state $\langle s_m, s_n \rangle \in S_{\widetilde{MN}}$, I need to detect whether 5.3.a) $s_m \in S_{\widetilde{M}}$ is an end state of $\widetilde{M}$ or 5.3.b) progress from state $\langle s_m, s_n \rangle$ to a state $\langle s'_m, s'_n \rangle$ is possible such that $s_m \neq s'_m$. I introduce the predicate $progress(sys, s)$, returning true if the latter is true for a subsystem $sys$. I use this predicate to define the set $Sys_{\widetilde{MN}}(s) \subseteq \{M, N\}$ in Equation 5.2 which contains all the subsystems where progress is possible in state $s$ of SIA $\widetilde{MN}$.

$$Sys_{\widetilde{MN}}(\langle s_m, s_n \rangle) = \Big\{ sys \in \{M, N\} \mid progress(sys, \langle s_m, s_n \rangle) \Big\} \tag{5.2}$$

With the set $Sys_{\widetilde{MN}}(s)$ as defined in Equation 5.2 and the set of end states as defined in Equation 3.5, in Equation 5.3 I define $Sys^{block}_{\widetilde{MN}}(\langle s_m, s_n \rangle)$, the set of subsystem identifiers that are permanently blocking in state $\langle s_m, s_n \rangle$ of the composed SIA $\widetilde{MN}$.

$$\begin{aligned} Sys^{block}_{\widetilde{MN}}(\langle s_m, s_n \rangle) = \Big\{ M \mid s_m \notin S^{end}_{\widetilde{M}} \wedge M \notin Sys_{\widetilde{MN}}(\langle s_m, s_n \rangle) \Big\} & \\ \cup \ \Big\{ N \mid s_n \notin S^{end}_{\widetilde{N}} \wedge N \notin Sys_{\widetilde{MN}}(\langle s_m, s_n \rangle) \Big\} & \end{aligned} \tag{5.3}$$

Using the set $Sys^{block}_{\widetilde{MN}}(\langle s_m, s_n \rangle)$, defined in Equation 5.3, I define the set of permanent blocking states $S^{block}_{\widetilde{MN}}$ of the composed SIA $\widetilde{MN}$ in Equation 5.4.

$$S^{block}_{\widetilde{MN}} = \Big\{ \langle s_m, s_n \rangle \in S_{\widetilde{MN}} \mid \exists sys \in \{M, N\} \ . \ sys \in Sys^{block}_{\widetilde{MN}}(\langle s_m, s_n \rangle) \Big\} \tag{5.4}$$

Informally, this means that a state $\langle s_n, s_m \rangle \in S_{\widetilde{MN}}$ is permanently blocking if at least one subsystem is not in an end state and can perform no more actions.

Following Definition 5.4, Definition 5.5, and Definition 5.6 I can conclude that a PNSC composed of the processes $M$ and $N$ is alive if and only if $S^{block}_{\widetilde{MN}} = \emptyset$ with the set $S^{block}_{\widetilde{MN}}$ defined in Equation 5.4.

#### 5.1.1.3 Permanent Blocking Analysis on an Assembly of Processes

In this section I extend the permanent blocking analysis from two subsystems to an arbitrary number of subsystems. By incrementally applying the folding operation $\otimes$, the initial PNSC dependency graph is lost because shared actions are turned to internal actions in the SIA of the composed process (see Definition 3.2 and 3.4). From a software engineering point of view this might be desirable because it allows to structure big systems hierarchically and allows to reuse code as black boxes where only the interface (specified by a SIA) is known. Once a system is composed and guaranteed to be free of permanent blocking, the internal actions can be removed if they do not influence the blocking behaviour of a system. This can serve to drastically reduce the number of states of the composed system and reduce the behavioural description to the interaction protocol of the composed process. A technique to achieve this is proposed by Pace et al. in [62] but will not be further discussed in this dissertation.

In order to apply the permanent blocking analysis, described by Equation 5.4, to an assembly of incrementally composed processes, two pieces of information must be propagated throughout the process of incrementally folding subsystems with the operator $\otimes$: First, the dependency graph of the PNSC is required in order to retain the dependencies of the subsystems despite the composition. This preserves the information of which ports are shared by which processes. Second, a state $s$, of a SIA composed out of $n$ subsystems, has to be described by the tuple $\langle s_1, \dots, s_n \rangle$ to analyse the permanent blocking state of the initial subsystem and not the incrementally built composition. These state tuples are extended at each step of the incremental composition by a corresponding state of the subsystem that is composed with the system.

With the propagation of this information, the permanent blocking analysis can be expanded to $n$ subsystems. Hence, there is no necessity to perform the analysis at each step of the incremental folding operation and can now be applied on the resulting composed system, described by the SIA $\widetilde{J} = \widetilde{N}_1 \otimes \widetilde{N}_2 \otimes \cdots \otimes \widetilde{N}_n$. Specifically, the set of system identifiers describing subsystems where progress is possible, defined in Equation 5.2, is extended to $n$ subsystems in Equation 5.5.

$$Sys_{\widetilde{J}}(\langle s_1, \dots, s_n \rangle) = \Big\{ sys \in \{N_1, \dots, N_n\} \mid progress(sys, \langle s_1, \dots, s_n \rangle) \Big\} \tag{5.5}$$

Similarly, the set of system identifiers describing subsystems that are permanently blocking at a particular state of the composed system, defined in Equation 5.3, is expanded to $n$ subsystems in Equation 5.6.

$$\begin{aligned} Sys^{block}_{\widetilde{J}}(\langle s_1, \dots, s_n \rangle) = \Big\{ & N_i \mid 1 \leq i \leq n \\ & \wedge \; s_i \notin S^{end}_{\widetilde{N}_i} \; \wedge \; N_i \notin Sys_{\widetilde{J}}(\langle s_1, \dots, s_n \rangle) \Big\} \end{aligned} \tag{5.6}$$

Finally, the set of permanent blocking states defined in Equation 5.4 is expanded to $n$ subsystems in Equation 5.7.

$$S^{block}_{\widetilde{J}} = \Big\{ \langle s_1, \dots, s_n \rangle \in S_{\widetilde{J}} \mid \exists i \in \{1, \dots, n\} \; . \; N_i \in Sys^{block}_{\widetilde{J}}(\langle s_1, \dots, s_n \rangle) \Big\} \tag{5.7}$$

The price to pay for propagating state information of subsystems is threefold: First, the propagation of state information requires an increased memory space as the list of involved subsystems in the state tuple $\langle s_1, \dots, s_n \rangle$ constantly grows. Second, because permanent blocking situations are not detected when they occur, permanent blocking states are propagated and consequently multiplied. This leads to an ever growing list of detected permanent blocking situations that may only point to a single cause. Third, because state information of the subsystems needs to be preserved, it is not directly possible to reduce the state space by reducing internal actions. However, it is only necessary to preserve the set of subsystem identifiers $Sys(e)$, attributed to each edge representing an internal action, because internal actions cannot cause deadlocks (they are the result of a non-permanent blocking communication). For example, it should be possible to adopt the reduction technique by Pace et al. [62] to reduce the data structure used for the analysis.

### 5.1.2 Deadlock Analysis

In the previous section I established a method to analyse the blocking behaviour of synchronous communication. In this section I discuss how to distinguish between a *deadlock* of multiple subsystems and *lonely blocking* of a single subsystem.

The crossroad example discussed in Section 3.3 is a typical example of a deadlock situation. To give an example of a lonely blocker, let's consider the PNSC as depicted in Figure 5.2. The two processes $M_2$ and $N_2$ share the ports $a$ and $b$, and process $M_2$ has a port $c$ that is not (yet) connected to any other process. Their corresponding SIAs $\widetilde{M}_2$ and $\widetilde{N}_2$ share the action $a$, action $b \in \mathcal{A}^O_{\widetilde{N}_2}$ is ignored, and action $c \in \mathcal{A}^I_{\widetilde{M}_2}$ is open.

Initially, the two SIAs start in their respective initial state $s_0$ and $s_3$. Transitions, labelled by open actions, are assumed to always be triggered by the (unknown) environment. Hence, SIA $\widetilde{M}_2$ performs the transition $\langle s_0, c, s_2 \rangle$. State $s_2$ is, according to

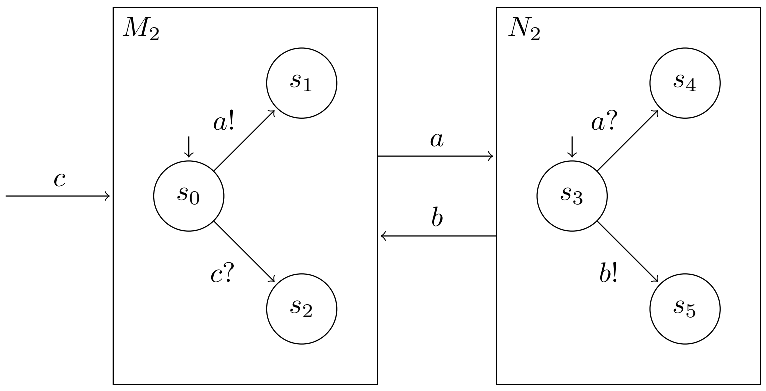


FIGURE 5.2: An example of a PNSC where the process $M_2$ is alive and process $N_2$ is lonely blocking in state $s_3 \in S_{\widetilde{N_2}}$.

Equation 3.5, an end state where SIA $\widetilde{M_2}$ can reside indefinitely. However, due to the fact that SIA $\widetilde{M_2}$ has reached an end state and finished its processing, SIA $\widetilde{N_2}$ will block and wait to synchronize on action $a$ in state $s_3$ indefinitely. State $s_3$ is not an end state and therefore a permanent blocking state which makes process $N_2$ potentially permanent blocking while process $M_2$ is alive (no permanent blocking states in its corresponding SIA $\widetilde{M_2}$). Process $N_2$ is a lonely blocker.

In Definition 2.1 I defined the four conditions that must hold simultaneously for a deadlock to occur. Condition 1 (mutual exclusion) is implicitly satisfied by the produce and consume semantics of the process model. Condition 2 (no pre-emption) must be satisfied by the implementation of the model. For condition 3 and 4 I have to consider the semantics of synchronous communication as it is defined by the model. Due to the symmetric semantics of synchronous communication, two processes have to be ready simultaneously in order to communicate a message from one to another. This can be understood as follows: The receiving process makes a typed buffer available for the sending process to write to. The sender can only write to the buffer if the type of the message matches. The resources in such a system are the buffers and the message tokens. If a particular buffer is not available, it is held by the receiver. If a message token is not currently available for transmission, it is held by the sender.

Applied on the PNSC model I observe that the held resources are defined by the signature of a processes, i.e. the set of ports. More precisely, the set of output ports $\mathcal{P}_N^O$ correspond to the typed message tokens held by process $N$ and the set of input ports $\mathcal{P}_N^I$ correspond to the typed buffers held by process $N$. The type of a buffer or a message token is defined by the name of the corresponding port. The SIA of a process in a PNSC, on the other hand, defines the required resources by a process. The resource a process is waiting on depends on the state, the SIA of the process is currently in. An input action corresponds to a request for a typed message token and an output action corresponds to a request for a typed buffer.

Formally, I define the predicate $\mathcal{H}(N)$ in Equation 5.8. It is the set of resources *held* by process $N$ (and its corresponding SIA $\widetilde{N}$).

$$\mathcal{H}(N) = \mathcal{P}^O_N \cup \mathcal{P}^I_N \tag{5.8}$$

Note that with synchronous communication the set of resources that are held by a process is static and is not depending on the state of the process as it would be the case with asynchronous communication. Further, I define the predicate $\mathcal{W}(\widetilde{N}, s)$ in Equation 5.9. It is the set of resources SIA $\widetilde{N}$ (of process $N$) is *waiting* for in state $s$.

$$\begin{aligned}\mathcal{W}(\widetilde{N}, s) = \{a_{buffer} \mid \forall d \in \delta_{\widetilde{N}} \;.\; d = \langle s, a, \_\rangle \;\wedge\; a \in \mathcal{A}^O_{\widetilde{N}}\} \\ \cup \{a_{token} \mid \forall d \in \delta_{\widetilde{N}} \;.\; d = \langle s, a, \_\rangle \;\wedge\; a \in \mathcal{A}^I_{\widetilde{N}}\}\rangle\end{aligned} \tag{5.9}$$

For deadlock condition 3 (hold and wait) to hold in a particular state $s \in S_{\widetilde{N}}$ of SIA $\widetilde{N}$ an input or output action must be enabled (an action is enabled if condition defined in Equation 3.4 holds). Hence, the following condition must be satisfied:

$$\forall s' \in S_{\widetilde{N}}, \exists a \in \mathcal{A}^O_{\widetilde{N}} \cup \mathcal{A}^I_{\widetilde{N}} \;.\; \langle s, a, s'\rangle \in \delta_{\widetilde{N}} \tag{5.10}$$

In order to check whether deadlock condition 4 (circular wait) is holding, I need to check the wait dependencies of subsystems in permanent blocking states of the composed SIA. As the folding operation $\otimes$ is applied incrementally, in an arbitrary order, on each SIA representing a process in a PNSC, I always consider two processes at a time: The composed process that was constructed out of a subset of the processes of the initial PNSC by the incremental folding operation and another process that is not yet incorporated in the composed process but is part of the PNSC. To detect a circular wait of two processes $M$ and $N$ with corresponding SIAs $\widetilde{M}$ and $\widetilde{N}$ I consider all the permanent blocking states of the composed SIA $\widetilde{MN}$ and check whether both subsystems are blocked in their corresponding state. A permanent blocking state $\langle s_m, s_n\rangle \in S_{\widetilde{MN}}$ of SIA $\widetilde{MN}$ is a deadlock state if the predicate $dl_2(M, N, \langle s_m, s_n\rangle)$, as defined in Equation 5.11, returns true. The hold $\mathcal{H}$ and wait $\mathcal{W}$ predicates are defined in Equation 5.8 and Equation 5.9, respectively.

$$dl_2(M, N, \langle s_m, s_n\rangle) = (\mathcal{H}(M) \cap \mathcal{W}(\widetilde{N}, s_n)) \neq \emptyset \;\wedge\; (\mathcal{H}(N) \cap \mathcal{W}(\widetilde{M}, s_m)) \neq \emptyset \tag{5.11}$$

A circular wait is not necessarily limited to two participants. More than two processes could be involved in a deadlock. However, the information of other involved processes that are incorporated in the composed process, is lost due to the incremental folding operation. In the next section I address this issue.

#### 5.1.2.1 Deadlock Analysis on an Assembly of Processes

As mentioned before, the deadlock condition 4 (circular wait) may involve more than two subsystems. The crossroad example described by Figure 3.8 is such a case. By incrementally folding the processes together, the presented analysis will detect a deadlock at the last step, when folding the system $\widetilde{P}_{NW} \otimes \widetilde{P}_{NE} \otimes \widetilde{P}_{SE}$ with the system $\widetilde{P}_{SW}$. In reality however, the deadlock is caused by the four individual systems being in a circular wait.

With Equation 5.11 I defined the deadlock condition 4 (circular wait) for two subsystems $M$ and $N$ in state $\langle s_m, s_n \rangle$. With the predicate $dl$, as defined in Equation 5.12, this is expanded to $n$ subsystems. The ordered set $(N_1, \dots, N_n)$ describes a dependency from subsystem $N_i$ to $N_{i+1}$ and from $N_n$ to $N_1$ to complete the circle. The state of the composed system is described by the tuple $\langle s_1, \dots, s_n \rangle$, with $s_i$ being a state from the SIA $\widetilde{N}_i$ of subsystem $N_i$.

$$\begin{aligned} dl((N_1, \dots, N_n), \langle s_1, \dots, s_n \rangle) = {} & (\mathcal{H}(N_n) \cap \mathcal{W}(\widetilde{N}_1, s_1)) \neq \emptyset \\ & \wedge \Big( \forall N_i | i \in \{1, \dots, n{-}1\} \; . \; (\mathcal{H}(N_i) \cap \mathcal{W}(\widetilde{N}_{i+1}, s_{i+1})) \neq \emptyset \Big) \end{aligned} \tag{5.12}$$

By applying the permanent blocking and deadlock analysis, described in Section 5.1.1.3 and this section, respectively, on the example of Figure 5.1 in Section 5.1 I am now able to identify all cases of permanent blocking and can distinguish between lonely blocking and deadlocks: Using Equation 5.5, Equation 5.6, and Equation 5.7, I compute the set of permanent blocking states $S^{block}_{\widetilde{O}_{1234}} = \{s_{0235}\}$ and the set of subsystem identifiers $Sys^{block}_{\widetilde{O}_{1234}}(s_{0235}) = \{O_1, O_3, O_4\}$, indicating the permanent blocking subsystems at state $s_{0235}$. Using the initial dependency graph, $Sys^{block}_{\widetilde{O}_{1234}}(s_{0235})$, and Equation 5.12 I can identify process $O_1$ as lonely blocker, permanently blocking in state $s_0 \in S_{\widetilde{O}_1}$ on action $a \in \mathcal{A}_{\widetilde{O}_1}$. Further, I find that processes $O_3$ and $O_4$ are in a deadlocking situation, permanently blocking in state $s_3 \in S_{\widetilde{O}_3}$ on action $c \in \mathcal{A}_{\widetilde{O}_3}$ and state $s_5 \in S_{\widetilde{O}_4}$ on action $d \in \mathcal{A}_{\widetilde{O}_4}$, respectively.

## 5.2 Implementation of the Permanent Blocking Analysis

In this section I describe in detail how two aspects of the permanent blocking analysis can be implemented: First, the computation of the set $Sys_{\widetilde{MN}}(s)$ as defined in Equation 5.5

will be discussed in Section 5.2.1. Second, the algorithm to distinguish between lonely blockers and deadlock situation is addressed in Section 5.2.2.

### 5.2.1 Algorithm to Compute $Sys(s)$

While it is fairly simple to compute the set of permanent blocking states of a composed system (as defined in Equation 5.4) it is not so trivial to compute the set of subsystems from which progress is possible in a particular state of the composed system (as defined in Equation 5.2). This is because it is not enough to observe the transitions from one state in order to decide if a subsystem is progressing or not. A path of successor states and transitions needs to be considered in order to take into account that a subsystem can be temporarily blocked in one state but will still be able to progress in a reachable successor state. In the following I am going to describe an algorithm that allows to compute the set $Sys(s)$, as defined in Equation 5.2, at each state $s$ of a composed SIA in linear time complexity.

A straightforward approach to compute $Sys(s)$ is to follow all paths in the composed SIA and count the involved subsystems along the paths. However, in order to follow all distinct paths in a graph one has to identify all simple cycles in the graph, a problem that is currently known to be solvable in bounded time by $\mathcal{O}((V+E)(C+1))$ [63] where $V$ is the number of vertices, $E$ the number of edges, and $C$ the number of cycles in the graph.

Fortunately, I am not explicitly interested in the enumeration of all the paths themselves, but only whether for a state $s$ there exists a path starting with $s$ that includes actions of all subsystems. This test can be simplified by exploiting the properties of *strongly connected* graphs, i.e. graphs where every vertex is reachable from every other vertex in the graph.

For example, if I have a graph component $X$ (i.e. a subgraph) in a SIA that is a strongly connected graph, that means that every state $s \in X$ can reach every state $s' \in X$. Further, this means that every state $s \in X$ has an outgoing path that includes transitions that are triggered by actions of all the subsystems involved in $X$.

From these observations I devise Algorithm 5.1 to compute the set $Sys(s)$ of subsystems where progress is possible in a particular state $s$ of a composed system. In the following, the functions, called in each line of Algorithm 5.1, are explained in detail.

*Line 1:* **Transform a SIA into a Graph**

$$\langle g_{sia}, v_{init} \rangle \leftarrow \texttt{initSiaGraph}(\widetilde{N}, s_0)$$

**Algorithm 5.1** Permanent Blocking Analysis

```
Require:
    $\widetilde{N}$, the SIA of a PNSC $N$
    $s_0$, the initial state of SIA $\widetilde{N}$

1: $\langle g_{sia}, v_{init} \rangle \leftarrow$ initSiaGraph($\widetilde{N}, s_0$)
2: $Cluster \leftarrow$ getConnectedComponents($g_{sia}$)
3: $g_{cond} \leftarrow$ getClusterGraph($g_{sia}, Cluster$)
4: propagateSubSys($g_{cond}, v_{init}$)
5: condensed2Sia($g_{cond}, g_{sia}, Cluster, \widetilde{N}$)
```

The function `initSiaGraph` takes a SIA $\widetilde{N}$ and its initial state $s_0$ as input arguments and produces the tuple $\langle g_{sia}, v_{init} \rangle$ as an output. A vertex $v$ of the graph $g_{sia}$ corresponds to a state $s \in S_{\widetilde{N}}$. I write $v \equiv s$ to indicate the correspondence. A directed edge $e$ is defined by a sequence of two vertices, written as $(v_1, v_2)$, where vertex $v_2$ is a direct successor of vertex $v_1$. An edge $e = (v_1, v_2)$ of the graph $g_{sia}$ corresponds to a transition $\langle s_1, -, s_2 \rangle \in \delta_{\widetilde{N}}$ where $v_1 \equiv s_1 \in S_{\widetilde{N}}$ and $v_2 \equiv s_2 \in S_{\widetilde{N}}$. The vertex $v_{init}$ of graph $g_{sia}$ corresponds to the initial state of SIA $\widetilde{N}$: $v_{init} \equiv s_0 \in S_{\widetilde{N}}$. To each edge $e$ a set of subsystem identifiers $Sys(e)$ is attached. For a subsystem identifier $M$ with SIA $\widetilde{M}$ to be in the set $Sys(e)$, attached to the edge $e = (v_1, v_2)$, the following must hold:

$$M \in Sys(e) \text{ iff } \exists \langle s_1, a, s_2 \rangle \in \delta_{\widetilde{N}} \,.\, a \in \mathcal{A}_{\widetilde{M}}$$

with $v_1 \equiv s_1 \in S_{\widetilde{N}}$ and $v_2 \equiv s_2 \in S_{\widetilde{N}}$.

*Line 2:* **Compute Connected Components**

$$Cluster \leftarrow \texttt{getConnectedComponents}(g_{sia})$$

The function `getConnectedComponents` takes a graph $g_{sia}$ as an input argument and produces a set $Cluster$ as an output. The set $Cluster$ contains sets of vertices where each set holds all vertices that belong to a strongly connected component in the graph $g_{sia}$. This computation is feasible in linear time $\mathcal{O}(V+E)$ by the algorithm proposed by Tarjan [64] where $V$ is the number of vertices and $E$ the number of edges in the graph.

*Line 3:* **Condense Graph**

$$g_{cond} \leftarrow \texttt{getClusterGraph}(g_{sia}, Cluster)$$

The function `getClusterGraph` takes the graph $g_{sia}$ and the set $Cluster$ as input parameters and produces the graph $g_{cond}$ as an output. In the resulting graph $g_{cond}$, called *condensed graph*, each strongly connected component is contracted into a single vertex. This reduces the state space of the graph and removes all cycles except for self-loops on the contracted vertices. Further, edges with the same source and target vertex are merged (i.e. each vertex has at maximum one self-loop). When merging two edges $e_1$ and $e_2$ into an edge $e$, the set $Sys(e)$ attached to a merged edge $e$ is defined as follows:

$$Sys(e) = Sys(e_1) \cup Sys(e_2)$$

Finally, all self-loops are removed to transform the graph into a DAG. The edge attributes $Sys(e)$ of the self-loops are assigned to the corresponding vertex as attribute $Sys(v)$. For vertices where no self-loop was present the set $Sys(v)$ is empty.

*Line 4:* **Propagate Subsystem Indentifiers**

$$\texttt{propagateSubSys}(g_{cond}, v_{init})$$

The function `porpagateSubSys` takes the condensed graph $g_{cond}$ and the vertex $v_{init}$ as input parameters and updates the set $Sys(v)$ of each vertex with subsystem identifiers, which can perform an action from the current vertex and each successor vertex. This is accomplished by traversing the graph and propagating the edge attributes $Sys(e)$. The graph traversal is done by a Depth-first Search (DFS) in linear time ($\mathcal{O}(V+E)$ where $V$ is the number of vertices and $E$ the number of edges in the graph). This is described in detail by Algorithm 5.2 where the following functions are used:

`target`$(e)$ returns the target vertex of edge $e$.

`setVisited`$(v)$ marks a vertex $v$ as visited.

`isVisited`$(v)$ checks whether a vertex $v$ was already visited by the algorithm.

`incident`$(g, v)$ returns the incident edges of a vertex $v$ in a graph $g$.

I have to check for visited vertices because the condensed graph can include alternate paths. The function `propagateSubSys` is called recursively on all direct successors of the current vertex (line 8) and assigns to each vertex a set of subsystem identifiers (line 9). By updating the set of subsystem identifiers $Sys(v)$ of the vertex $v$ in line 9, I return the future of the path from a state corresponding to $v$: If $sys \in Sys(v)$, then an action $a \in \mathcal{A}_{\widetilde{sys}}$ can trigger a transition along a path starting from a state corresponding to $v$.

**Algorithm 5.2** Propagate Subsystem Identifiers

```
Require: Function arguments
    $g_{cond}$, the condensed graph (line 3 of Algorithm 5.1)
    $v$, vertex to traverse the graph from

function propagateSubSys($g_{cond}$, $v$)
    if isVisited($v$) then
        return $Sys(v)$
    end if
    setVisited($v$)
    $E \leftarrow$ incident($g_{cond}, v$)
    for all $e \in E$ do
        $res \leftarrow$ propagateSubSys($g_{cond}$, target($e$))
        $Sys(v) \leftarrow Sys(e) \cup res$
    end for
    return $Sys(v)$
end function
```

*Line 5:* **Assign Vertex Attributes to States in SIA**

$$\texttt{condensed2Sia}(g_{cond}, g_{sia}, Cluster, \widetilde{N})$$

The function `condensed2Sia` takes the annotated, condensed graph $g_{cond}$, the graph $g_{sia}$, the set of clusters $Cluster$, and the SIA $\widetilde{N}$ as input parameters and computes the set $Sys(s)$ for each state $s \in S_{\widetilde{N}}$. This is achieved by first assigning the vertex attributes $Sys(v)$ of $g_{cond}$, generated in the previous step, to each corresponding vertex in $g_{sia}$. The set $Cluster$, returned by the function `getConnectedComponents` in line 2 of Algorithm 5.1, is used to establish the correspondence between the vertices in the two graphs. Finally, the set $Sys(s)$ is computed for each state $s \in S_{\widetilde{N}}$ by a one-to-one assignment of each corresponding set $Sys(v)$ in graph $g_{sia}$.

The algorithm to identify permanent blocking states as described by Algorithm 5.1 is of linear time complexity ($\mathcal{O}(S + T)$ with $S$ denoting the number of states and $T$ the number of transitions of a SIA) because the graph cycles were eliminated in the process of line 2 and 3.

### 5.2.2 Algorithm to Compute $dl\Big((N_1, \ldots, N_n), s\Big)$

In order to distinguish between subsystems that are lonely blocking and subsystems that are involved in a deadlock situation a graph representing the blocking dependencies must be used. This is achieved as follows:

The permanent blocking analysis returns the set of permanent blocking states (see Equation 5.7) and the set of blocking subsystems (see Equation 5.6) for each permanent blocking state. Using this information I identify the respective actions in each subsystem that are prevented from triggering the transitions. With the help of the dependency graph of the PNSC and the list of blocking actions in each permanent blocking state the wait dependencies of each subsystem in the permanent blocking situation can be represented as a graph. This corresponds to the ordered set of subsystems $(N_1, \ldots, N_n)$, used in the predicate $dl$, as defined in Equation 5.11. Using the same clustering approach as with the permanent blocking analysis (line 2 and 3 of Algorithm 5.1) I distinguish between vertices that are involved in a cycle and vertices that stand alone and, consequently, distinguish between a deadlock situation and lonely blockers.

## 5.3 Chapter Summary

In this chapter I introduced a static analysis for a system of interconnected processes, each process described by a SIA, that allows to detect permanent blocking states. The permanent blocking analysis is performed in linear time complexity $\mathcal{O}(S+T)$ where $S$ denotes the number of states and $T$ the number of transitions in a composed system. However, $T$ and $S$ grow exponentially due to the state explosion problem of Labelled Transition System (LTS) composition. In order to control the state explosion, transitions, labelled by internal actions, need to be removed. Research has been done in this area and there exist promising approaches (e.g. [62]) that could be applied to the work presented in this dissertation. The applicability is not addressed in this dissertation and is postponed for future work.

A further analysis (linear time complexity) allows to distinguish between two types of permanent blocking: *Deadlocks*, where a circular wait is the cause for blocking, and *lonely blocking* where a process is blocking without causing other processes to block.

Regarding scalability, the composition operation of SIAs causes an exponential growth of the state space of the composed system. Given that the composition operation is applied incrementally on each process of a PNSC, the state space grows very fast. This is not a problem for small systems that are closed because an imposed order on input actions causes a lot of states to be unreachable which keeps the state space small. For bigger, open systems, however, the state space has to be reduced by removing internal actions and merging states solely involving transitions triggered by internal actions. This allows to compose processes of a PNSC into a process that can be used as a component in a hierarchical system.

A future work will be to remove internal actions in order to reduce the state space of folded SIAs.

Chapter 6

# Streamix - An Instantiation of PNSCs as a Coordination Language

In this chapter I introduce the coordination language Streamix. Streamix is based on the Process Network with Synchronous Communication (PNSC) model which I introduced in Chapter 3 and extended in Chapter 4. Therefore, Streamix represents one possible instance of the PNSC model and serves as proof of concept.

The inspiration for Streamix stems from the coordination language S-Net [10]. One of the achievements of S-Net is the capability of modelling networks in a structured manner by employing binary operators. Streamix adopts the concept of structured programming and uses similar operators to model networks. The concept of the underlying model of Streamix, however, differs largely from S-Net due to the differences of their respective targeted application fields. While Streamix is suitable for Cyber-physical Systems (CPSs), where complex, reactive interaction patterns pose a challenge for structured system composition, S-Net is mostly intended for best-effort applications with transformational data-processing aspects (e.g. systems where computational chunks can easily be pipelined).

Structured programming has become a very successful programming paradigm as it provides locality of a program's control flow. Similar concepts of locality are desired for the specification and development of concurrent and parallel systems. In the domain of CPSs or embedded computing it is challenging to identify such structured compositions since control flow tends to be driven by concurrently acting reactive components with often circular dataflow relations.

A structured network provides a sense of locality and allows to observe and understand a part of the network without having to consult other, potentially distant, parts of the network. This is achieved by keeping port-to-port connections local. In contrast to this, a non-structured network would rather be created with global port-to-port connection tables which are hard to read and understand. For a modular development it is important to understand the behaviour of an isolated sub-entity, independent of where it is going to be integrated, which is why a structured approach is key.

Streamix aims to enforce structure, support reactive data processing, and allow persistent state and synchronisation points within behavioural components. Streamix allows to model streaming networks where the topology of the network imposes dependencies between components and where message streams are coordinated within the network. Streamix is a representative of the exogenous coordination model [33] in order to assure clear separation of concerns.

The chapter is structured as follows: Section 6.1 gives an overview of the coordination model of the language which consists of three layers, namely the computational components, the routing network, and the extra-functional requirements layer. Section 6.2 describes the box abstraction, an abstraction used to represent computational components. Section 6.3 describes how such boxes are instantiated as nets and how their interface is defined. Section 6.4 describes how to compose multiple nets with different network operators. Finally, in Section 6.6 I discuss limitations of the coordination language and in Section 6.7 I summarise the chapter.

## 6.1 Coordination Model

An application can benefit from a parallel hardware architecture if it is decomposable into multiple chunks, called *computational components* throughout the chapter, where some or all of those can be run, possibly at the same time, on different parts of the parallel architecture. The computational components may be required to interact with each other which must be handled by some sort of communication structure, called *routing network* throughout the chapter. A routing network can be a simple connection between two computational components or it can include complex synchronisation and routing operations to coordinate multiple computational components. If the decomposition is performed to such a granularity that the only concern of one computational component is to execute a specific functionality and all communication and synchronisation of data happens through routing networks placed outside the computational component, perfect separation of concerns with respect to behaviour and coordination is achieved. Streamix does not require such a decomposition as it targets CPSs where computational components are often of reactive nature which are hard to decompose [13]. The separation is further underlined by using a conventional programming language to describe the behaviour of the content of a computational component and a coordination language to describe the routing networks.

To summarize, the general coordination model described above consists of *computational components*, *routing networks*, and an *extra-functional requirements layer*. In the following each layer is described in more detail.

### 6.1.1 Computational Components

The computational components are oblivious of any other computational component and also to the fact that they are coordinated by routing networks. This corresponds, according to the classification of Arbab [65], to an exogenous coordination model. Computational components follow a triggering semantics that is imposed by the coordination model but they are not altered by the coordination language in any way. A computational component is considered to be a black box by the coordination language and can have persistent state or be stateless (pure).

### 6.1.2 Routing Network

The concerns of the coordination language are the synchronisation and routing of information that is passed between computational components through routing networks. In this chapter, any information or data that is transmitted over a routing network will be referred to as *message token*, independent of the complexity or the type of the information. Routing networks can be composed out of multiple primitives to form a complex coordination construct. Such primitives are referred to as *coordination components* throughout this chapter. Coordination components can be either implicitly generated or explicitly described by the coordination language. In the latter case, the coordination language defines primitives to perform explicit coordination actions within the coordination model. Common examples are coordination components that perform synchronization, copy, or merge operations. The most basic coordination component is a channel, where message tokens are transmitted from one end to the other with an implicit synchronisation between the two ends. Channels are buffered and follow the *First In, First Out (FIFO)* principle. The routing networks serve as coordination instances between computational components that glue the computational components together and are hence the glue-code between the computational components.

### 6.1.3 Extra-functional Requirements Layer

Another aspect of coordination is the question *when* the computational components are executed. This depends on the data dependencies between the computational components but also on the time-criticality of the system. Data dependencies are given out of construction by the network topology. For computational components without particular timing constraints, the scheduler of the runtime system of the coordination model decides when computational components are executed. This is also true in the case of time-critical systems, however, when deadlines must be respected the scheduler is not only influenced by the data dependencies but also by imposed timing requirements on the computational components. Such timing requirements are provided by the third layer of the language architecture, the *extra-functional requirements layer*. In general, the *extra-functional requirements layer* allows to annotate computational components or routing networks in order to ensure requirements that are linked to the extra-functional behaviour of the coordination of the application.

## 6.2 Box Abstraction

In order do describe how a computational component interacts with coordination components, it is associated to a construct I call *box* throughout this chapter. A box has input and output ports, corresponding to the input and output parameters of a computational component. A box is of type Multiple Input, Multiple Output (MIMO). I use the term *mode* of a port to talk about its direction, i.e. whether it is an input or an output port. Further, a Synchronous Interface Automaton (SIA) is associated with a box. While the computational component describes the behaviour of the box, a SIA describes the interaction of the box with its environment.

Boxes need to be instantiated. An instance of a box is called a *net* in Streamix. Similar to a class in an object-oriented programming language serving to create objects, a box in Streamix serves as a code template to create nets. A net corresponds to a process of a PNSC as defined by Definition 3.1. Nets are hierarchical and can include other nets and networks of nets. This is further discussed in Section 6.3.

### 6.2.1 User-defined Boxes

Boxes are either user-defined or fulfil a predefined function. User-defined boxes serve as an abstraction for computational components that describe a custom functionality of an application. Such computational components are usually provided by a domain expert and implemented in a programming language suitable for the specific problem.

A user-defined box is declared and assigned to a symbol which can later be used to spawn a net instance of the declaration. A box declaration links the source code of a computational component to the box and assigns the input and output parameters of the computational component with the input and output ports of the box. For example, the following box declaration links the input parameter `x` and the output parameter `y` of the computational component `fa` to the input port `x` and output port `y` of the box `A`.

```
A = box fa( in x, out y )
```

The names of the computational component parameters must not necessarily be the same as the names of the corresponding box ports. If so, this must be specified in the box declaration. In the following example, the parameters `x` and `y` of computational component `fa` are linked to the ports `x_box` and `y_box` of box `B`.

```
B = box fa( in x_box( x ), out y_box( y ) )
```

This allows to reuse the same computational component in a different environment by declaring a new box and assigning it to a different symbol.

### 6.2.2 Implicit Boxes

In Chapter 3 I introduced an interface model, called SIA, that allows to describe the interaction of processes with their environment. In Chapter 5 I then introduced an analysis based on SIAs that allows to check for freedom of permanent blocking within the process network. SIAs are part of the PNSC model where two basic primitives are used to model an application: processes and synchronous channels. The model offers a separation of computational and coordination concerns by describing an application as a network of processes where processes represent computational components and channels, connecting processes, represent coordination components. In Chapter 4 I extended the model with Cross-criticality Interfaces (CCIs) that allow coordination of processes with respect to communication coupling or rate-control. Streamix implements CCIs as coordination components and provides more implicitly generated coordination components. In this section I will describe these coordination components that serve to route and synchronize message tokens in the network. Each coordination component represents a process in the PNSC model and is therefore an implicitly instantiated net, based on a Streamix box. Consequently, a SIA and a computational component is associated with it. The details of the implementation of each coordination component is discussed in Chapter 7.

#### 6.2.2.1 FIFO Buffers

Streamix follows a stream processing concept in the sense that message tokens are communicated over First-in, First-out (FIFO) channels. A channel $c_i$ is spawned between two ports with matching names (see Section 6.3 for more detail on the connection semantics of ports) and has a length $len(c_i)$. By default, no length of the channels is specified. However, as Streamix targets CPSs where controllability of the system is an important aspect it is reasonable to keep the length of each channel fix. Therefore, each port $p_i$ in a user-defined box declaration can be annotated with a fixed buffer length $len(p_i)$ by appending the desired length in square brackets to the port declaration. A FIFO has to have a minimal length of one. The box `Kb` for example, uses a buffer length of two for the channel linked to port `a`, an unspecified buffer length for the channel linked to port `b`, and a buffer length of three for the channel linked to port `c`.

```
Kb = box fk( in a[2], out b, decoupled in c[3] )
```

By writing to a channel, a message token is put into the channel and by reading from a channel a message token is removed from the channel. If a channel is empty, the read access is blocking until a message token is written into the channel. If a channel is full, the write access is blocking until a message token is read from the channel.

As described in Section 4.1, due to mixed-criticality requirements it is sometimes necessary to remove the blocking semantics of the PNSC model. This can be achieved by annotating a port in a user-defined box declaration with the keyword `decoupled`. In the following example, the box `Kd` has a decoupled output port `b` and a decoupled input port `c`.

```
Kd = box fk( in a, decoupled out b, decoupled in c )
```

As described in Section 4.1.2, decoupling an input port in synchronisation prevents this port from triggering a box and can lead to duplication of message tokens. Decoupling an output port in synchronisation removes the back-pressure imposed by consumers and can lead to losses of message tokens. Note that a box must have at least one input port that is triggering or at least one output port where back-pressure can be exert. In Section 4.1.3 I defined the SIAs for all combinations of FIFO buffers with and without decoupled inputs and outputs: Please refer to Figure 3.14 for an example of a SIA for a blocking FIFO buffer, to Figure 4.4 for a FIFO with decoupled input (the output port of the connecting producing net is decoupled), to Figure 4.5 for a FIFO with decoupled output (the input port of the connecting consuming net is decoupled), and to Figure 4.6 for a FIFO where input and output is decoupled (and consequently the output and input port of the connected producing and consuming net).

Note that the length of the buffer is related to the coupling of a port. The length of a channel $c_i$, where each end is connected to a port ($p_1$ and $p_2$, respectively), is defined as follows:

$$len(c_i) = \begin{cases} 1 & \text{if } p_1 \text{ and } p_2 \text{ are decoupled} \\ max(len(p_1), 1) & \text{if } p_2 \text{ is decoupled} \\ max(len(p_2), 1) & \text{if } p_1 \text{ is decoupled} \\ max(len(p_1) + len(p_2), 1) & \text{otherwise} \end{cases} \tag{6.1}$$

#### 6.2.2.2 Routing Node

Routing nodes are spawned implicitly whenever more than two ports are connected. Multiple connections to an input port of a net spawn a summing node and multiple connections to an output port of a net spawn a diverging node. Hence, a routing node can either be a diverging node, a summing node or a combination of any number of both. In general, a routing node can have any number of input and output ports.

A diverging node is a special case of a routing node which has one input port and two or more output ports. A message token arriving at the input port is copied to each output port. As long as the writing process to an output is blocked, the diverging node is not accepting any more message tokens at its input port. This means before the diverging node is able to process any further message token it has to complete the copy process to each output.

An example of a diverging node with its corresponding SIA is depicted in Figure 6.1.

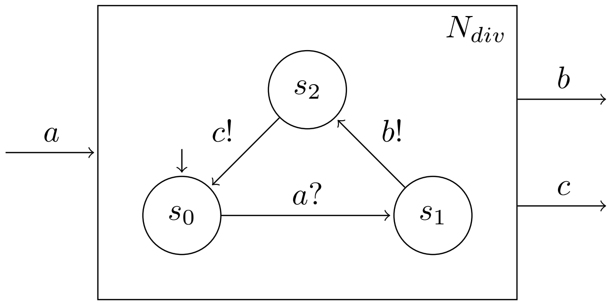


FIGURE 6.1: An example of a diverging node with an input port $a$ and two output ports $b$ and $c$.

Another special case of a routing node is a summing node. It has one output port and two or more input ports. Message tokens are read at the input ports according to the first-come first-serve policy and are made available at the output port in the same order as they were read. If message tokens are available on multiple input ports, they are read in arbitrary order and are processed non-deterministically. The first-come first-serve policy requires a summing node to peak at an input port whether a message token is available before accessing it in order to prevent a potential blocking on one input port where no message token is available while on another port message tokens would be available. This leads to non-deterministic behaviour [23]. If the output is blocking, no further message tokens are accepted at the input until the currently blocked message token is written.

An example of a summing node with its corresponding SIA is depicted in Figure 6.2.

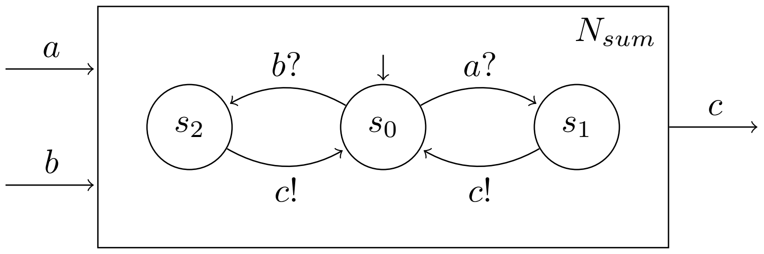


FIGURE 6.2: An example of a summing node with input ports $a$ and $b$ and an output port $c$.

By combining summing and diverging nodes, a routing node allows a one-to-many, many-to-one, and many-to-many mapping of ports. For a one-to-one mapping of two ports, no routing node is necessary.

An example of a combination of the two node types with its corresponding SIA is shown in Figure 6.3.

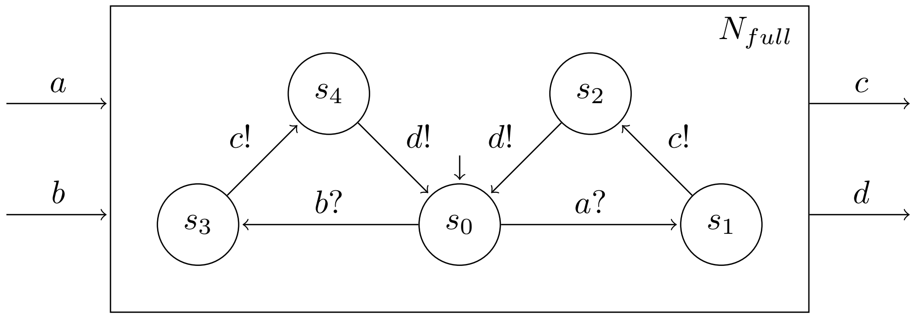


FIGURE 6.3: An example of a routing node as combination of summing and diverging nodes with input ports $a$ and $b$ and output ports $c$ and $d$.

#### 6.2.2.3 Temporal Firewall

A temporal firewall is a special type of communication channel (see Section 4.2.1 for details on the model). It imposes a fixed rate on the transmission of message tokens, i.e. at a specified rate, a single message token is copied from the input port to the output port. The input port and the output port of a temporal firewall is decoupled to guarantee that no blocking is possible for any connected net. This may lead to duplication or loss of message tokens: Let $r_{clk}$ be the rate of the temporal firewall, $r_p$ the rate of a producing net which is connected to an input port of the temporal firewall, and $r_c$ the rate of a consumer net which is connected to an output port of the temporal firewall. A message token duplication occurs if $r_p < r_{clk}$, i.e. the rate of the producer is lower than the rate of the temporal firewall or if $r_{clk} < r_c$, i.e. the rate of the temporal firewall is lower than the rate of the consumer. A message token loss occurs if $r_p > r_{clk}$, i.e. the rate of the producer is higher than the rate of the temporal firewall or if $r_{clk} > r_c$, i.e. the rate of the temporal firewall is higher than the rate of the consumer. A temporal firewall is implicitly spawned when declaring a net as a time-triggered instance (see Section 6.4.5 for more details).

#### 6.2.2.4 Rate-control Guard

Another special type of communication channel is the rate bounded channel (see Section 4.2.2 for the model description). It is a FIFO channel, as specified by the port declaration of the connecting ports, extended with a rate-control guard (the length and the coupling attributes of the channel do not change). Simply put, a rate bounded channel only accepts a new message token if a certain time interval has passed between the arrival instances of the two consecutive message tokens. Such channels are implicitly spawned when declaring a net as rate bounded. I will describe the Streamix syntax to declare a rate bounded net in Section 6.4.5.

### 6.2.3 Interaction Protocol of a User-defined Box

In a box declaration, the order of the port list is important for two reasons: First, depending on the implementation of the Run-time System (RTS), the parameters of a computational component might be matched according to the index corresponding to the position of a port in the port list of the box declaration. This is not necessarily true, as a mapping could also be achieved through name matching (as it is done with the runtime system described in Chapter 7 where C macros are used to access channels by name) but it is good to have a possible identification rule that does not rely on names. The second reason relates to the SIA description of a box. In addition to a computational component, each box has a SIA that is associated to the box declaration.

The order of the ports is used to automatically generate a SIA description of a box if none is provided by the user. The algorithm to automatically generate a SIA is based on the assumption that the parameters of the computational component, corresponding to the ports of a box, are accessed in the same order as the ports are declared in the box declaration. An automatically generated SIA $\widetilde{K}$, corresponding to the box declaration

```
K = box fk( in a, out b, out c )
```

is a circle graph with three states $S_{\widetilde{K}} = \{0, 1, 2\}$ and three transitions $\langle 0, a, 1\rangle$, $\langle 1, b, 2\rangle$, and $\langle 2, c, 0\rangle$. Each action in the set of actions $\mathcal{A}_{\widetilde{K}} = \{a, b, c\}$ corresponds to the port of the box declaration with the same name. State 0 is the initial state. The protocol, described by such a SIA is deterministic.

If the default SIA is not representing correctly how a computational component interacts with its environment, the user must provide a custom SIA that is associated with the box. To do so, a language defined by the following grammar can be used:

| | |
|---|---|
| ⟨*sia_decl*⟩ | ⇒ **sia** ⟨*sia_name*⟩ **{** ⟨*state_decl*⟩ *[* ⟨*state_decl*⟩ *]** **}** |
| ⟨*state_decl*⟩ | ⇒ ⟨*state_name*⟩ **:** ⟨*transition*⟩ *[* **\|** ⟨*transition*⟩ *]** |
| ⟨*transition*⟩ | ⇒ ⟨*action_name*⟩ ⟨*action_mode*⟩ **.** ⟨*state_name*⟩ |
| ⟨*action_mode*⟩ | ⇒ **?** |
| | \| **!** |
| | \| **;** |
| ⟨*sia_name*⟩ | ⇒ identifier |
| ⟨*state_name*⟩ | ⇒ identifier |
| ⟨*action_name*⟩ | ⇒ identifier |

The name of the user-defined SIA must be the same as the name of the computational component and each state declaration symbol can only occur once. A state declaration can, however, have multiple transitions. The first state declaration corresponds to the initial state of the SIA. For example, the default SIA $\widetilde{K}$ as described above would be described with the language as follows:

```
sia fk {
  0: a?.1
  1: b!.2
  2: c!.0
}
```

However, if the protocol of computational component `fk` does not specify in which order the ports `b` and `c` are accessed, the SIA is defined as follows:

```
sia fk {
  0: a?.1
  1: b!.2 | c!.3
  2: c!.0
  3: b!.0
}
```

Here, from state `1` two paths are possible to reach the initial state `0`: either by the transitions $\langle 1, b, 2\rangle$ and $\langle 2, c, 0\rangle$ or by the transitions $\langle 1, c, 3\rangle$ and $\langle 3, b, 0\rangle$. If there is an internal decision, based on an internal state, deciding whether port `b` or port `c` is accessed, internal actions need to be used to describe this behaviour:

```
sia fk {
  0: a?.1
  1: int1;.2 | int2;.3
  2: b!.0
  3: c!.0
}
```

In state `1` the protocol is non-deterministic because the choice whether transition $\langle 1, int1, 2\rangle$ or $\langle 1, int2, 3\rangle$ is taken depends on an internal state of the computational component that is not accessible by the language.

### 6.2.4 Box Annotations and Grammar

Streamix provides two keywords to annotate elements with information dedicated for the RTS. A box can be annotated with the keyword `pure` if the computational component is purely functional. This information can be useful in the runtime system: e.g. it is easy to relocate a pure computational component as no state information has to be relocated with it whereas a relocation of a computational component with persistent state is much harder because state information has to be moved as well.

Further, a box can be annotated with the keyword `static` in order to indicate that no dynamic operations should be performed with instances of this box. Again, this is an annotation that is useful for the runtime system if automatic proliferation of computational components is performed in order to exploit parallel architectures.

Putting everything together, the following grammar defines a box declaration in Streamix. Note that ports in a box declaration can be assigned to *port collections*. Why this is useful and sometimes necessary will be explained in Section 6.3.

| | | |
|---|---|---|
| ⟨*box_assign*⟩ | ⇒ | ⟨*box_name*⟩ **=** ⟨*box_decl*⟩ |
| ⟨*box_decl*⟩ | ⇒ | *[* **static** *]* *[* **pure** *]* **box** ⟨*impl_name*⟩ **(** ⟨*box_port_list*⟩ **)** |
| ⟨*box_port_list*⟩ | ⇒ | ⟨*box_port_decl*⟩ *[* **,** ⟨*box_port_decl*⟩ *]** |
| ⟨*box_port_decl*⟩ | ⇒ | *[* **decoupled** *]* *[* ⟨*port_collection*⟩ *]* ⟨*port_mode*⟩ ⟨*port_name*⟩ *[* **(** ⟨*port_name_alt*⟩ **)** *]* *[* **[** ⟨*buffer_len*⟩ **]** *]* |
| ⟨*port_mode*⟩ | ⇒ | **in** |
| | \| | **out** |
| ⟨*buffer_len*⟩ | ⇒ | $\mathbb{Z}^+$ |
| ⟨*impl_name*⟩ | ⇒ | identifier |
| ⟨*port_name*⟩ | ⇒ | identifier |
| ⟨*port_name_alt*⟩ | ⇒ | identifier |

## 6.3 Nets: Instantiations of Boxes

A net $N$ can be the instantiation of a single box in the simplest case or a combination of networks via network combinators. The communication interface of each net $N$ consists of a set of ports: $\mathcal{P}(N)$. Each port $p_i \in \mathcal{P}(N)$ is either an input or output port, specified by a mode attribute: $mode(p_i) \in \{input, output\}$.

Based on that I define predicates for input and output ports as follows:

$$\mathcal{P}^I(N) = \{ \; p_i \mid p_i \in \mathcal{P}(N) \;\wedge\; (mode(p_i) = input) \; \} \tag{6.2}$$

$$\mathcal{P}^O(N) = \{ \; p_i \mid p_i \in \mathcal{P}(N) \;\wedge\; (mode(p_i) = output) \; \} \tag{6.3}$$

A port $p_i$ is locally identified by its name.

A network is created by instantiating boxes as nets and by connecting the ports of the nets together with directed channels. Channels have two ends where each end connects to a single port of a net. The two ends must be connected to ports of opposite mode which gives the channel a flow-direction. In order to create a connection between two ports with a channel, the port names must match. The connection semantics of nets is later explained in detail in Section 6.4 where network operators are described.

The tuple $< port\_name, port\_mode >$ must be unique in the port set of a net. This means, however, that a net can have two ports with the same name.

A network topology as depicted in Figure 6.4 is rather common in the world of CPSs due to their reactive nature. Streamix poses no constraints on the flow direction of connections, hence, a connection between two nets can be composed out of multiple channels with different directions. This results in a network where the flow-direction is

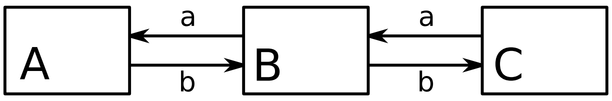


FIGURE 6.4: An example of a simple network where three components are interconnected with no obvious data flow direction.

not clearly defined and channel directions may become ambiguous.

### 6.3.1 Flow Direction Ambiguities

As described in the beginning of this section, channels are directed. As nets can have multiple input and output ports, a connection between two components can be composed out of multiple channels with different directions. This results in a network where the flow-direction is not clearly defined and channel directions may become ambiguous.

As an example, let's consider a net $B$ with two input ports $a$ and $b$ and two output ports $a$ and $b$ as depicted in Figure 6.4. The interface of such a net is not clearly defined as there are four possible ways to place the channels at the interface of the net (see Figure 6.5). Hence a connection to another, similar net would be ambiguous.

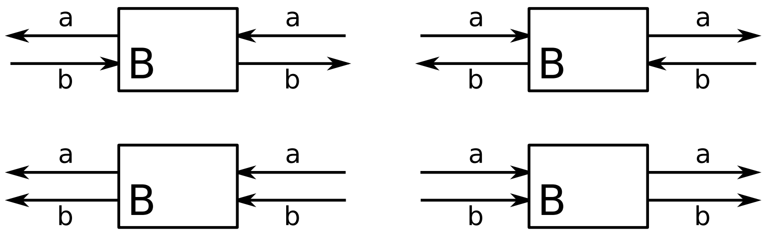


FIGURE 6.5: An example of an ambiguous flow direction. The net $B$ can be connected in four different ways.

Note that the connection of $B$ is only ambiguous when inspected locally. By unfolding the complete network and assuming that all ports are connected (this is a requirement for a network to be valid), the direction of each channel is clearly defined (as in the example of Figure 6.4). However, this is not sufficient because in a structured program all parts of the program need to be unambiguous, independent of their surrounding elements.

In order to achieve an unambiguous flow direction or to help the programmer to structure the code, ports can be grouped into two separate collections: A *left (L)* or a *right (R)* collection. Note that the names *left* and *right* refer to the left and right side of a network operator (see Section 6.4) and do not necessarily relate to a schematic representation of a net. This grouping is optional and can be omitted if the flow direction is unambiguous. The grouping of the ports into the two collections depends on the context of the program and each collection can hold any number of input or output ports or can be empty. The grouping provides a logical flow-direction due to the constraint that ports in a left collection of one net can only connect to ports in a right collection of another net and vice versa. The logical flow-direction is unrelated to the real flow-direction of messages as channels can be of arbitrary direction. An analogy from the real world would be the transmission of electrical signals on a ribbon cable with a female and a male connector on each end. While the cable is directed, each wire in the cable can communicate electrical signals in an individual direction.

A third collection, called *side (S)*, allows its ports to broadcast message tokens to other ports in a side collection. Side-ports are explained in more detail in Section 6.3.3.

The collection of a port is described as $col(p_i) \in \{L, R, S\}$. I use the following predicates for port collections:

$$\mathcal{P}_L(N) = \{\ p_i \mid p_i \in \mathcal{P}(N) \ \wedge\ (col(p_i) = L)\ \} \tag{6.4}$$

$$\mathcal{P}_R(N) = \{\ p_i \mid p_i \in \mathcal{P}(N) \ \wedge\ (col(p_i) = R)\ \} \tag{6.5}$$

$$\mathcal{P}_S(N) = \{\ p_i \mid p_i \in \mathcal{P}(N) \ \wedge\ (col(p_i) = S)\ \} \tag{6.6}$$

The set of ports not grouped into a collection is described by the predicate:

$$\mathcal{P}_0(N) = \Big\{p_i \mid p_i \in \mathcal{P}(N) \ \wedge\ \big(col(p_i) \neq L\big) \ \wedge\ \big(col(p_i) \neq R\big) \ \wedge\ \big(col(p_i) \neq S\big)\Big\} \tag{6.7}$$

Figure 6.6 depicts a schematic representation of a net where ports are grouped into the three different collections.

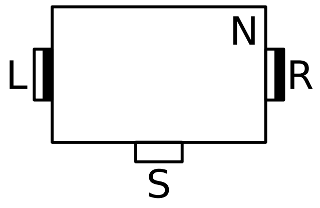


FIGURE 6.6: Schematic representation of a *net* with the three optional port groupings *left (L)*, *right (R)*, and *side (S)*.

An abstract representation of the network depicted in Figure 6.4 is shown in Figure 6.7 where ports are grouped in a left and a right collection. The connections are represented as undirected lines as the real flow direction can only be known by inspecting the port definitions in the declarations of two connecting nets.

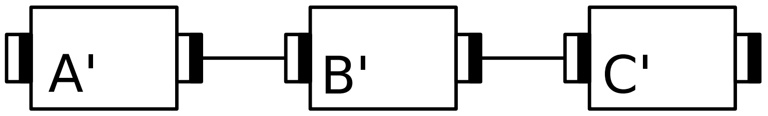


FIGURE 6.7: The representation of the network of Figure 6.4 with the use of left and right collections.

In the following I will use a notation that combines the sets defined above: Combining the subscript symbol, indicating the port collection (as it is used in Equation 6.4, Equation 6.5, and Equation 6.6), with the superscript symbol, indicating the port mode (as it is used in Equation 6.2 and Equation 6.3), correspond to the intersection operation of both sets, e.g. $\mathcal{P}_L^O = \mathcal{P}_L \cap \mathcal{P}^O$.

### 6.3.2 Self-loop Connection

The exact semantics of port connections if nets are combined with network operators is described in Section 6.4. It is, however, possible for a net to form a connection with itself. Let $\hat{\mathcal{P}}_0^I$ be the set of all input ports of $N$ that are not assigned to a collection and let $\hat{\mathcal{P}}_0^O$ be the set of all output ports of $N$ that are not assigned to a collection. A self-loop connection is defined by Definition 6.1.

**Definition 6.1** (Self-loop Connection)**.** *The self-loop connections of a net $N$ are defined by the set*

$$\mathcal{P}_F^H = \hat{\mathcal{P}}_0^I \cap \hat{\mathcal{P}}_0^O$$

As an example, let's consider the following box declaration.

```
C = box fc( in x, out x )
```

Once this box is instantiated as net, a self-loop is formed from the output to the input port, named x in this example. If one wants to declare a box that can be instantiated multiple times and connected in a chain, port collections have to be used:

```
C = box fc( left in x, right out x )
```

This prevents self-loops from being established as no matching ports are found in the set of all non-grouped ports $\hat{\mathcal{P}}_0(C)$.

### 6.3.3 Side-port Connection

In Streamix side-ports, i.e. ports in the set $\mathcal{P}_S$, serve the purpose of stream broadcasts. This means that side-ports of a net $N_1$ and a net $N_2$ are implicitly connected by all serial and parallel composition operators which are defined in Section 6.4.2 and Section 6.4.3. Side-port connections are defined by Definition 6.2.

**Definition 6.2** (Side-port Connection)**.** *The set of side-ports $\mathcal{P}_S(N)$ of a resulting net $N$ of any parallel or serial composition of two nets $N_1$ and $N_2$ is defined by the tuple*

$$\mathcal{P}_S = \langle \mathcal{P}_S^I, \mathcal{P}_S^O, \mathcal{P}_S^{\{IO\}} \rangle$$

*where each element is defined as follows:*

- $\mathcal{P}_S^{\{IO\}} = (\mathcal{P}_S^I(N_1) \cap \mathcal{P}_S^O(N_2)) \cup (\mathcal{P}_S^O(N_1) \cap \mathcal{P}_S^I(N_2))$*: the set of bi-directional side ports of $N$ is the intersection of all side ports of opposite mode of $N_1$ and $N_2$.*
- $\mathcal{P}_S^I = (\mathcal{P}_S^I(N_1) \cup \mathcal{P}_S^I(N_2)) \setminus \Big((\mathcal{P}_S^I(N_1) \cap \mathcal{P}_S^I(N_2)) \cup \mathcal{P}_S^{\{IO\}}\Big)$*: the set of input side ports of $N$ is the union with no duplicates of all input side ports of $N_1$ and $N_2$, excluding any port of the set $\mathcal{P}_S^{\{IO\}}$.*
- $\mathcal{P}_S^O = (\mathcal{P}_S^O(N_1) \cup \mathcal{P}_S^O(N_2)) \setminus \Big((\mathcal{P}_S^O(N_1) \cap \mathcal{P}_S^O(N_2)) \cup \mathcal{P}_S^{\{IO\}}\Big)$*: the set of output side ports of $N$ is the union with no duplicates of all output side ports of $N_1$ and $N_2$, excluding any port of the set $\mathcal{P}_S^{\{IO\}}$.*

In contrast to all other port connections (described in Section 6.3.2 and Section 6.4), connected side-ports are not hidden from the interface of the resulting net. Consequently, they are propagated with each composition and connect to any side-port with the same name in the network. Because of this, multiple ports are connected together and routing nodes are spawned. This results in a potential situation where either input or output side-ports can connect to the port of a resulting net. This is indicated by the superscript '$\{IO\}$' of the set $\mathcal{P}_S^{\{IO\}}$.

Let $p_1 \in \mathcal{P}_S^I(N_1)$ be a side-port of mode input in net $N_1$ and $p_1 \in \mathcal{P}_S^I(N_2)$ be a side-port of mode input in net $N_2$. When combining the two networks $N_1$ and $N_2$ with any serial or parallel combination operator, a diverging node is spawned and the port $p_1 \in \mathcal{P}_S^I(N)$ of the resulting net $N$ is of mode input. Similarly, two side ports $p_2 \in \mathcal{P}_S^O(N_1)$ and $p_2 \in \mathcal{P}_S^O(N_2)$ spawn a summing node and result in a port $p_2 \in \mathcal{P}_S^O(N)$. However, two side ports $p_3 \in \mathcal{P}_S^I(N_1)$ and $p_3 \in \mathcal{P}_S^O(N_2)$ or $p_4 \in \mathcal{P}_S^O(N_1)$ and $p_4 \in \mathcal{P}_S^I(N_2)$ spawn a general routing node and result in a port $p_3 \in \mathcal{P}_S^{\{IO\}}(N)$ or $p_4 \in \mathcal{P}_S^{\{IO\}}(N)$, respectively, where the mode is input or output.

Because of their property to broadcast across the whole network, side-ports may cause a producer net to block for a long time, e.g. if a lot of nets consume from this particular side-port and are waiting to be executed but only few resources are available to execute them. To prevent this situation, all side-ports are decoupled at the output. Hence a component can never be blocked due to back-pressure through a side-port.

### 6.3.4 Net Interface

All ports at a net interface are open, i.e. available to form a connection with ports of other nets. The set of connected ports $\mathcal{P}^H$ is hidden from the interface of a net. The interface of a net $N$ is defined by Definition 6.3.

**Definition 6.3** (Net Interface)**.** *A net $N$ is defined by the tuple*

$$\mathcal{P}(N) = \langle \mathcal{P}_0^I, \mathcal{P}_0^O, \mathcal{P}_L^I, \mathcal{P}_L^O, \mathcal{P}_R^I, \mathcal{P}_R^O, \mathcal{P}_S \rangle$$

*where each element is defined as follows (where $\mathcal{P}^H$ is the set of connected ports, hidden from the interface):*

- $\mathcal{P}_0^I = \hat{\mathcal{P}}_0^I \setminus \mathcal{P}^H$*: the set of non-grouped input ports of $N$*
- $\mathcal{P}_0^O = \hat{\mathcal{P}}_0^O \setminus \mathcal{P}^H$*: the set of non-grouped output ports of $N$*
- $\mathcal{P}_L^I$*: the set of left-grouped input ports of $N$*
- $\mathcal{P}_L^O$*: the set of left-grouped output ports of $N$*
- $\mathcal{P}_R^I$*: the set of right-grouped input ports of $N$*
- $\mathcal{P}_R^O$*: the set of right-grouped output ports of $N$*
- $\mathcal{P}_S$*: the set of side-grouped ports of $N$. For further information refer to Definition 6.2 in Section 6.3.3*

### 6.3.5 Net Declaration and Prototyping

Every net must be declared before it is used. A net declaration is either a box declaration that is assigned to a symbol (see Section 6.2), a wrapper declaration (see Section 6.4.6), a net assignment (see Section 6.4.1), or a net prototype. A net prototype allows to define the interface of a net. This can be used as a check, in order to see whether a network declaration returns the expected net. In order to prototype a net, the keyword `net` is used, followed by the name of the net and the net port declaration. The following net prototype describes the interface of a net `N`.

```
N = net( left in x, right out x )
```

For net prototypes it is mandatory to always provide a port collection for each port. For a net prototype to be valid, the port list of the net prototype must match to the port list of the corresponding net: Let $\mathcal{P}_{Np}$ be the set of ports of a net prototype $N$ and $\mathcal{P}_{Nn}$ be the set of ports of a net $N$ then it must hold that

$$\begin{aligned} &\forall p_i \in \mathcal{P}_{Np}, \exists p_j \in \mathcal{P}_{Nn} \;.\; (mode(p_i) = mode(p_j)) \;\wedge\; (col(p_i) = col(p_j)) \;\wedge\; (p_i = p_j) \\ \wedge\; &\forall p_i \in \mathcal{P}_{Nn}, \exists p_j \in \mathcal{P}_{Np} \;.\; (mode(p_i) = mode(p_j)) \;\wedge\; (col(p_i) = col(p_j)) \;\wedge\; (p_i = p_j) \end{aligned} \tag{6.8}$$

The complete grammar for net declarations and net assignments is provided in the following. Note that every Streamix scope must have one and only one `connect` statement followed by a net.

| | | |
|---|---|---|
| ⟨*program*⟩ | ⇒ | ⟨*net_decls*⟩ **`connect`** ⟨*net*⟩ |
| ⟨*net_decls*⟩ | ⇒ | [ ⟨*net_decl*⟩ ]* |
| ⟨*net_decl*⟩ | ⇒ | ⟨*box_assign*⟩ |
| | \| | ⟨*net_assign*⟩ |
| | \| | ⟨*net_proto*⟩ |
| | \| | ⟨*wrap_decl*⟩ |
| ⟨*net_proto*⟩ | ⇒ | **`net`** ⟨*net_name*⟩ **`(`** ⟨*net_port_list*⟩ **`)`** |
| ⟨*net_port_list*⟩ | ⇒ | ⟨*net_port_decl*⟩ [ **`,`** ⟨*net_port_decl*⟩ ]* |
| ⟨*net_port_decl*⟩ | ⇒ | ⟨*port_collection*⟩ ⟨*port_mode*⟩ ⟨*port_name*⟩ |
| ⟨*port_collection*⟩ | ⇒ | **`left`** |
| | \| | **`right`** |
| | \| | **`side`** |
| ⟨*port_mode*⟩ | ⇒ | **`in`** |
| | \| | **`out`** |
| ⟨*net_name*⟩ | ⇒ | identifier |
| ⟨*port_name*⟩ | ⇒ | identifier |

## 6.4 Network Composition

This section describes the operators that allow to interconnect nets in a structured manner. I propose two basic grouping operators that allow to combine two nets by either forcing a connection or preventing a connection. The connective grouping is called *serial composition* and the non-connective grouping is called *parallel composition*. For both operators, there exist two variants each with a slightly different connection semantics. A third type of network operator, called *wrapper*, allows scoping and reorganisation of port assignments. A forth type allows to control the execution rate of nets and the communication rate of message tokens.

The focus of the network operators is twofold:

1. The operators aim at structuring the network by providing a sense of locality, meaning that information necessary to understand a local part of the network is kept local.
2. Connections between nets are to be kept explicit and not hidden by an implicit connection semantics of the operators.

### 6.4.1 Net Assignments

In Streamix, a net is either a *composed net* or an *abstract net*. When composing nets with network operators, a composed net is created. An abstract net is either the instance of a box, a wrapper, or a net assignment. Consequently, when assigning a composed net to a net symbol, this net symbol represents an abstract net that can be further instantiated in network compositions. A net is assigned to a net symbol by the assign operator '='. The Streamix grammar provides an overview on all network operators.

```
⟨net_assign⟩      ⇒ ⟨net_name⟩ = ⟨net⟩
⟨net⟩             ⇒ ⟨abstract_net⟩
                  | ⟨composed_net⟩
⟨abstract_net⟩    ⇒ ⟨box_name⟩
                  | ⟨net_name⟩
                  | ⟨wrap_name⟩
⟨composed_net⟩    ⇒ ⟨net⟩ . ⟨net⟩
                  | ⟨net⟩ : ⟨net⟩
                  | ⟨net⟩ | ⟨net⟩
                  | ⟨net⟩ ! ⟨net⟩
                  | ( ⟨net⟩ )
                  | tt [ ⟨time_decl⟩ ] ( ⟨net⟩ )
                  | tb [ ⟨time_decl⟩ ] ( ⟨net⟩ )
⟨time_decl⟩       ⇒ [ ⟨time⟩ s ] [ ⟨time⟩ ms ] [ ⟨time⟩ us ] [ ⟨time⟩ ns ]
⟨box_name⟩        ⇒ identifier
⟨net_name⟩        ⇒ identifier
⟨wrap_name⟩       ⇒ identifier
```

$\langle time \rangle \qquad \Rightarrow \mathbb{Z}^+$

The differentiation between abstract and composed nets is an important aspect that allows to change the semantics of the composition operators, depending on whether an operator is applied on abstract or composed nets. This occurs when building a composed net with both, parallel and serial composition operators and when applying rate-control operators on nets. I discuss the former case in Section 6.4.4 where the operator precedence is described and the latter case in Section 6.4.5 where rate-control operators are described.

### 6.4.2 Serial Composition

The serial composition is a grouping operation enforcing connections between two operands. Two types of serial compositions are possible: one that is enforcing local connections and one that is less strict and allows bypassing. The former uses the operator '.' whereas the latter uses the operator ':'.

The port connection semantics is equivalent for both serial combination operators. It is defined by Definition 6.4.

**Definition 6.4** (Connection of Serial Composition)**.** *The port connections of a serial combination where $N_1$ is the left operand and $N_2$ is the right operand is defined by the set*

$$\mathcal{P}^H = \mathcal{P}^H_0 \cup \mathcal{P}^H_*$$

*where $\mathcal{P}^H_0$ and $\mathcal{P}^H_*$ are defined as follows, with $\mathcal{P}^H_F$ defined in Definition 6.1*

- $\mathcal{P}^H_0 = (\mathcal{P}^I_0(N_1) \cap \mathcal{P}^O_0(N_2)) \cup (\mathcal{P}^O_0(N_1) \cap \mathcal{P}^I_0(N_2)) \cup \mathcal{P}^H_F$*: the set of non-grouped hidden ports of $N$ is the intersection of all ports of opposite mode that are not grouped in any collection of $N_1$ and $N_2$, excluding self-loops.*
- $\mathcal{P}^H_* = (\mathcal{P}^I_R(N_1) \cap \mathcal{P}^O_L(N_2)) \cup (\mathcal{P}^O_R(N_1) \cap \mathcal{P}^I_L(N_2))$*: the set of grouped hidden ports of $N$ is the intersection of all ports of opposite mode that are grouped in the right collection of $N_1$ and in the left collection of $N_2$.*

*No implicit routing nodes are spawned by a serial combination (side-ports aside), hence, the following must hold:*

- $P^I_0(N_1) \cap P^I_0(N_2) = \emptyset$
- $P^O_0(N_1) \cap P^O_0(N_2) = \emptyset$

- $P_L^I(N_1) \cap P_L^I(N_2) = \emptyset$
- $P_L^O(N_1) \cap P_L^O(N_2) = \emptyset$
- $P_R^I(N_1) \cap P_R^I(N_2) = \emptyset$
- $P_R^O(N_1) \cap P_R^O(N_2) = \emptyset$

The difference between the locality enforcing serial connection and the one allowing bypasses is that all ports of the resulting net of the former $N_{local} = N_1 \ . \ N_2$ (as defined in Definition 6.5) are assigned to a port collection, i.e. $P_0(N_{local}) = \emptyset$, while the resulting net of the latter $N_{bypasss} = N_1 \ : \ N_2$ (as defined in Definition 6.6) can have unassigned ports.

**Definition 6.5** (Net Interface of Locality Enforcing Serial Composition)**.** *The resulting net of a locality enforcing serial composition of two boxes $N_1$ and $N_2$, written as $N_1 \ . \ N_2$, is defined by the tuple*

$$\mathcal{P}(N_1 \ . \ N_2) = \langle \mathcal{P}_0^I, \mathcal{P}_0^O, \mathcal{P}_L^I, \mathcal{P}_L^O, \mathcal{P}_R^I, \mathcal{P}_R^O, \mathcal{P}_S \rangle$$

*where each element is defined as follows, with $\mathcal{P}_0^H$ and $\mathcal{P}_*^H$ defined in Definition 6.4:*

- $\mathcal{P}_0^I = \emptyset$*: the set of non-grouped input ports of $N$ is empty.*
- $\mathcal{P}_0^O = \emptyset$*: the set of non-grouped output ports of $N$ is empty.*
- $\mathcal{P}_L^I = \mathcal{P}_L^I(N_1) \cup (\mathcal{P}_0^I(N_1) \setminus \mathcal{P}_0^H)$*: the set of left-grouped input ports of $N$ is the union of left-grouped input ports of $N_1$ with non-grouped input ports of $N_1$, excluding non-grouped hidden ports of $N$.*
- $\mathcal{P}_L^O = \mathcal{P}_L^O(N_1) \cup (\mathcal{P}_0^O(N_1) \setminus \mathcal{P}_0^H)$*: the set of left-grouped output ports of $N$ is the union of left-grouped output ports of $N_1$ with non-grouped output ports of $N_1$, excluding non-grouped hidden ports of $N$.*
- $\mathcal{P}_R^I = \mathcal{P}_R^I(N_2) \cup (\mathcal{P}_0^I(N_2) \setminus \mathcal{P}_0^H)$*: the set of right-grouped input ports of $N$ is the union of right-grouped input ports of $N_2$ with non-grouped input ports of $N_2$, excluding non-grouped hidden ports of $N$.*
- $\mathcal{P}_R^O = \mathcal{P}_R^O(N_2) \cup (\mathcal{P}_0^O(N_2) \setminus \mathcal{P}_0^H)$*: the set of right-grouped output ports of $N$ is the union of right-grouped output ports of $N_2$ with non-grouped output ports of $N_2$, excluding non-grouped hidden ports of $N$.*
- $\mathcal{P}_S$*: for further information refer to Definition 6.2 in Section 6.3.3*

*The operator '.' is non-associative and non-commutative. A locally enforcing serial composition is valid iff the following conditions are satisfied:*

- $P_R(N_1) \setminus P_*^H = \emptyset$
- $P_L(N_2) \setminus P_*^H = \emptyset$

The schematic representation of a locality enforcing serial composition is shown in Figure 6.8. The Figure illustrates that all ports of the composition are assigned to its immediate neighbour, i.e. no net can be bypassed. Note, however, that no direction is indicated which means that message tokens can flow from the right operand to the left operand and vice versa.

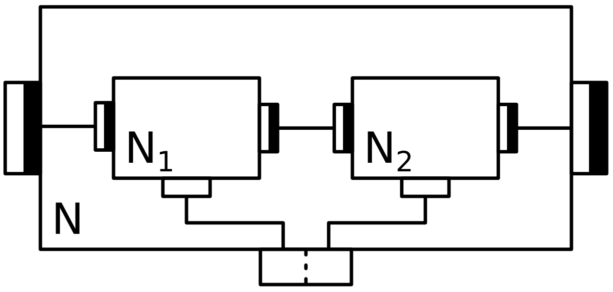


FIGURE 6.8: Locality enforcing serial composition of two nets $N_1$ and $N_2$, written as $N = N_1 \ . \ N_2$.

**Definition 6.6** (Net Interface of Bypassing Serial Composition). *The resulting net of a serial composition with bypassing of two boxes $N_1$ and $N_2$, written as $N_1 \ : \ N_2$, is defined by the tuple*

$$\mathcal{P}(N_1 \ : \ N_2) = \langle \mathcal{P}_0^I, \mathcal{P}_0^O, \mathcal{P}_L^I, \mathcal{P}_L^O, \mathcal{P}_R^I, \mathcal{P}_R^O, \mathcal{P}_S \rangle$$

*where each element is defined as follows, with $\mathcal{P}_0^H$ and $\mathcal{P}_*^H$ defined in Definition 6.4:*

- $\mathcal{P}_0^I = (\mathcal{P}_0^I(N_1) \setminus \mathcal{P}_0^H) \cup (\mathcal{P}_0^I(N_2) \setminus \mathcal{P}_0^H)$*: the set of non-grouped input ports of $N$ is the union of non-grouped input ports of $N_1$ and $N_2$, excluding hidden ports of $N$.*
- $\mathcal{P}_0^O = (\mathcal{P}_0^O(N_1) \setminus \mathcal{P}_0^H) \cup (\mathcal{P}_0^O(N_2) \setminus \mathcal{P}_0^H)$*: the set of non-grouped output ports of $N$ is the union of non-grouped output ports of $N_1$ and $N_2$, excluding hidden ports of $N$.*
- $\mathcal{P}_L^I = \mathcal{P}_L^I(N_1) \cup (\mathcal{P}_L^I(N_2) \setminus \mathcal{P}_*^H)$*: the set of left-grouped input ports of $N$ is the union of left-grouped input ports of $N_1$ with left-grouped input ports of $N_2$, excluding grouped hidden ports of $N$.*

- $\mathcal{P}_L^O = \mathcal{P}_L^O(N_1) \cup (\mathcal{P}_L^O(N_2) \setminus \mathcal{P}_*^H)$: *the set of left-grouped output ports of $N$ is the union of left-grouped output ports of $N_1$ with left-grouped output ports of $N_2$, excluding grouped hidden ports of $N$.*
- $\mathcal{P}_R^I = (\mathcal{P}_R^I(N_1) \setminus \mathcal{P}_*^H) \cup \mathcal{P}_R^I(N_2)$: *the set of right-grouped input ports of $N$ is the union of right-grouped input ports of $N_1$ with right-grouped input ports of $N_2$, excluding grouped hidden ports of $N$.*
- $\mathcal{P}_R^O = (\mathcal{P}_R^O(N_1) \setminus \mathcal{P}_*^H) \cup \mathcal{P}_R^O(N_2)$: *the set of right-grouped output ports of $N$ is the union of right-grouped output ports of $N_1$ with right-grouped output ports of $N_2$, excluding grouped hidden ports of $N$.*
- $\mathcal{P}_S$: *for further information refer to Definition 6.2 in Section 6.3.3*

*The operator ':' is associative and non-commutative.*

The schematic representation of a serial composition that allows bypassing is shown in Figure 6.9. Note that in contrast to Figure 6.8, where a locality enforcing serial composition is illustrated, here, bypassing is possible.

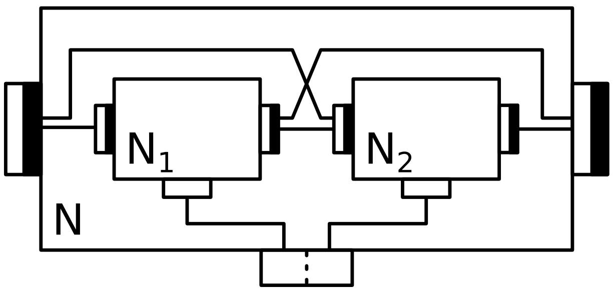


FIGURE 6.9: A serial composition of two nets $N_1$ and $N_2$, written as $N = N_1 \ : \ N_2$ where bypassing is allowed.

### 6.4.3 Parallel Composition

The parallel composition is a grouping operation of two operands which prevents any implicit connection between the two operands, independent of the names or the groupings of their ports. Two types of parallel composition are possible, a deterministic version which uses the operator '|' and a non-deterministic version which uses the operator '!'.

No implicit port connection is allowed with the parallel composition of two nets $N_1$ and $N_2$ (with the exception of side-ports). Hence, the conditions defined in Equation 6.9

must be satisfied.

$$\begin{aligned}\mathcal{P}_0^O(N_1) \cap \mathcal{P}_0^I(N_2) &= \emptyset \\ \mathcal{P}_L^O(N_1) \cap \mathcal{P}_L^I(N_2) &= \emptyset \\ \mathcal{P}_R^O(N_1) \cap \mathcal{P}_R^I(N_2) &= \emptyset \end{aligned} \tag{6.9}$$

Further, as Streamix targets CPSs where predictability is important, the parallel composition operator '|' does not allow to implicitly instantiate a summing node as described in Section 6.2.2.2. This is because such a summing node has a non-deterministic behaviour. Hence, for a deterministic parallel composition of two nets $N_1$ and $N_2$ the condition as defined in Equation 6.10 must hold.

$$\begin{aligned}\mathcal{P}_0^O(N_1) \cap \mathcal{P}_0^O(N_2) &= \emptyset \\ \mathcal{P}_L^O(N_1) \cap \mathcal{P}_L^O(N_2) &= \emptyset \\ \mathcal{P}_R^O(N_1) \cap \mathcal{P}_R^O(N_2) &= \emptyset \end{aligned} \tag{6.10}$$

Diverging nodes as described in Section 6.2.2.2, however, have a deterministic behaviour and can be spawned implicitly in a parallel composition. The resulting net of a deterministic parallel composition is defined in Definition 6.7.

**Definition 6.7** (Net Interface of Deterministic Parallel Composition)**.** *The resulting net of a deterministic parallel composition of two nets $N_1$ and $N_2$, written as $N = N_1 \mid N_2$, is defined by the tuple*

$$\mathcal{P}(N_1 \mid N_2) = \langle \mathcal{P}_0^I, \mathcal{P}_0^O, \mathcal{P}_L^I, \mathcal{P}_L^O, \mathcal{P}_R^I, \mathcal{P}_R^O, \mathcal{P}_S \rangle$$

*where each element is defined as follows:*

- $\mathcal{P}_0^I = \mathcal{P}_0^I(N_1) \cup \mathcal{P}_0^I(N_2)$*: the set of non-grouped input ports of $N$ is the union of non-grouped input ports of $N_1$ and $N_2$.*
- $\mathcal{P}_0^O = \mathcal{P}_0^O(N_1) \cup \mathcal{P}_0^O(N_2)$*: the set of non-grouped output ports of $N$ is the union of non-grouped output ports of $N_1$ and $N_2$.*
- $\mathcal{P}_L^I = \mathcal{P}_L^I(N_1) \cup \mathcal{P}_L^I(N_2)$*: the set of left-grouped input ports of $N$ is the union of left-grouped input ports of $N_1$ and $N_2$.*
- $\mathcal{P}_L^O = \mathcal{P}_L^O(N_1) \cup \mathcal{P}_L^O(N_2)$*: the set of left-grouped output ports of $N$ is the union of left-grouped output ports of $N_1$ and $N_2$.*
- $\mathcal{P}_R^I = \mathcal{P}_R^I(N_1) \cup \mathcal{P}_R^I(N_2)$*: the set of right-grouped input ports of $N$ is the union of right-grouped input ports of $N_1$ and $N_2$.*

- $\mathcal{P}_R^O = \mathcal{P}_R^O(N_1) \cup \mathcal{P}_R^O(N_2)$*: the set of right-grouped output ports of* $N$ *is the union of right-grouped output ports of* $N_1$ *and* $N_2$*.*

- $\mathcal{P}_S$*: for further information refer to Definition 6.2 in Section 6.3.3*

Non-determinism can not always be avoided. For this reason, Streamix offers a non-deterministic parallel composition operator '!' where the condition defined in Equation 6.10 is *no* requirement. The resulting net of a non-deterministic parallel composition is equivalent to the deterministic one as defined in Definition 6.8. Further, as with the deterministic parallel composition, implicit connections are prevented, hence the condition defined in Equation 6.9 must be satisfied.

**Definition 6.8** (Net Interface of Non-deterministic Parallel Composition)**.** *The resulting net of a non-deterministic parallel composition of two nets* $N_1$ *and* $N_2$*, written as* $N = N_1 \ ! \ N_2$*, is defined by the tuple*

$$\mathcal{P}(N_1 \ ! \ N_2) = \langle \mathcal{P}_0^I, \mathcal{P}_0^O, \mathcal{P}_L^I, \mathcal{P}_L^O, \mathcal{P}_R^I, \mathcal{P}_R^O, \mathcal{P}_S \rangle$$

*where each element is defined by Definition 6.7*

Note that the name of the parallel network combinator can be misleading as it suggests that two nets are configured in parallel and can be executed in parallel. This is not necessarily the case because the channels connecting nets can be of arbitrary direction and may create arbitrary dependencies.

The parallel composition is schematically represented in Figure 6.10. It is important to note, that in a parallel composition no connection is established between any input and output ports, excluding ports in side-port collections.

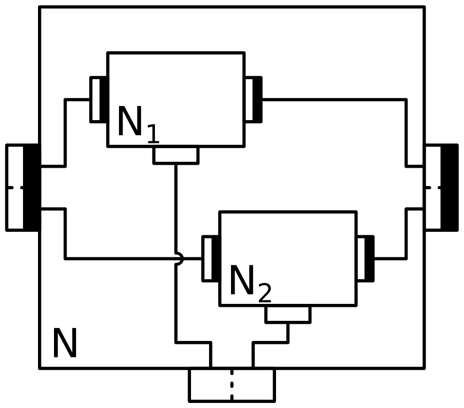


FIGURE 6.10: Parallel composition of two nets $N_1$ and $N_2$ written as $N_1 \mid N_2$.

### 6.4.4 Operator Precedence

The order of precedence of the operators is defined as follows: The serial composition precedes the parallel composition. Further, the strict versions of the composition operators precede the less strict versions. Together, this defines the operator precedence:

$$. \;>\; : \;>\; | \;>\; !$$

An example where the precedence is applied is shown in Figure 6.11. On the left side two sequences $A_1 \,.\, B_1 \,.\, C_1$ and $A_2 \,.\, B_2 \,.\, C_2$ are created which are then joined by the parallel composition. On the right side the parallel compositions are enforced by the parentheses, resulting in a serial composition with full connectivity.

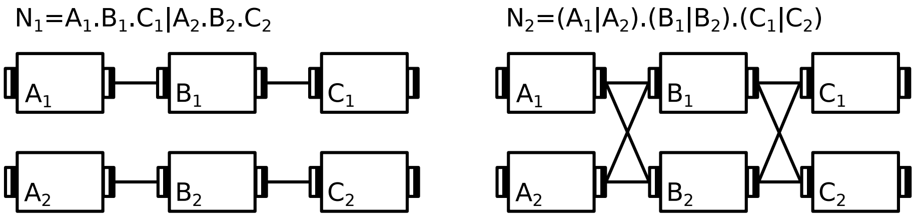


FIGURE 6.11: Two examples of connection graphs where the operator precedence is illustrated.

As described in Section 6.4.1, I distinguish between *abstract* nets and *composed* nets. Further, I distinguish between operators that enforce port connections, i.e. the serial composition operators '.' and ':', and operators that prevent port connections, i.e. the parallel composition operators '|' and '!'. The enforced connectivity property of serial composition operators is distributive over parallel composition operators in composed nets. This means that given a composed net as depicted on the right side of Figure 6.11, the full connectivity is mandatory for the composed net to be valid, e.g. there must be at least one valid port connection between net $A_1$ and net $B_1$ as well as between $A_1$ and $B_2$. In other words, if an operator implies a connection between two nets at least one pair of ports must form a connection between the two nets. Otherwise, the two nets are incompatible and cannot be connected with the operator in question.

To check this property of serial composition operators I define the predicate $abst(\mathcal{P})$ which returns a set of mutual distinctive abstract nets, where the ports in port set $\mathcal{P}$ belong to. Formally, the condition defined in Equation 6.11 must hold for a serial composition operation $N_1 \,.\, N_2$ or $N_1 : N_2$ to be valid.

$$\forall net_i \in abst(\mathcal{P}(N_1)),\ \forall net_j \in abst(\mathcal{P}(N_2))\ .\ \mathcal{P}^H(net_i) \cap \mathcal{P}^H(net_j) \neq \emptyset \tag{6.11}$$

### 6.4.5 Time-controlled Nets

Streamix allows to control the execution of nets with time annotations. Based on the model described in Chapter 4 the triggering semantics of a net can be changed from event-triggered, where message tokens trigger nets upon arrival on an input port, to time-triggered, where a periodic clock signal is triggering the net. This is achieved with the following Streamix syntax. Let `N` be a defined Streamix net, then the expression

```
tt[1s](N)
```

changes the triggering semantics of `N` from event-triggered to time-triggered where a clock signal causes the net `N` to trigger every second. This is achieved by replacing all channels, connected to `N`, by synchronised temporal firewalls (see Section 6.2.2.3). The period of each temporal firewall is specified by the time argument of the time-triggered operator (one second in the example above). Note that the `tt` operator has no effect on a net with no interaction with its environment, i.e. if $\mathcal{P}(N) = \emptyset$.

Two time-triggered nets $N_1$ and $N_2$ can be composed into a new net $N$ with the operators '.', ':', '|', and '!' as defined in Definition 6.5, Definition 6.6, Definition 6.7, and Definition 6.8, respectively. When two time-triggered nets $N_1$ and $N_2$, each framed by temporal firewalls, are composed with a serial composition operator (see Section 6.4.2) connecting temporal firewalls are merged if their respective period is equivalent.

As an example let's consider the two boxes

```
N1 = box fn1( in x, out y )
N2 = box fn2( in y, out z )
```

and instantiate them in a network as follows

```
tt[1s](N1).tt[1s](N2)
```

The resulting network is illustrated in Figure 6.12 where the channel $y$ between $N_1$ and $N_2$ is a single temporal firewall.

A detailed illustration of the resulting network is depicted in Figure 6.12. Note that the connections from the time-triggered net to the temporal firewalls are internally changed in order to avoid naming conflicts.

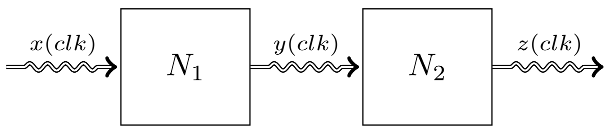


FIGURE 6.12: A time-triggered instance of the box `N1` and `N2` where all channels are temporal firewalls with period $clk = 1s$.

However, when instantiating the boxes $N_1$ and $N_2$ in a network where the triggering rates are different, temporal firewalls cannot be merged and are cascaded.

```
tt[1s](N1).tt[2s](N2)
```

This situation is illustrated in Figure 6.13.

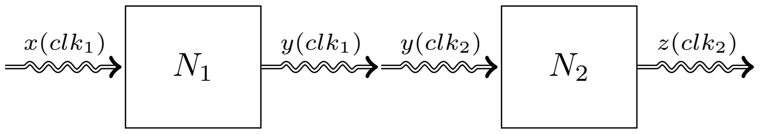


FIGURE 6.13: A time-triggered instance of the box `N1` and `N2` where the channel $y$ is modelled by two temporal firewalls, with the periods $clk_1 = 1s$ and $clk_2 = 2s$, respectively.

Another way of controlling the triggering semantics by a time annotation is the keyword `tb`. It allows to impose a rate bound on each input channel of a net. This causes to spawn a rate bounded FIFO (see Section 6.2.2.4) instead of a normal FIFO. For example, the following syntax imposes a rate bound on each input channel of net `N`.

```
tb[200ms](N)
```

Here the minimal time interval between two consecutive message tokens is set to 200 milliseconds.

Both operators `tt` and `tb` are distributed on all abstract nets when applied on a composed net. Consequently, the network defined by the following expression

```
tt[1s](N1.N2)
```

is equivalent to the network

```
tt[1s](N1).tt[1s](N2)
```

of which the resulting network is illustrated in Figure 6.12.

### 6.4.6 Wrapper

A *wrapper* $W$ has an interface and a body. The interface consists of a set of ports $\mathcal{P}(W)$. The body can contain all types of net declarations as listed in Section 6.3.5 and must define one network $N_W$ where the network interface consists of a set of ports $\mathcal{P}(N_W)$. This one network is declared with the keyword `connect`.

A wrapper acts as a static scope for all names defined inside the wrapper body. These names are only visible in the scope of the wrapper and any wrapper that is declared in this scope. As only ports that are explicitly defined in the interface of the wrapper are visible outside of the scope of the wrapper, also side ports have to be propagated manually to be accessible outside. Hence, the wrapper allows to create and control subnet broadcasts of side-ports. The instance of a wrapper declaration is a net and can be used in any network operator.

As an example, let's declare the following wrapper `W`.

```
wrapper W( in x, out y ) {
  A = box fa( in x, out z )
  B = box fb( in z, out y )
  connect A.B
} net( left in x, right out y )
```

Here, the boxes `A` and `B` are not accessible outside of the body of the wrapper `W`. Note that the interface of the net prototype corresponds to the interface of the net defined after the `connect` statement. If this does not match, the wrapper is invalid. However, the interface of the wrapper is not required to match with the interface of the net, as it is the case in the example above.

The purpose of a wrapper is to allow the construction of arbitrary networks and to provide a scoping facility. In order to create arbitrary networks, the wrapper allows to

1. create a connection between a port $p_i \in \mathcal{P}(N_W)$ and a port $p_j \in \mathcal{P}(W)$
2. create a connection between a port $p_i \in \mathcal{P}(W)$ and a port $p_j \in \mathcal{P}(W)$

A schematic representation of the wrapper connections is depicted in Figure 6.14. The grey circle around the net $N$ represents the possibility of rearranging its port collections on the wrapper interface $W$.

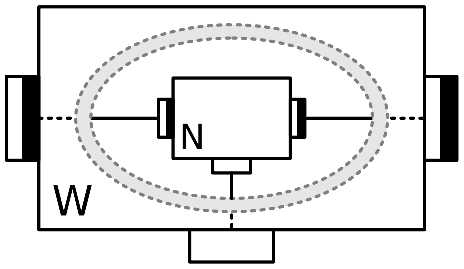


FIGURE 6.14: A schematic representation of the scoping mechanism where ports from any collection of the net $N$ can be connected to ports from any collection of the wrapper $W$.

The different port connections inside a wrapper are implicitly established by name and mode matching. In order to achieve arbitrary connections, a wrapper allows to rename ports by appending a list of names to a port declaration. As an example, let's declare the following wrapper `Wp`.

```
wrapper Wp( in a(x), out b(y) ) {
  A = box fa( in x, out z )
  B = box fb( in z, out y )
  connect A.B
} net( left in x, right out y )
```

Here, the ports `a` and `b` in $\mathcal{P}(Wp)$ are renamed to `x` and `y`, respectively, in order to connect to the corresponding ports in $\mathcal{P}(N_{Wp})$.

If multiple ports are connected, routing nodes are spawned accordingly. For example, in the following wrapper declaration a diverging node and a summing node is spawned:

```
wrapper Wf( in a(w, y), out b(x, z) ) {
  A = box fa( in w, out x )
  B = box fb( in y, out z )
  connect A|B
} net( left in w, left in y, right out x, right out z )
```

The port $a \in \mathcal{P}(Wf)$ connects to a diverging node with two output ports, connecting to $w \in \mathcal{P}(A)$ and $y \in \mathcal{P}(B)$, respectively. On the right side, the ports $x \in \mathcal{P}(A)$ and $z \in \mathcal{P}(B)$ connect to a summing node and the output port of the summing node connects to $b \in \mathcal{P}(Wf)$.

Let $\mathcal{P}_{int}(p)$ describe the set of port names that are associated to the wrapper port $p$ ($p \notin \mathcal{P}(W) \rightarrow \mathcal{P}_{int}(p) = \emptyset$). Then the predicate $wrap\_name\_match(p, p')$, as defined in Equation 6.12 returns true if two ports $p$ and $p'$ have matching names.

$$wrap\_name\_match(p,p') \ := \ (p = p') \ \vee \ p \in \mathcal{P}_{int}(p') \ \vee \ p' \in \mathcal{P}_{int}(p) \tag{6.12}$$

Using the predicate $wrap\_name\_match(p, p')$ defined in Equation 6.12 I further define the two following predicates: The predicate $wrap\_connect\_by(p, p')$, as defined in Equation 6.13, returns true if two ports $p$ and $p'$ in a wrapper interface can connect.

$$\begin{aligned} wrap\_connect\_by(p,p') \ := \ & wrap\_name\_match(p,p') \\ & \wedge \ (p \in \mathcal{P}(W) \ \wedge \ p' \in \mathcal{P}(W) \ \wedge \ mode(p) \neq mode(p')) \end{aligned} \tag{6.13}$$

The predicate $wrap\_connect\_net(p, p')$, as defined in Equation 6.14, returns true if two ports $p$ and $p'$ can connect where one is part of the wrapper interface and one is part of the internal net of the wrapper.

$$\begin{aligned} wrap\_connect\_net(p,p') \ &:= \ wrap\_name\_match(p,p') \\ &\qquad \wedge \ (p \in \mathcal{P}(N_W) \ \wedge \ p' \in \mathcal{P}(W) \ \wedge \ mode(p) = mode(p')) \end{aligned} \tag{6.14}$$

Note that port connections in a wrapper are independent of their port collection. This means that a wrapper can regroup ports into arbitrary port collections. The complete grammar of the wrapper is defined as follows.

| | | |
|---|---|---|
| ⟨*wrap_decl*⟩ | ⇒ | **`wrapper`** ⟨*wrap_name*⟩ **`(`** ⟨*wrap_port_list*⟩ **`)`** **`{`** ⟨*program*⟩ **`}`** **`net`** **`(`** ⟨*net_port_list*⟩ **`)`** |
| ⟨*wrap_port_list*⟩ | ⇒ | ⟨*wrap_port_decl*⟩ *[* **`,`** ⟨*wrap_port_decl*⟩ *]** |
| ⟨*wrap_port_decl*⟩ | ⇒ | *[* ⟨*port_collection*⟩ *]* ⟨*port_mode*⟩ ⟨*port_name*⟩ *[* **`(`** ⟨*alt_port_list*⟩ **`)`** *]* |
| ⟨*alt_port_list*⟩ | ⇒ | ⟨*port_name*⟩ *[* **`,`** ⟨*port_name*⟩ *]** |
| ⟨*program*⟩ | ⇒ | ⟨*net_decls*⟩ **`connect`** ⟨*net*⟩ |
| ⟨*port_collection*⟩ | ⇒ | **`left`** |
| | \| | **`right`** |
| | \| | **`side`** |
| ⟨*port_mode*⟩ | ⇒ | **`in`** |
| | \| | **`out`** |
| ⟨*port_name*⟩ | ⇒ | identifier |
| ⟨*wrap_name*⟩ | ⇒ | identifier |

## 6.5 Describing a Cyber-physical System with Streamix

In this section I demonstrate vehicle platooning as an example of a CPS with different interactions of components. The basic idea of vehicle platooning is to coordinate the cruising speed of, in series driving, vehicles to achieve a more resourceful driving. Bergenhem et al. describe different types of platooning systems [66]. In Figure 6.15 I show a possible interaction scenario of different car components, relevant to vehicle platooning, with a particular focus on the braking mechanism only.

The bottom elements represent more elementary braking components, having higher criticality than those more on the top level. The *anti-lock braking system* (ABS) receives a control signal for a desired braking action but the ABS then decides on its own when to assert and release braking pressure (Break) based on feedback over the revolution

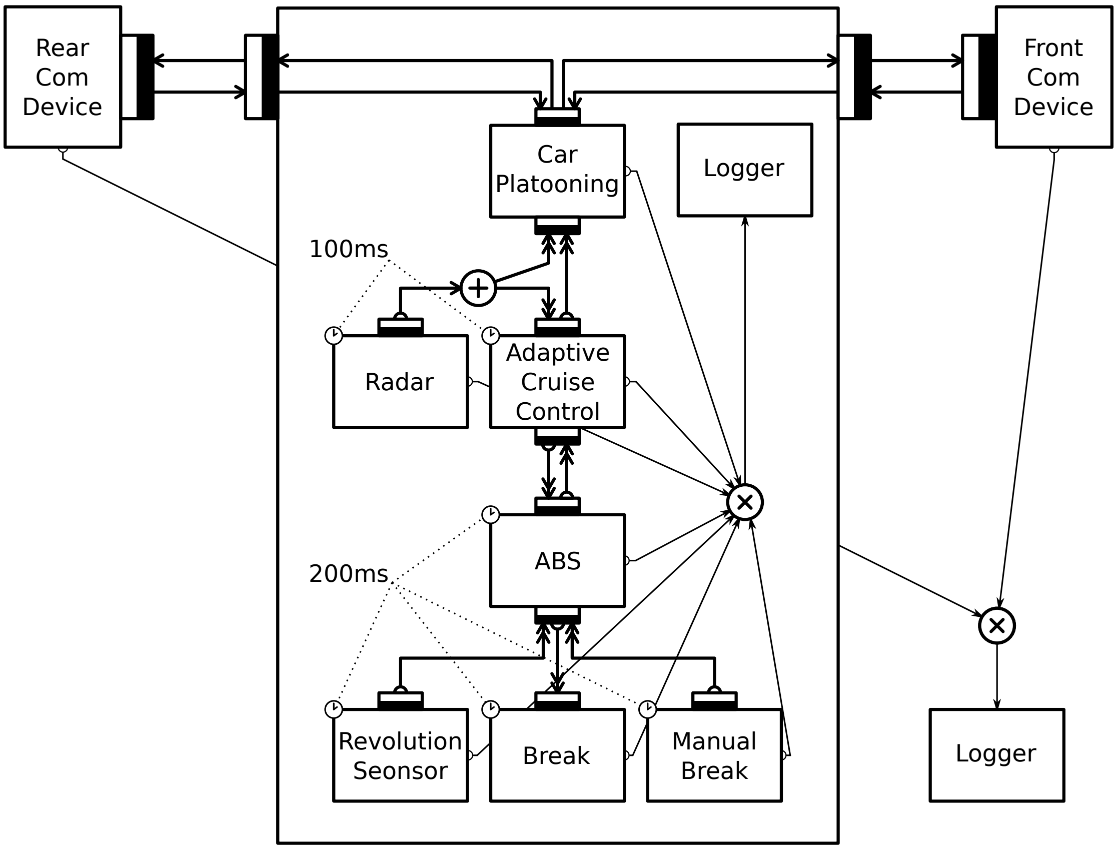


Figure 6.15: Structured representation of the car platooning example.

sensors of the wheels. Besides the manual braking control we assume an adaptive cruise control system which uses a radar to measure the distance to its front vehicle and starts automatic braking requests to keep a certain minimal distance. While the adaptive cruise control system is a safety feature, the car platooning system on top of it acts to optimise driving economy. The car platooning system may communicate with the car platooning system of other cars through the communication devices to achieve better efficiency.

What we see from this example is that communication between components is bi-directional and individual components act as reactive systems on their own. Such communication patterns cannot be mapped to acyclic directed computation graphs. However, the Streamix network operators achieve a structuring of the network by implicitly grouping components together and keeping port connections local.

Figure 6.16 describes the Streamix program of the car platooning application. Lines 1-6 describe the breaking control of the application where boxes are declared and instantiated: In line 6 a net assignment to the symbol `N_ABS` is performed and the boxes `RevolutionSensor`, `Break`, `ManualBreak`, and `ABS` are instantiated as nets and composed with the help of serial and parallel composition operators. The operator `tt`

```
1  RevolutionSensor = box f_rs( out speed, side out log )
2  Break = box fb( in break_abs, side out log )
3  ManualBreak = box f_mb( out break_cmd, side out log )
4  ABS = box f_abs( in speed, in break_cmd, in dc_abs,
      out break_abs, out abs, side out log )
5
6  N_ABS = tt[200ms]( ( RevolutionSensor | Break
      | ManualBreak ).ABS )
7
8  Radar = box f_ds( out dist, side out log )
9  AdaptiveCriuseControl = box f_dc( in abs, in dist_dc,
      out dc_cpa, out dc_abs, side out log )
10 wrapper W_Radar( out dist_dc( dist ),
      out dist_cpa( dist ), side out log ) {
11   connect Radar
12 } net( right out dist, side out log )
13
14 N_ACC = tt[100ms]( W_Radar:AdapdiveCruiseControl )
15
16 Logger = box f_log( side in log )
17 CarPlatooning = box f_cpa( decoupled in dist_cpa,
      decoupled in dc_cpa, in com_front_rcv,
      out com_front_send, in com_rear_rcv,
      out com_rear_send, side out log )
18
19 wrapper Control( out com_front_send, in com_front_rcv,
      out com_rear_send, in com_rear_rcv ) {
20   connect N_ABS.N_ACC.CarPlatooning|Logger
21 } net( right out com_front_send, right in com_front_rcv,
      right out com_rear_send, right in com_rear_rcv )
22
23 ComFront = box f_com( out com_front_rcv( com_rcv ),
      in com_front_send( com_send ), side out log )
24 ComRear = box f_com( out com_rear_rcv( com_rcv ),
      in com_rear_send( com_send ), side out log )
25
26 connect ComRear.Control.ComFront|Logger
```

FIGURE 6.16: The Streamix program describing the car platooning application depicted in Figure 6.15.

imposes a time-triggered semantics on each net of this composition where the triggering rate is set to 5Hz.

Lines 8-14 describe the adaptive cruise control application and the connected radar controller. A wrapper `W_Radar` (lines 10-12) is used to spawn a routing node that allows to copy the signal `dist` of the radar and serve it to the adaptive cruise control net and the car platooning net. In line 14 an instance of the box `AdaptiveCruiseControl` and the wrapper `W_Radar` is composed to a new net and assigned to the symbol `N_ACC`.

The two nets are instantiated as time-triggered nets with a trigger rate of 10Hz. Note that a bypassing serial composition operator is used. This allows to propagate the distance signal to the car platooning net (line 20).

Lines 16-21 describe the composition of the control system. A wrapper `Control` is used to restrict the broadcast of the side-port signal `log` which allows to connect a separate instance of the box `Logger` to the communication devices (line 26). Line 20 instantiates the net symbols `N_ABS` and `N_ACC` and the boxes `CarPlatooning` and `Logger` as nets. Note that the net instance `Logger` has no other connection than a side-port connection and is therefore composed to the network by a parallel composition operator. The box `CarPlatooning` is instantiated as an event-triggered net. However, as all the signals from the adaptive cruise control net are decoupled (line 17), the car platooning net is only triggered by message tokens arriving form the communication devices `ComFront` and `ComRear`.

Lines 23-26 describe the composition of the complete car platooning application, including the communication devices. Note that in line 26 the box `Logger` is again composed to the network to provide a separate logging instance only for the communication devices. Further note that the communication devices use the same implementation function `f_com` but port renaming is used to create two boxes with different signatures.

## 6.6 Discussion

The focus of the coordination language Streamix lies in providing a concise syntax to model reactive networks of computational components. Streamix offers the following contributions.

**Exogeneous coordination model with explicit timing semantics**
Streamix combines the exogenous coordination model, where the behavioural aspects are separated from aspects of coordination, with explicit timing semantics. This allows to apply the concept of *separation of concerns* ([65]) in the domain of CPSs. An exogenous coordination model is achieved with a black-box approach where boxes, possibly implemented by a third party and written with a separate programming language, are coordinated by network elements which connect boxes and impose implicit and explicit synchronisation on the boxes.

**Structured network composition for a reactive data processing model**
Streamix provides an inherent structured composition of the network with the use of

network operators and makes this feature available in the world of CPSs. In Streamix, the concept of grouping ports in two separate collections, called *left* and *right*, enables a concise description of network topologies based on reactive data processing.

**Explicit decoupling of box triggers and back-pressure to implement mixed-criticality systems**
The computing model of Streamix allows the modelling of mixed-criticality systems with the help of CCIs. CCIs allow interaction between systems of different criticality but prevent the interference of a lower critical component with a higher critical component. This is achieved by introducing a decoupling mechanism and an automatic derivation of channel implementations based on the chosen decoupling. The decoupling of ports is described in the coordination language and a computational component is oblivious of whether it is accessing a decoupled or a blocking channel.

## 6.7 Chapter Summary

In this chapter I introduced the exogenous coordination language Streamix. Streamix is an instantiation of the model PNSC, introduced in Chapter 3 and of its extension, introduced in Chapter 4. The language allows to instantiate CCIs and control triggering semantics of components, as described in Chapter 4, through dedicated operators and keywords. Different network composition operators allow to compose components in a hierarchical and structured manner. I demonstrated the expressiveness of Streamix by modelling a car platooning application with special focus on the braking mechanism.

# Chapter 7

# Toolchain for Streamix

In this chapter I will give some technical details about the implementation of the compiler of the coordination language Streamix, the permanent blocking analysis, a runtime system for Streamix programs, and a preprocessor for the runtime system. In order to build an application, with computational components being written in C/C++ and the coordination aspects being described by Streamix, the Streamix toolchain is used as described in the following list.

1. The Streamix compiler `smxc` takes a Streamix file as input and produces a network dependency graph, annotated with network information such as whether inputs and outputs are decoupled, clock rates, and whether boxes are pure and/or static. Further, for each user-defined computational component a graph file is produced, describing the Synchronous Interface Automaton (SIA) of the computational component. Optionally, the compiler takes user-defined SIA descriptions of computational components as an input if the interaction protocol of the computational component does not follow a default behaviour.

2. The Run-time System (RTS) preprocessor `smxrtsp` takes the network dependency graph, generated by the Streamix compiler, as an input and generates compilable C code, using function calls and macros defined in the Streamix runtime system library. Additionally, the RTS preprocessor generates the dependency graph of the Process Network with Synchronous Communication (PNSC) where only synchronous channels are used and First-in, First-out (FIFO) channels are represented as processes. It further produces SIA description files for implicitly generated PNSC processes such as channels and routing nodes.

3. The permanent blocking analysis `smxsia` takes the PNSC dependency graph and the interface descriptions of all processes as input (this includes automatically

generated SIA files by the compiler and the runtime system preprocessor as well as the user-defined SIAs passed to the Streamix compiler) and checks the network for potential permanent blocking situations.

4. Finally, the executable application code is created by compiling the computational component implementations, the generated files by the runtime system preprocessor, and by linking the runtime system library and the libraries used by the RTS.

Figure 7.1 provides a schematic overview of the build process described above. The white boxes represent the executables described above and the unframed elements represent inputs and outputs. The inputs in the grey area represent the inputs provided by the user.

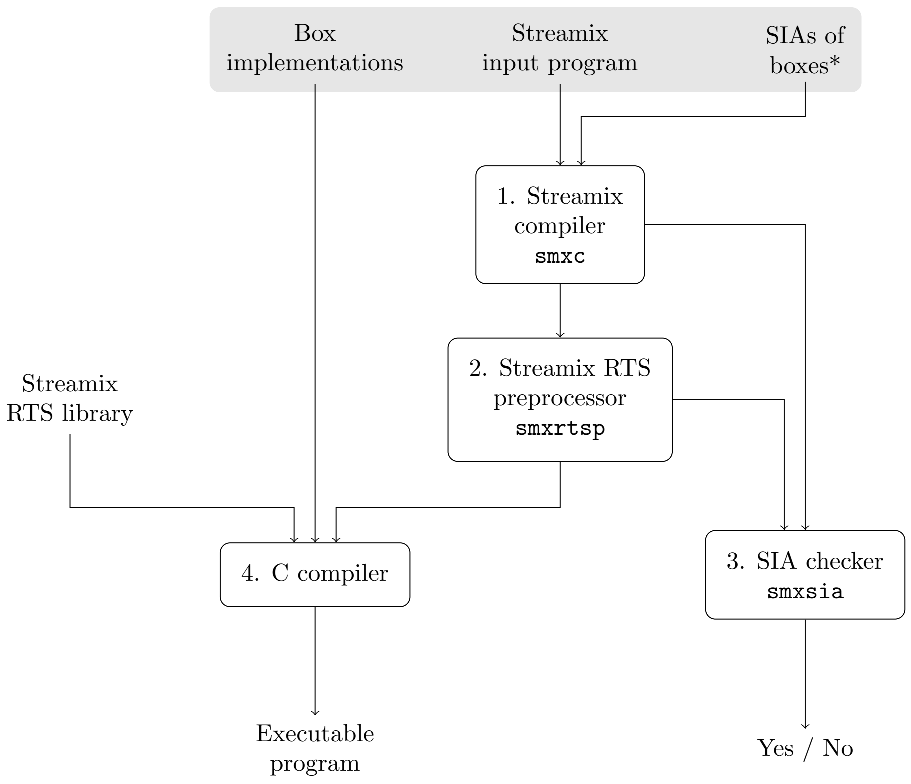


FIGURE 7.1: A schematic overview of the toolchain for Streamix.

I provide a root project with a link to all tools and some examples as a git repository on GitHub. The source code for each tool is provided as a separate git repository where the code is commented respecting the *Doxygen*[1] syntax. For each repository the API documentation can be generated with the command `make doc`. Here are the links to the different repositories:

**Root Project:** `https://github.com/moiri/streamix`

[1]The homepage of the *Doxygen* project: `http://www.stack.nl/~dimitri/doxygen/`

**The RTS Library:** `https://github.com/moiri/streamix-rts`

**`smxc` - The Compiler:** `https://github.com/moiri/streamix-c`

**`smxrtsp` - The RTS Preprocessor:**
`https://github.com/moiri/streamix-graph2c`

**`smxsia` - The Permanent Blocking Analysis:**
`https://github.com/moiri/streamix-sia`

The following list provides a brief overview of the examples that are available on GitHub. I kept the examples simple to perform isolated tests with the network operators provided by Streamix. All examples can easily be adapted, e.g. decoupling of ports, adding rate bounds or temporal firewalls, due to the exogenous nature of Streamix. Each example can be compiled with the command `make`. The command `make run` executes the compiled example and the command `make valgrind` uses *Valgrind*[2] to execute and analyse the compiled example.

**`class.smx`** performs a serial composition of four nets where two nets refer to the same box declaration. This example serves to show the capability of Streamix to instantiate the same box declaration multiple times and allowing to concatenate the nets due to port collections.

**`copy.smx`** demonstrates the implicit creation of a diverging routing node due to a parallel composition of two nets with the same input name.

**`dboth.smx`** serves to demonstrate the communication over a FIFO channel that is decoupled on its input and output.

**`din.smx`** serves to demonstrate the communication over a FIFO channel that is decoupled on its input.

**`dout.smx`** serves to demonstrate the communication over a FIFO channel that is decoupled on its output.

**`merge.smx`** demonstrates the implicit creation of a summing routing node due to a parallel composition of two nets with the same output name. Note that the non-deterministic parallel composition operator must be used here.

**`simple.smx`** performs a serial composition between two nets and demonstrates port renaming capabilities of Streamix.

[2]The project homepage of *valgrind* can be found here: `http://valgrind.org/`

**`stream.smx`** performs a serial composition between two nets and demonstrates how to specify a buffer length in Streamix. The example requires inputs by the user over the standard input stream. The program terminates once the character `ESC` is received by the consumer.

**`tb.smx`** demonstrates a rate-bound communication over a decoupled channel.

**`tcp.smx`** serves as a simple example to show the support of Streamix for complex components. Note that in this example the port order of the box declarations does not correspond to the communication protocol of the interacting processes. In order to prevent an error message from the permanent blocking analysis tool `smxsia` a custom SIA description is provided in the file `tcp.sia`. I did this intentionally to provide an example where a custom SIA description is required.

**`tt.smx`** demonstrates the support of Streamix for temporal firewalls and the time-triggered communication scheme.

This chapter is structured as follows: First, in Section 7.1 I describe the RTS library of Streamix to give the reader an understanding of how a Streamix program is mapped onto a software architecture and then run on a hardware platform. Section 7.2 gives technical insight on the compiler of the coordination language Streamix. In Section 7.3 I describe the RTS preprocessor and in Section 7.4 I provide an overview of the implementation of the permanent blocking analysis. Finally, in Section 7.5 I discuss the result and the limitations of the implementation and in Section 7.6 I summarise the chapter.

## 7.1 The Streamix RTS Library

The RTS of Streamix defines the execution model of a Streamix program. It maps Streamix language elements to run-time constructs that can be executed on a hardware platform. The Streamix RTS library provides a set of functions and macros that allows to create, initialise, execute, and destroy such run-time constructs.

In the following I describe a prototype of an RTS library for Streamix which I implemented as a proof of concept for the Streamix language. The RTS is implemented in C and targets x86 architectures running a Linux operating system. The RTS library uses POSIX threads [67] and the *zlog*[3] library. Timers which I use for the rate-control mechanism, are created with the *timerfd* API which limits the RTS to Linux systems with kernel version 2.6.25 and newer.

[3]The homepage of the *zlog* project: `https://github.com/HardySimpson/zlog`

A Streamix program defines, simply put, a network of computational components which interact with each other via channels. In the RTS each computational component is represented as a POSIX thread and each channel is a static structure with mutex variables that allow to model the blocking behaviour of FIFO channels. The RTS provides function macros to connect threads by channels according to the Streamix network. I chose mutex variables for the implementation because it is a simple and easy to understand mechanism to handle concurrent access on shared resources. It has, however, the limitation that unbuffered synchronous communication between two computational components is not supported, i.e. all channels are of type FIFO with a minimal length of one.

### 7.1.1 Implementation of Computational Components

The RTS does not distinguish between implicitly generated and user-defined computational components. The only difference is that the function describing the behaviour of an implicitly generated computational component is part of the RTS library whereas a user-defined computational component is associated to a user-defined function. A user-defined Streamix box (see Section 6.2.1) is linked to the user-defined function by name matching. For each computational component a POSIX thread and a static interface structure is created. The interface structure represents the interface of a Streamix net. Upon creation of the thread the function `start_routine_net()` is called. This function is defined in Algorithm 7.1. It takes a pointer to the function `ccomp()` and

**Algorithm 7.1** PThread Start Routine of Computational Component

**Require:**
`ccomp()`, the function defining the behaviour of the computational component
$\mathcal{P}$, the net interface structure, associated to the computational component where $\mathcal{P}^I$ and $\mathcal{P}^O$ are the sets of input and output ports, respectively.

```
1: function start_routine_net(ccomp(), $\mathcal{P}$)
2:     $state \leftarrow SMX\_NET\_CONTINUE$
3:     while $state = SMX\_NET\_CONTINUE$ do
4:         $state \leftarrow$ ccomp($\mathcal{P}$)
5:         $state \leftarrow$ smx_net_update_state($\mathcal{P}^I, state$)
6:     end while
7:     smx_net_terminate($\mathcal{P}^O$)
8:     return $Null$
9: end function
```

the net interface handler $\mathcal{P}$ as input arguments and returns *Null*. The net interface handler $\mathcal{P}$ is a void pointer to a C structure with a field for each input and output port of the net, associated to the computational component. The sets of input and output ports are represented by the symbols $P^I$ and $P^O$, respectively. Each port field in the C structure points to a connecting channel structure. The fields have the same names as the Streamix ports which allows to access the corresponding channel by name instead of an index (this allows a user-friendly access to channels). The interface structure also contains an array of channel pointers which allows to access the channels by index rather than by name (this allows a machine-friendly access to channels). The function `ccomp()` implements the behaviour of the computational component represented by the thread and takes the net interface structure as argument.

The body of the function consists of a while loop that is repeatedly making a call to the function `ccomp()` (line 4). The state variable is either updated by the return value of the function `ccomp()` or by the function `smx_net_update_state()` (line 5). The function `ccomp()` can return three values:

*SMX_NET_RETURN* lets the RTS decide whether to terminate the thread or not. The decision is made by the `smx_net_update_state()` (line 5) function.

*SMX_NET_CONTINUE* forces the RTS to continue the repeated execution of the computational component function.

*SMX_NET_END* forces the RTS to terminate the thread.

The function `smx_net_update_state()` takes the set of input ports $\mathcal{P}^{\mathcal{I}}$ and the *state* variable as arguments and returns a state value. The function returns *SMX_NET_END* if all connected triggering producers are terminated or if the function `ccomp()` sets the *state* variable to *SMX_NET_END*. Otherwise the `smx_net_update_state()` returns *SMX_NET_CONTINUE*. Consumers connected to the thread are signalled if the thread terminates. This is accomplished through the function `smx_net_terminate()` (line 7). It takes the set of output ports $\mathcal{P}^O$ as argument and has no return value.

To instantiate a computational component as a thread the RTS provides the following functions as macros:

`SMX_NET_CREATE`$(f)$ takes the function name $f$ of the computational component as input argument and returns a pointer to the interface handler structure of a net, i.e. a Streamix instance of the computational component. The macro allocates memory space for the net interface structure.

`SMX_NET_INIT`$(f, \mathcal{P}, indegree, outdegree)$ takes the function name $f$ of the computational component, the net interface structure, the number of input ports, and the number of output ports as arguments (no return value). The macro allocates memory space for channel arrays. These are needed to access channels by index instead of accessing them by name.

`SMX_NET_RUN`($\mathcal{P}, f$) takes the net interface structure $\mathcal{P}$ and the function name $f$ of the computational component as arguments and returns a POSIX thread identifier. The macro creates a POSIX thread with the function `start_routine_net`, as defined in Algorithm 7.1, and a pointer to the net interface structure as argument to the start routine.

`SMX_NET_DESTROY`($f$) takes the function name $f$ of the computational component and the net interface structure as argument and frees the allocated memory space.

### 7.1.2 Implementation of Channels

Each channel is represented by a C structure. Hence, in contrast to computational components, channels are passive elements and only serve as memory locations to store message tokens in order to achieve decoupling in time and space (see Section 4.1). A channel structure consists of the following elements:

**type** defines the type of a FIFO buffer that is associated to the channel. It can either be a FIFO buffer that is fully coupled in synchronisation, fully decoupled or partly decoupled (see Section 4.1.2).

**name** is a human readable identifier that is used in log messages. It is derived from the port name of the connected producer net.

**state** indicates the state of the channel. The following states are possible:

*SMX_CHANNEL_READY* indicates that the channel is *ready* and has available data.

*SMX_CHANNEL_PENDING* indicates that the channel is *pending* and is currently empty and blocking. If the channel is decoupled at the output, this state is not possible.

*SMX_CHANNEL_END* indicates that the producer of this channel has terminated. This state is set by the function `smx_net_terminate` of the producer net.

**fifo** is a pointer to a FIFO buffer structure. In this structure the arriving message tokens are stored until they are consumed by the consumer. The FIFO structure is a circular buffer of static length. In addition to the circular buffer, the FIFO structure has one more space to store the last message token if the circular buffer is empty. This backup storage is used to duplicate the last message token if the output of the channel is decoupled. A counter variable indicates the number of occupied spaces in the circular buffer.

**guard** is a pointer to a structure that serves to bound the communication rate on this channel (see Subsection 7.1.2.2).

**collector** is a pointer to a structure that serves to notify a summing routing node (a node with multiple inputs) if a message token on any of the channels connecting to a input port is available (see Subsection 7.1.2.3).

In order to protect critical sections such as the state of the channel or the count variable in the FIFO structure, mutex (short for mutual exclusion) variables are used. Producer and consumer threads are synchronized by conditional variables:

It is only possible to read from a channel if the channel is *ready*, i.e. the channel state is set to *SMX_CHANNEL_READY*. Otherwise the read access is blocking until a message token is written to the channel and the channel becomes *ready*. If a channel is decoupled on its input, the channel is either *ready* or its producer is terminated, i.e. the channel state is set to *SMX_CHANNEL_END*, but the channel is never *pending*.

It is only possible to write to a channel if the counter in the FIFO structure is lower than the length of the circular buffer, i.e. there is still space in the circular buffer. Otherwise the write operation is blocking until space is made available in the buffer. An exception to this is when the channel is decoupled on its input. In this case, if all spaces are occupied, the tail of the FIFO buffer is overwritten with the new message token. Whenever a message token is written to the channel, the channel becomes *ready*. This is signalled to the consumer thread connected to this channel.

The Streamix RTS library provides the following function macros to create and destroy channels.

`SMX_CHANNEL_CREATE`(*len*, *type*) takes the specified length *len* of the channel buffer and the channel type *type* as arguments and returns a pointer to the created channel structure. The macro allocates the space for the channel and the FIFO structure according to the specified length of the buffer. It further initialises the state of the channel and the pthread mutex variables. The state is set to *SMX_CHANNEL_PENDING* if the input is coupled in synchronisation. If the input is decoupled in synchronisation, the state is set to *SMX_CHANNEL_READY*.

`SMX_CHANNEL_DESTROY`(*ch*) takes a pointer to the channel structure *ch* as argument and frees the allocated memory for the channel structure and the FIFO structure.

In order to connect the channels to nets according to the dependency graph, specified by the Streamix program, the following macro functions are provided by the RTS:

`SMX_CONNECT`$(\mathcal{P}, ch, f, p_{name}, p_{mode})$ takes a pointer to a net interface structure $\mathcal{P}$, a pointer to a channel structure $ch$, the computational component function name $f$, the port name $p_{name}$, and the port mode $p_{mode}$ (e.g. *in* or *out*) as arguments (no return value). The macro assigns the channel structure pointer to the corresponding port in the net interface structure. This allows to access a channel from the computational component function by name identification (easy access for a human).

`SMX_CONNECT_ARR`$(\mathcal{P}, ch, f, p_{mode})$ takes a pointer to a net interface structure $\mathcal{P}$, a pointer to a channel structure $ch$, the computational component function name $f$, and the port mode $p_{mode}$ as arguments (no return value). The macro increases the port counter and assigns the channel structure pointer to the corresponding index port in the net interface structure. This allows to access a channel from the computational component function by index identification (easy access for a machine).

When programming the computational component function, a domain expert must use predefined macros of the RTS library to access the connected channels. Note that for all types of channels the same macro is used. Hence, the computational component function is oblivious to the blocking semantics of the channel as well as to where the other end of the channel is connected. The port identification in the macros is achieved via macro concatenation by the preprocessor operator '##'. If a port name is passed as argument that cannot be found in the net interface structure, the C compiler fails with an error message. The interface handler is passed as a void pointer to the *start routine* of a thread. The computational component function name is passed as argument to a macro in order to typecast the interface handler to the net interface structure within the macro. The following two macro functions are provided to access a channel:

`SMX_CHANNEL_READ`$(\mathcal{P}, f, p_{name})$ takes a pointer to the interface handler $\mathcal{P}$, the computational component function name $f$, and the port name $p_{name}$ as arguments and returns a pointer to a message token structure. The macro can only access input ports. Depending on the type of the connected channel, the message token is removed from the buffer in the channel or is duplicated. Whether the access is blocking depends on availability of data and the communication coupling specified by the Streamix program.

`SMX_CHANNEL_WRITE`$(\mathcal{P}, f, p_{name}, msg)$ takes a pointer to the interface handler $\mathcal{P}$, the computational component function name $f$, the port name $p_{name}$, and a pointer to a message token *msg* as arguments (no return value). The macro can only access output ports. Depending on the type of the connected channel, the message token is appended to the buffer in the channel or is overwriting an existing message token in the channel buffer. Whether the access is blocking depends on availability of space and the communication coupling specified by the Streamix program.

In order to work with message tokens the RTS library provides the following macro functions to create and destroy message tokens:

`SMX_MSG_CREATE`$(data, size, copy, free)$ takes a pointer to a data structure *data*, the size of the data structure *size*, a *copy* function pointer, and a $free$ function pointer as input arguments and returns a pointer to the created message token structure. The *copy* function defines the copy operation that is performed on the data stored inside the message token structure once the message token is copied. This can happen because of the decoupling of an input port of a net or because of a diverging routing node. The *copy* function must take a void pointer to the data structure and the size of the data structure as arguments and return a void pointer to the copied data structure. If the *copy* function argument is set to $Null$, a `memcpy`() is performed on the data structure instead. The $free$ function defines the operation that is performed on the data structure once a message token is destroyed. This can happen if a message token is overwritten due to the decoupling of an output port of a computational component or because message tokens are dismissed in a rate-bound communication protocol. The $free$ function must take a void pointer to the data structure as argument and return nothing. If the $free$ function argument is set to $Null$, a `free`() is performed on the data structure instead.

`SMX_MSG_DESTROY`$(msg)$ takes a pointer to a message token structure *msg* as argument and frees the allocated message token structure and performs the user-defined free function on the data structure stored in the message token.

#### 7.1.2.1 Implementation of Time-triggered Communication

In Section 4.2.1 I described the PNSC extension to achieve a time-triggered network by introducing temporal firewalls. In the Streamix RTS, temporal firewalls with the same rate are grouped into one single timer net that is associated to a single POSIX thread. This allows to only use one timer for all temporal firewalls and automatically synchronizes the temporal firewalls. The start routine of such a thread is defined by the function `start_routine_tf`(), described in Algorithm 7.2. The function `start_routine_tf`$(\mathcal{P}, \mathcal{T})$ takes the interface of the assembled temporal firewalls $\mathcal{P}$,

**Algorithm 7.2** PThread Start Routine of a Temporal Firewall

**Require:**

$\mathcal{P}$, the net interface structure, associated to the timer net where $\mathcal{P}^I$ and $\mathcal{P}^O$ are ordered sets of input and output ports, respectively.

$\mathcal{T}$, a periodic timer structure with a period specified by the time-triggered Streamix operator.

```
1: function start_routine_tf($\mathcal{P}$, $\mathcal{T}$)
2:     $state \leftarrow SMX\_NET\_CONTINUE$
3:     smx_tf_enable($\mathcal{T}$)
4:     while $state = SMX\_NET\_CONTINUE$ do
5:         $msg \leftarrow$ smx_tf_read_inputs($\mathcal{P}^I$)
6:         smx_tf_write_outputs($\mathcal{P}^O$, $msg$)
7:         smx_tf_wait($\mathcal{T}$)
8:         $state \leftarrow$ smx_net_update_state($\mathcal{P}^I, SMX\_NET\_RETURN$)
9:     end while
10:    smx_net_terminate($\mathcal{P}^O$)
11:    return $Null$
12: end function
```

and the timer $\mathcal{T}$ as input arguments and returns $Null$. The interface $\mathcal{P}$ of a timer net $TN$ is a set of tuples $\langle p^i_k, p^o_k \rangle$ where $p^i_k$ is an input port and $p^o_k$ an output port of the $k$-th temporal firewall grouped in the timer net $TN$. $\mathcal{P}^{\mathcal{I}}$ and $\mathcal{P}^{\mathcal{O}}$ each represent an ordered set where a port $p^i_k \in \mathcal{P}^I$ and a port $p^o_k \in \mathcal{P}^O$ belong to the $k$-th temporal firewall in a timer net. The timer $\mathcal{T}$ is a periodic timer where the period is set to a time interval $t_{tt}$, specified by the `tt` operator of the Streamix language (see Section 6.4.5).

The behaviour of a timer net, as described by Algorithm 7.2, performs the following actions: Before the thread starts to execute the while loop, the timer $\mathcal{T}$ is armed with the function `smx_tf_enable`($\mathcal{T}$) which takes the timer structure $\mathcal{T}$ as input argument and returns nothing. The timer $\mathcal{T}$ is triggering periodically without being armed again. The same as any other computational component, the body of the function consists of a while loop where the stop condition depends on the value of the state variable. The state variable is influenced by the function `smx_net_update_state`() which returns the state $SMX\_NET\_TERMINATE$ if all producer nets, connected to the timer net, are terminated (see Section 7.1.1). In the while loop, the timer net is first reading from the input of each temporal firewall and stores the message tokens in the set $msg$ (line 5). This is accomplished by the function `smx_tf_read_inputs`(). Next, all message tokens are written to the output of each temporal firewall with the function `smx_tf_write_outputs`() (line 6). The function then blocks until timer $\mathcal{T}$ expires (line 7). Finally, the state variable is updated with the function `smx_net_update_state`() and the iteration starts at the beginning.

To detect whether a producer net, connected to a temporal firewall, has missed its deadline, the function `smx_tf_read_inputs()` checks whether a message token is available in the buffer of the channel connecting the producer net to the temporal firewall. If no message token is available, the producer was not able to terminate its execution before the deadline (due to Property 4.2) and the following error message is produced:

```
producer on channel '<channel>' missed its deadline
```

where `<channel>` is a human readable identifier of the channel connecting the producer to the temporal firewall. To detect whether a consumer net, connected to a temporal firewall has missed its deadline, the function `smx_tf_write_outputs()` checks whether a message token was overwritten in the buffer of the channel connecting the temporal firewall to the consumer net. If this is the case, the consumer net has missed its deadline as it was not able to consume the message token from the previous round (see Property 4.1). In this case the following error message is produced:

```
consumer on channel '<channel>' missed its deadline
```

where `<channel>` is a human readable identifier of the channel connecting the temporal firewall to the consumer net.

The RTS library provides the macro function `SMX_TF_CREATE`$(t_s, t_{ns})$ to create the timer interface structure. It takes the time specification in seconds $t_s$ and nanoseconds $t_{ns}$ as input parameters and returns a pointer to the created timer structure. The macro function `SMX_TF_DESTROY`$(\mathcal{T})$ takes a pointer to a timer interface structure $\mathcal{T}$ as input argument and frees the allocated memory. The macro function `SMX_TF_RUN`$(\mathcal{T})$ takes a pointer to the timer interface structure $\mathcal{T}$ as input parameter and creates a thread, associated with the routine defined in Algorithm 7.2. The individual temporal firewalls are added dynamically to the timer interface structure through the macro function `SMX_CONNECT_TF`$(\mathcal{T}, ch_{in}, ch_{out})$. This macro takes a pointer to the timer interface structure $\mathcal{T}$, a pointer to the input channel $ch_{in}$, and a pointer to the output channel $ch_{out}$ as input parameters.

#### 7.1.2.2 Implementation of Rate-bounded Communication

In Section 4.2.2 I described the rate-control extension of the PNSC model. In the Streamix RTS the rate-control is achieved with the help of a timer (provided by the *timerfd* library) that is accessed through the guard structure, assigned to the guard field of the channel structure. The guard structure consists of a file-descriptor pointing to the timer and the minimal inter-arrival time specified by the Streamix program. If no rate-control is specified on a channel, the guard field of the channel structure is set to $Null$.

If the output port, connected to a rate-controlled channel, is not decoupled, back-pressure is excerpt on the producer in order to not exceed the maximal rate: When performing a blocking write operation to a channel where a rate-control is specified, the write process not only blocks if the channel buffer is full but it also blocks until the timer has reached the specified minimal inter-arrival time. The timer is reset and armed after a message token has successfully been written to the rate-controlled channel buffer.

In the case of communication decoupling where the output port of the producer is decoupled, the rate-control is achieved by discarding message tokens. In Section 4.2.2 I introduced four protocols to achieve rate-bounded communication in a decoupled channel. From the four protocols I implemented the unbuffered Minimal Inter-release Time (MIRT) protocol because it does not affect the latency of a transmitted message tokens and guarantees that the communication rate never exceeds the specified bound. Because the protocol is unbuffered, no separate thread is required to handle the rate-control: The decision of whether a message token is discarded is made immediately after a write operation. A message token is written to the rate-bounded channel only if the minimal inter-arrival time is exceeded, i.e. the timer has reached its threshold. Otherwise a message token is discarded. The timer is reset and armed after a message token has successfully been written to the rate-controlled channel buffer.

The RTS library provides the macro function `SMX_CONNECT_GUARD`$(ch, t_s, t_{ns})$ which takes a pointer to a channel structure $ch$, the minimal inter-arrival time in seconds $t_s$, and the minimal inter-arrival time in nanoseconds $t_{ns}$ as input arguments (no return value). The minimal inter-arrival time is stored in a timespec structure (`time.h` library) where time is specified by two integer values: one specifying the seconds and one the nanoseconds. The macro allocates the memory for the guard structure, initializes and arms the timer, and assigns the structure pointer to the guard field of the channel structure that was passed as an argument.

#### 7.1.2.3 Implementation of Routing Nodes

Routing nodes are computational components that are implicitly created by the Streamix coordination language (see Section 6.2.2.2). Thus, the computational component function is provided by the RTS library. In addition to this, the net interface structure of a routing node is slightly different to the net interface structure of a user-defined box: First, ports are not accessed by name matching but by index matching. Hence, it is sufficient to store the connecting channel structure pointers in an array instead of a specific field for each port. Second, because a summing routing node is non-deterministic it is allowed to peak whether a message is available on a channel before performing a blocking read. Hence, instead of using the state variable of the channel structure as blocking condition an alternative conditional variable is used. This variable is stored in a collector structure.

A collector structure is initialised with the macro function `SMX_NET_RN_INIT`($\mathcal{P}$) which takes a pointer to the routing node interface structure $\mathcal{P}$ as an argument, allocates the memory space for a collector structure and assigns the collector structure to the corresponding field in the routing node interface structure. The collector structure has a *state* field and a *count* field. The *state* field can take the values *SMX_CHANNEL_READY*, *SMX_CHANNEL_PENDING*, and *SMX_CHANNEL_END* (the same as the state field of the channel structure). The *count* field indicates the number of available message tokens on all channels connected to the input ports of the routing node. The fields *state* and *count* of the collector structure form a critical section and are protected by a mutex.

The macro function `SMX_CONNECT_RN`($\mathcal{P}, ch$) is used to assign the collector structure of a routing node to a channel. It takes a pointer to the net interface structure $\mathcal{P}$ of the routing node and a pointer to a channel structure *ch* as input arguments (no return value). If the collector structure of a channel is not $Null$, a write access to this channel increases the *count* field by one and changes the collector state to *SMX_CHANNEL_READY*.

In order to access a channel that is connected to the input port of a routing node, the implementation does not use the macro function `SMX_CHANNEL_READ`() as a user-defined box implementation would. Instead a custom read access is performed that is not blocking on the channel state but on the state of the collector structure assigned to the net interface structure of the routing node: As long as the state of the collector structure of the routing node is *SMX_CHANNEL_PENDING* the read access is blocking. This allows a control-node to trigger whenever a message token is available on any channel connected to its input ports.

Every message token that reaches a routing node is copied to each channel, connected to an output port of the routing node. The write operation is equivalent to the operation performed by the macro function `SMX_CHANNEL_WRITE`().

## 7.2 `smxc` - The Streamix Compiler

The Streamix compiler *smxc* is implemented in C. Its main job is to read an input Streamix file, i.e. a hierarchical network description written in the language Streamix (as defined in Chapter 6), and translate it into a flat dependency graph of the network. This generated dependency graph holds all the necessary information for the runtime system to generate a multi-threaded application. The compiler consists of a scanner, a parser, a context checker, a network generator, and a SIA generator. When executing the compiler `smxc` one input file, describing the Streamix program, is passed as argument. The following options can be specified:

**`-h`** displays the usage message and exits the program.

**`-v`** displays the version number and exits the program.

**`-s <path>`** specifies the path to a file where the SIA descriptions of the user-defined boxes are stored. This option expects a path to a file as argument.

**`-S`** skips the automatic SIA generation step (no SIA graphs are produced).

**`-p <path>`** specifies the build path to the folder where the generated output files are stored. This option expects a path to a folder as argument. The default path is `./build/`.

**`-o <file>`** specifies the filename of the generated network dependency graph. This option expects a filename as argument. The default name is `out.<format>` where `<format>` is specified by the `-f` option.

**`-f <format>`** specifies the format of the produced output files. This option expects either `gml` [68] or `graphml` [69] as argument. The default format is `graphml` [69].

Whenever the compiler detects an error or an inaccuracy in the Streamix program an error or a warning message is produced. Such a message is of the form

```
<filename>: <line#>: <type>: <message>
```

where `<filename>` indicates the name of the file that caused the message, `<line#>` the line number, and `<type>` whether the message is a *warning* or an *error*. In the following subsections I will describe each error or warning message in detail but, for the sake of readability, I will omit all information but the `<message>` itself. In the messages I will use different tags as placeholders for information that depends on the Streamix program producing the messages:

**`<symbol>`** stands for a user-defined symbol.

**`<net>`** stands for a user-defined, human-readable identifier of a net.

**`<id>`** stands for an internal identifier of a net.

**`<port>`** stands for a user-defined, human-readable identifier of a port.

**`<sia>`** stands for a user-defined, human-readable identifier of a SIA.

**`<state>`** stands for a user-defined, human-readable identifier of a SIA state.

**`<action>`** stands for a user-defined, human-readable identifier of an SIA action.

The Streamix scanner is generated with *flex*, the fast lexical analyzer generator[4]. The flex definition file in the Streamix compiler repository can be found in the file `streamix.lex`. The Streamix parser is generated with *bison*, a general purpose parser generator[5]. The parser is defined in the file `streamix.y` in the Streamix compiler repository. In the parsing step, the Streamix program is represented as an Abstract Syntax Tree (AST) which is then passed to the context checker. In the following subsection I will describe the context checker, the graph analyser and the SIA generator in more detail.

### 7.2.1 The Streamix Context Checker

In order to check the context of the symbols in the Streamix program, the compiler uses the AST generated by the parser and iterates through each node. Every defined symbol in the Streamix program is stored into a hash table (library *uthash*[6]) where each symbol has a unique key $\langle name, scope, mode \rangle$ to avoid collisions. *name* describes the name of the symbol, *scope* the scope in which the symbol is defined, and *mode* describes the direction of ports. The *mode* is omitted for symbols that do not describe ports. For ports, *mode* is added to the key string in order to prevent two ports with the same name and the same mode in a scope. The hash table is then used to check whether a symbol is used in the right context of the Streamix program. If any of the following rules does not hold, the compiler produces an error message:

1. A symbol must be defined before it is used, otherwise an *error* message is produced:

   ```
   use of undeclared identifier '<symbol>'
   ```

[4]The project homepage of *flex* can be found here: https://github.com/westes/flex
[5]The project homepage of *bison* can be found here: https://www.gnu.org/software/bison/
[6]The project homepage of *uthash* can be found here: http://troydhanson.github.io/uthash/

In the following I use '. . . ' to indicate that something else is required for a correct syntax which is omitted here to keep the examples minimal. A symbol can be defined through a

- box assignment, e.g. box $A$ in `A = fa box(...)`
- net assignment, e.g. net $N$ in `N = ...`
- wrapper declaration, e.g. wrapper $W$ in `wrapper W(...)...`
- port declaration in a box, wrapper or net prototype, e.g. port $x$ in `A = box fa( in x )`

2. A symbol must be unique in its scope, otherwise the following *error* message is produced:

   ```
   redefinition of '<symbol>'
   ```

   The uniqueness of a symbol depends on its *name*, its *scope*, and its *mode* if the symbol is a port. Scopes, in the following denoted as `<scope>`, are defined as follows:

   - The body of a wrapper has its own scope, e.g. `wrapper W(...){<scope>}net(...)`.
   - The port interface declaration of a box, wrapper, or net prototype has its own scope, e.g. `A = box fa(<scope>)`.

3. The symbol name of a net prototype must match the symbol name of a net definition in this scope. Otherwise the following *error* message is produced:

   ```
   undefined reference to net '<symbol>'
   ```

4. Net prototypes must have a net definition with a matching symbol name. The port list of a net prototype must match with its corresponding net definition, hence the condition as defined in Equation 6.8 must hold. If the condition does not hold, the following *error* message is produced:

   ```
   conflicting types for '<symbol>'
   ```

### 7.2.2 Generation of Dependency Graph

At the same time as the context check of symbols is performed, the Streamix program is checked for connection errors and the dependency graph is generated with all valid connections. In order to have a powerful graph tool at hand, the *igraph*[7] library is

[7]The project homepage of *igraph* can be found here: http://igraph.org/c/

used to represent the dependency graph. As Streamix allows a hierarchical structure of networks, in a first step a local dependency graph is built for each net at each hierarchy level. Only in a second step, the network is flattened into one big dependency graph.

The nodes of a dependency graph represent net instances and edges between different nodes are used to represent the interactions of the net instances. When building the local dependency graphs on one hierarchy level, for each net an instance structure is generated and attached to its corresponding node in the graph structure. The net, represented by a single node, can be a box with an associated user-defined implementation, or the net itself can be a network of a different hierarchical level, composed out of other nets. An instance structure is defined by the tuple $\langle id, name, type \rangle$. The *id* is a unique identifier corresponding to the id of the graph node that is representing the instance in the dependency graph. The *name* is the human readable identifier, given to the instance by the programmer (this name is not necessarily unique). The *type* is used internally to distinguish between different types of instances (e.g. wrapper, boxes, nets). The following checks are performed in order to verify whether the program respects the Streamix connection rules as defined in Section 6.4:

1. At least one input port of a net must be triggering. This means that it is not allowed to decouple all input ports if the net is event-triggered. Otherwise the following error message is produced:

   ```
   all input ports are decoupled
   ```

2. Modes of ports with matching names must be of opposite direction when forming one of the following connections:

   - A self-loop connection as defined in Definition 6.1.
   - A serial connection as defined in Definition 6.4.
   - A bypass connection of wrappers as defined in Equation 6.13.

   Modes of ports with matching names must be of the same direction when forming connections between a wrapper and its inner net as defined in Equation 6.14. The port mode of two ports with matching names does not matter when forming a side-port connection as defined in Definition 6.2. If a port mode conflict is detected, the following *error* message is produced:

   ```
   conflicting modes of ports '<port>' in '<net>'(<id>)
      and '<net>'(<id>)
   ```

3. A serial connection must spawn at least one channel. If the serial connection operator is used, it must perform a connection. Otherwise the compiler assumes

that the intention of the programmer is not represented by the actual connection and throws the following *error* message:

```
no port connection in serial combination
   '<net>(<id>).<net>(<id>)'
```

4. The deterministic parallel composition operator '|' (see Definition 6.7) is not allowed to spawn a summing control flow node (see Section 6.2.2.2), i.e. two output ports with the same name and the same collection (or no collection) are not allowed. This is indicated by the *error* message

```
nondeterminism on deterministic operation
   '<net>(<id>)|<net>(<id>)', use '!' instead
```

5. The locality enforcing serial composition operator '.' (see Definition 6.5) must connect all ports that are not assigned to a port collection. A port is not assigned to a port collection if the condition defined in Equation 6.7 holds. If such ports cannot be connected, the following *error* message is produced:

```
unconnected port '<port>' in '<net>'(<id>) of serial
   combination '<net>.<net>'
-> for bypassing, use operator ':' or a wrapper
```

6. If a net is framed by temporal firewalls through the operator `tt` (see Section 6.4.5) but the net has no open ports, the operator has no effect as the net is not interacting with any other net. To indicate this situation the following *warning* is produced:

```
net has no open ports, operator 'tt' has no effect
```

Once the local dependency graphs are built on each hierarchy, they are combined into one big dependency graph where all hierarchies are removed. If a local net, defined by a local dependency graph, is instantiated multiple times on a higher hierarchy level, every instance in the local dependency graph has to be replicated recursively in the flattened graph. On the flattened dependency graph the following two final connection checks are performed:

1. Every flow control node must have at least one output and at least one input. If this condition is not met the following *error* message is produced:

```
single mode in routing node 'smx_rn'(<id>)
```

   Because of side-port connections and the parallel composition, it is possible that routing nodes are generated where all connected ports are of the same mode. If such a routing node is not further connected, the aforementioned error occurs.

2. In the final dependency graph all ports must be connected. Otherwise the *error* message

   ```
   port '<port>' in '<net>'(<id>) is not connected
   ```

   is produced.

In the resulting flattened dependency graph each vertex represents either an implicitly created routing node or a user-defined box. Every hierarchical element is replaced by its net definition. This graph of the network is then written to a file. The language in which the graph is defined in the output file can be chosen between the Graph Modelling Language (GML) [68] and the GraphML [69] format which is based on the Extensible Markup Language (XML). The GraphML format is preferred because it is more completely implemented in the *igraph* library. The produced dependency graph has the following attributes:

**Vertex Attributes**

**`id`** (number) is an internal unique identifier of the net.

**`label`** (string) is a human-readable identifier of the net.

**`func`** (string) is the function name of the computational component implementation.

**`static`** (bool) indicates whether the net is static or not (see Section 6.2.4).

**`pure`** (bool) indicates whether the net is pure or not (see Section 6.2.4).

**Edge Attributes**

**`id`** (number) is an internal unique identifier of the channel.

**`label`** (string) is a human-readable identifier of the channel.

**`nsrc`** (string) is a human-readable identifier of the source port of the channel (the value is set to "smx_null" if it is identical to `label`).

**`ndst`** (string) is a human-readable identifier of the destination port of the channel (the value is set to "smx_null" if it is identical to `label`).

**`dsrc`** (bool) indicates whether the source port is decoupled.

**`ddst`** (bool) indicates whether the destination port is decoupled.

**`len`** (number) is defining the length of the channel, i.e. the space for `len` number of message tokens.

**`sts`** (number) is the timing information in seconds of the connected producer net.

**`stns`** (number) is the timing information in nanoseconds of the connected producer net.

**`dts`** (number) is the timing information in seconds of the connected consumer net.

**`dtns`** (number) is the timing information in nanoseconds of the connected consumer net.

**`type`** (number) indicates the type of timing annotations (0: no timing, 1: time-triggered, 2: time-bound).

### 7.2.3 Generation of Interaction Protocol Descriptions

Optionally, the compiler also accepts a SIA description file where the interaction protocol of user-defined boxes is described (see Section 6.2.3). This SIA description file is read by the Streamix compiler and then scanned, parsed, and a graph, representing the SIA is generated. As with the Streamix language, the scanner is produced with *flex* and the parser with *bison*. A user-defined SIA is associated to a user-defined box by matching the name of the computational component implementation with the name of the SIA. The context of symbols is checked as follows:

- Each SIA name must be unique in a Streamix program (the same as the function name of the computational component implementation). If a SIA is detected where the name matches an already existing SIA, the following *error* message is produced:

  ```
  redefinition of '<sia>'
  ```

- Each state in a SIA must be defined with a unique name. Otherwise the following *error* message is produced:

  ```
  redefinition of '<state>'
  ```

- Each state in a SIA must be defined before it can be used in a transition definition. Otherwise the following *error* message is produced:

  ```
  use of undeclared identifier '<state>'
  ```

- Each action in a SIA must correspond to a port of the corresponding Streamix box. If no such correspondence can be found, the following *error* message is produced:

  ```
  action '<action>' does not match the box signature
  ```

For each box in the Streamix program, the compiler produces a graph file representing the SIA of the box. Such a graph is either created according to the definition provided by the programmer or the SIA is generated automatically, as described in Section 6.2.3. The representation language of the SIA output graph files is specified as either GML [68] or GraphML [69] (the same format as for the dependency graph is used). SIA states are represented by graph vertices and SIA transitions are represented by graph edges. A SIA graph has the following attributes:

**Graph Attributes**

**`sia`** (string) is an internal unique identifier of the SIA.

**`name`** (string) is a human-readable identifier of the SIA.

**Edge Attributes**

**`name`** (string) is an internal unique identifier of the SIA transition.

**`pname`** (string) is a human-readable identifier of the action, labelling the transition.

**`mode`** (character) describes the SIA action type where '!' stands for an output action, '?' for an input action and ';' for an internal action.

## 7.3 `smxrtsp` - The Streamix RTS Preprocessor

The Streamix RTS preprocessor `smxrtsp` is implemented in C and uses the *igraph* library[8]. It takes the network dependency graph, generated by the Streamix compiler, as an input and produces a C program that creates and initialises the network, described by the network dependency graph, using macros defined in the Streamix RTS (see Section 7.1). The benefit from this is that the initialising step of a final program is much more performant because no parsing of the entire network graph is necessary as this was already performed by the RTS preprocessor. Further, the RTS preprocessor generates a C header file that defines the net interface structures for each net. This allows to use macro concatenations in the RTS library to access ports by name without having to perform a lookup during runtime.

The code generated by the Streamix RTS preprocessor can be linked with the Streamix RTS system library and compiled together with the user-defined computational component function definitions into the final executable program.

[8]The project homepage of *igraph* can be found here: http://igraph.org/c/

The second purpose of the Streamix RTS preprocessor is to extend the network dependency graph such that it corresponds to a PNSC model. In order to achieve this, each channel, represented by an edge in the dependency graph, needs to be represented as a PNSC process. Hence, for each edge $e$ in the dependency graph a new vertex $v$ and two new edges $e_1$ and $e_2$ are introduced. Let $source(e)$ describe the source vertex of an edge $e$ and $target(e)$ describe the target vertex of an edge $e$, then vertex $v$ must be connected such that the following condition holds:

$$source(e) = source(e_1) \ \land \ target(e) = target(e_2) \ \land \ target(e_1) = v \ \land \ source(e_2) = v$$

Edge $e$ is removed from the graph. Further, all the vertex and edge attributes are removed and replaced by an attribute `sia`. The vertex attribute `sia` is a unique identifier (string) of the net or the FIFO channel, represented by the vertex. The edge attribute `sia` is a unique identifier (string) of the synchronous channel, represented by the edge. These are the only attributes that are relevant for the SIA checker (described in Section 7.4). In order to cope with the different scopes of a Streamix program and to prevent naming conflicts in the flattened dependency graph (described in Section 7.2.2) an internal naming scheme is used.

In a last step, the RTS preprocessor generates SIA description graphs for every implicitly generated process, i.e. for routing nodes and for FIFO channels. The interaction protocol of a routing node is defined in Section 6.2.2.2. The interaction protocols of the different types of FIFO channels are defined in Section 3.3.1 and Section 4.1.2.

When executing the Streamix RTS preprocessor `smxrtsp` it takes the network dependency graph `input.smx` generated by the Streamix compiler as input and allows the following options:

**`-h`** displays the usage message and exits the program.

**`-v`** displays the version number and exits the program.

**`-p <path>`** specifies the build path to the folder where the generated output files are stored. This option expects a path to a folder as argument. The default path is `./build/`. The generated SIA description files are stored in a folder `sia` at the build path. The following output files are generated:

  **`build/input.c`** contains the main function where threads are spawned for each net and nets are connected by channels according to the dependency graph.

  **`build/boxgen.c`** defines the start routine of a thread for each box where the computational component function provided by the domain expert is called.

**build/boxgen.h** defines the net interface structure for each computational component function, providing an interface to access the channels from within the function.

**build/sia_input.graphml** describes the PNSC dependency graph where each FIFO channel is represented as a vertex in order to work with the SIA checker.

**build/sia/smx_rn_i.graphml** describes the SIAs for each routing node.

**build/sia/smx_ch_i.graphml** describes the SIAs for each channel.

**-f <format>** specifies the format of the produced output files. This option expects either `gml` [68] or `graphml` [69] as argument. The default format is `graphml` [69].

## 7.4 `smxsia` - The Permanent Blocking Analysis

The permanent blocking analysis `smxsia` is a prototype implementation of the permanent blocking analysis, described in Chapter 5. It is implemented in python and makes use of the igraph library[9]. In order to perform the permanent blocking analysis, all SIA graph description files generated by the Streamix compiler and the Streamix RTS preprocessor are composed incrementally. This process is described by Algorithm 7.3. In the algorithm, the function `readSia` reads a SIA graph description file and returns

**Algorithm 7.3** Incremental SIA composition

```
Require:
    $g_{nw}$, the dependency graph produced by the RTS preprocessor

1: $sia_1 \leftarrow$ readSia()
2: while $sia_2 \leftarrow$ readSia() do
3:     $shared \leftarrow$ getShared($g_{nw}, sia_1, sia_2$)
4:     $sia_1 \leftarrow$ foldSia($sia_1, sia_2, shared$)
5:     $g_{nw} \leftarrow$ abstractPNSC($g_{nw}, sia_1, sia_2, shared$)
6: end while
```

a graph structure representing the SIA. The network dependency graph produced by the RTS preprocessor is used to identify shared actions of SIAs defined in Equation 3.6. This is possible because of the relation between PNSC process ports and SIA actions as defined in Section 3.2.3. This is implemented in the function `getShared` which takes the dependency graph $g_{nw}$ and two SIAs as input and returns the shared actions *shared*. The function `foldSia` takes two SIAs and the shared actions between the SIAs as input arguments and returns the composed SIA. The composition is performed with the composition operator $\otimes$ defined in Definition 3.4. The function `abstractPNSC` takes

[9]The project homepage of *igraph* can be found here: `http://igraph.org/python/`

the dependency graph $g_{nw}$, two SIAs, and the set of shared actions between the two SIAs as input arguments and returns the abstracted dependency graph where the two processes corresponding to the SIAs passed as input argument are abstracted according to Definition 3.2. The loop is repeated until all SIA description files are read.

Finally, the permanent blocking analysis as described in Section 5 is performed on the resulting composed SIA. If a permanent blocking situation is detected, error messages are produced to indicate the permanent blocking state and the name of the action that is blocked.

When executing the SIA permanent blocking analysis `smxsia` it takes the network dependency graph and all SIA description files as input arguments and allows the following options:

**`-h`** displays the usage message and exits the program.

**`-f <format>`** specifies the format of the produced output files. This option expects either `gml` [68] or `graphml` [69] as argument. The default format is `graphml` [69].

**`-o <file>`** specifies the filename of the generated SIA description graph. This option expects a filename as argument. The default name is `out.<format>` where `<format>` is specified by the `-f` option.

## 7.5 Discussion

All the tools presented in this chapter are prototypes and only serve as a proof of concept. That being said, I spent quite some effort to implement unit and integration tests and performed memory analysis with the *valgrind* tool[10]. The source code of the RTS and the compiler are well documented, however, the source for the permanent blocking analysis and the RTS preprocessor are lacking in this respect.

In terms of features, it is especially noteworthy that from within a computational component function, channels are accessed through one macro for read access and one macro for write access, independently of the channel that is connected. This represents well the exogenous property of the coordination language Streamix. Further, it is very handy that channels can be accessed through name matching. I achieved this without any cost in terms of runtime performance because I made use of macro concatenations of the C preprocessor.

[10]The project homepage of *valgrind* can be found here: `http://valgrind.org/`

### 7.5.1 Scheduler of the Streamix RTS

Even though the focus of my thesis is not on the scheduling aspect of Streamix programs, it cannot be left unmentioned. The Streamix RTS does not provide a scheduler but relies on the Completely Fair Scheduler (CFS) provided by the linux kernel. This is not a practical solution for a time-critical system but it serves its purpose for a prototype implementation that aims at testing the usability of the language Streamix.

It is interesting to note that newer versions of linux kernels (version 3.14 and newer) include a real-time scheduler that supports Earliest Deadline First (EDF) scheduling which allows to check whether a set of tasks is schedulable (given the Worst-case Execution Time (WCET) of the tasks).

The close relation between software and hardware of Cyber-physical Systems (CPSs) and its implications on multi/many-core programming is discussed in a recent paper by Castrillon et al. [5] where they survey existing models and tools.

### 7.5.2 Order of SIA Composition

The SIA checker composes the SIAs in the order they are provided as input parameters. It does this after the Streamix compiler and RTS preprocessor have generated the flattened out dependency graph of the PNSC. Given that Streamix provides operators that indicate whether two nets are connected or not (serial and parallel composition operators), it would make sense to perform SIA composition operations at compile time of the Streamix compiler. This would allow to perform further checks with Streamix net prototyping and simplify intermediate composed SIAs to reduce the state space. Further, as mentioned in Chapter 5, reducing the state space of composed SIAs should also be possible by removing internal actions after a composition.

### 7.5.3 Static and Pure Nets

The coordination language Streamix allows to annotate nets with the keywords *static* and *pure*. The current RTS prototype does not make use of these annotations. The reason is that the current implementation builds a static network of connected POSIX threads but does not change the network dynamically at runtime. Consequently, no relocation or proliferation of nets is done.

## 7.6 Chapter Summary

In this chapter I provided technical details on the set of tools I implemented for the Streamix coordination language project. This includes the compiler `smxc`, the RTS preprocessor `smxrtsp`, the RTS library, and the permanent blocking analysis `smxsia`. For more detailed information for specific functions and structures, please refer to the documentation in the source code, provided on GitHub[11].

[11]Streamix root project: `https://github.com/moiri/streamix`

# Chapter 8

# Related Work

In this chapter I discuss existing work which I used as an inspiration for my thesis or is related to some aspects of it. Section 8.1 discusses the topic of interface theory and related models that served as an inspiration for the model of Chapter 3. Section 8.3 relates the coordination aspects of Streamix and its underlying model, discussed in Chapter 3, Chapter 4, and Chapter 6, to existing coordination models and languages. Finally, Section 8.4 relates the permanent blocking analysis, described in Section 5, to existing work in this area and I discuss specific properties of languages and models, mentioned in previous section, with respect to permanent blocking.

## 8.1 Interface Theory

For complex component-based system designs, compatibility checks need to consider the behaviour of components and their reaction to the environment they are placed in. This problem was addressed by Lynch and Tuttle when they introduced Input/Output Automata (IOAs) [70]. They present a model that allows to describe the behaviour of an algorithm and to compose the descriptions hierarchically. The description is done in the form of an automaton that captures the order of input, output, and internal actions of an algorithm. The concept was later adopted by de Alfaro and Henzinger [15] for their work on Interface Automata (IAs), a model to describe the interface of components. While IAs are syntactically similar to IOAs, the composition operation is defined differently. This is due to the conceptual difference between the two models with respect to the assumptions made on the environment. An IOA has to accept any input of the environment, independent of the state it is in. This lies in contrast to an IA which assumes a helpful environment. This means that IOAs must be able to cope with any environment, while, in the case of IAs, the compatibility check returns a valid result if there is at least one environment that is compatible. IOAs are called input-enabled or pessimistic, while IAs are called optimistic. As a consequence, the input operations of IAs become blocking. IAs and IOAs have been combined by Larsen et al. in their work on Interface Input/Output Automata (IIOAs) [16]. By formally separating *assumptions* and *specifications* of a component, the authors eliminate the blocking inputs of IAs. The separation is achieved by describing each aspect with a separate automaton. The work was further extended in [18] where they introduce Modal Input/Output Automata (MIOAs) in order to distinguish between *must* and *may* modalities of transitions. The modalities are used to indicate whether an instantiation of a MIOA must support the transitions of the generic system description or may be omitted.

Synchronous Interface Automata (SIAs), introduced in this dissertation, are syntactical similar to IAs. There is, however, a fundamental difference in the blocking semantics of output actions. Whenever an output action is produced by an IA it must be accepted by the environment and cannot be blocked by an action that is controlled by another component (output or internal action). A state where an output action cannot be served without blocking is called an error state: The error state of a composed IA $P \otimes Q$ is a state where $P$ produces an output that cannot be served by an input of $Q$ or where $Q$ produces an output that cannot be served by an input of $P$. This semantics allows to guarantee that if a composed IA has no non-autonomously reachable error states, autonomous actions of one component are always served by the other. IAs are, however, a simplification of the synchronous communication semantics and the model can therefore not be used to detect permanent blocking states. SIAs, on the other hand, have blocking input and output actions in order to model synchronous communication and therefore allow to check for the possibility of being blocked in a blocking state indefinitely. Simply put, IOAs and IIOAs are neither blocking on inputs nor outputs, IAs are only blocking on inputs, and SIAs are blocking on inputs and outputs.

Hennicker and Knapp [17] developed a theory based on both, IAs and MIOAs where they not only focus on a pairwise component analysis but consider the interoperability of an assembly of components. They propose to check that the assumption of each component on its environment (the rest of the network) is met, i.e. given an assembly of components $A = \{C_1, \dots, C_n\}$ and an operation $\bigotimes\langle A \rangle$ denoting the n-ary composition of the assembly of components, they compute $\bigotimes\langle A \setminus C_i \rangle \otimes C_i$ for each component $C_i$.

## 8.2 Mixed-criticality Models

In Chapter 4 I described an extension of the Process Network with Synchronous Communication (PNSC) model to allow time-triggered processes to coexist and interact with event-triggered processes. To my knowledge the PNSC model is the first model to incorporate time-triggered and event-triggered semantics where interaction between the two is possible with Cross-criticality Interfaces (CCIs). However, the co-existence of time-triggered and event-triggered communication paradigms on the same system has already been discussed in the past: Pop et al. proposed to separate communication into two phases and statically pass them as input to the scheduler [71]. The work of Ferreira et al. on FTT-CAN is similar to this concept [24]. Steiner proposed a scheduler for mixed-critical CPSs using the concept of *schedule porosity* [25]. Obermaisser proposed an implementation of the Controller Area Network (CAN) bus, which is event-driven, on a system based on the time-triggered communication principle using a tunnelling approach [26]. In terms of mixed-criticality systems, Burns et al. provide a review document that is constantly updated and maintained [2].

Tan et al. made the argument that given that the human society is event-driven, future Cyber-physical Systems (CPSs) should be event-triggered [46]. Given that today most critical applications employ the Time-triggered Architecture (TTA) because of its predictability and fault-tolerance it is unlikely that the time-triggered model will be replaced completely. But because of efficiency reasons and because event-triggered models represent the physical world better the interaction of the two systems is becoming more frequent [61].

## 8.3 Coordination Models and Languages

Since the publication of the first coordination language Linda [32], a multitude of coordination languages have been proposed, based on the same general principle of tuple space [50, 51]. In their landmark survey on coordination languages [41] Papadopoulos and Arbab distinguish between data-driven and control-driven coordination languages where languages based on the tuple space model fall into the former classification. Streamix is a control-driven coordination language because the underlying PNSC model is component-based and corresponds to the black-box approach where the component itself is unaware of being coordinated (such a model is also called an exogenous coordination model [33]). In the following I will describe several coordination models and languages in more detail and compare them to Streamix and its underlying model PNSC.

### 8.3.1 S-Net

A first approach of coordinating streaming applications was introduced with StreamIt [8], a language that allows to create a loosely structured streaming network by interconnecting computational components, called filters, with network constructors such as split-join, feedback, or simple one-to-one streams. The fully fledged coordination language S-Net [10] is also based on streaming networks but unlike StreamIt, S-Net is exogenous and achieves a tighter structuring with network operators. S-Net uses binary operators to construct a streaming network from functional components, interlinked by First-in, First-out (FIFO)-channels. S-Net aims at exploiting the inherent pipelined network structure of streaming applications to run components in parallel. Feedbacks are avoided by replicating components dynamically and building a pipeline with the replicas. As a consequence, S-Net relies on the requirement that all components are pure, i.e. purely functional and therefore without persistent state, and of type Single Input, Single Output (SISO). S-Net features an extensive type system which allows to build arbitrary networks through concepts like flow inheritance and sub-typing. The concept of S-Net to use binary operators to construct a network out of pure SISO components is very appealing because it allows to describe networks in a very concise and structured way and imposes a fixed information flow from the left to the right. However, the complex type system of S-Net makes it exceedingly hard to understand how message tokens are routed in an S-Net network, which contradicts the conciseness of the network operators to some extent. Also, the requirement of only accepting pure components makes S-Net not very suitable for applications with legacy code because it is hard to transform components with persistent state into a system of stateless components.

Streamix is similar to S-Net with respect to their blocking semantics as they both are similar to the model of Kahn Process Network (KPN) [23]. Further, Streamix borrows the concept of using binary operators to describe a network in a structured way from S-Net. The underlying model of the two coordination languages, however, differs largely because S-Net targets transformational systems and aims at exploiting parallel architectures by dynamically proliferating components while Streamix targets CPSs where predictability and analysability is key. One major difference is that Streamix allows components with persistent state, internal synchronisation points, and multiple inputs and multiple outputs while Streamix restricts components to be pure and of type SISO. Consequently, the semantics of the network operators are different in both languages: The network operators of S-Net are specifically designed to allow the programmer to explicitly control how parts of the application are executed concurrently on a multicore architecture. While the operators in Streamix also provide information on whether there is a direct dependency between operands or not, they serve more the purpose of letting a programmer compose reactive networks in a structured manner. By doing this, some of the expressiveness of the language is lost because dependencies are not immediately observable through the operators. What is gained, however, is a convenient and concise way of describing reactive systems. Also, Streamix does not rely on type routing which preserves observable connectivity information at the level of the operators and, thus, provides more local network information than S-Net does. In order to preserve analysability of the system, the underlying PNSC model provides an interface language (SIAs) to describe the interaction behaviour of components with their environment.

### 8.3.2 BIP

The coordination language Behaviour, Interaction, Priorities (BIP) [72, 73] is a language that allows to compose and coordinate concurrent components. A three-layered approach allows a clear separation between computation and coordination. The behaviour layer is an automata-based model that describes the behaviour of a component and the interaction of the component with its interface. The interaction layer describes connections between the interfaces of components. The priority layer is used to impose scheduler constraints. Connections in BIP are stateless, as are the basic connections in the PNSC model presented in this dissertation. By extending the PNSC model, however, a library of modular, potentially stateful channels is provided (e.g. FIFO channels). Further, connections in the PNSC model are one to one while in BIP multiple ports can form a connection. The interaction layer specifies the level of synchronisation (e.g. rendez-vous, broadcast). The interaction model of BIP does not support a decoupling mechanism as described in Section 4.1.2. In BIP, the composition of components is achieved through user-defined connectors where port connections are explicitly listed. In Streamix this is avoided with the help of network operators.

An interesting application of BIP is described in [74] where the behaviour layer is extended by timed automata. This extended BIP model is then used to first, describe a platform independent timing behaviour of a component and second, to represent the physical timing behaviour of a hardware platform. Using a composition operation it can then be checked whether the timing behaviour of the application is compatible with the timing behaviour of the hardware platform. Given the similarity between the two models, it should be possible to apply this method on the PNSC model.

### 8.3.3 Giotto

Giotto [56] is a coordination language that allows the implementation of concurrent real-time tasks while providing *composable I/O behaviour*. The aim is to provide an abstraction for real-time systems to make real-time software less dependent on hardware and to control I/O jitter. Giotto is based on the Logical Execution Time (LET) model where tasks are executed within a statically allocated time slot. At the beginning of the LET inputs are made available and outputs are released at the end of the LET of a task. The time instants of input and output operations are independent of the execution time as it is not specified when and where tasks are executed.

The extended PNSC model achieves the LET behaviour by replacing input and output channels of processes by temporal firewalls (see Section 4.2.1). The coordination language Streamix allows to instantiate such temporal firewalls with the `tt` operator on a single net or a composed net (see Section 6.4.5).

### 8.3.4 Ptolemy and Ptides

The focus of the Ptolemy [3] project lies on providing a unified modelling language that allows to integrate and compose heterogeneous real-time systems. Ptolemy is based on an actor-oriented design [9] where actors are executed concurrently and communicate with other actors by sending message tokens through ports (not to be confused with the actor model [29]). It allows to model heterogeneous systems by assigning different predefined model semantics, called Model of Computations (MoCs), to assemblies of actors, called platforms. Ptolemy supports many different MoCs, i.a. KPN [23], the LET model of Giotto [56], Synchronous Data Flow (SDF) [11] (e.g. the underlying model of StreamIt [8]), or the discrete-event model [75]. A detailed description of Ptolemy and an extensive list of available MoCs is provided in [55].

In contrast to the Ptolemy project, the presented approach in this dissertation is not an assembly of submodels but one single model where a modular composition operation applied on processes preserves the model properties, independent of the triggering semantics or communication coupling of the individual processes. It is clear that the here presented model is not as diverse as the Ptolemy project but it provides well defined interfaces between interacting subsystems with different triggering semantics and provides flexible control over communication coupling which allows the design of mixed-criticality systems. Further, in contrast to the underlying model of Ptolemy where one actor models a simple task, the PNSC allows to model complex tasks with multiple synchronisation points within.

One MoC of Ptolemy that is particularly interesting and needs to be considered for future work on the PNSC model is Ptides [54]. Ptides extends the discrete-event MoC and allows to describe the timing behaviour in a network of event-triggered actors. This is achieved by distinguishing between physical time and model time where the former is tied to sensors and actuators, i.e. components that interact with the physical world, and the progression of the latter is specified within the model. It is a promising approach to loosen the coupling between the software development of time-critical systems and the hardware platform where the system is executed.

## 8.4 Deadlocks and Permanent Blocking

In this section I discuss research related to the analysis of deadlocks and permanent blocking presented in Chapter 5. I will discuss methods already mentioned in the sections before but also mention approaches that are not related to already discussed related work. Neither of the methods discussed in the following make a distinction between deadlocks and permanent blocking but always use the term deadlock, in some cases in accordance with our definition but more often as a synonym for what we describe as permanent blocking.

The work on interface theory, presented in Section 8.1, aims at checking the interoperability of components. The goal is to check whether a component is compatible with the environment it is placed in. In [16, 18], Larsen et al. relate mutual deadlock freeness of two components to whether they are violating each other's assumptions or not. However, in their work on an assembly theory [17], Hennicker and Knapp point out that neither IAs nor MIOAs imply deadlock, thus an analysis of permanent blocking, as presented in Chapter 5, is needed. Note that they use examples of lonely blockers and address them as deadlocks. Further note that because MIOAs are input-enabled, a component has to be able to cope with any input in any state. This delegates the handling of unexpected inputs to the component implementation and thus prevents permanent blocking at the interface as long as the implementation is correct.

While interface theory is component centric and uses interface specification to check for compatibility of components, session types use a communication-centric approach and specify the protocol between components in order to check for compatibility. Bartoletti et al. present a survey on behavioural contracts in [76] where session types are related to IAs and classified as a subset of IAs. Further, they define binary asymmetric compliance relations, amongst others, the notion of progress. Their definition differs from the one given in this dissertation in two points: Firstly, they require each subsystem to reach a success state where we consider any action as a contribution to progress. Secondly, they do not make a distinction between shared and open actions because they restrict the notion of progress to two participants.

When it comes to CPSs, one often distinguishes between two types of tasks: simple tasks (*S-task*) and complex tasks (*C-task*) [43]. An S-task has no synchronization point within while a C-task has one or multiple blocking synchronization points. An example of a streaming network composed purely out of S-tasks is the Ptolemy II project [9]. Zhou and Lee present a statical deadlock analysis of such a streaming network in [12]. The approach is based on *causality interfaces* where port to port dependencies are described as functions. A function $d(n)$ describes the output rate with the parameter $n$ describing the input rate. The approach is general in the sense that the multiplicity of each input-output relation is not static. Operators are used to compose the dependencies in order to form causality interfaces. To detect deadlocks, causality loops are identified and checked against a certain criteria. Given that the analysis is based on circular dependencies, the approach is suitable to identify deadlocks but not permanent blocking.

As described by Lee et al. in [9], the triggering semantics of Ptolemy II actors (an equivalent to processes as described in this dissertation) requires actors to have no synchronization point. This imposes more restrictions on the actor implementation than the model presented in this dissertation does on process implementations. Hence, their approach to detect deadlocks cannot simply be applied to my model. In order to apply their analysis to PNSCs, each process in a PNSC would have to be split into multiple Ptolemy II actors to eliminate blocking synchronization points within the processes. This is a difficult task because it would require to transform processes with persistent state into a system of stateless components.

Kroening et al. present an approach to guarantee deadlock-freedom in C/Pthread programs by statically analysing the code [77]. As their analysis is tailored for a specific technology (C/Pthread) it is applied on a lower level of abstraction than the work presented in Chapter 5 which is not restricted to a particular programming language. Furthermore, the approach checks for cycles in a lock-graph and therefore focuses on deadlocks but not permanent blocking.

A topic related to static detection of permanent blocking situations is preventing them out of construction due to inherent properties of the model. The coordination language BIP [72] uses an underlying model [40] that avoids permanent blocking by construction: BIP describes the behaviour of components with an automata-based model and composes components based on different synchronization protocols (e.g. rendez-vous, broadcast). Freedom of permanent blocking is guaranteed with priority rules that allow to describe alternative actions, should an interaction be prevented from occurring. As a consequence, this means that priority rules have to be met by the implementation of the component and hence, handling of permanent blocking is delegated to the implementation.

Another coordination language, offering freedom of deadlocks by construction, is S-Net [10]. S-Net uses binary operators to construct a streaming network from functional components, interlinked by FIFO-channels. Freedom of deadlocks is guaranteed due to the absence of feedback loops. Feedbacks are avoided by dynamically, potentially infinitely, replicating components and arranging them in a pipeline. S-Net is not completely free of permanent blocking because an implementation with finite buffer sizes can cause starvation, which is equivalent to what we call a lonely blocking situation. Later, feedback loops were introduced to S-Net [78], for which to preserve the guarantee of deadlocks freedom they had to introduce the assumption of infinite buffer length on the feedback channel.

Communicating Sequential Processes (CSP) is a process calculus introduced by Tony Hoare where atomic processes, which can be considered as labelled transition systems, communicate over a set of channels [30]. In his book, A. W. Roscoe [79] presents a refined theory of CSP and discusses deadlocks in Chapter 13 where he proposes a number of design rules to avoid deadlocks. The proposed methods are all based on a local analysis, involving only a small collection of processes. He states that, ultimately, a state space exploration is required to decide on global deadlock-freedom which corresponds to the approach taken in this thesis by using the underlying model described in Chapter 5.

A method that is often used in industry is the so-called *Ostrich Algorithm* as mentioned by Tanenbaum and Bos [80]. The rationale behind this is that permanent blocking is happening so rarely that it is not worth the price to invest resources to handle such cases. However, ignoring the potential problem is not a suitable solution for critical systems which are addressed by the work presented in this dissertation.

# Chapter 9

# Conclusion and Outlook

In this chapter I conclude my dissertation by providing a summary of the accomplishments, discussing the result, and giving an outlook for future research to improve and refine aspects of the work presented in this dissertation.

## 9.1 Summary of the Dissertation

In my work I introduced the coordination model *Process Network with Synchronous Communication (PNSC)* with the aim to describe concurrent, time-critical Cyber-physical Systems (CPSs). The coordination model provides a separation between the concerns of computation and coordination (i.e. communication and synchronisation). This is achieved by a component-based design where computational components are abstracted by an interface description. I introduced Synchronous Interface Automata (SIAs), an automata-based interface language that allows to clearly define the communication protocol of interacting components and incrementally compose interfaces of components. The interaction of components uses a blocking semantics that allows to model streaming networks where complex components, i.e. components with persistent state and internal synchronisation points, interact over synchronous communication channels. An extension of the model supports an event-triggered as well as a time-triggered execution strategy and allows interaction between components independent of the execution strategy. A communication decoupling mechanism allows to control the level of interference and thus, provides support for mixed-criticality systems. Further, the extended model provides rate-control mechanisms for communication and computation to allow the specification of timing requirements of an application. Computational components are oblivious of the coordination that is excerpt on them by the model, hence, the domain expert who implements the computation component is not required to know the context in which the component will be used. To statically check for freedom of permanent blocking in an assembly of interacting computational components, I introduced an analytic method that uses the SIA interface description of a composed system to identify states in the system that can cause permanent blocking.

As an instance of the coordination model PNSC I designed the coordination language Streamix. Streamix is an exogenous coordination language that allows to construct networks of concurrent processes in a structured and hierarchical way. It accomplishes this by using network operators that allow to compose simple networks into more complex ones. The language uses the extended PNSC model as a backbone to detect permanent blocking situations in a composed network. Streamix supports all the features of the extended PNSC model and adds syntax and structure to it. I implemented a prototype of a compiler, a Run-time System (RTS) preprocessor, an RTS, and a SIA model checker to allow to build executable applications with the Streamix coordination language.

## 9.2 Discussion of the Results

The main priority of a coordination model is to separate the concerns of computation and coordination. The PNSC model provides separation of concerns out of construction because of the process abstraction through SIAs and the communication decoupling in space: The behavioural description of a process is solely tied to the interaction protocol specified by its SIA which describes the temporal ordering of read and write operations of the process. Read and write operations are performed on ports without any knowledge of what is connected to the port. This is by itself not a novelty as the principle of using channels as coordination primitives, or glue-code, between components has been used in many instances (e.g. [10, 53]) and the concept of interface theories was initially proposed by de Alfaro and Henzinger [14]. What is new however, is the combination of coordination primitives and an interface theory that allows to model a network of complex, reactive components with a blocking semantics of interacting processes that is fundamentally different from interface theories such as Interface Automata (IAs) [15]. The permanent blocking analysis, introduced in this dissertation, is based on the blocking semantics of SIAs and is a novel approach to statically detect permanent blocking situations in a network of concurrent processes. The coordination model PNSC offers Cross-criticality Interface (CCI) to selectively decouple communication in synchronisation and remove the blocking behaviour in order to prevent unwanted interference between subsystems. This mechanism, in conjunction with clock signals, is further used to create temporal firewalls that allow to create subnets of time-triggered processes, similar to Giotto [56]. The novelty of the PNSC model is the concept of selectively introducing temporal firewalls where required and allow interaction between time-triggered and event-triggered processes. PNSCs also allow to control the communication rate of single channels in order to increase the predictability of event-triggered communication.

All the above mentioned coordination aspects are applied on processes without any requirement of changing the implementation of the processes. The RTS implementation for the coordination language Streamix achieves this by providing one singe macro function for read access and one single macro function for write access to channels connected to the process. The semantics of the read and write operations differs, depending on the coordination elements that are specified in the Streamix program. The macro functions allow to access the channels by name mapping and position mapping where the former tends to be more convenient for humans while the latter is more convenient for machines.

The coordination language Streamix adds syntax to the PNSC model and allows to structure a network of processes by using network operators. Structure in a program is provided by keeping information local. This means that when composing a network out of processes, the composition operation must hold as much information about the connections of the individual processes as possible. To gain expressiveness, Streamix provides multiple composition operators to describe different types of grouping, e.g. parallel grouping where no connections are established between processes or serial grouping, i.e. enforcing connections between processes. It is a difficult act to balance between providing convenience for the programmer and keeping the expressiveness of an operator: An example in Streamix is the *bypassing serial composition* operator as defined in Definition 6.6. It relaxes the more strict requirements of the *locally enforcing serial composition* operator to allow the construction of more complex connections in a network (e.g. bypassing processes). Such networks can also be built with the wrapper operator (see Section 6.4.1) but at the cost of more writing work. However, the advantage of the wrapper is that it provides a better code structure as local connections are kept local. There is no definitive answer on which option is the better.

The PNSC model (and therefore also Streamix) targets mixed-criticality CPSs. As a conclusion, let me summarize the main properties of the model that support this claim:

**Time-criticality of CPSs** A main requirement to give guarantees on timing requirements is predictability and analysability. The PNSC achieves the former through rate-control, i.e. limiting the communication rate and imposing a fixed rate to achieve a time-triggered communication semantics. The latter is achieved with interface abstraction of the SIA model that allows to check whether a PNSC is free of permanent blocking situations.

**Mixed-criticality of CPSs** The CCIs provided by the extended PNSC model allow to describe a system that simultaneously operates with event-triggered and time-triggered communication. The corner stone of the CCIs is the selectively applicable mechanism to decouple communication in synchronisation that allows interaction between components with different criticality levels while preventing unwanted interference.

**The Reactive Nature of CPSs** The PNSC model copes with the reactive nature of CPS by allowing complex processes with persistent state and internal synchronisation points. SIAs allow to preserve a detailed understanding of how processes interact and serve as instrument to detect permanent blocking situations in a network. Further, the coordination language Streamix provides convenient network operators that allow to structure a network of processes in a concise way without removing the possibility to describe bidirectional communication between processes. Because a PNSC process can be of arbitrary complexity, the model is expected to be well suited to support legacy code.

## 9.3 Outlook

One of the achievements of my work is the interface model for blocking communication and based on it the permanent blocking analysis that finds potential permanent blocking states in a composed PNSC. While the analysis performed on a composed PNSC is of linear time complexity $\mathcal{O}(S + T)$, where $S$ denotes the number of states and $T$ the number of transitions in a composed system, the number of states $S$ and transitions $T$ grow exponentially with respect to the number of subsystems. This problem is known as the state space explosion of automata composition. There are existing approaches that help to alleviate this problem (e.g. reducing internal actions) which needs to be addressed in future work.

The SIA model, serving as a backbone of the permanent blocking analysis, was inspired by the work of de Alfaro and Henzinger on IAs [15]. A property of IAs is *independent implementability* which means that if an interface is compatible with a system it can be refined separately and remains compatible. No refinement for SIAs was defined throughout this dissertation and it is left to future work to define refinement conditions suitable for SIAs.

The software development of a time-critical system is still often tightly coupled to a specific hardware platform because a time-critical application must be guaranteed to produce a correct result before a specified deadline. As the execution time of an application is heavily influenced by the hardware platform the application is running on, such guarantees are usually given by extensive testing of the software application in conjunction with its hardware platform. This leads to the problem that whenever something has to be changed (in hardware or software) these extensive tests have to be performed again. The aim is to loosen the tight coupling between the software application and the hardware platform of a time-critical system in order to facilitate the development and maintenance process of cyber-physical systems. This would alleviate the necessity of expensive, holistic testing and allow the efficient redeployment of developed applications on different platforms.

A coordination model, such as the one presented in this dissertation, can help to get closer to this goal with separation of concerns: Computational components can be checked for correctness independently and the coordination model allows to assemble the components into a network by keeping a tight control on the interactions of the components. The time-triggered communication semantics of a PNSC allows to design a predictable system where each time-triggered process can be analysed individually. In the current model it must be guaranteed that the Worst-case Execution Time (WCET) of the process implementation respects the specified timing constraints of the model. As future work it would be interesting to extend SIAs with a timed automata model to describe the timing behaviour of a process (similar to the Behaviour, Interaction, Priorities (BIP) [74] approach mentioned in Section 8.3.2).

Given that the PNSC model allows to describe systems that operate simultaneously with time-triggered and event-triggered components, timing specifications solely based on rate-control are not sufficient to fully describe time-critical systems. The specification of latency must also be supported for an event-triggered subnetwork. This is an aspect that was not covered in this dissertation. An interesting approach related to this problem was introduced with Ptides [54]. It remains to be seen whether a similar approach can be adapted to work with complex processes where the interaction of a process with its environment is described by a (timed) automata model. Further, in parallel to the work presented in this dissertation we (Kirner and Maurer) published a paper that discusses the problem of specifying timing requirements in streaming networks and identified some pitfalls when it comes to latency specifications in streaming networks [37]. One challenge is to split global, system-wide timing requirements into meaningful chunks that can be attributed to computational components. This is a problem that has, to my knowledge, not yet been addressed.

Last but not least, I give an outlook on research for the coordination language Streamix and the toolset I built around it. For my thesis it served as a proof of concept to show the usability of the PNSC and SIA model. However, to exploit all the features offered by the language the RTS must support a mixed-criticality scheduler for multi-core architectures. This is a huge research field (Burns and Davis provide a review on the topic [2]) and requires a thorough analysis.

# Bibliography

[1] Thomas A. Henzinger and Joseph Sifakis. The Embedded Systems Design Challenge. In *FM 2006: Formal Methods*, pages 1–15. Springer, Berlin, Heidelberg, August 2006. doi: 10.1007/11813040_1.

[2] Alan Burns and Rob Davis. Mixed Criticality Systems: A Review. *Department of Computer Science, University of York, Tech. Rep*, December 2016.

[3] Johan. Eker, Jorn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming Heterogeneity - The Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2002.805829.

[4] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical Systems: The Next Computing Revolution. In *Design Automation Conference*, pages 731–736, June 2010. doi: 10.1145/1837274.1837461.

[5] Jeronimo Castrillon, Lothar Thiele, Lars Schorr, Weihua Sheng, Ben Juurlink, Mauricio Alvarez-Mesa, Angela Pohl, Ralph Jessenberger, Victor Reyes, and Rainer Leupers. Multi/Many-core Programming: Where Are We Standing? In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 1708–1717, San Jose, CA, USA, 2015. EDA Consortium.

[6] Edward A. Lee. Cyber Physical Systems: Design Challenges. In *2008 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, May 2008. ISBN 978-0-7695-3132-8. doi: 10.1109/ISORC.2008.25.

[7] Robert Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7):491–541, July 1997. ISSN 0001-5903, 1432-0525. doi: 10.1007/s002360050095.

[8] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Compiler Construction*, number 2304 in Lecture Notes in Computer Science, pages 179–196. Springer Berlin Heidelberg, January 2002. ISBN 978-3-540-43369-9 978-3-540-45937-8.

[9] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-Oriented Design of Embedded Hardware and Software Systems. *J CIRCUIT SYST COMP*, 12(03):231–260, June 2003. ISSN 0218-1266. doi: 10.1142/S0218126603000751.

[10] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. Asynchronous Stream Processing with S-Net. *Int J Parallel Prog*, 38(1):38–67, February 2010. ISSN 0885-7458, 1573-7640. doi: 10.1007/s10766-009-0121-x.

[11] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987. ISSN 0018-9219. doi: 10.1109/PROC.1987.13876.

[12] Ye Zhou and Edward A. Lee. A Causality Interface for Deadlock Analysis in Dataflow. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, EMSOFT '06, pages 44–52, New York, NY, USA, 2006. ACM. ISBN 978-1-59593-542-7. doi: 10.1145/1176887.1176895.

[13] David Harel and Amir Pnueli. On the Development of Reactive Systems. In *Logics and Models of Concurrent Systems*, NATO ASI Series, pages 477–498. Springer, Berlin, Heidelberg, 1985. ISBN 978-3-642-82455-5 978-3-642-82453-1. doi: 10.1007/978-3-642-82453-1_17.

[14] Luca de Alfaro and Thomas A. Henzinger. Interface Theories for Component-Based Design. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software*, number 2211 in Lecture Notes in Computer Science, pages 148–165. Springer Berlin Heidelberg, October 2001. ISBN 978-3-540-42673-8 978-3-540-45449-6. doi: 10.1007/3-540-45449-7_11.

[15] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 109–120, New York, NY, USA, 2001. ACM. ISBN 978-1-58113-390-5. doi: 10.1145/503209.503226.

[16] Kim G. Larsen, Ulrik Nyman, and Andrzej Wąsowski. Interface Input/Output Automata. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, number 4085 in Lecture Notes in Computer Science, pages 82–97. Springer Berlin Heidelberg, August 2006. ISBN 978-3-540-37215-8 978-3-540-37216-5. doi: 10.1007/11813040_7.

[17] Rolf Hennicker and Alexander Knapp. Moving from Interface Theories to Assembly Theories. *Acta Informatica*, 52(2-3):235–268, March 2015. ISSN 0001-5903, 1432-0525. doi: 10.1007/s00236-015-0220-7.

[18] Kim G. Larsen, Ulrik Nyman, and Andrzej Wąsowski. Modal I/O Automata for Interface and Product Line Theories. In Rocco De Nicola, editor, *Programming Languages and Systems*, number 4421 in Lecture Notes in Computer Science, pages 64–79. Springer Berlin Heidelberg, March 2007. ISBN 978-3-540-71314-2 978-3-540-71316-6. doi: 10.1007/978-3-540-71316-6_6.

[19] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Resource Interfaces. In Rajeev Alur and Insup Lee, editors, *Embedded Software*, number 2855 in Lecture Notes in Computer Science, pages 117–133. Springer Berlin Heidelberg, October 2003. ISBN 978-3-540-20223-3 978-3-540-45212-6. doi: 10.1007/978-3-540-45212-6_9.

[20] Lothar Thiele, Ernesto Wandeler, and Nikolay Stoimenov. Real-time Interfaces for Composing Real-time Systems. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, EMSOFT '06, pages 34–43, New York, NY, USA, 2006. ACM. ISBN 978-1-59593-542-7. doi: 10.1145/1176887.1176894.

[21] Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Timed Interfaces. In Alberto Sangiovanni-Vincentelli and Joseph Sifakis, editors, *Embedded Software*, number 2491 in Lecture Notes in Computer Science, pages 108–122. Springer Berlin Heidelberg, October 2002. ISBN 978-3-540-44307-0 978-3-540-45828-9. doi: 10.1007/3-540-45828-X_9.

[22] Thomas A. Henzinger and Slobodan Matic. An Interface Algebra for Real-Time Components. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 253–266, April 2006. doi: 10.1109/RTAS.2006.11.

[23] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *In Information Processing'74: Proceedings of the IFIP Congress*, volume 74, pages 471–475, 1974.

[24] Joaquim Ferreira, Piaulo Pedreiras, Luis Almeida, and José A. Fonseca. The FTT-CAN Protocol for Flexibility in Safety-Critical Systems. *IEEE Micro*, 22(4):46–55, July 2002. ISSN 0272-1732. doi: 10.1109/MM.2002.1028475.

[25] Wilfried Steiner. Synthesis of Static Communication Schedules for Mixed-Criticality Systems. In *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, pages 11–18, March 2011. doi: 10.1109/ISORCW.2011.12.

[26] Roman Obermaisser. Reuse of CAN-Based Legacy Applications in Time-Triggered Architectures. *IEEE Transactions on Industrial Informatics*, 2(4):255–268, November 2006. ISSN 1551-3203. doi: 10.1109/TII.2006.885920.

[27] Hermann Kopetz and Roman Obermaisser. Temporal Composability. *Computing Control Engineering Journal*, 13(4):156–162, August 2002. ISSN 0956-3385. doi: 10.1049/cce:20020401.

[28] Fred J. Pollack. New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies (Keynote Address). In *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 978-0-7695-0437-7.

[29] Gul Abdulnabi Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. Phd, MIT, Massachusetts, US, June 1985.

[30] Charles A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21 (8):666–677, August 1978. ISSN 0001-0782. doi: 10.1145/359576.359585.

[31] Open MPI: Open Source High Performance Computing. `https://www.open-mpi.org/`.

[32] David Gelernter and Nicholas Carriero. Coordination Languages and Their Significance. *Commun. ACM*, 35(2):97–107, February 1992. ISSN 0001-0782. doi: 10.1145/129630.129635.

[33] Farhad Arbab. Composition of Interacting Computations. In Dina Goldin, Scott A. Smolka, and Peter Wegner, editors, *Interactive Computation*, pages 277–321. Springer Berlin Heidelberg, January 2006. ISBN 978-3-540-34666-1 978-3-540-34874-0.

[34] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Publishing Company, Incorporated, 2nd edition, 2011. ISBN 978-1-4419-8236-0.

[35] Simon Maurer and Raimund Kirner. Coordination with Structured Composition for Cyber-physical Systems. In *Parallel Computing: On the Road to Exascale*, volume 27 of *Advances in Parallel Computing*, pages 615 – 624, Edinburgh, UK, September 2015. IOS Press. ISBN 978-1-61499-620-0. doi: 10.3233/978-1-61499-621-7-615.

[36] Simon Maurer and Raimund Kirner. Cross-criticality Interfaces for Cyber-physical Systems. In *Proc. 1st IEEE Int'l Conference on Event-Based Control, Communication, and Signal Processing*, pages 1–8, Krakow, Poland, June 2015. IEEE. doi: 10.1109/EBCCSP.2015.7300670.

[37] Raimund Kirner and Simon Maurer. On the Specification of Real-time Properties of Streaming Networks. In *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, Kärnten, Austria, October 2015.

[38] Simon Maurer, Raimund Kirner, and Olga Tveretina. Static Deadlock Analysis of Process Networks with Synchronous Interface Automata. *Ready for Submission*, 2017.

[39] Luca de Alfaro and Thomas A. Henzinger. Interface-Based Design. In Manfred Broy, Johannes Grünbauer, David Harel, and Tony Hoare, editors, *Engineering Theories of Software Intensive Systems*, number 195 in NATO Science Series, pages 83–104. Springer Netherlands, 2005. ISBN 978-1-4020-3530-2 978-1-4020-3532-6. doi: 10.1007/1-4020-3532-2_3.

[40] Gregor Gössler and Joseph Sifakis. Composition for Component-Based Modeling. In *Formal Methods for Components and Objects*, pages 443–466. Springer, Berlin, Heidelberg, November 2002. doi: 10.1007/978-3-540-39656-7_19.

[41] George A. Papadopoulos and Farhad Arbab. Coordination Models and Languages. In *Advances in Computers*, volume 46, pages 329–400. Elsevier, 1998. ISBN 0065-2458.

[42] Edward. G. Coffman, M. J. Elphick, and Arie Shoshani. System Deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971. ISSN 0360-0300. doi: 10.1145/356586.356588.

[43] Hermann Kopetz. Task Management. In *Real-Time Systems: Design Principles for Distributed Embedded Applications*, pages 218–221. Springer Publishing Company, Incorporated, 2nd edition, 2011. ISBN 978-1-4419-8236-0.

[44] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. ISSN 1539-9087. doi: 10.1145/1347375.1347389.

[45] Steve Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 239–243, December 2007. doi: 10.1109/RTSS.2007.47.

[46] Ying Tan, Steve Goddard, and Lance C. Pérez. A Prototype Architecture for Cyber-physical Systems. *SIGBED Rev.*, 5(1):26:1–26:2, January 2008. ISSN 1551-3688. doi: 10.1145/1366283.1366309.

[47] Hermann Kopetz. The Time-triggered Architecture. In *Real-Time Systems: Design Principles for Distributed Embedded Applications*, pages 325–339. Springer Publishing Company, Incorporated, 2nd edition, 2011. ISBN 978-1-4419-8236-0.

[48] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003. ISSN 0360-0300. doi: 10.1145/857076.857078.

[49] Lachlan Aldred, Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. On the Notion of Coupling in Communication Middleware. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, number 3761 in Lecture Notes in Computer Science, pages 1015–1033. Springer Berlin Heidelberg, January 2005. ISBN 978-3-540-29738-3 978-3-540-32120-0.

[50] Davide Rossi, Giacomo Cabri, and Enrico Denti. Tuple-based Technologies for Coordination. In Andrea Omicini, Franco Zambonelli, Matthias Klusch, and Robert Tolksdorf, editors, *Coordination of Internet Agents*, pages 83–109. Springer Berlin Heidelberg, January 2001. ISBN 978-3-642-07488-2 978-3-662-04401-8.

[51] Andrea Omicini and Mirko Viroli. Coordination Models and Languages: From Parallel Computing to Self-Organisation. *Knowledge Eng. Review*, 26:53–59, 2011. ISSN 1469-8005. doi: 10.1017/S026988891000041X.

[52] Farhad. Arbab, Ivan Herman, and Per Spilling. An Overview of Manifold and Its Implementation. *Concurrency: Pract. Exper.*, 5(1):23–70, February 1993. ISSN 1040-3108. doi: 10.1002/cpe.4330050103.

[53] Farhad Arbab. Reo: A Channel-based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14(03):329–366, June 2004. ISSN 1469-8072. doi: 10.1017/S0960129504004153.

[54] Patricia Derler, Thomas Huining Feng, Edward A. Lee, Slobodan Matic, Hiren D. Patel, Yang Zhao, and Jia Zou. PTIDES: A Programming Model for Distributed Real-Time Embedded Systems. Technical Report UCB/EECS-2008-72, EECS Department, University of California, Berkeley, May 2008.

[55] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation Using Ptolemy II*. Ptolemy.org, 2014.

[56] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A Time-Triggered Language for Embedded Programming. In *Embedded Software*, number 2211 in Lecture Notes in Computer Science, pages 166–184. Springer Berlin Heidelberg, January 2001. ISBN 978-3-540-42673-8 978-3-540-45449-6.

[57] Jan F. Groote and Alex Sellink. Confluence for Process Verification. *Theoretical Computer Science*, 170(1):47–81, December 1996. ISSN 0304-3975. doi: 10.1016/S0304-3975(96)80702-X.

[58] Hermann Kopetz. Temporal Control Versus Logical Control. In *Real-Time Systems: Design Principles for Distributed Embedded Applications*, pages 82–84. Springer Publishing Company, Incorporated, 2nd edition, 2011. ISBN 978-1-4419-8236-0.

[59] Hermann Kopetz. The Message Concept. In *Real-Time Systems: Design Principles for Distributed Embedded Applications*, pages 88–91. Springer Publishing Company, Incorporated, 2nd edition, 2011. ISBN 978-1-4419-8236-0.

[60] Davis Powell. Time/Event Triggering is Orthogonal to State/Event Observation. Workshop Presentation, October 2002.

[61] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A Distributed Run-Time Environment for the Kalray MPPA®-256 Integrated Manycore Processor. *Procedia Computer Science*, 18:1654–1663, January 2013. ISSN 1877-0509. doi: 10.1016/j.procs.2013.05.333.

[62] Gordon J. Pace, Frédéric Lang, and Radu Mateescu. Calculating $\tau$-Confluence Compositionally. In *Computer Aided Verification*, pages 446–459. Springer, Berlin, Heidelberg, July 2003. doi: 10.1007/978-3-540-45069-6_41.

[63] Donald B. Johnson. Finding All the Elementary Circuits of a Directed Graph. *SIAM J. Comput.*, 4(1):77–84, March 1975. ISSN 0097-5397. doi: 10.1137/0204007.

[64] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, June 1972. ISSN 0097-5397. doi: 10.1137/0201010.

[65] Farhad Arbab. What Do You Mean, Coordination? In *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, pages 11–22, 1998.

[66] Carl Bergenhem, Steven Shladover, Erik Coelingh, Christoffer Englund, and Sadayuki Tsugawa. Overview of Platooning Systems. In *Proceedings of the 19th ITS World Congress, Oct 22-26, Vienna, Austria (2012)*, 2012.

[67] The Open Group and IEEE. *Information Technology - Portable Operating System Interface (POSIX) Base Specifications, Issue 7*. IEEE Std 1003.1-2008. IEEE, September 2009. ISBN 978-0-7381-6032-0.

[68] Michael Himsolt. GML: A portable Graph File Format. Techincal report, Universität Passau, 94030 Passau, Germany, 1996.

[69] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and Scott M. Marshall. GraphML Progress Report Structural Layer Proposal. In *Graph Drawing*, Lecture Notes in Computer Science, pages 501–512. Springer, Berlin, Heidelberg, September 2001. ISBN 978-3-540-43309-5 978-3-540-45848-7. doi: 10.1007/3-540-45848-4_59.

[70] Nancy A. Lynch and Mark R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 137–151, New York, NY, USA, 1987. ACM. ISBN 978-0-89791-239-6. doi: 10.1145/41840.41852.

[71] Traian Pop, Petru Eles, and Zebo Peng. Holistic Scheduling and Analysis of Mixed Time/Event-triggered Distributed Embedded Systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, pages 187–192, New York, NY, USA, 2002. ACM. ISBN 1-58113-542-4. doi: 10.1145/774789.774828.

[72] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 3–12, September 2006. doi: 10.1109/SEFM.2006.27.

[73] Simon Bliudze and Joseph Sifakis. The Algebra of Connectors - Structuring Interaction in BIP. *IEEE Transactions on Computers*, 57(10):1315–1330, October 2008. ISSN 0018-9340. doi: 10.1109/TC.2008.26.

[74] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. Model-based Implementation of Real-time Applications. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '10, pages 229–238, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-904-6. doi: 10.1145/1879021.1879052.

[75] Edward A. Lee. Modeling Concurrent Real-Time Processes Using Discrete Events. *Annals of Software Engineering*, 7(1-4):25–45, October 1999. ISSN 1022-7091, 1573-7489. doi: 10.1023/A:1018998524196.

[76] Massimo Bartoletti, Tiziana Cimoli, and Roberto Zunino. Compliance in Behavioural Contracts: A Brief Survey. In Chiara Bodei, Gian-Luigi Ferrari, and Corrado Priami, editors, *Programming Languages with Applications to Biology and Security*, number 9465 in Lecture Notes in Computer Science, pages 103–121. Springer International Publishing, 2015. ISBN 978-3-319-25526-2 978-3-319-25527-9. doi: 10.1007/978-3-319-25527-9_9.

[77] Daniel Kroening, Daniel Poetzl, Peter Schrammel, and Björn Wachter. Sound Static Deadlock Analysis for C/Pthreads. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 379–390, September 2016.

[78] Clemens Grelck and Alex Shafarenko. S-Net Language Report. Technical Report 2.1, version 2.0. Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, AL10 9AB, United Kingdom, August 2013.

[79] Bill Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. ISBN 978-0-13-674409-2.

[80] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems.* Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014. ISBN 978-0-13-359162-0.