Verification of Weakly-Hard Requirements on Quasi-Synchronous Systems

Gideon Smeding

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : l'arrêté du 07 août 2006

Présentée par

Gideon Smeding

Thèse dirigée par **Joseph Sifakis**
et codirigée par **Gregor Goessler**

préparée au sein **INRIA Rhône Alpes**
et de **l'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Verification of Weakly-Hard Requirements on Quasi-Synchronous Systems

Vérification de propriétés faiblement dures des systèmes quasi-synchrones

Thèse soutenue publiquement le **19 Décembre 2013**,
devant le jury composé de :

Marc Pouzet
Professeur, UPMC / ENS, Rapporteur
Xavier Rival
Chargé de Recherche, INRIA / ENS, Rapporteur
Fran cois Vernadet
Professeur, INSA Toulouse / LAAS, Examinateur
Joseph Sifakis
Professeur, EPFL, Directeur de thèse
Gregor Goessler
Chargé de Recherche, INRIA, Co-Directeur de thèse

Abstract

The synchronous approach to reactive systems, where time evolves by globally synchronized discrete steps, has proven successful for the design of safety-critical embedded systems. Synchronous systems are often distributed over asynchronous architectures for reasons of performance or physical constraints of the application. Such distributions typically require communication and synchronization protocols to preserve the synchronous semantics. In practice, protocols often have a significant overhead that may conflict with design constraints such as maximum available buffer space, minimum reaction time, and robustness.

The quasi-synchronous approach considers independently clocked, synchronous components that interact via communication-by-sampling or FIFO channels. In such systems we can move from total synchrony, where all clocks tick simultaneously, to global asynchrony by relaxing constraints on the clocks and without additional protocols. Relaxing the constraints adds different behaviors depending on the interleavings of clock ticks. In the case of data-flow systems, one behavior is different from another when the values and timing of items in a flow of one behavior differ from the values and timing of items in the same flow of the other behavior. In many systems, such as distributed control systems, the occasional difference is acceptable as long as the frequency of such differences is bounded. We suppose hard bounds on the frequency of deviating items in a flow with, what we call, weakly-hard requirements, e.g., the maximum number deviations out of a given number of consecutive items.

We define relative drift bounds on pairs of recurring events such as clock ticks, the occurrence of a difference or the arrival of a message. Drift bounds express constraints on the stability of clocks, e.g., at least two ticks of one per three consecutive ticks of the other. Drift bounds also describe weakly-hard requirements. This thesis presents analyses to verify weakly-hard requirements and infer weakly-hard properties of basic synchronous data-flow programs with asynchronous communication-by-sampling when executed with clocks described by drift bounds. Moreover, we use drift bounds as an abstraction in a performance analysis of stream processing systems based on FIFO-channels.

Résumé

L'approche synchrone aux systèmes réactifs, où le temps global est une séquence d'instants discrets, a été proposée afin de faciliter la conception des systèmes embarqués critiques. Des systèmes synchrones sont souvent réalisés sur des architectures asynchrones pour des raisons de performance ou de contraintes physiques de l'application. Une répartition d'un système synchrone sur une architecture asynchrone nécessite des protocoles de communication et de synchronisation pour préserver la sémantique synchrone. En pratique, les protocoles peut avoir un coût important qui peut entrer en conflit avec les contraintes de l'application comme, par exemple, la taille de mémoire disponible, le temps de réaction, ou le débit global.

L'approche quasi-synchrone utilise des composants synchrones avec des horloges indépendantes. Les composants communiquent par échantillonnage de mémoire partagée ou par des tampons FIFO. On peut exécuter un tel système de façon synchrone, où toutes les horloges avancent simultanément, ou de façon asynchrone avec moins de contraintes sur les horloges, sans ajouter des protocoles. Plus les contraintes sont relâchées, plus de comportements se rajoutent en fonction de l'entrelacement des tics des horloges. Dans le cas de systèmes flots de données, un comportement est différent d'un autre si les valeurs ou le cadencement ont changé. Pour certaines classes de systèmes l'occurrence des déviations est acceptable, tant que la fréquence de ces événements reste bornée. Nous considérons des limites dures sur la fréquence des deviations avec ce que nous appelons les exigences faiblement dures, par exemple, le nombre maximal d'éléments divergents d'un flot par un nombre d'éléments consécutifs.

Nous introduisons des limites de dérive sur les apparitions relatives des paires d'événements récurrents comme les tics d'une horloge, l'occurrence d'une difference, ou l'arrivée d'un message. Les limites de dérive expriment des contraintes entre les horloges, par exemple, une borne supérieure de deux tics d'une horloge entre trois tics consécutifs d'une autre horloge. Les limites permettent également de caractériser les exigences faiblement dures. Cette thèse présente des analyses pour la vérification et l'inférence des exigences faiblement dures pour des programmes de flots de données synchrones étendu avec de la communication asynchrone par l'échantillonnage de mémoire partagée où les horloges sont décrites par des limites de dérive. Nous proposons aussi une analyse de performance des systèmes répartis avec de la communication par tampons FIFO, en utilisant les limites de dérive comme abstraction.

Contents

Glossary of Symbols and Notation

$\mathcal{S}$	domain of signals ordered by subset inclusion	17
χ_x	counter function of the signal x	19
η_x	dater function of the signal x	20
$\mathcal{S}^{\mathcal{V}}$	the domain of behaviors with events in $\mathcal{V}$	21
$\mathcal{P}^{\mathcal{V}}$	the domain of processes with events in $\mathcal{V}$	22
$\pi_{\mathcal{V}}$	projection of processes and transition systems to events in $\mathcal{V}$	22
$\parallel$	composition operator of processes and labelled transition systems	22,70
$X_{x/y}$	relative counter function of signal x w.r.t. y	22
$\sqsubseteq$	partial order relation	29
$\sqcup, \sqcap$	least upper bound (supremum), greatest lower bound (infimum)	29
$\top, \bot$	top and bottom elements of a lattice	29
$\operatorname{gfp} \Pi$	the greatest fix-point of the mapping Π	40
$C \rightleftharpoons A$	Galois connection between concrete domain C and abstract domain A	30
α, γ	abstraction and concretization functions of a galois connection	30
$\mathcal{X}^{\mathcal{V}}$	the domain of sets of relative counter functions of events $\mathcal{V}$	31
$C^u_{x/y}, C^l_{x/y}$	upper and lower clock bounds on signal x w.r.t. y	32
$\mathcal{C}^{\mathcal{V}}$	the domain of clock bounds for events in $\mathcal{V}$	32
$D^u_{x/y}, D^l_{x/y}$	upper and lower drift bounds on signal x w.r.t. y	35
$\mathcal{D}^{\mathcal{V}}$	the domain of drift bounds for events in $\mathcal{V}$	36
$\hat{x}$	clock signal of the signal x	69
$[\![x = e]\!]$	labelled transition system of core sampling network equation $x = e$	30
$\mathcal{L}^{\mathcal{V}}$	the domain of languages of transition systems with alphabet $\wp(\mathcal{V})$	89
$\mathbb{N}^\infty$	natural numbers extended with ∞	112
$\mathcal{F}$	the domain of non-decreasing functions over $\mathbb{N}^\infty$	112
$\vee, \wedge$	point-wise maximum and minimum of functions in $\mathcal{F}$	113
$\circ$	function composition	116
f^{-1}	pseudo-inverse of $f \in \mathcal{F}$	117
$\otimes, \overline{\otimes}$	min- and max-plus convolution	121, 122
$f, f^{\overline{*}}$	sub- and super-additive closure	122, 122
$\oslash, \overline{\oslash}$	min-plus deconvolution	121, 122

Chapter 1

Introduction

Since their invention computers have permeated our daily lives to the extend that society has become dependent on them, even for *safety-critical* tasks. It is not the machines first identified as computers that we rely most on, e.g. personal computers and laptops, but rather *embedded systems*; the computers embedded in machines that perform tasks without being seen. For example, the anti-lock brake systems (ABS) of our car, the smoke-detector in our kitchen, the pacemaker on grandfather's heart, the control software of a nuclear power plant, and so on.

The safety of such embedded systems are ensured by systematic engineering processes. In some industries, such as avionics, the design, development and maintenance processes are standardized through certification. Validation of systems — checking if a system behaves as intended — is an essential part of such engineering processes, both during the design and at deployment.

Embedded systems are increasingly *distributed* for reasons of performance, robustness, or physical constraints. They are distributed in the sense that there is no global time-reference by which components agree on the order of events; components interact *asynchronously*. Examples of such distributed systems include network on chips (NoC), drive-by-wire architectures employed in modern cars, and distributed control systems in rail transportation.

The validation of asynchronous systems is complicated by their *non-deterministic* behavior. That is, system behavior depends on the interleaving of events as observed and generated by different parts of the system. In this thesis, we explore abstractions and verification methods to facilitate the design of such distributed systems with the aim of distributing synchronous systems.

1.1 Context

1.1.1 Reactive Systems

Reactive systems were identified [HP85] as a class of systems that are notoriously hard to design. They interact continuously with their environment at the speed

of their environment. Several aspects complicate the design of such systems:

- They are often used for *safety-critical* applications, where system failure may have grave consequences. Consequently, the correctness of a design must be validated before deployment insofar this is possible.
- In case of embedded systems where the system interacts with physical processes, the system operates under *real-time* constraints because the system must react on time.
- They are often subject to *resource constraints*; the system must be constructed within a budget of available resources.

Recently, with the advance of computing in general and networking in particular, the size and complexity of reactive systems has exploded. It has been widely noted [Kop08] that engineering practice lacks the models to effectively reason with such complex systems. [Sif11] notes three challenges in particular: (1) the combined modelling of the computational and physical aspects of a system, (2) the need for a compositional approach to design for scalability, and (3) the efficient use of resources for mixed-criticality systems.

1.1.2 Synchronous Approach to Reactive Systems

The synchronous programming paradigm [BB91] was proposed to facilitate the development of reactive systems and since proven to be successful [BCE+03]. Prominent examples of synchronous programming languages include Lustre [HCRP91] and its commercial implementation Scade, Signal [LB87], and Esterel [BG92]. Synchronous programming languages aim to simplify the programming of reactive embedded systems through the adoption of a parallel model of computation based on a high-level timing model. Moreover, the languages have mathematically defined semantics to permit formal verification of properties with the help of automatic tools.

The synchronous languages are based on the *synchrony hypothesis* that states that computation and calculation are instantaneous. The synchrony hypothesis leads to a logical model of time, where time evolves by discrete steps. The logical time is often made explicit as a *clock signal* that drives components within its *clock domain* similarly to the clocks in digital circuits. Figure 1.1 depicts a schematic reactive program driven by a global clock signal.

Of course the "real" world does not behave synchronously: neither computation nor communication are instantaneous. Moreover, synchronous programs are often compiled to a synchronous program where reactions that are parallel in the synchronous model actually occur in sequence. This discrepancy is resolved by designing systems with sufficient margins in the timing requirements so that the deployed system will react on time even if reactions are not actually simultaneous.

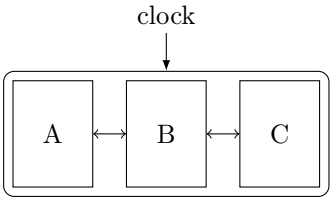


Figure 1.1: Three processes communicating synchronously on a clock signal.

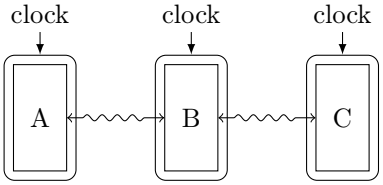


Figure 1.2: A GALS system where independently clocked components communicate asynchronously across clock domains.

1.1.3 GALS Systems

In distributed systems it quickly becomes infeasible or too costly to cope with asynchrony through design with margins in timing or, conversely, because an asynchronous design can be cheaper and/or more performant. Modern processors, for example, are sometimes partitioned in separate clock domains, because of clock skew clock signals arrive at different times due differences in travel distance [VHR+08] or to optimize power usage by dynamic (clock) frequency scaling [SMB+02]. Distribution can also be a consequence of physical constraints when, for example, the physical distance between two components prevents timely communication.

Globally asynchronous, locally synchronous (GALS) systems were originally defined by Chapiro [Cha84] to unify the synchronous and asynchronous modes of interaction found in distributed hardware. GALS systems consist of synchronous components that communicate with each other by asynchronous means. Each synchronous component has its own, independent clock. Figure 1.2 depicts a GALS variation of the synchronous program depicted in Figure 1.1

The semantics of GALS systems are defined by the asynchronous primitives used to communicate between clock domains and the origin and nature of the clocks. Teehan [TGL07] distinguishes three GALS design styles by the nature of the clocks: pausible clocks where clocks can be controlled to synchronize across domains, asynchronous clocks where clocks are independently generated and have no known relation between them and mesochronous clocks that have a known relation but cannot be controlled, e.g., clocks that have a bounded phase drift, skew, or fixed rate difference.

1.1.4 Distribution of Synchronous Programs

One approach to the design of GALS is to first design the system leveraging the synchronous approach and then distribute the synchronous design. The distribution of a synchronous program on a GALS architecture consists in the decomposition and subsequent asynchronous recomposition of the program. Each synchronous sub-system in the asynchronous composition has its own time-steps that may interleave arbitrarily with each other. In general, such a distribution may behave differently than the original synchronous program.

Benvensite et al. [BCL99, BCCSV03] identified the following conditions under which the asynchronous composition of synchronous systems behaves like their synchronous composition, that is, the synchronous behavior can be reconstructed from the asynchronous behavior:

- each system in the composition must be *endochronous* (self-timed) to know, based on its state, whether its inputs are present, as absence/presence of signals cannot be detected; and
- each pair of components in the composition must be *isochronous* (same-timed), i.e., they must agree on the values of all shared variables (or on the shared state).

It is possible to distribute a synchronous program if one ensures the distributed components and their compositions are both endo- and isochronous. Endochrony can, in general, only be determined through model checking. In practice it is therefore more common to *desynchronize* a system through program transformations that ensure endo- and isochrony of the distributed components. That is, to design the system using a synchronous language, repartition its components into clock domains that interact asynchronously, and introduce synchronization protocols. Figure 1.3 schematically depicts the distribution of the synchronous program of Figure 1.1 through the introduction of controllers and communication protocols that ensure endo- and isochrony.

1.1.5 Quasi-Synchronous Systems

In [Cas01] Caspi narrates how analog (unclocked) circuits in control systems were gradually replaced with digital (clocked) circuits and software, leading to a particular kind of GALS systems named *quasi-synchronous* systems. In such systems, synchronous programs are composed asynchronously using communication-by-sampling, where independently clocked synchronous programs communicate by reading and writing shared memory. All clocks have the same period with a bounded drift and, consequently, some writes may be overwritten before they are read and some may be read multiple times. The systems are designed under the quasi-synchrony hypothesis: for any pair of clocks one can tick at most twice between any two ticks of the other. Thus, the number of overwritten and duplicated messages is bounded.

Loosely Time-Triggered Architectures (LTTA) [BCG$^+$02] provide a quasi-synchronous platform with independently clocked computing resources that

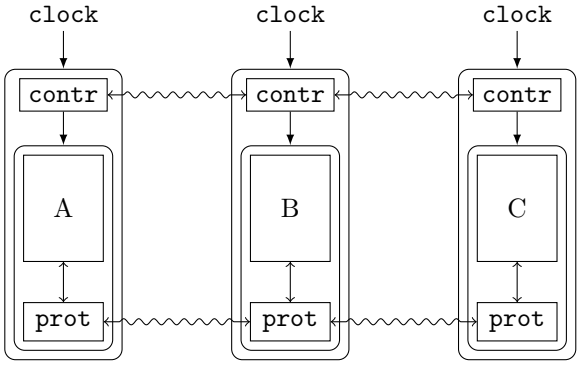


Figure 1.3: A distributed synchronous program with controllers to synchronize the clocks and protocols for the transfer of data.

communicate by sampling and proposes a semantic-preserving distribution method. The quasi-synchrony hypothesis of LTTAs allows for light-weight synchronization protocols and throughput guarantees.

1.2 Motivation and Objectives

The motivation of this work starts with the observation that embedded systems are increasingly designed as distributed GALS systems. Such systems are either built as GALS systems from the ground up or designed as synchronous systems that are subsequently distributed.

In the first case, system validation in general is severely complicated by the large (possibly infinite) number of possible interleavings of events. Testing based verification is marred by large number of possible execution scenarios as well as the practical problems of controlling or even observing the order of events in distributed systems. Formal verification methods face the state-space explosion problem that originates in the asynchronous composition of distributed components.

In the second case, that is, the distribution of synchronous systems, both testing and formal verification of the original design benefits from the synchrony hypothesis to avoid the aforementioned problems faced for GALS systems. The validity of the distribution, however, depends on a semantics-preserving desynchronization. In practice desynchronization entails overhead in reaction time, (memory) space, use of network bandwidth, and loss of robustness. That is, the distributed system must exchange extra messages, introduce buffers, and delay execution to be semantically equivalent to the original synchronous program.

The quasi-synchronous approach with communication-by-sampling suggests a mixed approach where synchronous programs are distributed without a strict preservation of the synchronous semantics. With communication-by-sampling the behavior depends on the interleaving of write and read operations, i.e., de-

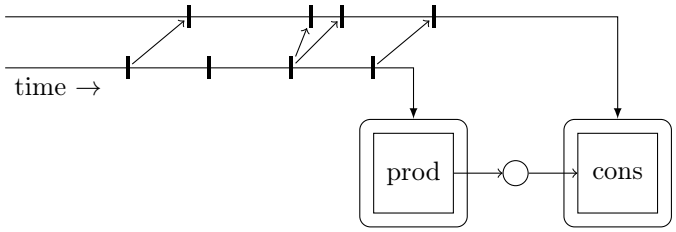


Figure 1.4: A producer whose output is read by a consumer through communication-by-sampling.

pending on the interleaving of clock ticks. For example, the system depicted in Figure 1.4 depicts an independently clocked producer and consumer that communicate by sampling. Depending on occurrence times of the producer's and consumer's clocks, messages may be lost (overwritten if the consumer under-samples the memory cell) or duplicated (if the consumer over-samples the memory cell).

If clocks are stable enough, such a distribution may still behave much like the original system and diverge only occasionally. Moreover, not all divergence from the synchronous behavior may imply a malfunction. We deem such a mixed approach especially useful for embedded systems of certain types:

- *Robust* systems must maintain a quality of service in spite of disruptions. Non-blocking communication-by-sampling is *robust* in the sense that disruptions where one component stops functioning do not stop the complete system.This is especially relevant in cases where parts may stop responding completely. In normal circumstances the system would need to provide guarantees on reactivity, e.g., reacting within a given number of clock cycles.

- In *control systems* a controller directs a physical process through actuators based on information from sensors. Control systems are designed with tolerances to deal with the finite accuracy of sensors and actuators, as well as quantization and signal processing in the controller.

- In *weakly-hard real-time systems* [BBL01] deadlines for reactions may occasionally be missed, as long as a hard bound is respected, e.g., one out of ten deadlines may be missed. Similar constraints can be considered for any error: reactions with the wrong values, absence of reactions, collisions, etc.

The requirements on these kind of systems fit a pattern: within a given time-interval the number of some event must be bounded. The number of events can be bounded from below, e.g., a time-out: react at least once or sample a sensor-value at least thrice within the interval, or it can be bounded from above, e.g., the maximum number of missed deadlines. We call these

weakly-hard requirements as a generalization of Bernat's notion of weakly-hard real-time systems.

Consider a synchronous system design that is partitioned into multiple independently clocked subsystems that communication by sampling. Starting from its synchronous execution where all clocks tick simultaneous, we can move to quasi-asynchronous systems by weakening the synchrony of clocks. Assuming the synchronous design never violates its requirements, its quasi-synchronous distribution with will, in general, fail more and more frequently as constraints on the clocks are relaxed.

This work envisions a mixed approach to the design of GALS systems where systems are designed principally designed as synchronous systems, but with foresight of its future distribution. In order to realize such distributions, we need to (1) describe the degree of (quasi-)synchrony of clocks that drive the system; (2) express weakly-hard requirements on the frequency of events; and (3) verify weakly-hard requirements for a quasi-synchronous distribution.

1.3 Contributions and Outline

This thesis introduces a framework for the distribution of synchrononous programs over a loosely time-triggered architecture with communication-by-sampling, where the semantics of the distributed system are allowed to diverge from the original synchronous program as specified by drift bounds on the clocks. We provide methods to verify and infer weakly-hard properties of such systems that are necessary for the development of safety-critical systems. This thesis also gives a method to infer performance characteristics of FIFO-channel based distributions.

Chapter 2 introduces a discrete event model that is the formal foundation to describe the processes of GALS systems throughout this work.

Chapter 3 defines clock and drift bounds as abstract descriptions of processes. Drift bounds, the first original contribution of this thesis, are is a generalization of the quasi-synchrony hypothesis. A drift bound limits (from below and/or above) the number of ticks of one clock in for every N consecutive ticks of another clock. By deriving drift bounds as an abstraction for the possible behaviors of clocks, we naturally derive useful properties such as the transitivity of bounds.

Drift bounds turn out to be surprisingly versatile when applied to the occurrence of any event and not just clocks. In particular, we show how they can also be used to express weakly-hard requirements (see Chapter 6).

We also show how drift bounds can be used to describe aspects of resources and streams. Chapter 4 uses drift bounds in this capacity for a novel performance analysis of GALS systems that communicate over FIFO channels. The analysis is similar to real-time calculus [TCN00] but, in contrast to the abstractions used in real-time calculus (namely arrival and resource curves), drift bounds preserve the correlation of events which allows us to derive better (local) backlog bounds.

Chapter 5 defines sampling networks, a synchronous data-flow language extended with an asynchronous communication-by-sampling primitive. Non-blocking communication-by-sampling permits an arbitrary interleaving of clocks, ranging from (global) synchrony, to quasi-synchrony, to unbounded asynchrony, but the arbitrary interleaving of clocks will change the behavior. It is therefore crucial to analyse the behavior of a distributed sampling network. Chapter 6 introduces methods to verify and infer weakly-hard properties (in the form of drift bounds) of a sampling network when executed in an environment described by drift bounds. We provide an exact analysis, based on a full state-space exploration, and an abstract-interpretation based analysis that trades accuracy for speed of analysis.

Chapter 4 is independent from Chapters 5 and 6, but all depend on the event-model introduced by Chapters 2 and the abstractions of Chapter 3.

The analyses have been implemented as prototypes based on a library of operators for non-decreasing function. The operators and their properties are described in the Appendix A.

Chapter 2

A Discrete Event Model for GALS Systems

This chapter introduces an event model for reactive systems with discrete events. The aim is to facilitate high-level reasoning and modeling of GALS systems. The basic event model captures the occurrence of recurring events over time. Events model the ticks of a clock, transitions between states, as well as the values of boolean flows.

A relational view on events is constructed that only shows the interleaving of the occurrences of events, while hiding the real-time itself. This us to isolate logical and temporal behavior insofar the temporal behavior can influence logical behavior. The motivation is that digital systems cannot observe real-time directly, but only through some discretized notion, namely the occurrence of events. In the clock-centric view of GALS systems in particular, behavior is strongly determined by the interleaving of clock ticks.

Consider for example the producer-consumer example of Figure 1.4 in the introduction. We can model the system with four events: the ticks of the respective clocks that activate the consumer and producer, the event of a loss of a message and the event where a message is duplicated. The loss and duplication of messages is determined by the interleaving of the clocks' ticks. The model introduced in this chapter allows us to reason with precisely this kind of systems and abstract from the real-time aspects when needed.

2.1 Signals, Counter and Dater Functions

The event model describes the behavior of a system through the presence of *events*, such as the tick of a clock. Events are unvalued, discrete and atomic, viz. they occur at a single indivisible time instant and carry no information other than their presence. A signal is a set of dates in (in $\mathbb{R}$) at which a recurring event the event *occurs*.

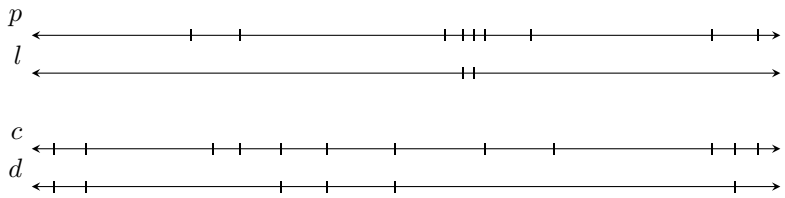


Figure 2.1: Possible behavior of the producer-consumer system of Figure 1.4 with signals p (producer clock/written), l (lost), c (consumer clock) and d (duplicate) depicted on a timeline.

Definition 1 (Signal ($\mathcal{S}$))**.** *The signal $x \in \mathcal{S}$ is a countable set of events in $\mathbb{R}$ such that it is zeno-free, i.e., that only a finite number of events may occur before any time instant $t \in R$:*

$$\forall t \in \mathbb{R} : \ |\{u \in x \mid u \leq t\}| \in \mathbb{N}$$

An event of a signal x is said to *occur* at time time t, if it contains an event at that time, i.e., if $t \in x$. Conversely, the event $t \in x$ is said to be an *occurrence* of the event x. Generally, the signal is named after its event, e.g., signal x contains the occurrences of x.

Signals group the occurrences of recurring events such as all the ticks of a single clock or the arrivals of messages in one particular stream. Consider a few abstract example signals that are revisited later on:

- the empty signal $\emptyset$ where no event occurs;
- a finite signal $\mathit{fin} = \{t_1, t_2\}$ of two events $t_1, t_2 \in \mathbb{R}$; and
- an infinite, periodic signal $\mathit{per} = \{an + b \mid n \in \mathbb{N}\}$ with $a, b \in \mathbb{R}$.

The following examples illustrate the restrictions of signals on event sets:

- The set $\mathit{cont} = (s, e)$ consists of a continuous event from $s \in \mathbb{R}$ to $e \in \mathbb{R}$ is not a proper signal, because it has an uncountable number of events.
- The set $\mathit{zeno} = \{\frac{1}{2^n} \mid n \in \mathbb{N}\}$ contains an infinite number of events just after time 0. It is not a proper signal because this is a zeno-event.

For a concrete example, consider the producer and consumer of Figure 1.4. Each is triggered by their respective clocks signals p and c. The signals l and d respectively refer to the events where a message is lost (overwritten) or a message is duplicated (read twice) in case of non-blocking communication. Figure 2.1 depicts the signals that describe a possible behavior of both variants of the producer-consumer system.

Chapter 5 shows how signal model can model systems with boolean values despite the fact that events are unvalued. This is achieved by associating a boolean stream with a clock signal that indicates its presence and a value signal that indicates the moments the values in the stream are true.

2.1.1 Counter Function

A signal can also be described by counting the total number of events over time, because the only distinguishing element of events in one signal is the time at which they occur. Such a representation is often more convenient than sets of dates.

Let sets $\mathbb{R}^\infty$ and $\mathbb{N}^\infty$ denote the set of rational numbers extended with the limits $-\infty$ and ∞ and the set of natural number extended with the limit ∞ respectively. The counter function is then defined as follows.

Definition 2 (Counter function (χ_x))**.** *The counter function $\chi_x \in \mathbb{R}^\infty \to \mathbb{N}^\infty$ counts the number of events $\chi_x(t)$ of a signal x that have occurred up to time $t \in \mathbb{R}^\infty$:*

$$\chi_x(t) = |\{u \in x \mid u \leq t\}|$$

Lemma 1 (Properties of counter function)**.** *The counter function*

1. *starts at zero ($\chi_x(-\infty) = 0$); and*
2. *is non-decreasing ($t \leq u \implies \chi_s(t) \leq \chi_s(u)$).*

Counter functions simplify reasoning with signals, particularly in the case of data-flow systems. Consider, for example, a variant of the produce-consumer system depicted in Figure 1.4 where messages are sent over a FIFO channel. Let the events w and r denote two signals that describe the number of messages written to (pushed) and the number of read (pulled) elements from the FIFO channel, then the buffer contains $\chi_w(t) - \chi_r(t)$ messages at any time t.

Revisiting the example signals, we obtain the following counter functions:

- the counter function of the empty signal $\chi_\emptyset(t) = 0$ for all $t \in \mathbb{R}^\infty$;
- the counter function of the finite signal

$$\chi_{fin}(t) = \begin{cases} 0 & \text{if } t < t_1 \\ 1 & \text{if } t_1 \leq t < t_2 \\ 2 & \text{if } t_2 \leq t \end{cases}$$

- the counter function of the periodic signal $\chi_{per}(t) = \max(0, \lfloor \frac{t-b}{a} \rfloor)$.

Figure 2.2 shows the counter functions of the clock signals considered earlier for the producer-consumer system.

2.1.2 Dater Function

A signal can also be described by a dater function that gives the time of each consecutive event. If the number of events is bounded, then the time of events beyond the last is ∞.

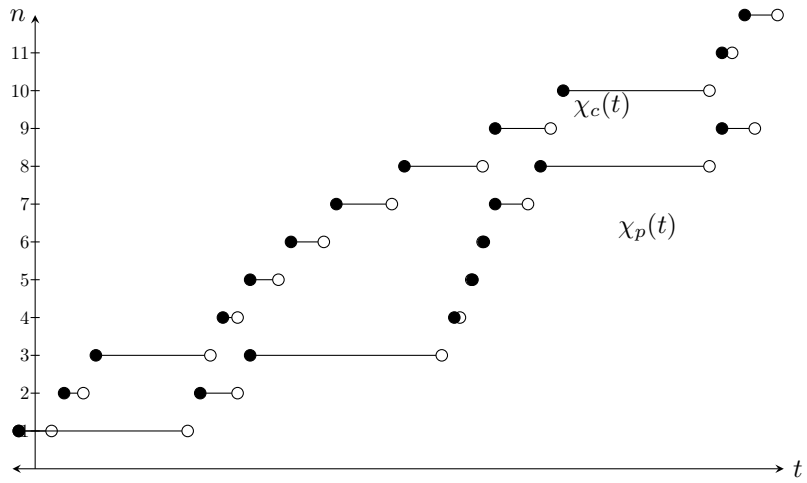


Figure 2.2: The counter functions of clock signals p and c depicted in Figure 2.1.

Definition 3 (Dater function (η_x))**.** *The dater function $\eta_x \in \mathbb{N}^\infty \to \mathbb{R}^\infty$ gives the time of the n-th occurrence of signal x and is defined as*

$$\eta_x(n) = \inf\{t \in \mathbb{R}^\infty \mid \chi_x(t) \geq n\}$$

Lemma 2 (Properties of dater function)**.** *The dater function*

1. *starts at $-\infty$ ($\eta_x(0) = -\infty$);*
2. *eventually reaches ∞ ($\eta_x(n) = \infty$ for $n > |x|$); and*
3. *is non-decreasing ($n \leq m \implies \eta_s(n) \leq \eta_s(m)$).*

Again, this representation is particularly helpful when studying FIFO channels. Consider again the producer consumer system that communicate with a FIFO channel where events w and r describe the writing and reading of messages to and from the FIFO channel respectively. Then, the $\eta_r(n) - \eta_w(n)$ gives amount of time that the $n \in \mathbb{N}$-th element spent in the FIFO channel before bing read.

Revisiting the example signals we obtain the following counter functions:

- the dater function $\eta_\emptyset(n) = \infty$ for $n > 0$ and $\eta_\emptyset(0) = -\infty$ for the empty signal;
- for the finite signal the dater function

$$\eta_{fin}(n) = \begin{cases} -\infty & \text{if } n = 0 \\ t_1 & \text{if } n = 1 \\ t_2 & \text{if } n = 2 \\ \infty & \text{otherwise} \end{cases}$$

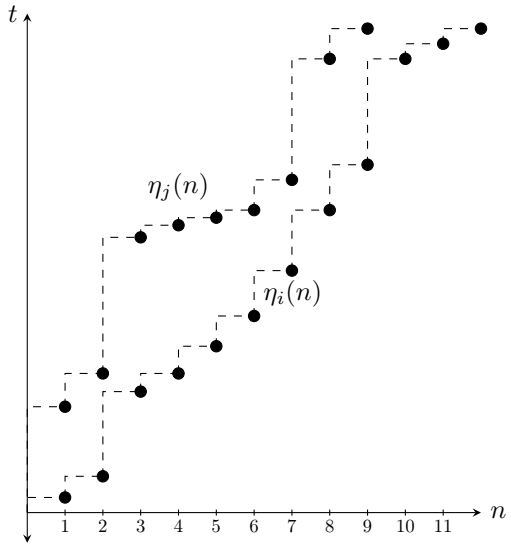


Figure 2.3: The dater functions of signals p and c depicted in Figure 2.1.

- for the periodic signal $\eta_{per}(0) = -\infty$ and $\eta_{per}(n) = an + b$ for $n > 0$

Figure 2.3 shows the dater functions of the producer-consumer clock signals corresponding to the counter functions depicted in Figure 2.2.

The original signal can easily be reconstructed from its dater function. To do so, one takes the times at which the counter function increases or, equivalently, the date (as defined by the dater function) of each event:

$$x = \{\eta_x(n) \mid n \in \mathbb{N}\} \setminus \{-\infty, \infty\}$$

Thus, we have gone full circle: from signal to counter function, from counter function to dater function, and from counter function back to the signal. As a consequence, a signal may be defined by its counter or dater function. In the sequel we consider signals defined as a set of events or a counter or dater function.

Note that the countability of the events and Zeno-freedom is crucial. For example, the Zeno and continuous event sets have the same dater function if t_1 approaches zero, yet they describe different sets of events.

2.2 Behaviors and Processes

Until now we have only considered single signals. To model systems with multiple signals, albeit a finite number, we use vectors of signals. For convenience the vector is indexed by a set $\mathcal{V}$ of events or, rather, their names.

Definition 4 (Behavior ($\mathcal{S}^{\mathcal{V}}$))**.** *A* behavior *is a vector* $\boldsymbol{x} \in \mathcal{S}^{\mathcal{V}}$ *of* $|\mathcal{V}|$ *signals indexed by the events of* $\mathcal{V}$*.*

A behavior can be regarded as a trace of the system that contains all relevant events that occur while the system is running. Systems are modelled by *processes* that consist of the set of all behaviors that the system may exhibit.

Definition 5 (Process ($\mathcal{P}$))**.** *A* process $P \subseteq \mathcal{S}^{\mathcal{V}}$ *is a set of behaviors that describes a system with events* $\mathcal{V}$*. The set* $\mathcal{P}^{\mathcal{V}}$ *consists of all processes with events* $\mathcal{V}$*.*

Processes may be combined to define new processes, facilitating a compositional construction of processes.

Assuming the processes P and Q describe the same events, i.e., $P, Q \in \mathcal{P}^{\mathcal{V}}$, they may be composed by intersecting the behavior: $P \cap Q$. Such a composition restricts the behaviors to those possible in both behaviors.

The projection operator $\pi_{\mathcal{W}} \in \mathcal{P}^{\mathcal{V}} \to \mathcal{P}^{\mathcal{W}}$ hides all events not in $\mathcal{W}$ and adds events that are in $\mathcal{W}$ but not in $\mathcal{V}$. That is, the projection operator can project a process from any dimension (indexed by event names) to any other dimension, not just reduce the dimension. Added signal are completely unrestricted: they can have any possible behavior.

Definition 6 (Projection ($\pi_{\mathcal{W}}$))**.** *The projection* $\pi_{\mathcal{W}}(P)$ *of a process* $P \in \mathcal{P}^{\mathcal{V}}$ *is defined such that*

$$\pi_{\mathcal{W}}(P) = \{\boldsymbol{x} \in \mathcal{S}^{\mathcal{W}} \mid \forall \boldsymbol{y} \in P,\ \forall v \in \mathcal{V} \cap \mathcal{W} :\ x_v = y_v\}$$

For convenience we define the composition operator $\parallel$ to compose two processes, synchronizing only the events in both processes. It is, in fact, derived from the intersection and projection.

Definition 7 (Composition ($\parallel$))**.** *The composition* $(P \parallel Q) \in \mathcal{P}^{\mathcal{V} \cup \mathcal{W}}$ *of* $P \in \mathcal{P}^{\mathcal{V}}$ *and* $Q \in \mathcal{P}^{\mathcal{W}}$ *is defined such that*

$$P \parallel Q = \pi_{\mathcal{V} \cup \mathcal{W}}(P) \cap \pi_{\mathcal{V} \cup \mathcal{W}}(Q)$$

Note that the composition operator is commutative, associative and idempotent.

2.3 The Relative Counter Function

The counter and dater functions are closely linked. In fact, one might notice that the counter function is a kind of inverse of the dater function. It is not a real inverse because $\eta_x(\chi_x(t)) \neq t$, namely when there is no event at time t in signal x ($t \notin x$). However, the following equalities and inequalities do hold.

Lemma 3 (Compositions of counter and dater functions)**.** *Let* x *be a signal then the following (in)equalities hold for the compositions of its dater and counter functions*

1. $\eta_x(\chi_x(t)) = t$ *for all* $t \in x$
2. $\eta_x(\chi_x(t) + 1) \leq t$ *for all* $t \in \mathbb{R}$*; and*
3. $\chi_x(\eta_x(n)) = n$ *for all* $n \in \mathbb{N}$ *such that* $n \leq |x|$.

Let us now introduce relative counter functions, defined as a composition of a counter and a dater function. The original motivation for relative counter functions was to formulate what values are received when communicating by sampling. Given a valuation function $v \in \mathbb{N} \to \mathcal{V}$ such that $v(n)$ gives the n-th value written to the shared memory by the producer, we south to define another valuation function v' such that $v'(n)$ is the value of the n-th sample read by the consumer. To formulate the relation between v and v' we need the total number of writes at the time of ever n-th sample. The relative counter function expresses precisely this quantity.

Definition 8 (Relative Counter Function ($X_{x/y}$))**.** *The relative counter function that gives the total number of* $X_{x/y}(n)$ *events of signal* x *is present up to the* n*-th event of signal* y*:*

$$X_{x/y}(n) = \chi_x(\eta_y(n))$$

With the relative counter function the sampled values are defined as by a simple composition $v' = v \circ X_{p/c}$ where p and c are the clock signals of the producer and consumer. Note that this formulation implies instantaneous communication in case the producer and consumer are activated simultaneously.

Relative counter functions inherit most of the counter functions' properties. The most important difference being that relative counter functions can increase by more than one in a single step.

Lemma 4 (Properties of relative counter functions)**.** *The relative counter function* $X_{x/y}$ *of all signals* x *and* y

1. *starts at zero* ($X_{x/y}(0) = 0$)*;*
2. *is non-decreasing* ($n \leq m \implies X_{x/y}(n) \leq X_{x/y}(m)$)*; and*
3. *if* $x = y$ *it is the identity function* ($x = y \implies X_{x/y}(n) = n$ *for* $n \leq |x|$).

In Appendix A we define a set $\mathcal{F} \subseteq \mathbb{N}^\infty \to \mathbb{N}^\infty$ of non-decreasing functions that start at the origin; the set of all possible relative counter functions. It also defines a point-wise ordering ($\leq \in \mathcal{F} \times \mathcal{F}$) and various operators over functions in $\mathcal{F}$.

Figure 2.4 depicts the relative counter functions of the producer-consumer system's clock signals. Observe how $X_{c/p}$ jumps from 4 to 7, because there are three occurrences of c between the second and third occurrence of p. We can also see that only the (non-strict) order of ticks is preserved, in the sense that if e.g. $X_{c/p}(2) = 4$ we do not know if the fourth event in c occurred simultaneously with the second event in p, or if it preceded the second event in p.

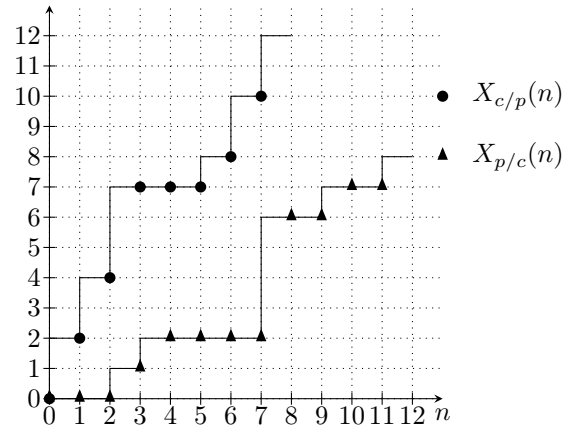


Figure 2.4: The relative counter functions of signals p and c from Figure 2.1.

Aside from communication-by-sampling, we find that the relative counter function provides an excellent abstraction from the temporal behavior that allows us to concentrate only on the observable and often controllable interleavings of events.

2.3.1 Transitivity

The relative counter functions are related by transitivity: the relative counter functions $X_{x/y}$, $X_{x/z}$ and $X_{z/y}$ of any three signals x, y and z are not independent.

Lemma 5 (Transitivity of relative counter functions)**.** *Let x, y and z be signals, then the relative counter function $X_{x/y}$ is bounded such that for all $n \in \mathbb{N}^\infty$*

$$X_{x/z}(X_{z/y}(n)) \leq X_{x/y} \leq X_{x/z}(X_{z/y}(n) + 1)$$

Proof. The proof of the lower bound follows from the fact that function composition is associative and that $(\eta_z \circ \chi_z)(t) \leq t$. Thus we derive $X_{x/z} \circ X_{z/y} = \chi_x \circ \eta_z \circ \chi_z \circ \eta_y \leq \chi_x \circ \eta_y = X_{x/y}$.

The upper bound is proven similarly, but now $\eta_z(\chi_z(t)+1) \geq t$, and therefore $X_{x/z}(X_{z/y}(n) + 1) = \chi_x(\eta_z(\chi_z(\eta_y(n)) + 1)) \geq \chi_x(\chi_y(n)) = X_{x/y}(n)$. □

Intuitively one may think of the relative counter function $X_{x/y}$ both as an imperfect (as observed at occurrences of y) counter function for x and an imperfect dater function for y (with occurrences of x as time units). Stated thusly, the composition $X_{x/z} \circ X_{z/y}$ is in fact reminiscent of the definition of the relative counter function itself with $X_{x/z}$ as the counter function of x and $X_{z/y}$ as the dater function of y, both using occurrences of z as a unit of time. The inequalities of Lemma 5 are due to the quantization of time by z.

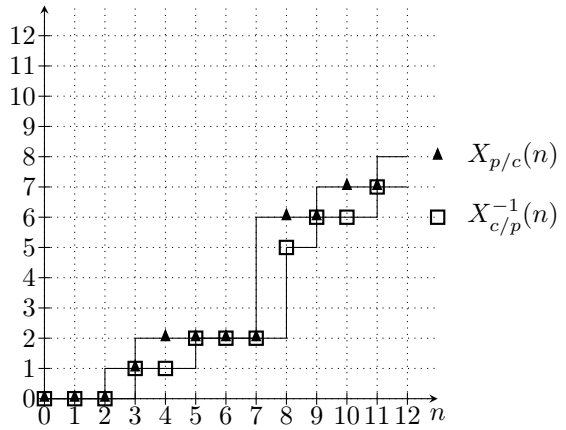


Figure 2.5: The relative counter function $X_{p/c}$ and the pseudo inverse $X^{-1}_{c/p}$ for signals p and c from Figure 2.1.

2.3.2 Pseudo-Symmetry

Maintaining the intuition of a relative counter function $X_{x/y}$ as an imperfect counter function of x with time units y, we can also perform an operation analogous to the definition of the (real) dater function in Definition 3:

$$X^{-1}_{x/y}(n) = \inf\{m \in \mathbb{N}^\infty \mid X_{x/y}(m+1) \geq n\}$$

This is defined in Appendix A as the *pseudo-inverse* of functions in $\mathcal{F}$. It is essentially a discretized version of the operation that defines the dater function. The intuitive interpretation would suggest that $X^{-1}_{x/y}$ defines an imperfect dater function for x with time units in y just like the counter function $X_{y/x}$ The example in Figure 2.5 shows that this relation indeed holds for the relative counter functions of signals p and c. But it also shows that, for the instances where the signals c and p occur simultaneously, $X^{-1}_{c/p} \neq X_{p/c}$. The following lemma formalizes and proves this relation.

Lemma 6 (Pseudo-symmetry of relative counter function)**.** *Let x and y be signals then, for all $n \in \mathbb{N}^\infty$*

$$X^{-1}_{y/x}(n) \leq X_{x/y}(n) \leq X^{-1}_{y/x}(n) + 1$$

Proof. We will prove this by deriving equal upper and lower bounds of the pseudo-inverse. First note that $X^{-1}_{x/y}(n)$
$= \min\{m \in \mathbb{N} \mid (\chi_x \circ \eta_y)(m+1) \geq n\} = \min\{m \in \mathbb{N} \mid \eta_y(m+1) \geq \eta_x(n)\}$ because $\chi_x(\eta_x(n)) = n$ and $\eta_x(\chi_x(t)) \leq t$ imply $\chi_x(x) \geq y \Leftrightarrow x \geq \eta_x(y)$. Then, since $\eta_y(\chi_y(t)+1) \geq t$ implies $x \geq \chi_y(y) \Rightarrow \eta_y(x+1) \geq y$, we have the lower

bound

$$\begin{aligned}
&\min\{m \in \mathbb{N} \mid \eta_y(m+1) \geq \eta_x(n)\} \\
\leq &\min\{m \in \mathbb{N} \mid m \geq (\chi_y \circ \eta_x)(n)\} \\
= &\min\{m \in \mathbb{N} \mid m \geq X_{y/x}(n)\} \\
= &X_{y/x}(n)
\end{aligned}$$

and, since $\chi_y(\eta_y(n)) = n$ implies $\eta_y(x) \geq y \Rightarrow x \geq \chi_y(y)$,

$$\begin{aligned}
&\min\{m \in \mathbb{N} \mid \eta_y(m+1) \geq \eta_x(n)\} \\
\geq &\min\{m \in \mathbb{N} \mid m+1 \geq (\chi_y \circ \eta_x)(n)\} \\
= &\min\{m \in \mathbb{N} \mid m \geq X_{y/x}(n) - 1\} \\
= &X_{y/x}(n) - 1
\end{aligned}$$

□

2.4 Synchronous Processes

Taken literally, synchrony implies simultaneity of events. However, such a notion adds little meaning in the context of our signals: if the events of two signals are simultaneous, the signals are equal. In stead, we define two signals in a behavior to be synchronous if there is a third signal in the behavior that contains the events of both signals. The third signal effectively functions as a clock for the two synchronous signals. Extending this notion, we define synchronous behaviors and processes to possess an hierarchical relationship with a single root that acts as the clock for the whole system.

Definition 9 (Synchronous Process)**.** *The behavior $\boldsymbol{x}$ of a system events $\mathcal{V}$ is synchronous if there is a hierarchy $\leq \in \mathcal{V} \times \mathcal{V}$ such that*

1. *it is a partial order with a single greatest element (root clock); and*
2. *it implies a subset relation on the signals: $x \leq y \implies x_i \subseteq x_j$.*

A process is synchronous all its behaviors satisfy the same hierarchy.

An *asynchronous* behavior or process has no such hierarchical relation between signals. GALS systems are compositions of synchronous systems, resulting in a system with multiple hierarchies. Note that any process can be equipped with a set of hierarchies: namely the trivial partial order where no two signals are comparable. Hence, technically, any process is GALS.

2.5 Conclusion

2.5.1 Related Work

Our signals and processes are closely related to those in the tagged-signal model first introduced by Lee and Sangiovanni-Vincentelli in [LSV98]. They define

tagged-signals as values paired with a *tag* from an (partially or totally) ordered set. Our model can be interpreted as an instance of the tagged-signal model, where values are in the trivial singleton set, i.e., there is only a single possible value, and events are dated by $\mathbb{R}$. In [BCC+07] Benveniste et al. investigates the structure of different tag sets in a variant of the tagged-signal model to model heterogeneous (in model of communication and computation) systems. Again our model can be seen as an instance of this variant as well by using a tag structure based on $\mathbb{R}$ and trivial set of values. Signals are then represented by a partial function that closely resembles the dater function. The tagged-signal model, however, does not define counter and dater functions.

Both counter and dater functions have been used in max-plus algebra [BCOQ92] and related work to model the activations of transitions in marked graphs (conflict-free Petri nets) or similar models. The most prominent difference, is our restriction to a single event per time instant due to the origins in signals. While our definitions could easily be adjusted to a similar effect, but it would weaken some of the bounds on relative counter functions because the composition $\chi_x \circ \eta_x$ would no longer be the identity function. Most notably, it would invalidate the transitive and pseudo-symmetric upper bounds. That is, with simultaneous occurrences of the same event $X_{i/j}(n) \not\leq X_{i/k}(X_{k/j}(n)+1)$ although $X_{i/k}(X_{k/j}(n)) \leq X_{i/j}(n)$ would still hold and $X_{i/j}(n) \not\leq [X_{j/i}]^{-1}(n) + 1$.

2.5.2 Discussion

The main contribution of this chapter consists of the relative counter function that is the basis of the work presented in the remainder of this document. While the relative counter function purposefully isolates our reasoning from real-time, it is always possible to add a clock signal that represents a discretization of real-time to make it an observable quantity.

Relative counter functions seem to have more potential than we explored in this thesis. For example, one could establish a notion of equivalence between behaviors based on the comparison of the relative counter functions. It seems this could be a useful notion of equivalence to compare a synchronous program with its asynchronous implementation, because it abstracts from the exact timing of events and compares only the interleavings of events.

Observe that one might construct a relative dater function in analogy to the relative counter function, i.e., a composition $\eta_x \circ \chi_y$ for two signals x and y. It is, however, unclear what aspects of the system these values represent and what practical applications such a relative dater functions should have.

Chapter 3

Abstract Domains for GALS Systems

In this chapter we introduce abstractions to efficiently reason with processes. In particular, the abstractions help us deal with the non-determinism of the root clocks in GALS systems by characterizing the possible interleavings of clock ticks in a GALS architecture:

- *Clock bounds* $C_{x/y}(n) \in \mathbb{N}$ limit, for each pair (x, y) of clock signals, the maximum and minimum, total number of ticks of x up to the n-th clock of y. Clock bounds include, for example, difference bounds on the number of ticks of a pair of clocks, e.g., the upper bound $C^u_{x/y}(n) = n + 5$ is satisfied if, and only if, x can be at most five ticks ahead of y.

- *Drift bounds* $D_{x/y}$ limit, for each pair (x, y) of clock signals, the maximum and minimum number of ticks of x in any interval of Δ ticks of y. Drift bounds can, for example, express the quasi-synchrony hypothesis that requires there be no more than two ticks of one clock for every two of another.

We also found that drift bounds are useful to describe aspects of resources, streams, and quality-of-service constraints.

Clock and drift bounds are used to describe guarantees of the environment, e.g., the drift bounds between clocks or the availability of resources such as a network connection. The bounds are also used to describe guarantees on system behavior in the context of an environment, e.g., the quality of service of a system. The analyses presented in this thesis infer guaranteed drift and clock bounds of a system when executed in an environment with guaranteed drift and clock bounds.

Such analyses consist in the *abstract interpretation* [CC77] of a program using drift and clock bounds as the abstract domains. That is, the analyses evaluate a system's abstract behavior — drift and clock bounds that describe

system behavior — in the context of an environment's abstract behavior. Abstract interpretation enables analysis of programs with infinitely many behaviors and unbounded state-space by, in general, over-approximating the set of behaviors or state space, with elements from the abstract domain. Thus, in general, abstract interpretation yields a sound (conservative approximation) but incomplete (imprecise) analysis method.

3.1 The Domain of Processes

The set of all possible processes $\mathcal{P}$ is the *domain* of processes. This section extends the formal framework of of processes and shows how to introduce abstractions for such processes. It then introduces matrices of relative counter functions as an abstraction for processes.

3.1.1 Complete Lattices

Complete lattices are partially ordered sets where each subset has unique least upper and greatest lower bounds. All domains considered in this work are complete lattices including the domain of processes. Let us quickly review the essential properties of complete lattices. For a more extensive introduction we refer to [DP02].

Definition 10 (Partial order ($\sqsubseteq$))**.** *A partially ordered set $\langle S, \sqsubseteq \rangle$ consists of the set S and the transitive, reflexive and anti-symmetric relation $\sqsubseteq \in S \times S$.*

The domain of processes is partially ordered by the subset relation. That is, with $P \sqsubseteq Q \iff P \subseteq Q$ the process P is lesser or equal to Q, if all behaviors in P are also in Q.

Definition 11 (Upper and lower bounds)**.** *Let $\langle S, \sqsubseteq \rangle$ and $S' \subseteq S$ then*

- *The* least upper bound $s = \bigsqcup S'$ *is the least element in S larger than or equal to all elements in S'.*
- *The* greatest lower bound $s = \bigsqcap S'$ *is the greatest element in S less than or equal to all elements in S'.*

The least upper and greatest lower bounds of a powerset ordered by the subset relation are defined by the union ($\sqcup = \cup$) and intersection ($\sqcap = \cap$) respectively. For example the greatest lower bound $P \sqcup Q = P \cup Q$ of two processes P and Q consists of all behaviors both in P and Q.

Complete lattices are partially ordered sets equipped with a least upper and greatest lower bound. Moreover, any subset of a complete lattice has both a least upper and greatest lower bound.

Definition 12 (Complete lattice ($\top, \bot$))**.** *A complete lattice $\langle S, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ consists of*

- *The partially ordered set* $\langle S, \sqsubseteq \rangle$
- *The least (or bottom) element* $\bot = \bigsqcup \emptyset = \bigsqcap S$
- *The greatest (or top) element* $\top = \bigsqcap \emptyset = \bigsqcup S$

The powerset $\wp(A)$ ordered by the subset relation is a complete lattice with empty set as a least element ($\bot = \emptyset$) and a greatest element that is the set of all elements ($\top = A$).

3.1.2 Abstractions and their Domains

Abstractions provide a limited view of concrete objects that exposes only specific aspects of system behavior. For example, the projection of a process where we consider only a subset of the signals in the complete system is an abstraction of the unprojected process. The Galois connection (see e.g. [NNH05, Smi10]) formalizes the relation between objects from a concrete domain and objects from an abstract domain as follows.

Definition 13 (Galois connection ($C \rightleftharpoons A, \alpha, \gamma$))**.** *Let* $\langle C, \sqsubseteq^C \rangle$ *and* $\langle A, \sqsubseteq^A \rangle$ *be partially ordered set and let* $\alpha \in C \to A$ *and* $\gamma \in A \to C$ *a pair of monotone functions such that*

$$\forall a \in A, c \in C : \ \alpha(c) \sqsubseteq^A a \iff c \sqsubseteq^C \gamma(a)$$

Then the pair of functions forms a Galois connection, written $C \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} A$.

The C is the concrete domain and A is the abstract domain. The abstraction function α relates a concrete object in C to its abstraction in A. The concretization function γ does the reverse. Abstractions can be chained [Smi10] because Galois connections are transitive relations:

$$C \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} A \text{ and } A \underset{\gamma'}{\overset{\alpha'}{\rightleftharpoons}} B \implies C \underset{\gamma \circ \gamma'}{\overset{\alpha' \circ \alpha}{\rightleftharpoons}} B$$

The simplest abstraction for processes that we use, is the hiding of events in a process using the abstraction function. That is, let $\mathcal{V} \subseteq \mathcal{W}$, then the domain $\mathcal{P}^{\mathcal{V}}$ of processes with events $\mathcal{V}$ is an abstraction for the domain $\mathcal{P}^{\mathcal{W}}$ of processes with events $\mathcal{W}$. The projection function defines the Galois connection: the concretization function is $\pi_{\mathcal{W}} \in \mathcal{P}^{\mathcal{V}} \to \mathcal{P}^{\mathcal{W}}$ and abstraction function is $\pi_{\mathcal{V}} \in \mathcal{P}^{\mathcal{W}} \to \mathcal{P}^{\mathcal{V}}$.

In general there are multiple Galois connections between a concrete domain and an abstract domain. For example, there is always a connection $\alpha(c) = \bot^A$ and $\gamma(a) = \top^C$. However, a Galois connection is uniquely determined by either the abstraction or the concretization function as expressed by the following theorem from [Smi10].

Theorem 1. *Let* $C \underset{\gamma}{\overset{\alpha'}{\rightleftharpoons}} A$ *denote a Galois connection between the partially ordered sets* $\langle C, \sqsubseteq^C \rangle$ *and* $\langle A, \sqsubseteq^A \rangle$*, then*

- $\alpha(c) = \bigsqcap^A \{a \mid a \in A, c \sqsubseteq^C \gamma(a)\}$
- $\gamma(a) = \bigsqcup^C \{c \mid c \in C, \alpha(c) \sqsubseteq^A a\}$

This theorem simplifies the construction of Galois connection through implicit definitions of the abstraction and concretization functions in terms of the other.

3.1.3 The Abstract Domain of Relative Counter Functions

As a first step towards other abstractions, we define the domain of relative counter functions as an abstraction of processes. The domain of relative counter functions abstracts from time, keeping only the interleaving of events.

Like the representation of behaviors as a vector of counter or dater functions, we group the relative counter functions in a matrix. Let the behavior of a system with events $\mathcal{V}$ be described by a column vector $\boldsymbol{\chi} \in (\mathbb{R}^\infty \to \mathbb{N}^\infty)^{|\mathcal{V}|\times 1}$ of counter functions and the corresponding row vector $\boldsymbol{\eta} \in (\mathbb{N}^\infty \to \mathbb{R}^\infty)^{1\times|\mathcal{V}|}$ of dater functions, then we define the matrix $X \in \mathcal{F}^{|\mathcal{V}|\times|\mathcal{V}|}$ of counter functions as their product:

$$X = \begin{pmatrix} \chi_1 \\ \vdots \\ \chi_n \end{pmatrix} \circ (\eta_1 \cdots \eta_n) = \begin{pmatrix} \chi_1 \circ \eta_1 & \cdots & \chi_1 \circ \eta_n \\ \vdots & \ddots & \vdots \\ \chi_n \circ \eta_1 & \cdots & \chi_n \circ \eta_n \end{pmatrix}$$

Abusing notation, we access elements of the matrix using the names of events. So $X_{i/j}$ is equivalent to X_{x_i/y_j} in the context of a behavior $\boldsymbol{x}$. Elements of the domain consist of sets of relative counter function matrices, each corresponding to one or more behaviors.

Definition 14 (Domain of Relative Counter Function Matrices ($\mathfrak{X}$))**.** *Let $\mathfrak{X}^\mathcal{V}$ denote the domain of all sets of relative counter function matrices with dimensions $N \times N$ indexed by elements from $\mathcal{V} = \{x1, ..., x_N\}$, ordered by the subset relation, with $\mathcal{P}^\mathcal{V} \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} \mathfrak{X}^\mathcal{V}$ defined by*

$$\alpha(P) = \{(\chi_{x_1}, ..., \chi_{x_N}) \circ (\eta_{x_1}, ..., \eta_{x_N})^T \mid (x_1, ..., x_N) \in P\}$$

As for processes, we specify the dimension (number of signals) of the domain in superscript when needed. The concrete domain $\mathcal{P}^\mathcal{V}$ of processes with signals $\mathcal{V}$ is related to the abstract domain $\mathfrak{X}^\mathcal{V}$ of relative counter function matrices.

The concretization function is defined implicitly by the abstraction using Theorem 1 such that a Galois connection is formed. In this case, the concretization of a set of relative counter function matrices consists of all behaviors that have relative counter functions matching one of the matrices.

It is worth noting that sets of relative counter function matrices contain less information than the original process. In fact, a single relative counter function matrix abstracts from an infinite number of behaviors. That is, we can obtain an

infinite number of behaviors with the same interleavings of events by stretching and compressing time.

The intersection of two processes P and Q in $\mathcal{P}^N$ (with the same number of signals) can be applied directly to the abstraction, as

$$\alpha^{\mathcal{X}}(P \sqcap Q) = \alpha^{\mathcal{X}}(P) \sqcap \alpha^{\mathcal{X}}(Q)$$

Note that, like for the processes, $\cap = \sqcap$ and $\cup = \sqcup$.

The projection operator is also lifted to the domain of relative counter functions where it changes the dimension of the relative counter function matrices. The projection $\pi_{\mathcal{V}} \in \mathcal{X}^{\mathcal{W}} \to \mathcal{X}^{\mathcal{V}}$ is defined such that

$$\pi_{\mathcal{V}}(A) = \left\{ X \;\middle|\; \begin{array}{c} X \in \mathcal{X}^V, X' \in A \\ \forall x, y \in (\mathcal{V} \cap \mathcal{W}) :\; X_{x/y} = X'_{x/y} \end{array} \right\}$$

for all $A \in \mathcal{X}^N$. As for the domain of concrete processes, the projection defines a Galois connection between domains $\mathcal{X}^{\mathcal{V}}$ and $\mathcal{X}^{\mathcal{W}}$ such that $\mathcal{V} \subseteq \mathcal{W}$.

The lifting of projection to relative counter functions also extends the composition operator $\|$ to relative counter functions with

$$A \parallel B = \pi_{\mathcal{V} \cup \mathcal{W}}(A) \cap \pi_{\mathcal{V} \cup \mathcal{W}}(B)$$

for $A \in \mathcal{X}^{\mathcal{V}}$ and $B \in \mathcal{X}^{\mathcal{W}}$.

3.2 The Domain of Clock Bounds

The domain $\mathcal{X}$ is not a very practical abstraction as its elements consist, in general, of infinitely large sets of relative counter functions. Clock bounds provide an abstraction that describes processes by the (point-wise) maximum of all relative counter functions.

Definition 15 (Clock bound)**.** *The bounds $C^l_{x/y}$ and $C^u_{x/y}$ in $\mathcal{F}$ are the lower resp. upper clock bounds of signals x and y such that for all $n \in \mathbb{N}^\infty$*

$$C^l_{x/y}(n) \leq X_{x/y}(n) \leq C^u_{x/y}(n)$$

Clock bounds limit the total number of events at every occurrence of another signal, e.g., the upper clock bound $C^u_{x/y}(n) = m$ states that at the n-th occurrence of y, x may have occurred at most m times. It constitutes a direct bound on the relative counter function $X_{x/y}$.

Elements from the *domain* of clock bounds describe all processes that respect a pair of upper and lower drift bound matrices.

Definition 16 (Clock bound domain ($\mathcal{C}$))**.** *The abstract domain $\mathcal{C}^N$ of clock bounds consists of the product of upper and lower bound matrices in $\mathcal{F}^{N \times N}$, ordered by the element-wise comparison and $\mathcal{X} \underset{\gamma^{\mathcal{C}}}{\overset{\alpha^{\mathcal{C}}}{\rightleftarrows}} \mathcal{C}$ defined by*

$$\gamma^{\mathcal{C}}(C) = \{X \mid X \in \mathcal{X}^N, \forall 1 \leq i, j \leq N :\; C^l_{i/j} \leq X_{i/j} \leq C^u_{i/j}\}$$

Clock bounds can model general difference bounds such as $\chi_i(t) - \chi_j(t) \geq B$ through $C^u_{i/j}(n) = B + n$ as well as relative clock rates such as $\chi_i(t) \leq \chi_j(t)R$ through $C^u_{i/j}(n) = nR$, or combinations of both difference bounds and relative rates. Even rates or difference bounds that change over time can be modelled, as long as the behavior is eventually periodic (to have a finite representation of the bound). For example, the clocks constraints of N-synchronous Kahn networks [CDE$^+$06] and, in particular, the clock envelopes of [CMPP08] can be presented as a clock bounds.

3.2.1 Composition of Clock Bounds

Like relative counter function abstraction we lift the projection and composition of processes to the clock bound abstraction. This enables the compositional description of systems with clock bounds and enables us to work with (infinitely) large sets of behaviors represented by their clock bounds.

First, the intersection and union of clock bound matrices naturally arise from the Galois connection because the . The intersection consists in the greatest lower bound $\sqcap^{\mathcal{C}}$ and takes the element-wise, point-wise infima. That is, for all $A, B \in \mathcal{C}$ and $n \in \mathbb{N}^\infty$ the upper bound of the intersection of A and B is defined such that

$$[A \sqcap^{\mathcal{C}} B]^u_{i/j}(n) = \min(A_{i/j}(n), B_{i/j}(n))$$

The lower bound is defined similarly, but with the point-wise maximum.

As for the relative counter function matrices, the projection operator $\pi_{\mathcal{W}}$ removes and adds dimensions (indexed by event names). Newly added signals are completely unbounded. Consider, for example, the projection $\pi_{\mathcal{W}}(C)$ of clock bound matrices $C \in \mathcal{C}^{\mathcal{V}}$ where $\mathcal{W} = \{x_1, ..., x_N, y_1, ...\}$ and $\mathcal{V} = \{x_1, ..., x_N\}$, then

$$[\pi_{\mathcal{V}}(C)]^u = \begin{pmatrix} C^u_{x_1/x_1} & \cdots & C^u_{x_1/x_N} & \top^{\mathcal{F}} & \cdots & \top^{\mathcal{F}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ C^u_{x_N/x_1} & \cdots & C^u_{x_1/x_N} & \top^{\mathcal{F}} & \cdots & \top^{\mathcal{F}} \\ \top^{\mathcal{F}} & \cdots & \top^{\mathcal{F}} & \top^{\mathcal{F}} & \cdots & \top^{\mathcal{F}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \top^{\mathcal{F}} & \cdots & \top^{\mathcal{F}} & \top^{\mathcal{F}} & \cdots & \top^{\mathcal{F}} \end{pmatrix}$$

and similarly for the lower bounds. The important consequence is that the composition operator $C \parallel C' = \pi_{\mathcal{V}\cup\mathcal{W}}(C) \cap \pi_{\mathcal{V}\cup\mathcal{W}}(C')$ with $C \in \mathcal{C}^{\mathcal{V}}$ and $C' \in \mathcal{C}^{\mathcal{W}}$ results in a matrix that contains three submatrices: (1) the matrix for signals $\mathcal{V} \setminus \mathcal{W}$ only in C, (2) the matrix for signals $\mathcal{W} \setminus \mathcal{V}$ only in C', and (3) the matrix for signals $\mathcal{V} \cap \mathcal{W}$ shared by C and C'. The upper bound of the composition is as follows:

$$[C \parallel^{\mathcal{C}} C']^u = \begin{pmatrix} C & & \top \\ & \pi_{\mathcal{V}\cap\mathcal{W}}(C) \sqcap \pi_{\mathcal{V}\cap\mathcal{W}}(C') & \\ \top & & C' \end{pmatrix}$$

The lower bound of the product looks similar, but with $\bot$ in the place of $\top$, for the lower bound.

3.2.2 Properties of Clock Bounds

Clock bounds inherit the transitivity and pseudo-symmetry of relative counter functions. These properties are direct consequences of Lemmas 5 and 6.

Lemma 7 (Transitivity of clock bounds)**.** *Let $C \in \mathcal{C}^{\mathcal{V}}$ be a pair of lower and upper clock bound matrices then, for all $X \in \gamma^{\mathcal{C}}(C)$, all $i, j, k \in \mathcal{V}$ and all $n \in \mathbb{N}^\infty$*

$$C^l_{i/k}(C^l_{k/j}(n)) \leq X_{i/j}(n) \leq C^u_{i/k}(C^u_{k/j}(n) + 1)$$

Computing the transitive closure of clock bounds is closely related to the verification of the schedulability of a data-flow system such as SDF [LM87] and CSDF [BELP96] as well as the N-synchronous Kahn networks [CDE$^+$06]. In particular, if any upper bounds in the diagonal of the transitive closure is less than the identity ($\exists i \in \mathcal{V}, n \in \mathbb{N}^\infty :\ D^u_{i/i}(n) < n$), the system has a deadlock.

Lemma 8 (Pseudo-symmetry of clock bounds)**.** *Let $(C^l, C^u) \in \mathcal{C}^{\mathcal{V}}$ be a pair of lower and upper clock bound matrices then, for all $X \in \gamma^{\mathcal{C}}$, all $i, j \in \mathcal{V}$ and all $n \in \mathbb{N}^\infty$*

$$[C^u_{j/i}]^{-1}(n) \leq X_{i/j}(n) \leq [C^l_{j/i}]^{-1}(n) + 1$$

Compositions often yield clock bound matrices that are imprecise in the sense that there exist other, smaller bounds that abstract the same process. The practical result of transitivity and symmetry of clock bounds, is that we can use them to tighten the bounds of the composition.

3.3 The Domain of Drift Bounds

Clock bounds cannot model time-invariant properties such as a bound on bursts or jitter. Neither can they express quasi-synchronous clock constraints. For those, we introduce drift bounds.

Drift bounds limit the incline of the relative counter function for each interval. As shown in Figure 3.1, the constraint imposed by a drift bound can be

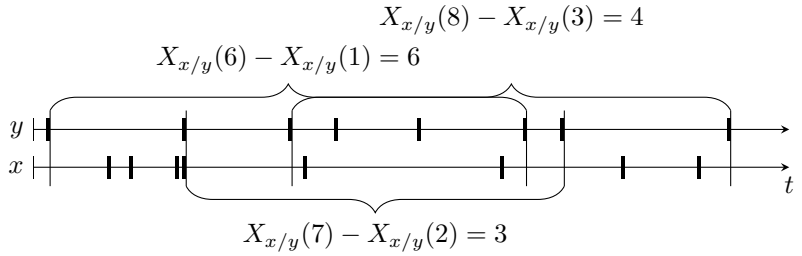


Figure 3.1: Interpretation of a drift bound as a sliding window: bounding the number of occurrences of x in every interval of 5 occurrences of y.

interpreted as a sliding window constraint. In the figure, x occurs at most six times and at least 3 times in every interval of 5 occurrences of y. Drift bounds generalize this to a more versatile mechanism to apply different constraints for each interval size.

Definition 17 (Drift bound)**.** *The bounds $D^l_{x/y}$ and $D^u_{x/y}$ in $\mathcal{F}$ are a lower resp. upper drift bound of the signals x and y such that, for all interval sizes $\Delta \in \mathbb{N}^\infty$ and all $n \in \mathbb{N}^\infty$*

$$D^l_{x/y}(\Delta) \leq X_{x/y}(n + \Delta) - X_{x/y}(n) \leq D^u_{x/y}(\Delta)$$

Drift bounds can describe a wide variety of concepts depending on the significance of the bounded signals.

- In case of communication-by-sampling, a guaranteed drift bound on the clock signals p and c of producer resp. consumer can be used to find bounds on the variability of communicated values. Let v and v' in $\mathbb{N} \to \mathbb{R}$ denote valuation functions for the producer resp. consumer such that $v(n)$ gives the n-th produced value (at time χ_p). Let $V \in \mathbb{N} \to \mathbb{R}$ be a variability bound such that $|v(n + \Delta) - v(n)| \leq V(\Delta)$ for all $n, \Delta \in \mathbb{N}$ (note the similarity with drift bounds). Then the variability of the values read by the consumer is bounded such that, for all $n, \Delta \in \mathbb{N}$,

$$|v'(n + \Delta) - v'(n)| \leq [V \circ D^u_{p/c}](\Delta)$$

- Interpreting clocks either as the availability of resources or the arrivals and departure of packets in a stream processing network, drift bounds can express bounds on the availability of resources and burstiness of arrival streams. This application is explored in the next chapter.

- The requirements for weakly hard real-time systems, where deadlines may be missed occasionally but not too often, can also be expressed as drift bounds. Let e be the error event that occurs when a deadline is missed and c a reference clock representing either an observer's activations or a quantization of real-time. A drift bound $D^u_{e/c}$ such that $D^u_{e/c}(100) = 10$

expresses the constraint on behaviors (e, c) that, e.g. 10 deadlines may be missed for every 100 time units. This principle can be generalized to any type of error, resulting in a notion of quality of service (QoS).

We define the drift bound domain in a similar fashion as the clock bound domain through a galois connection with the domain of relative counter functions.

Definition 18 (Drift bound domain (D))**.** *The abstract domain $\mathcal{D}^N$ of drift bounds consists of the product of all upper and lower drift bound matrices in $\mathcal{F}^{N\times N}$, ordered by the element-wise comparison and $\mathcal{X} \underset{\gamma^{\mathcal{D}}}{\overset{\alpha^{\mathcal{D}}}{\rightleftarrows}} \mathcal{D}$ defined by*

$$\gamma^{\mathcal{D}}(D) = \left\{ X \;\middle|\; \begin{array}{c} X \in \mathcal{X}^N, \forall 1 \le i,j \le N : \ \forall n, \Delta \in \mathbb{N}^\infty : \\ D^l_{i/j}(\Delta) \le X_{i/j}(n+\Delta) - X_{i/j}(n) \le C^u_{i/j}(\Delta) \end{array} \right\}$$

The condition in the definition of $\gamma^{\mathcal{D}}$ can be rewritten using special operators that derive from network calculus [LT01], namely the min/max-plus deconvolutions ($\oslash$ and $\overline{\oslash}$):

$$\begin{aligned} D^u_{i/j}(\Delta) \ge [X_{i/j} \oslash X_{i/j}](\Delta) \quad &= \sup\{X_{i/j}(n+\Delta) - X_{i/j}(n) \mid n \in \mathbb{N}^\infty\} \\ D^l_{i/j}(\Delta) \le [X_{i/j} \overline{\oslash} X_{i/j}](\Delta) \quad &= \inf\{X_{i/j}(n+\Delta) - X_{i/j}(n) \mid n \in \mathbb{N}^\infty\} \end{aligned}$$

for all $X \in \mathcal{X}^N$, all $1 \le i,j \le N$ and all $\Delta \in \mathbb{N}^\infty$. The deconvolution operators are the dual of the min/max-plus convolution operators, which allows us to write

$$\begin{aligned} X_{i/j}(n) \le [X_{i/j} \otimes D^u_{i/j}](n) \quad &= \inf\{X_{i/j}(n-\Delta) - D^u_{i/j}(\Delta) \mid 0 \le \Delta \le n\} \\ X_{i/j}(n) \ge [X_{i/j} \overline{\otimes} D^l_{i/j}](n) \quad &= \sup\{X_{i/j}(n-\Delta) - D^l_{i/j}(\Delta) \mid 0 \le \Delta \le n\} \end{aligned}$$

While we will avoid the operators where possible, some proofs rely on their properties. Appendix A gives a more complete overview of the operators and their properties.

Drift bounds are composed with the same operators as clock bounds, except that we give operators the superscript $\mathcal{D}$ to distinguish them, .e.g., $D \sqcup^{\mathcal{D}} D'$.

3.3.1 Properties of Drift Bounds

Let us note some important properties of drift bounds that, eventually, can be used to find tighter bounds for any given set of bounds.

Like the clock bounds, drift bounds inherit transitivity and pseudo-symmetry of relative counter functions.

Lemma 9 (Transitivity of drift bounds)**.** *Let $D \in \mathcal{D}^N$ be a pair of drift bound matrices then, for all $X \in \gamma^{\mathcal{D}}(D)$, all $n, \Delta \in \mathbb{N}^\infty$ and all $1 \le i,j,k \le N$*

$$[D^l_{i/k} \circ D^l_{k/j}](\Delta) \le X_{i/j}(n+\Delta) - X_{i/j}(n) \le [D^u_{i/k} \circ (D^u_{k/j} + 1)](\Delta)$$

Lemma 10 (Pseudo-symmetry of drift bounds)**.** *Let $D \in \mathcal{D}^N$ be a pair of drift bound matrices then, for all $X \in \gamma^{\mathcal{D}}(D)$, all $n, \Delta \in \mathbb{N}^\infty$ and all $1 \leq i, j \leq N$*

$$[D^u]^{-1}_{i/j}(\Delta) \leq X_{i/j}(n+\Delta) - X_{i/j}(n) \leq [D^l]^{-1}_{i/j}(\Delta) + 1$$

Unlike clock bounds, drift bounds are additive. Recall the interpretation of drift bounds as constraints within a sliding window for each interval size. Take two such intervals and place them next to each other so that one starts where the other stops. It then stands to reason that the joined interval is bounded by the sum of the bounds for the two partial intervals. This is the foundation of additivity of bounds.

Lemma 11 (Additivity of drift bounds)**.** *Let $D \in \mathcal{D}^N$ be a pair of drift bound matrices then, for all $X \in \gamma^{\mathcal{D}}(D)$, all $n, \Delta, \Delta' \in \mathbb{N}^\infty$,*

$$D^l_{i/j}(\Delta) + D^l_{i/j}(\Delta') \leq X_{i/j}(n - (\Delta + \Delta')) - X_{i/j}(n) \leq D^u_{i/j}(\Delta) + D^u_{i/j}(\Delta')$$

Proof. By definition of the concretization, for all $X \in \gamma(D)$, all $n, m, \Delta, \Delta' \in \mathbb{N}^\infty$

$$D^l_{i/j}(\Delta) \leq X_{i/j}(n-\Delta) - X_{i/j}(n) \leq D^u_{i/j}(\Delta)$$

and

$$D^l_{i/j}(\Delta') \leq X_{i/j}(m-\Delta') - X_{i/j}(m) \leq D^u_{i/j}(\Delta')$$

then, for $n = m - \Delta'$, we obtain

$$\begin{aligned}
&D^l_{i/j}(\Delta) + D^l_{i/j}(\Delta') \\
\leq &X_{i/j}(n-\Delta) - X_{i/j}(n) + X_{i/j}(m-\Delta') - X_{i/j}(m) \\
= &X_{i/j}(m - (\Delta + \Delta')) - X_{i/j}(m) \\
\leq &D^u_{i/j}(\Delta) + D^u_{i/j}(\Delta')
\end{aligned}$$

□

Note that the upper and lower drift bounds are bounded from a different direction: upper bounds are sub-additive while lower bounds are super-additive.

3.3.2 Interaction of Clock and Drift Bounds

The drift and clock bounds seem quite similar, but capture different aspects of processes. If the two domains are combined, yielding a product domain, the question arises how the two domains interact. That is, we seek the drift and clock bounds of the composition $\gamma^{\mathcal{C}}(C) \cap \gamma^{\mathcal{D}}(D)$ for a pair $(C, D) \in \mathcal{C} \times \mathcal{D}$. Moreover, since the enumeration of all relative counter functions produced by the concretization functions is infeasible, we seek bounds that derive directly from the drift and clock bounds without a concretization step.

First, we consider the clock bound as it is affected by the clock bound. The drift bounds limit the incline of any relative counter function, therefore it also has an effect on the clock bound.

Lemma 12 (Drift bound on clock bound)**.** *Let* $(C, D) \in \mathcal{C}^N \times \mathcal{D}^N$ *then, for all* $X \in \gamma^{\mathcal{C}}(C) \cap \gamma^{\mathcal{D}}(D)$ *and all* $1 \leq i, j \leq N$

$$D^l_{i/j} \otimes C^l_{i/j} \leq X_{i/j} \leq C^u_{i/j} \otimes D^u_{i/j}$$

Proof. Recall that, by reformulating the concretization function of the drift bound abstraction with the min/max-plus convolution operators, we have

$$X_{i/j} \overline{\otimes} D^l_{i/j} \leq X_{i/j} \leq X_{i/j} \otimes D^u_{i/j}$$

for all $X \in \gamma^{\mathcal{D}}(D)$ and all $1 \leq i, j \leq N$. Using the fact that the min/max-plus operators are monotonic and that $X \in \gamma^{\mathcal{C}}(C)$, we substitute $X_{i/j}$ in the left- and right-hand-side with their lower ($C^l_{i/j}$) respectively upper ($C^u_{i/j}$) clock bounds to obtain the result. □

Second, we consider the effect of the clock bound on the drift bound. A drift bound is limited by the maximum incline that is possible for any relative clock that satisfies the clock bounds. In particular, if the upper and lower clock bounds are very close to one another, there is little room for bursts.

Lemma 13 (Clock bound on drift bound)**.** *Let* $(C, D) \in \mathcal{C} \times \mathcal{D}$ *then, for all* $X \in \gamma^{\mathcal{C}}(C) \cap \gamma^{\mathcal{D}}(D)$*, all* $1 \leq i, j \leq N$ *and all* $n, \Delta \in \mathbb{N}^\infty$

$$[C^l_{i/j} \overline{\oslash} C^u_{i/j}](\Delta) \leq X_{i/j}(n + \Delta) - X_{i/j}(n) \leq [C^u_{i/j} \oslash C^l_{i/j}](\Delta)$$

Proof. By definition of the min/max-plus deconvolutions, we have

$$[X_{i/j} \overline{\oslash} X_{i/j}](\Delta) \leq X_{i/j}(n + \Delta) - X_{i/j}(n) \leq [X_{i/j} \oslash X_{i/j}](\Delta)$$

for all $X \in \mathcal{X}$, all $n, \Delta \in \mathbb{N}^\infty$ and all $1 \leq i, j \leq N$. Using the monotonicity of the deconvolutions and the fact that $X \in \gamma^{\mathcal{C}}(C)$, we substitute $X_{i/j}$ by their lower and upper bounds to obtain the result. Note the order of the operants of the deconvolution is reversed, i.e., they are monotone for the left-hand-side operator and antitone for the right-hand-side. □

If we consider that a drift bound can be interpreted as a clock bound, i.e., for any $D \in \mathcal{D}$ $X \in \gamma^{\mathcal{D}}(D)$ implies $X \in \gamma^{\mathcal{C}}(D)$ (the reverse does not hold though) we can, in fact, restate both lemma's as properties of drift bounds. Lemma 12 then leads to the additivity of drift bounds and Lemma 13 leads to a notion very similar the causality problem treated in [MA10].

3.4 Abstract Interpretation of GALS Systems

Chapters 4 and 6 propose analyses for GALS system designs. In those analyses, the clock and drift bounds serve two purposes. Firstly they specify the behavior of the environment, i.e., of the input flows and clocks that drive the system. Secondly, the clock and drift bounds are used to express constraints on, or guarantees provided by, the GALS system when executed in that environment.

In other words, we seek to compute the abstraction $\alpha(P \parallel \gamma(a))$ of the system's process $P \in \mathcal{P}$ when constrained to an environment $a \in \mathcal{A}$, be it a clock bound, drift bound, or both.

Abstract interpretation [CC77, CC92] evaluates program behavior in an abstract domain to obtain an abstraction of its concrete behavior. Let $C \rightleftharpoons \mathcal{A}$ denote a Galois connection between the concrete domain C and abstract domain $\mathcal{A}$ with abstraction function α and concretization function γ. With concrete semantics described by a transfer function $f \in C \to C$ that relates objects (typically input and output) in the concrete domain C, abstract interpretation uses abstract transfer functions $f^{\#} \in \mathcal{A} \to \mathcal{A}$ defined such that $f^{\#} = \alpha \circ f \circ \gamma$ or approximated by an abstract transfer function such that $f^{\#}(a) \sqsupseteq \alpha \circ f \circ \gamma(a)$ for all $a \in \mathcal{A}$. The abstract transfer function yields (an approximation of) the abstraction of the output of the concrete transfer function for the given, abstracted, input.

In this thesis processes are specified by equations of the shape $x = e$ where x is an event name and e is an expression (with an unspecified syntax) over other events. Let $[\![x = e]\!]$ denote the process described by the equation $x = e$, i.e., the process of all behaviors that satisfy the equation. Systems defined by multiple equations $x_1 = e_1, ..., x_n = e_n$ are described by the processes that satisfy all the equations. That is, a process defined by the composition of the individual equation's processes:

$$[\![x_1 = e_1; ...; x_n = e_n]\!] = [\![x_1 = e_1]\!] \parallel ... \parallel [\![x_n = e_n]\!]$$

Our task is to compute the abstractions of such composed systems and the theory of abstract interpretation enables us to do this efficiently by composing abstractions.

3.4.1 Composition as a Fix-Point of Transfer Functions

In practice, computing the (concrete) process of a composition is infeasible because we lack finite representations of concrete processes. Even if finite representations are available, e.g., as finite transition systems (see Chapter 6, the representation usually does not scale to large systems. Therefore we propose an approximative method where each component is modelled by a mapping over the abstract domain that restricts abstracted behavior to those that are possible in the presence of the component. To handle cyclic relations, the components' mappings are combined in a fix-point computation.

Figure 3.2 schematically depicts the approximated abstraction of two composed processes. Pictured at the top is the approximated composition in the abstract domain $\mathcal{A}$, which may be the domain of clock bounds, the domain of drift bounds, or their combination. The lower left and right components depict the composition of the components with the abstraction of the global composition in the concrete domain. Because the abstraction only conservatively approximates the behaviors of the environment, each component will also conservatively approximate the behavior of the global system.

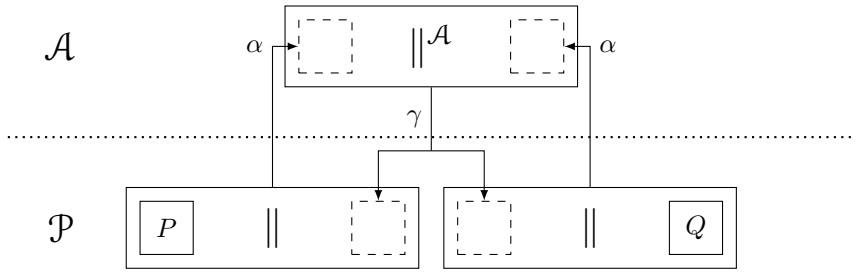


Figure 3.2: Approximation of the abstraction $\alpha(P \parallel Q)$ of a composition.

The Knaster-Tarski theorem [Tar55], stated below for the greatest fix-point, tells us that there is a unique greatest and a unique least fix-point for any monotonic function over a complete lattice. For a more complete introduction we refer to [DP02].

Theorem 2 (Knaster-Tarski)**.** *Let S denote a complete lattice and $\Pi \in S \to S$ a monotonic function, i.e., a function such that $s \sqsubseteq s' \implies \Pi(s) \sqsubseteq s'$ for all $s, s' \in S$, then the set of fix-points is a complete lattice with greatest fixed point*

$$\operatorname{gfp} \Pi = \bigsqcap \{s \mid s \in S,\ \Pi(s) \sqsupseteq s\}$$

The theorem enables us to state the following equality that defines a composition as a fix-point:

$$\begin{aligned} &[\![x_1 = e_1; ...; x_n = e_n]\!] \parallel [\![x'_1 = e'_1; ...; x'_m = e'_m]\!] \\ =& \operatorname{gfp}\left([\![x_1 = e_1; ...; x_n = e_n]\!]^{\|\mathcal{P}} \circ [\![x'_1 = e'_1; ...; x'_m = e'_m]\!]^{\|\mathcal{P}}\right) \end{aligned}$$

where the *concrete transfer function* $[\![x_1 = e_1; ...; x_n = e_n]\!]^{\|\mathcal{P}} \in \mathcal{P} \to \mathcal{P}$ of equations $x_1 = e_1, ..., x_n = e_n$ yields the process of the equations composed with its environment and is defined such that

$$[\![x_1 = e_1; ...; x_n = e_n]\!]^{\|\mathcal{P}}(P) = [\![x_1 = e_1; ...; x_n = e_n]\!] \parallel P$$

By itself, the expression of composition as a fix-point of transfer functions seems unnecessarily complicated. With the following theorem from [CC92], however, we can use it to justify a fix-point computation in the abstraction.

Theorem 3 (Sound over-approximation of fix-ponts)**.** *Let $C \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} \mathcal{A}$ denote a Galois connection between complete lattices C and $\mathcal{A}$, and $\Pi \in C \to C$ a monotonic function then*

$$\alpha(\operatorname{gfp}(\Pi)) \sqsubseteq \operatorname{gfp}(\alpha \circ \Pi \circ \gamma)$$

Consequently, the abstraction of a composed system can be approximated with the fix-point of their abstract transfer functions, i.e.,

$$\begin{aligned} &\alpha([\![x_1 = e_1; ...; x_n = e_n]\!]^{\mathcal{P}} \parallel [\![x'_1 = e'_1; ...; x'_m = e'_m]\!]^{\mathcal{P}}) \\ \sqsubseteq\ & \operatorname{gfp}\left([\![x_1 = e_1; ...; x_n = e_n]\!]^{\|\mathcal{A}} \circ [\![x'_1 = e'_1; ...; x'_m = e'_m]\!]^{\|\mathcal{A}}\right) \end{aligned}$$

where the *abstract transfer function* of equations $x_1 = e_1; ...; x_n = e_n$ is defined such that

$$[\![x_1 = e_1; ...; x_n = e_n]\!]^{\|\mathcal{A}} = \alpha \circ [\![x_1 = e_1; ...; x_n = e_n]\!]^{\|\mathcal{P}} \circ \gamma$$

This is the ideal abstract transfer function. Most of the time, we will use an approximative abstract transfer function, such that, for all $a \in \mathcal{A}$

$$[\![x_1 = e_1; ...; x_n = e_n]\!]^{\|\mathcal{A}}(a) \sqsupseteq (\alpha \circ [\![x_1 = e_1; ...; x_n = e_n]\!]^{\|\mathcal{P}} \circ \gamma](a)$$

That is, it over-approximates the possible behaviors.

By itself, the fix-point formulation of the approximation does not suffice to compute the result. The Kleene theorem [Kle52] permits the computation of the greatest and least fix-points of continuous (limit-preserving) mappings for infinite lattices by an iterative method. Kleene iteration formalizes the computation of the abstract composition through a decreasing chain of bounds described previously.

Theorem 4 (Kleene Iteration)**.** *Let S be a complete lattice and $\Pi \in S \to S$ an upper semi-continuous mapping, i.e.,*

$$\bigsqcap_{s \in S'} \Pi(s) = \Pi\left(\bigsqcap_{s \in S'} s\right)$$

for any non-empty, totally ordered subset $S' \subseteq S$, then

$$\operatorname{gfp} \Pi = \bigsqcap \{\Pi^n(\top) \mid n \in \mathbb{N}\} = \top \sqcap\ \Pi(\top)\ \sqcap\ \Pi(\Pi(\top))\ \sqcap ...$$

Assuming the involved abstract transfer functions are continuous, the fix-point approximation of the composition in Figure 3.2 can be computed with a Kleene Iteration.

3.4.2 Abstract Mappings for Properties

When approximative abstract transfer functions are used, we can exploit the properties of the abstract domains to define additional mappings that compensate (in part) for the pessimism of the abstract mappings resulting in tighter bounds. The basic problem is that the approximate transfer functions yield drift and clock bounds that are not tight, in the sense that there exist smaller bounds that have precisely the same concretization. An abstraction is tight if there exists no smaller bound that has the same concretization. Theoretically, the greatest tight abstraction of some $a \in \mathcal{A}$ can be obtained by computing $\alpha(\gamma(a))$ (see e.g. [Smi10]), but in practice this is often impossible to compute (e.g. because there is no finite representation of the concretization) or infeasible. The properties of clock and drift bounds provide an opportunity to at least improve the result by approximating the tight bounds.

To find tighter bounds, we use the properties of clock and drift bounds. The transitivity mapping, for example, ensures the final result is transitively closed,

and the transitive closure of clock or drift bounds have the same concretization as the original (not transitively closed) bounds. That is, we define abstract mappings, such as $\Pi_{trans} \in \mathcal{C} \to \mathcal{C}$ for transitivity of clock bounds that computes the transitive closure of clock bounds within the environment. By integrating such mappings in the fix-point the computation will yield better bounds by ensuring the resulting bounds are transitive. Formally, we defend the use of such mappings by representing them as approximations of $\alpha \circ \gamma$.

To apply a mapping, it is sufficient to show that a (continuous) mapping over-approximates the abstract identity in order to add it to the fix-point computation. The properties of clock and drift bounds (transitivity of Lemmas 7 and 9, symmetry of Lemmas 8 and 10, additivity of Lemma 11 and the interaction of Section 3.3.2) naturally lead to approximations of $\alpha \circ \gamma$, because the properties are formulated as drift or clock bounds on the relative counter functions that satisfy a given clock or drift bound. In other words, for a given abstraction $a \in \mathcal{A}$ (a clock bound, drift bound or their product) the properties give a tighter abstraction $a' \sqsubseteq a$ such that all behaviors in the concretization of a are also in a'.

Consider, for example, the transitivity of clock bounds presented in Lemma 7. Defining the mapping Π_{trans} such that for all $C \in \mathcal{C}^{\mathcal{V}}$ and all $n \in \mathbb{N}^{\infty}$

$$[\Pi_{trans}(C)]^l_{i/j}(n) = \max_{k \in \mathcal{V}} C^l(C^l(n))$$
$$[\Pi_{trans}(C)]^u_{i/j}(n) = \min_{k \in \mathcal{V}} C^u(C^u(n) + 1)$$

the Lemma 7 can be restated for all $C \in \mathcal{C}$, such that

$$\forall X \in \gamma(C), n \in \mathbb{N}^{\infty} : \ \Pi_{trans}(C)^l(n) \leq X(n) \leq \Pi_{trans}(C)^u(n)$$

That means, that all concretizations of $C \in \mathcal{C}$ are bounded by $\Pi_{trans}(C)^u$ and therefore $\Pi_{trans}(C) \sqsupseteq [\alpha \circ \gamma](C)$.

Such mappings are defined for all the introduced properties and are added to all fix-point computations to improve the bounds. That is, we include mappings for transitivity and symmetry of clock and drift bounds, additivity and causality of drift bounds and interaction between clock and drift bounds in all fix-points.

3.5 Conclusion

3.5.1 Related Work

In literature there are many concepts with similarities to drift and clock bounds. Drift and clock bounds can be used as a generalization of these notions by considering many different types of events such as the completion of tasks, expiration of a deadline, a change of system state, etc. The generalization enables the application of our results, such as the transitivity, to those concepts.

Weakly-hard real-time systems [Ber98, BBL01] permit the occasional deadline for the completion of a task to be missed as long as the system satisfies a

given set of constraints. The constraint $\binom{n}{m}$ expresses requires that at least n out of m consecutive deadlines are met. It can be expressed by a lower drift bound $\mathcal{D}^l_{s/d}(m) = n$ on the events s of satisfied deadlines with respect to deadlines d. The constraint $\overline{\langle n \rangle}$ requires that no n consecutive deadlines are ever missed, and can be expressed by a lower drift bound $D^l_{s/m}(2) = n$ that requires at least n occurrences of s (met deadlines) for every two missed deadlines (event m).

Real-time Calculus [TCN00] uses arrival curves to describe the maximum and minimum number of the arrivals of tasks for each time interval. The curves are a real-time version of drift bounds that constrain the (absolute) counter function for an event model that permits simultaneous events. The extension [PRT+10] also introduces a very similar but limited form of (relative) drift bounds. Chapter 4 uses drift bounds in a similar manner and compares the two domains in more detail.

Bertrane [Ber08] defines an abstract interpretation of imperfectly-clocked synchronous systems combining several abstract domains. The behavior of these systems is defined in a concrete domain of continuous boolean signals closely related to that of electronic circuits. The global changes-counting domain [Ber06] consists of constraints $(u, \delta) \in \mathbb{N}^\infty \times \mathbb{R}^\infty_{\geq 0}$ for each (boolean) signal that are satisfied if the signal changes at most u times in each (real) time-interval of size δ. The same constraint can be expressed with an upper drift bound $D^u_{c/t}(\delta) = u$ on the change event c with respect to a discretization of real-time t. The integral bound domain consist of constraints $(\delta, \alpha, \beta) \in \mathbb{R}^\infty_{\geq 0} \times \mathbb{N}^\infty \times \mathbb{N}^\infty$ on a boolean signal represented by real values in $\{0, 1\}$ that are satisfied if the integral of real-valued signal is bounded to the interval $[\alpha, \beta]$ for every interval of size δ. We express the same bound in Chapter 6 with a drift bound $D^l_{x/\hat{x}}(\delta) = \alpha$ and $D^u_{x/\hat{x}}(\delta) = \beta$ for the signal x of a boolean flow (synchronous signal) with respect to its clock $\hat{x}$.

3.5.2 Discussion

We introduced the system depicted in Figure 3.3 of abstractions for discrete event systems. Using the abstraction and concretization functions, we can freely combine components from different levels of the abstraction by concretizing the components and combining the resulting processes. That is, we describe the composition as the concretization of components and subsequent intersection at the same level of abstraction.

In the analysis of distributed systems clock and drift bound abstractions are used in two capacities:

- The abstractions can be used to express *guarantees* on the behavior of the environment of a system. Most notably in the case of (root) clocks and resources — the original purpose of the abstractions — drift bounds can model the availability of resources and stability of clocks. But also in the case of other events originating in the environment with bounded non-determinism.

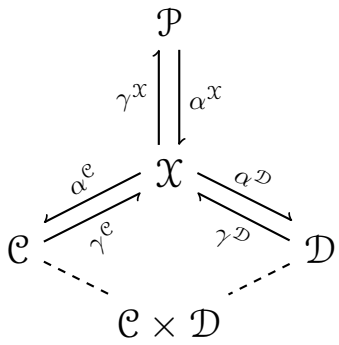


Figure 3.3: The Galois connections of the concrete domain $\mathcal{P}$ of processes with the abstract domains of (sets of) relative counter functions $\mathcal{X}$, clock bound $\mathcal{C}$, drift bound $\mathcal{D}$ and the product domain of both clock and drift bounds.

- The abstractions can be used to express *requirements* on the system. This mainly considers QoS requirements where errors may occur occasionally, but not too often.

Typically, one would analyze the behavior of a system when executed in an environment described by the abstractions. The analysis can be used in a purely informative fashion (e.g. to determine the frequency of certain events in a complex system) but also for the validation of requirements. We refer to the latter as verification to distinguish it from the more general concept of analysis. The analysis is naturally placed between guarantees and requirements: either to verify if the requirements are fulfilled by a system executed in an environment with the given guarantees, or to compute the guarantees provided by such a system.

The combination of such mappings modelling different sub-systems, is then used in a single fix-point computation that iteratively tightens the abstraction that describes system behavior.

Chapter 4

Analyzing Stream Processing Systems

Stream processing systems are GALS systems where the synchronous domains communicate over FIFO channels. Modern stream processing systems, such as multimedia applications and embedded automatic control systems, are often realized on heterogeneous, multi-processor architectures. Such architectures have a large design space, due to the choice in processors, the networks to connect them, the partitioning of software on the hardware, and the choice of scheduling regimes to allocate resources. As development progresses, architecture changes become increasingly expensive. It is therefore essential to evaluate the feasibility of a design at an early stage.

This chapter shows how to use the drift and clock bounds introduced in Chapter 3 to analyse the performance characteristics of stream processing systems. The clock and drift bound abstractions allow us to analyze systems in an environment with many unknown quantities: either because they have not yet been decided upon (early in the design phase) or due to the non-determinism inherent to distributed systems.

Consider, for example, the simple multimedia decoder depicted in Figure 4.1, that decodes audio and video parts of an incoming multimedia stream separately and then joins the decoded streams at the output. Each double arrow in the diagram represents a lossless channel with FIFO semantics (a stream). The PJD and BD components denote sources of the multimedia stream i and resources r and s. The greedy processing components (GPC) model the decoding of the input stream i, arriving periodically with bounded jitter and minimum inter-arrival time (PJD), on two separate processors, consuming resources r and s. The AND component joins the decoded streams resulting the output stream o.

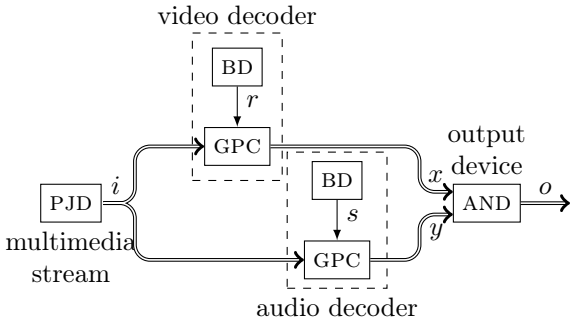


Figure 4.1: A multimedia decoder with separate audio and video processors.

4.1 Elements and Semantics of Stream Processing Systems

Stream processing systems are modelled by compositions of a few basic components that define relations between streams and resources. Components are processes (see Chapter 2) where signals model both (the availability of) resources and streams. This section defines the stream processing components as processes using the signals' counter functions. Many of the presented components are based on the components of the MPA real-time calculus toolbox [CKT03, HT07a].

4.1.1 Streams and Resources

Streams (the double arrows) tasks or messages to be treated using resources (single arrows). Both are modelled by signals. In the case of a stream, an event occurs on arrival or departure of a task or token. In the case of a resource, an event occurs when a resource is available.

The difference between departure and arrival events is a matter of perspective: occurrences of an incoming stream's signal are arrivals and occurrences of an outgoing stream's signal are departures. Connecting the output of one component with the input of another equates the signals thus one component's departure is another component's arrival event.

4.1.2 Source Components

Source components describe the behavior of the environment: the availability of resources and arriving streams. We capture bounds on availability and variability of arrivals with with clock and drift bounds. Such bounds can describe the non-deterministic behavior of the environment of a distributed system. Bounds can also be used to describe an environment whose exact behavior is unknown or has not (yet) been decided on.

The BD component

The BD component defines a resource with asymptotic rate (bandwidth or processing power) and a bounded, but variable, delay. In the long term, the resource defined by a bd component has a guaranteed rate but availability may be delayed occasionally. The rate and delay is defined with respect to a reference clock which can be any other signal. One possibility is to add a signal that models a discretization of real-time.

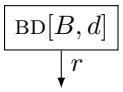


Figure 4.2: BD$[B, d]$

Formally the BD component defines the process below.

Definition 19 (BD)**.** *The* BD *component with bandwidth* $B \in \mathbb{R}$ *and delay* $d \in \mathbb{N}$ *describes a process* $\text{BD}_{B,d}$ *that consists of all behaviors* (r, k) *such that for all* $n \in \mathbb{N}^\infty$

$$\lfloor B(n-d) \rfloor \leq X_{r/k}(n) \leq \lceil Bn \rceil$$

The PJD component

The PJD component describes a stream that exhibits jitter, but is asymptotically stable. Moreover, it gives a lower bound on the time between each two consecutive events.

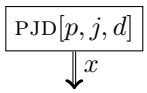


Figure 4.3: PJD$[p, j, d]$

The component has parameters $p, j, d \in \mathbb{N}$ for the period, jitter, and inter-arrival time respectively. Formally the BD component defines the process below.

Definition 20 (PJD)**.** *The* PJD *component with period* $p \in \mathbb{R}$ *and jitter* $j \in \mathbb{N}$ *and inter-arrival time* $d \in \mathbb{N}$ *describes a process that consists of all behaviors* (x, k) *such that for all* $n, \Delta \in \mathbb{N}^\infty$

$$\lfloor \Delta/p \rfloor - j \leq X_{x/k}(n+\Delta) - X_{x/k}(n) \leq \lceil \min(\Delta/p + j, \Delta/d) \rceil$$

4.1.3 The Greedy Processing Component

The most important component is the GPC which models the processing of a stream according to the availability of a resource. The incoming stream is

buffered and processed as soon as resources come available. The greedy processing component was first introduced in this form in [TCN00]. The original definition also provides a stream of remaining resources, which we have left out for simplicity.

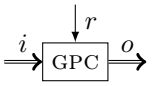


Figure 4.4: GPC

The behavior of the GPC is formalized below. It is a deterministic process as the output stream is defined as a function of the input stream and resource.

Definition 21 (GPC)**.** *The* GPC *component describes a process with behaviors* (i, o, r) *such that, for all* $t \in \mathbb{R}$

$$\chi_o(t) = \min\{\chi_i(u) + \chi_r(t) - \chi_r(u) \mid 0 \leq u \leq t\}$$

4.1.4 The Synchronous Join Component

The purely logical AND component — logical, because it is not dependent on resources — synchronously *joins* two streams. It is based on the component presented in [Wan06]. The AND component takes one arriving task of each stream x and y and combines them into a single output task on stream z.

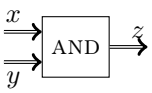


Figure 4.5: AND

Combining the tasks is instantaneous, so the total number of events in the outputs stream at any time is the minimum of the total number of events of the input streams.

Definition 22 (AND)**.** *The* AND *component defines a process with behaviors* (x, y, z) *such that for all* $t \in \mathbb{N}$

$$\chi_z(t) = \min(\chi_x(t), \chi_y(t))$$

4.1.5 Time-Division Multiple Access

The TDMA component defines a static schedule by which a resource is shared. That is, it periodically assigns access to a resource. For a TDMA component as depicted below, the first p resource units of resource r are assigned to s, the next p units to r, then again p units to s, etc. In effect it splits a resource in two.

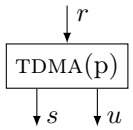


Figure 4.6: TDMA[p]

Definition 23 (TDMA)**.** *The* TDMA *component with period* $p \in \mathbb{N}$ *defines a process with behaviors* (r, s, u) *such that for all* $n \in \mathbb{N}$

$$X_{r/s}(n) = n + p\lfloor n/p \rfloor$$
$$X_{r/u}(n) = n + p\lceil n/p \rceil$$

4.1.6 The Delay Component

The DELAY component is a synchronous buffer that holds items for a number of ticks, imposing a fixed delay. The following diagram represents the component that delays events of the input stream i for d ticks of the reference clock k resulting in the output clock o.

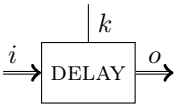


Figure 4.7: DELAY

The synchronous delay consists in a retiming of the counter function χ_i such that its timeline is shifted according to the occurrences of reference clock k with the given delay.

Definition 24 (DELAY)**.** *The* DELAY *component with delay* $d \in \mathbb{N}$ *defines a process with behaviors* (i, o, k) *such that for all* $n \in \mathbb{N}$

$$\chi_o(t) = \chi_i(\eta_k(\chi_k(t) - d))$$

4.2 Abstract Interpretation of Stream Processing Systems

Let us now formulate the abstract behavior and transfer functions of the stream processing elements. The system is to be analyzed in the product domain $\mathcal{C} \times \mathcal{D}$ of clock and drift bound domains introduced in Chapter 3.

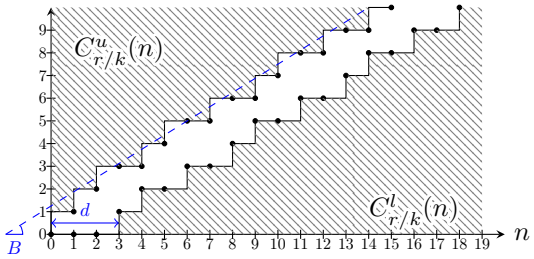


Figure 4.8: The clock bound imposed by a BD component.

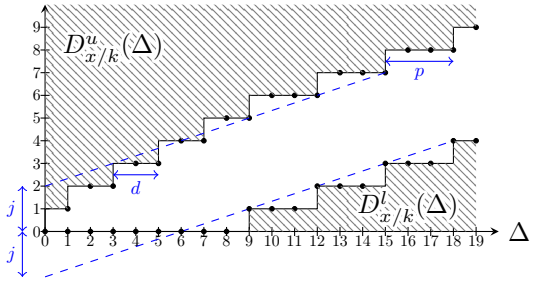


Figure 4.9: The drift bound imposed by a PJD component.

4.2.1 Source Components

Source components are non-deterministic processes defined by their abstraction. They directly formulate bounds on the relative counter functions of the signals in the source component process.

The BD component expresses a clock bound on the relative counter function $X_{r/k}$ of the resource signal r and reference signal k. Figure 4.8 depicts the clock bounds imposed by the BD component on the relative counter function of the resource signal with respect to the reference signal with the parameters $B = 3/4$ and $d = 3$.

As for the BD component, the PJD directly imposes a bound. Except that the PJD component imposes a drift bound. Figure 4.9 shows the drift bounds of a PJD component on the relative counter function of the resource signal with respect to the reference clock with period $p = 3$, jitter $j = 2$ and minimum inter-arrival time $d = 2$.

4.2.2 Greedy Processing Component

The greedy processing component imposes some direct bounds that relate the input stream and resource with the output stream. That is, the behaviors (i, o, r) of a GPC process in isolation (with unconstrained resources and input stream) satisfy the following bounds:

- the number of events in the output stream can never exceed the number

of events in the input stream, i.e., for all $n \in \mathbb{N}$

$$X_{o/i}(n) \leq n \leq X_{i/o}(n)$$

- output is produced when input is available: there must be at least one resource unit for each output event and, conversely, at most one output for each output event, i.e. for all $n \in \mathbb{N}$

$$X_{o/r}(n+1) - X_{o/r}(n) \leq 1 \leq X_{r/o}(n+1) - X_{r/o}(n)$$

Those bounds are, however, not very very precise: they fail to capture the greedyness of the processor resulting in the lack of a lower bound on the output stream with respect to the resource and input stream. Because the GPC is deterministic, however, an abstract system function can be used to obtain better bounds. That is, if bounds on the input or resources are known due to a composition with other components, they can be used to obtain bounds on the output by use of the following lemma.

Lemma 14 (Abstract GPC)**.** *Let $(i, o, r) \in$GPC be a behavior of a greedy processing component such that $(i, o, r) \in \gamma(C, D)$, then*

$$C^l_{i/r} \otimes id \leq X_{o/r} \leq C^u_{i/r} \otimes id$$
$$C^l_{r/i} \wedge id \leq X_{o/i} \leq C^u_{r/i} \wedge id$$

Proof. Let us translate the identity on χ_o to an identity on relative counter function $X_{o/r}$ by substituting real-time parameter t for relative time $\eta_r(n)$. Next, we substitute u by $\eta_r(\Delta)$ which is possible, because the minimum must occur exactly at each occurrence of r. Finally, $X_{r/r}$ is the identity function.

$$\begin{aligned}
&X_{o/r}(n) \\
=&\min\{\chi_i(u) + X_{r/r}(n) - \chi_r(u) \mid 0 \leq u \leq \eta_r(n)\} \\
=&\min\{X_{i/r}(\Delta) + X_{r/r}(n) - X_{r/r}(\Delta) \mid 0 \leq \Delta \leq n\} \\
=&\min\{X_{i/r}(\Delta) + id(n - \Delta) \mid 0 \leq \Delta \leq n\} \\
=&X_{i/r} \otimes id
\end{aligned}$$

The upper and lower bounds follow by substituting X by C^u and C^l respectively. □

The lemma allows us to formulate an abstract system function for the GPC component.

4.2.3 The Synchronous Join Component

Like the GPC component, the synchronous join imposes some bounds directly. Namely, bounds that ensure that the number of inputs (on both streams) can

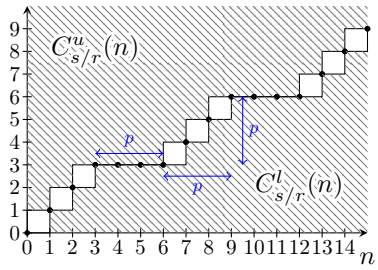


Figure 4.10: The clock bound imposed by a TDMA component.

not exceed the number of outputs. That is, each behavior (x, y, z) in the AND process satisfies the following bounds, for all $n \in \mathbb{N}$:

$$X_{z/x}(n) \leq n \leq X_{x/z}$$

and similar bounds on $X_{y/x}$ and $X_{x/y}$. In addition, tighter bounds can be derived if better bounds on the input streams are known using the property expressed by the following lemma.

Lemma 15 (Abstract AND)**.** *Let $(x, y, z) \in$AND be a behavior of a synchronous join process such that $(x, y, z) \in \gamma(C, D)$, then*

$$C^l_{x/k} \wedge C^l_{y/k} \leq X_{z/k} \leq C^u_{x/k} \wedge C^u_{y/k}$$

Proof. First we derive a relation on relative counter functions by substituting t by $\eta_k(n)$, which yields $X_{z/k} = X_{x/k} \wedge X_{y/k}$. The upper and lower bounds are then obtained by substituting X by its bounds. □

4.2.4 Time-Division Multiple Access

The periodic schedule of a TDMA component can in fact be modelled by clock bounds. Observe that, by Definition 23, the TDMA component consists in a fixed relative counter functions. The clock bounds can enforce such a requirement through equal upper and lower bounds, as is depicted in Figure 4.10. The equal upper and lower bounds force the clock s to occur synchronously with r for three ticks, and then pause for the same duration while u is given access, etc.

4.2.5 The Delay Component

The process of a delay component imposes the following bounds on the behaviors

- the upper and lower clock bounds are shifted by the delay:

$$C^l_{i/r}(n - d) \leq X_{o/r}(n) \leq C^u_{i/r}(n - d)$$

- the drift bound is unaffected by the delay because the same variability still occurs albeit at a later point in time, because of the delay:

$$D^l_{i/r}(\Delta) \leq X_{o/r}(\Delta) \leq D^u_{i/r}(\Delta)$$

4.2.6 Interpreting Bounds

If the drift and clock bounds of a system are known, they can be used to derive the following bounds on system performance:

1. the *backlog* that the system may accumulate while processing streams;
2. the *delay* incurred by streams traversing the system; and
3. the processing *throughput* of the system.

Backlog

Backlog is accumulated when an input stream arrives faster than it can be processed. It is the difference between the number of arrivals at an input stream and the number of departures at the input stream. The backlog between two streams is bounded by the maximal difference of their relative counter function from the diagonal.

Lemma 16 (Backlog bound)**.** *Let (x, y) be a behavior that satisfies the bounds (C, D). The backlog between signals x and y is bounded for all $t \in \mathbb{R}$*

$$\chi_x(t) - \chi_y(t) \leq \max\{n - C^l_{y/x}(n) \mid n \in \mathbb{N}\}$$

Proof. This follows from the observation that the difference $\chi_x(t) - \chi_y(t)$ is greatest when x has just ticked, it therefore suffices to observe the difference at each time instant $\eta_t(n)$ for all $n \in \mathbb{N}$. This results in

$$\begin{aligned}
&\max\{\chi_i(t) - \chi_y(t) \mid t \in \mathbb{R}^\infty\} \\
=&\max\{\chi_i(\eta_i(n)) - \chi_y(\eta_i(n)) \mid n \in \mathbb{R}^\infty\} \\
=&\max\{n - X_{y/i}(n)) \mid n \in \mathbb{R}^\infty\} \\
\leq&\max\{n - C^l_{y/i}(n)) \mid n \in \mathbb{R}^\infty\}
\end{aligned}$$

□

If arrivals are asymptotically faster than the departures, i.e., the system has insufficient capacity to treat the input, the backlog will accumulate indefinitely resulting in an infinite backlog.

Delay

Streams incur delay when they are buffered awaiting processing or while being processed. It is the difference between the arrival time and departure time.

With only the clock and drift bounds, it is impossible to obtain any bound on the real time, because the bounds only capture information on the relative occurrences events However, with another signal as reference clock, a bound on the delay with respect to that clock can be given.

Let (x, y, k) be a behavior that satisfies the bounds (C, D). The delay between signals x and y with respect to reference clock k is bounded for all $n \in \mathbb{N}^\infty$:

$$X_{k/c}(n) - X_{k/y}(n) \leq \sup\{C^u_{k/x}(n) - C^l_{k/y}(n) \mid n \in \mathbb{N}^\infty\}$$

The delay may become infinite if arrivals come faster than the system can process them. Note that high backlog does not automatically imply a high delay. For example, backlog may be large but delay small if the system has a large, constant capacity and input arrives slowly but with large bursts.

Asymptotic Rate

The throughput of a system is determined by the rate at which it produces output over the system's lifetime. That is, the asymptotic rate of the output. As for the delay bound, the asymptotic rate is defined with respect to a reference clock.

Let (x, k) be a behavior that satisfies the bounds $(\mathcal{C}, \mathcal{D})$ then the asymptotic rate is bounded by the limit

$$\lim_{n\to\infty} n/C^u_{x/k}(n)$$

4.3 Experiments

We have implemented the mappings and resulting fix-point algorithm in a prototype programmed in Python. At the core is a suite of operators on functions as documented in Appendix A and the theory developed in Chapter 3. More precisely, operators on eventually periodic functions that are represented by a finite transient part (the aperiodic introduction) and a periodic part with a finite period. The program consists of a few hundred lines of code, of which more than half is concerned with the operators.

In order to compare to conventional network calculus' analysis both systems were modelled with an additional clock that represented real-time. So all bounds on input streams and availability of resources were modelled with respect to the added real-time clock k.

4.3.1 Multimedia Decoder

The multimedia decoder of Figure 4.1 is described and analysed by the following code. The system consists of a list of mappings over bound matrices that correspond to the predicates introduced for the components in Section 4.1.

```
[k,i,x,y,o,r,s] = range(7) # identifiers

# declare system components
system = [
    pjd(i, 4, 24, 1, k),
    bd(r, 0.3, 3, k),
    bd(s, 0.3, 3, k),
```

```
    gpc(i, x, r),
    gpc(i, y, s),
    and_join(x, y, o),
]

bounds = solve(system) # fixpoint calculation

print backlog_bound(bounds, x, o)
```

Not visible in the above snippet, are the mappings described in Chapter 3, that handle the interaction between the bounds; the interaction between lower on upper bounds, transitivity, and the interaction of drift bounds and clock bound are applied by the solver implicitly because they are independent of the system's components.

The multimedia decoder is modelled with an input stream generated by a PJD component with period of 4, jitter of 24 and a minimum inter-arrival distance of 1. The two decoders had resources, generated by the BD components, with bandwidth of 0.3 units per time-unit and a drift of 3.

We calculate the buffer size using conventional network calculus using the curves for input stream i and resources r, s defined by the PJD and BD components with respect to the real-time clock k, e.g., for stream i the upper arrival input curve is $\alpha_i^u = D_{i/k}^u$. Then we derive a bound $\max\{\alpha_x^u(n) - \alpha_y^l(n) \mid n \in \mathbb{N}\}$ for the buffer size of stream x, where $\alpha_x^u = (\alpha_i^u \otimes \beta_r^u) \oslash \beta_r^l \wedge \beta_r^u$ and $\alpha_y^l = (\alpha_i^l \otimes \beta_s^l)$.

In this particular configuration our analysis calculated, in twelve iterations, an upper bound of 8 for the backlog of streams x and y, whereas the conventional method yields a bound of 18. We also confirmed that increasing the jitter of the input stream i has little effect on the estimated buffer size, while the conventional method's estimate grows quickly with jitter.

4.3.2 Shared Bus

A second example, depicted in Figure 4.11, shows a server that is connected to the outside world by a bus. The server responds to update requests that arrive over a bus and returns a reply over the very same bus. Conflict-free access to the bus from both sides is achieved through a time-division, multiple-access (TDMA) policy. That is, exclusive access to the bus is given periodically to the requester and the server. The server has a certain delay before it responds. If this delay corresponds to the TDMA period, then the bus does not introduce any additional delay since the reply to a received request can immediately be transmitted.

Two new components are used here: the TDMA component that splits a resources according to a fixed schedule, and the DEL component that imposes a fixed delay on a stream. In this example, the GPC components do not model processors or decoders, but the transmission of messages over the bus.

Experiments on the second example, shown below, where a server responds to requests over a bus, showed that the analysis effectively handles the regular nature of requests received over the TDMA bus.

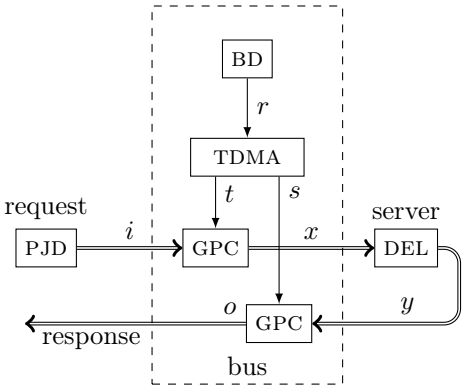


Figure 4.11: The stream i of update requests passes a bus, represented by resource r, to arrive at a server, which, after a fixed delay (DEL), sends a reply y back over the bus. The resource is shared with a fixed TDMA schedule, over the resources t and s.

```
[k,i,x,y,o,r,t,s] = range(8) # identifiers

# declare system components
system = [
    pjd(i, 9, 23, 1, r),
    bd(r, 0.33, 13, k),
    tdma(r, t, u, 9),
    gpc(i, x, t),
    delay(9, x, y, r),
    gpc(y, o, u),
]

bounds = solve(system) # fixpoint calculation

print backlog_bound(bounds, y, o)
```

Conventional analysis has no straightforward way to model this second example. The closest analysis we could make, again taking the drift bounds for input stream i and resource r, uses $\beta_t^u = \beta_s^u = \beta_r^u \otimes tdma^u \otimes id$ where $tdma^u(n) = 9\lceil n/16 \rceil$ and $\beta_t^l = \beta_s^l = \beta_r^l \otimes tdma^l \otimes id$ where $tdma^l(n) = 9\lfloor (n+7)/16 \rfloor$ for the upper (resp. lower) bounds for the shared bus resources. The delay is modeled according to [LT01] such that $\alpha_y^u = \alpha_x^u \otimes delay$ where $delay(n) = 0$ if $n \leq 9$ else ∞ and $\alpha_x^u = (\alpha_i^u \otimes \beta_t^u) \oslash \beta_t^l \wedge \beta_t^u$. Then we obtain a buffer size bound $\max\{\alpha_y^u(n) - \beta_s^l(n) \mid n \in \mathbb{N}\}$.

With a TDMA period of 9 and matching delay our analysis consistently showed, after 14 iterations, a backlog bound of 2 between streams y and o, whereas the backlog bound calculated by the conventional method is 7. Here the improved precision of our approach is due to the combined use of clock and drift bounds.

Clearly conventional analysis has a lower complexity because we model n

streams with n^2 bounds, whereas conventional analysis needs only n bounds. There is room for improvement because the effect of most bounds is local, there is no need to track all n^2 relations; the implementation could exploit system topology to reduce the number of bounds involved in the calculation. It should also be noted that conventional analysis does not need a fix-point computation if there are no cyclic dependencies, as is the case for the examples.

Experience suggests that the computational cost is most affected by the choice of curves. Combining curves with many different prime factors can lead to very long periods, which affects all further operations on the curve. Conventional analysis also suffers in this case.

4.4 Related Work

Existing formal verification techniques for real-time systems are based on model-checking of timed automata [HV06], algebraic techniques like Network calculus [LT01] and real-time calculus [CKT03], scheduling theory [HKO+93], or combinations thereof [HHJ+05].

Data-flow networks are represented naturally through relational constraints. The first such a notion is synchronic distance [Pet76] for Petri nets. The synchronic distance is the maximum number of times a transition may fire before another must be fired. If the Petri net models a data-flow system, typically modelled by conflict-free nets, the synchronous distance of a producer from a consumer (both modelled by transitions) corresponds to the maximum buffer occupation between them.

The affine clock calculus used in the validation and compilation of real-time applications [SGL99] programmed with a combination of SIGNAL and ALPHA, The calculus relates two clocks through a base clock. Typically, the arrival of complex tasks is related to their completions through a system clock that determines execution speed. These relations are expressed by affine transformations over the occurrence-times of events. The calculus then serves to determine synchronizability of two event streams.

The more recent work on n-synchrony [CDE+06] develops a similar concept to verify synchronizability of programs written in a synchronous data-flow language extended with statically scheduled sample (a periodic selection of elements in a flow) and merge (a combination of two flows defined by a static schedule) operations and buffers with FIFO semantics. The verification is based on a type-system and determines whether the system can be executed with finite buffer sizes. The clock envelopes introduced in [CMPP08] further develop this system, using clock abstractions called envelopes that permit more efficient verification at the cost of some over-estimated buffer sizes.

The clock bounds, introduced in this paper, express the same essential relations as the data-flow relations of the mentioned work. The formalization of the relations as bounds on relative counter functions however, is new. And it is precisely this formalization that allows us to combine clock bounds with drift bounds. Clock bounds are more general than the synchronic distance on conflict-

free nets, affine relations and clock envelopes, all of which can be represented by linear clock bounds in our model. Clock bounds are equal in expressiveness to the basic model of n-synchrony.

Drift bounds are essentially a relational variant of network calculus' arrival and resource curves. The problem of correlating variability of event streams, such as coincident bursts of arrival streams, has been studied before in various incarnations of network calculus.

In [RE08, PRT$^+$10] the case where streams are transported over a network as a single, joined stream, to be separated at arrival, is treated by tracking the correlations between sub-streams and the aggregated stream. The event count curves of [PRT$^+$10] are a special case of our drift bounds: a bound on a sub-stream with respect to the aggregate stream.

Correlated streams are also used in [HT07b] for the analysis of a fork-join scenario, where the load of a single event stream is split over several processing components and then merged in a scenario similar to the multimedia decoder of Figure 4.1. However, [HT07b] distribute frames over the processors for load distribution, like splitting traffic over two lanes and then merging it again, rather than the synchronous split join of our example.

Finally [WT05b] exploits correlations between the workload imposed by a task traversing a chain, which occur e.g. for tasks with variable payloads sizes that traverse a chain of processors: tasks that require many resources on the first processor because of a large payload, will also require more work for the next and vice versa. None of the above approaches uses a general relational model as presented in this paper.

Chapter 5

Sampling Networks

Sampling networks are comparable to the synchronous data-flow language Lustre. A sampling network consists of a network of processing nodes that operate on flows. Flows are sequences of values over time. A node defines the values of output flows based on the values of the input flows and the internal state of the node at that instant. Informally, one may think of a node as an actor that reads its inputs, performs its calculations, and writes its output at each activation. Nodes are activated by a clock, itself a boolean flow.

Consider the example depicted in Figure 5.1 that implements a distributed watchdog to detect system failure. It consists of two independently clocked partitions (of nodes) that communicate an alternating bit by sampling shared memory cells. System failure is detected if the alternating bit remains constant for too long.

The example consists of logical nodes depicted as combinational logical circuit nodes, the synchronous delay `pre` that delays a signal by one time-step (as per its clock), and the circular nodes are memory places used to communicate between the nodes in the domains of clocks `c` and `d`. The `alive` component is a composition of a `pre` and exclusive or that compares the last two values.

The following describes the sequence of values for each flow during an execution of the watchdog system, where *tt* and *ff* denote the boolean values true and false respectively and $\perp$ denotes absence of a value. The execution is shown over discrete time steps with instants $t = 1, 2, ..., 7$.

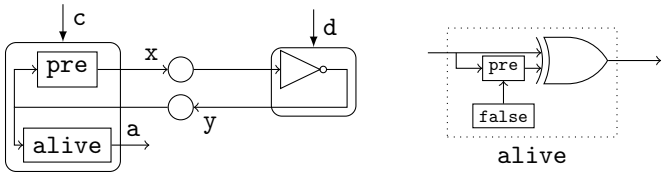


Figure 5.1: A distributed watchdog protocol to detect system failure.

t	1	2	3	4	5	6	7
c	*tt*	*tt*	*ff*	$\perp$	*tt*	*tt*	*tt*
d	*tt*	$\perp$	*tt*	$\perp$	*tt*	*ff*	$\perp$
x	*ff*	*tt*	$\perp$	$\perp$	*tt*	*ff*	*ff*
y	*tt*	$\perp$	*ff*	$\perp$	*ff*	$\perp$	$\perp$
a	*tt*	*tt*	$\perp$	$\perp$	*ff*	*tt*	*ff*

The values of flows `x` `y` and `a` are absent when their clocks are either absent, or false. Sampling of is done at the instants when the clock is present. When the reader and writer of a shared memory cell are active simultaneously the memory is written before it is read. Consequently, the false value of flow `y` at $t = 3$ is never sampled by the domain of clock `c` because at $t = 5$ (the first activation of c after $t = 3$) a new value has become available simultaneously. Moreover, the value of `x` is `true` at $t = 5$ because the synchronous buffer (`pre`) in domain `c` contains the value sampled at the previous activation of `c` (at $t = 2$).

In addition to the graphical representation of sampling networks, programs are defined textually with the syntax of Figure 5.2. The watchdog protocol, for example, is described as follows

```
alive s = s xor (false -> pre s)

partA y =
    let x = false -> pre y;
    in (x, alive y);

partB x = not x

system (c,d) =
    let (x, a) = partA (sample y on c);
        y = partB (sample x on d);
    in a
```

The main difference with the synchronous programming language Lustre [HCRP91] is the addition of a primitive for communication-by-sampling between differently clocked components without a common root clock.

5.1 Elements of a Sampling Network

Let us review the elements that make up a sampling network. Expressions are the basic syntactic element of a sampling network that, like nodes in the graph, define a flow based on input flows (variables). Constants and variables are mostly self-explanatory: constants create a sequence of the same value, and variables refer to input flows defined elsewhere.

Tuples combine multiple flows in a group. Note that there are only tuples of flows, rather than flows of tuples: the flows in a tuple need not be present at the same time.

$d ::=$	$\mathrm{f}\, p = e$	component definition
$\mid$	$d;\ d$	definition sequence
$p ::=$	$x \mid p\ \texttt{on}\ x \mid (p, ..., p)$	pattern
$e ::=$	κ	constant flow
$\mid$	x	variable
$\mid$	$(e, ..., e)$	tuple
$\mid$	$\texttt{let}\, p = e;\ p = e, ...;\ \texttt{in}\, e$	local definitions
$\mid$	$e\ op\ e$	buit-in opererator
$\mid$	$e\ \texttt{if}\ e\ \texttt{else}\ e$	mux operator
$\mid$	$\mathrm{f}\, e$	component instantiation
$\mid$	$e\ \texttt{on}\ x$	clocked expression
$\mid$	$\texttt{pre}\, e$	synchronous delay
$\mid$	$e \rightarrow e$	initialization
$\mid$	$e\ \texttt{when}\ e$	down-sampling
$\mid$	$\texttt{current}\, e$	up-sampling
$\mid$	$\texttt{sample}\, e$	asynchronous sampling
x, f		identifiers

Figure 5.2: Full syntax of a sampling network.

5.1.1 Primitive Operators

An operator expression applies a primitive functional operator to the consecutive values of its input flows. For example the following program depicts a single node with input and output flows.

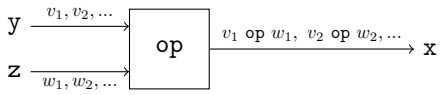


Figure 5.3: `x = y op z`

Primitive operators include the boolean conjunct, disjunct and negation as well as numerical operators such as addition and multiplication.

5.1.2 Synchronous Delay (`pre`)

The synchronous delay delays the input flow until the next tick of the clock. Or, said differently, the synchronous delay gives the previous value of the input flow. The output of a synchronous delay is undefined at the first time instant, because there is no value for the "previous" time instant. Figure 5.4 shows a delayed flow.

5.1.3 Initialization ($\rightarrow$)

The initialization expression defines a value for the first instant. It is typically used to define the otherwise undefined value of a delayed stream with a constant at the right-hand-side, e.g., `true -> pre x`. The following example shows the use of a pre-initialized synchronous delay.

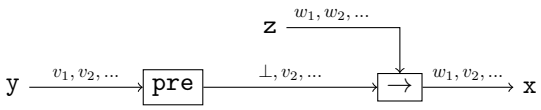


Figure 5.4: `x = y -> (pre z)`

5.1.4 Clocks and their Domains (`on`)

Each flow in a program is associated with a clock that indicates when the values in a flow occur. The clock `c` of a signal `x` is a boolean flow that has the value true exactly when `x` has a value.

The *domain* of a clock consists of all flows with a particular clock. In the graphical notation, clock domains are depicted as rectangles with rounded corners connected with a fat arrowhead to the endpoint of the clock. The origin of a flows (arrows) associates the flow to the clock. For example, the domain of

the clock `c` in the distributed watchdog in Figure 5.1, consists of flows `a`, `x` and `sample y`.

In the textual notation, flows are either explicitly associated to a clock using the `on` expression or pattern, or implicitly when the clock can be inferred from sub-expressions and the context. For example, the expression `x + y` defines a flow that is implicitly clocked by the same clock of flows `x` and `y`. Explicit annotation is typically necessary for sampling expressions. For example, the basic expression `sample x` has ambiguous semantics if it is not explicitly associated to a clock (e.g. `sample x on c`). The expression `x + sample y`, however, may be associated implicitly to the clock of `x`.

It is possible to write `x on c on d` for a flow `x` clocked by flow `c` which, in turn, is clocked by flow `d`. The flows of a tuple are all clocked individually, e.g., `(x on c, y on d)`. Clock annotations distribute over tuples, i.e., writing `(x, y) on c` is equivalent to `(x on c, y on c)`.

Flows cannot have arbitrary clocks, so explicit clock annotations must respect certain rules. For example, the expression `x on c + y on d` is not permitted because the added flows may not be present simultaneously at all times. Section 5.2.2 formalizes the constraints on clocks through the notion of well-clocked programs.

5.1.5 Sampling (`sample`, `when`, `current`)

The `sample` expression realizes communication between two different clock domains through the sampling of memory. At each tick of the clock of the input flow the value is written to a one-place buffer. At each tick of the output clock, the output value is the current value in the buffer. If there is no tick of the output clock for two subsequent inputs the buffered value is simply overwritten. Two ticks of the output clock without an interceding tick of the input will yield two subsequent copies of the buffered value. If both input and output are simultaneous writing the buffer precedes reading it. Consequently sampling a flow on its own clock (`x = sample y` where `x on c` and `y on c`) implies equality of flows `x` and `y`. The expression `sample (y on c) on d`, illustrated in Figure 5.5, creates a flow on clock `d` that consist of the most recent value of the flow `y`.

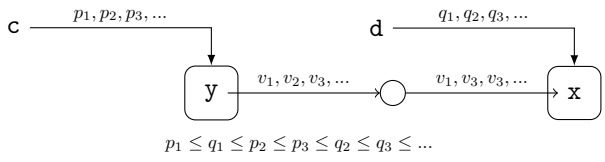
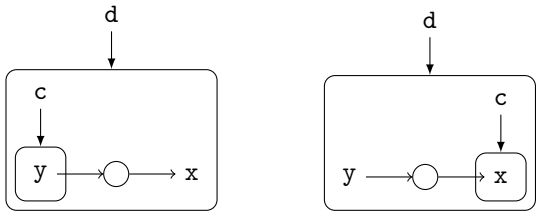


Figure 5.5: `x = sample (y on c) on d`

It is an asynchronous construct, because the two clock domains do not need to be synchronized in any way. As we will now show, the synchronous sampling primitives `when` and `current` of LUSTRE are special cases of our `sampling` primitive where the clocks are related hierarchically.



(a) `x = current (y on c) on d` (b) `x = (y on d) when c`

Figure 5.6: Synchronous sampling as a special case of asynchronous sampling.

The expression `current x` over-samples the flow `x` depending on the clock of the resulting flow. That is, given the clocked expression `current (y on c) on d`, the resulting flow interpolates the flow `y` by repeating the last value of `y`, if it is absent. As shown in Figure 5.6(a) it is defined by the following equivalence (by equality of the defined flows):

$$\texttt{current (y on c) on d} \;=\; \texttt{sample (y on c) on d} \quad \text{if } \texttt{c on d}$$

The expression `x when y` selects the values in flow `x` when the value in `y` is true. This constitutes down-sampling resulting in a new flow with clock `y`. More precisely (as depicted in Figure 5.6(b):

$$\texttt{(y on d) when c} \;=\; \texttt{sample (y on d) on c} \quad \text{if } \texttt{c on d}$$

The two synchronous sampling primitives can also be combined in a construction that is almost equivalent to the `sample` expression:

$$\texttt{sample (y on c) on d} \;=\; \texttt{current (y on c) when d}$$

Almost, because their combination requires the existence of a clock `e` such that `c on e` and `d on e`. In Figures 5.6(a) and 5.6(b), the hierarchical relation of the clocks is illustrated by nesting the clock domains of `c` and `d`.

The sampling primitive also simplifies synchronous composition of systems, i.e., within the same hierarchy, because it allows the communication between arbitrary levels within the hierarchy. Up- and down-sampling, on the other hand, only allows communication between the domain of a clock and the domain of its direct super- or sub-clock.

Consider Figure 5.7 that depicts a synchronous composition of the components `A` and `B`. Both have the same root-clock `r` but the components' clocks are different: `A` runs on `d`, itself a subclock of `c`, and `B` runs on `e`. With the sampling primitive, the composition is simple: `B (sample (A on d) on e)`. The relation between the clocks `d` and `e` is irrelevant to the composition itself. The same composition with up- and down-sampling, however, depends on the relation between the clocks in the hierarchy: `B ((current (current (A on d) on c) on r) when e)`. The `current` primitive up-samples a flow to the clock

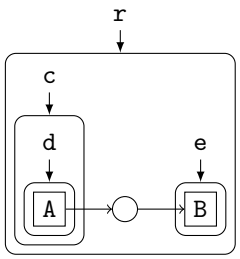


Figure 5.7: Communication between different clock domains within the same hierarchy with a single sampling primitive.

that is one level higher in the hierarchy and the `when` primitive down-samples a flow to a clock that is one level higher. If a change in the modules that define the clocks introduces or removes a level in the hierarchy, the composition must be adapted accordingly.

Although the sampling primitive subsumes the functionality of both the up- and down-sampling primitives, we maintain the latter two to facilitate the explicit construction of synchronous systems. The up- and down-sampling primitives often obviate the need for explicit clock annotations with `on`.

5.1.6 Local Definitions and Cycles (`let...in`)

The `let` expression binds variables to local definitions and hides the locally defined flows. Although it may also facilitate reuse and improve legibility (hiding prevents the cluttering of the name-space and enables variable shadowing) it is a necessary component to define cyclical circuits in the textual format. For example, a simple oscillator that has a single output that alternates between true and false and no input is defined as follows.

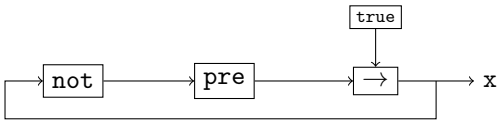


Figure 5.8: `let x = true -> pre (not x) in x`

5.1.7 Components

At the top-level a program consists in a sequence of component definitions. A component definition $fp = e$ consists in a composition of variables (a pattern) that declare names for input flows and an expression that defines the output flows.

Components defined at the top-level, are instantiated by inserting the expression e while substituting (taking care not to cause name-conflicts) the variables

in pattern p with the expressions e' in the instantiation fe'. Circular dependencies between components are not allowed, because they would result in an infinite expansion.

5.2 A Core Language

For the formal treatment of sampling networks we define a small core of expressions. For simplicity it is limited to boolean values and all expressions are clocked explicitly. That is, the values of flows must be in $\mathbb{B}$ and each flow expression $e\,\texttt{on}\,c$ is associated with a flow variable c that is its clock flow. Moreover, the structure of a program is completely flat: the program is reduced to collection of atomic equations without let-expressions. Each declaration binds a variable to a single primitive that operates only on variables, so expressions cannot be nested. The flattened program is very close to the graphical representation: each declaration corresponds to a node and each variable to an edge in the graph. The core language is as expressive as the full language but less convenient to write.

Definition 25 (Core sampling network)**.** *A core sampling network language consists in a set of equations of the form $x = e$ where x denotes a variable name and e an expression of the following syntax:*

$$
\begin{array}{rll}
e ::= & v\ \texttt{on}\ x & \textit{constant value} \\
\mid & x\ \texttt{or}\ y\ \texttt{on}\ z & \textit{disjunction} \\
\mid & x\ \texttt{and}\ y\ \texttt{on}\ z & \textit{conjunction} \\
\mid & \texttt{not}\ y\ \texttt{on}\ x & \textit{negation} \\
\mid & \texttt{pre}\ x\ \texttt{on}\ x & \textit{synchronous delay} \\
\mid & \texttt{sample}(x\ \texttt{on}\ x)\ \texttt{on}\ x & \textit{asynchronous sampling}
\end{array}
$$

where x, y, z are flow identifiers. Let $\mathcal{V}$ denote the set of all flow identifiers (variables) used in a program.

Not all syntactically correct programs are valid. Following [HCRP91] which defines the similar constraints on Lustre programs, sampling networks must satisfy the following criteria:

- A program may not redefine the same variable. That is, each equation in a program $\{x_1 = e_1; ...;\ x_n = e_n\}$ must bind different variables ($i \neq j \implies x_i \neq x_j$).
- Programs must be *acyclic* unless cycles are interrupted by a synchronous delay. That is, programs can not contain circular dependencies between flows, unless the cycle contains a synchronous delay. For example, the expression `let x = x and y; in x` is invalid because x depends on itself. The expression `let x = (pre x) and y; in x`, however, is valid because the loop is broken by a delay.

- Programs must be *well-clocked*. The arguments of primitive operators (negation, conjunction, and disjunction), delay respectively must have the same clock. For example, the expression `(x on c) and (y on d)` is illegal unless $c = d$ because the semantics are not well defined if either x or y is absent.

The following subsections formalize and clarify the last two constraints further.

5.2.1 Causality and Cycles

Programs are required to be acyclic because cyclical definitions of flows may be non-deterministic and contain deadlocks.

Consider again the cyclic expression `let x = x and y; in x` and note that there are flows for `x` and `y` that satisfy the equality: e.g. a pair of flows where `x` is always false and a pair where both flows are equal. Such a program is not deterministic; the input does not uniquely determine the output.

Similarly, we want to avoid cyclical clock definitions, such as in the simple program in Figure 5.9. Even though there is a single solution, the timing of the flows is non-deterministic.

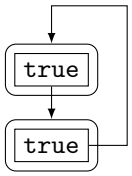


Figure 5.9: `let x = true on y; y = true on x; in x`

Additionally cyclic definitions may define deadlocked programs if there is no solution to the given equations. For example, the simple cyclic expression `let x = not x; in x` has no solutions since `x` cannot be true and false at the same time. While in this example the contradiction is easy to spot, finding contradictions in general expressions is a boolean satisfiability problem.

To avoid cyclical programs we require flows to be ordered by a precedence relation `pre` that can be interpreted loosely as the (partial) order in which values can be calculated within one time step. For example, in the expression `let x = y or z on c; in x`, each value of the flow `x` can be computed after the values of `y`, `z`, and `c` have become available/been calculated.

Definition 26 (Acyclic program ($\prec$))**.** *A program is acyclic if there exists a strict partial order $\prec \in \mathcal{V} \times \mathcal{V}$ over the variables in a program $x_1 = e_1$; ...; $x_n =$*

e_n; *such that, for all equations* $x_i = e_i$

$$\begin{aligned} x = v \texttt{ on } c &\implies c \prec x \\ x = y \texttt{ or } z \texttt{ on } c &\implies y, z, c \prec x \\ x = y \texttt{ and } z \texttt{ on } c &\implies y, z, c \prec x \\ x = \texttt{not } y \texttt{ on } c &\implies y, c \prec x \\ x = \texttt{pre } y \texttt{ on } c &\implies c \prec x \\ x = \texttt{sample}(y \texttt{ on } d) \texttt{ on } c &\implies y, d, c \prec x \end{aligned}$$

It is important to note that only the synchronous delay can break a cycle and not the sampling primitive, even though sampling also involves memory. The following distributed oscillator, for example, is still cyclic even though the system has a reasonable semantics if the clocks `c` and `d` do not occur simultaneously.

```
distr_osc (c,d) =
    let x = not (sample y) on c;
        y = sample x on d;
    in y
```

The requirement that programs be acyclic is a trade-off between expressiveness one one side and complexity on the other. There are useful programs with well-defined (and deterministic) semantics with loops. For example, a flip-flop could be defined as follows if cycles were permitted:

```
flipflop (r,s) =
    let q = not (r xor nq)
        nq = not (s xor q)
    in q
```

However, cyclic programs are easy to detect and correct (made acyclic) and can be compiled to efficient sequential code [HRR91]. The general problem of causality in the related problem of cyclic combinational circuits was shown to be NP-Hard by Malik [Mal93].

5.2.2 Well-Clocked Programs

Core expressions are clocked explicitly such that each flow (identified by its variable) is associated with a clock. The association between flows and their clocks has to be consistent with our expectations of the semantics: the operands in the expression `x or y` are expected to be available at the same time. That is, `x` and `y` must be associated with clocks that are present and true at the same time instants. To make sure the clocks are consistent we require a function that associates each flow with a clock such that the appropriate streams have matching clocks.

Additionally, the presence of the flow `x` defined by a program `let ... in x` must be determined by an input flow rather than a locally defined flow.

Definition 27 (Well-clocked program ($\hat{\cdot}$))**.** *A program is well-clocked if there exists a function* $\hat{\cdot} \in \mathcal{V} \to \mathcal{V}$ *from variable to variable in a program* $x_1 = e_1;\ ...;\ x_n = e_n$*; such that for all equations* $x_i = e_i$*,*

$$\begin{aligned}
x_i = v \texttt{ on } c &\implies \hat{x_i} = c \\
x_i = y \texttt{ or } z \texttt{ on } c &\implies \hat{x_i} = \hat{y} = \hat{z} = c \\
x_i = y \texttt{ and } z \texttt{ on } c &\implies \hat{x_i} = \hat{y} = \hat{z} = c \\
x_i = \texttt{not } y \texttt{ on } c &\implies \hat{x_i} = \hat{y} = c \\
x_i = \texttt{pre } y \texttt{ on } c &\implies \hat{x_i} = \hat{y} = c \\
x_i = \texttt{sample}(y \texttt{ on } d) \texttt{ on } c &\implies \hat{x_i} = c, \hat{y} = d
\end{aligned}$$

The above definition excludes some programs that might in fact be considered valid because clocks must be syntactically equal. Especially in a larger system, it is conceivable that two flows are clocked by different flows that are true at exactly the same instants and therefore do have the same clock with different names. The more precise alternative would be to require the clock flows to be equal rather than the same. Verifying the equality of two flows unfortunately is undecidable [CPHP87] and we therefore make the same choice as the designers of LUSTRE to require only syntactic equality of clocks.

The clock relation is functional: each variable is related to a clock, including variables that serve as clocks themselves. For example, the `when` expression depicted in Figure 5.6(b) defines a signal x such that $\hat{x} = c$ and, in turn, $\hat{c} = d$. Because the precedence relation ($\prec$) establishes a strict partial ordering for signals with respect to their clocks, the transitive closure of the clock function ($\hat{\cdot}$) defines set of *clock hierarchies* (or *forest* of clocks). The *root clocks* of the hierarchies are their own clock ($\hat{x} = x$). Root clocks are always input clocks as they cannot be defined without a cycle.

5.3 Semantics of a Sampling Network

The semantics of sampling networks are defined by a labelled state-transition system which states consist of the values stored in the sampling and delay components and transitions are labelled with the set of flow variables that are present and true at the time of the transition.

Labelled transition systems consist of a set of states and a set of directed, labelled transitions. It describes behavior as a sequence of transitions through the states.

Definition 28 (Labelled transition system)**.** *Let* $\langle S, \Sigma, S_0, \rightarrow \rangle$ *denote a labelled transition system with states* S*, alphabet* Σ*, initial states* $S_0 \subseteq S$*, and transitions* $\rightarrow \in S \times \Sigma \times S$*. The notation* $s \xrightarrow{\sigma} s'$ *is equivalent to* $(s, \sigma, s') \in \rightarrow$*.*

We only consider only a special case of labelled transition systems where all labels are sets of events that occur at the time of a transition. That is, we use transition systems $\langle S, \wp(\mathcal{V}), S_0, \rightarrow \rangle$ where $\mathcal{V}$ is a finite set of events.

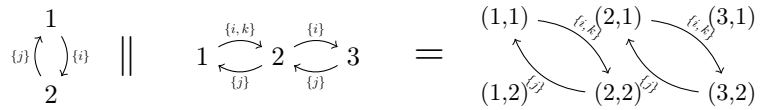


Figure 5.10: The composition of two labelled transition systems.

Composition of labelled transition systems is achieved by extending the composition operator $\|$, defined for processes in Section 2.2, to labelled transition systems. It composes two transition systems with named events synchronizing only on events in both events. Figure 5.10 gives an example composition.

Definition 29 (Composition ($\|$))**.** *The composition $A \parallel B$ of two transition systems is defined as $((s_A^0, s_B^0), S_A \times S_B, \Sigma_A \cup \Sigma_B, \rightarrow)$ where*

$$(s_a, s_b) \xrightarrow{\sigma_a \cup \sigma_b} (s'_a, s'_b) \iff s_a \xrightarrow{\sigma_a}_A s'_a,\ s_b \xrightarrow{\sigma_b}_B s'_b,\ \sigma_a \cap \Sigma_B = \sigma_b \cap \Sigma_A$$
$$(s_a, s_b) \xrightarrow{\sigma_a} (s'_a, s_b) \iff s_a \xrightarrow{\sigma_a}_A s'_a,\ s_b \in S_b,\ \sigma_a \cap \Sigma_B = \emptyset$$
$$(s_a, s_b) \xrightarrow{\sigma_b} (s_a, s'_b) \iff s_a \in S_a,\ s_b \xrightarrow{\sigma_b}_B s'_b,\ \emptyset = \sigma_b \cap \Sigma_A$$

The semantics of a sampling network are described as a composition of transition systems that define its elementary components. Each transition represents a global, discrete, time-step in which a signal is present. The hierarchical nature of flows implies there is at least one root clock present in each transition. A flow x is present during a transition $s \xrightarrow{\sigma} s'$, that is it has a value, if $\hat{x} \in \sigma$ and it is true if $x \in \sigma$.

Definition 30 (Transition system of a sampling network ($[\![x = e]\!]$))**.** *The labelled transition system of a sampling network with equations $\{x_1 = e_1, ..., x_n = e_n\}$ input flow variables I as the composition*

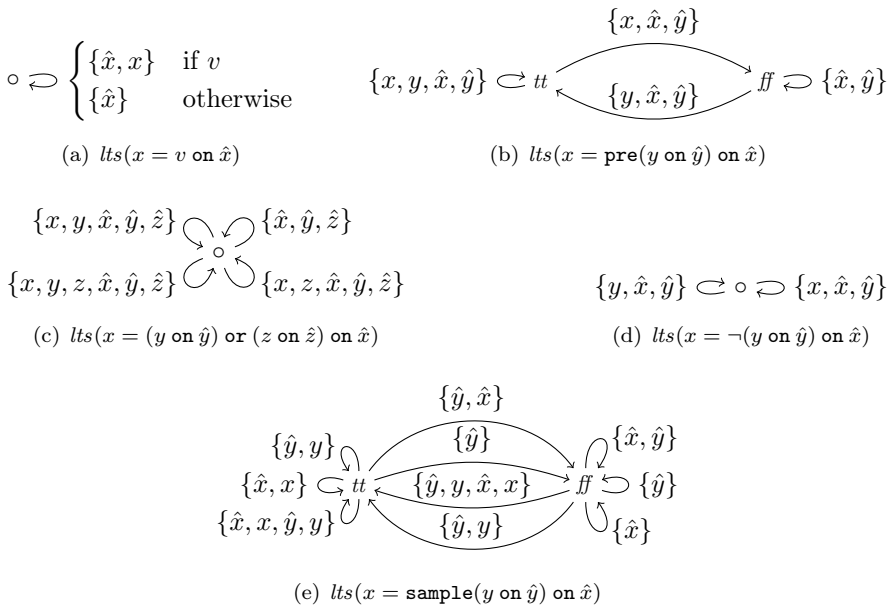
$$[\![x_i = e_i]\!] \parallel ... \parallel [\![x_n = e_n]\!]$$

where the labelled transition systems $[\![x = e]\!]$ for individual equations are defined in Figure 5.11.

These semantics only apply to non-cyclic and well-clocked programs. For example the transition system of the cyclic program `let x = not x on c in x` would consists of a single state and a single transition where `x` and its clock `c` are present at every transition. The transition system of the program `let c = false -> not (pre c on d); x = sample y on c; in x` would have transitions where `x` is present without its clock `c`.

5.4 Related Work

Sampling networks consists, for the most part, in synchronous data-flow primitives used in Lustre with some syntactical differences. The main difference



(a) $lts(x = v \texttt{ on } \hat{x})$

(b) $lts(x = \texttt{pre}(y \texttt{ on } \hat{y}) \texttt{ on } \hat{x})$

(c) $lts(x = (y \texttt{ on } \hat{y}) \texttt{ or } (z \texttt{ on } \hat{z}) \texttt{ on } \hat{x})$

(d) $lts(x = \neg(y \texttt{ on } \hat{y}) \texttt{ on } \hat{x})$

(e) $lts(x = \texttt{sample}(y \texttt{ on } \hat{y}) \texttt{ on } \hat{x})$

Figure 5.11: Labelled transition systems for sampling network components.

is the sampling primitive: while LUSTRE offers primitives for up- and down-sampling with `current` and `when` it requires the construction of a single clock hierarchy as was shown in Section 5.1.5. In [CMSW99] a quasi-synchronous system with communication-by-sampling is designed in Scade, a commercial implementation of LUSTRE, using a combination of up- and down-sampling to cross from one clock domain to another.

The data-flow language SIGNAL [LB87, AGA+95] enables the construction of polychronous systems where clocks are related in a forest of clocks rather than a clock hierarchy (tree of clocks). Such forests are created by locally joining flows with the `default` primitive that merges two differently clocked flows, preferring the left-hand-side if both are present at the same time. The default statement can be used to define communication by sampling in a similar fashion: `x = y default (pre x) when c`[1] and a clock constraint `x ^= ` defines the same flow as `x = sample y`. As depicted in Figure 5.12 it creates an additional clock `k` that is the union of clocks `c` and `e`.

[1] With adapted syntax; in signal we would write `x := y default x$1`

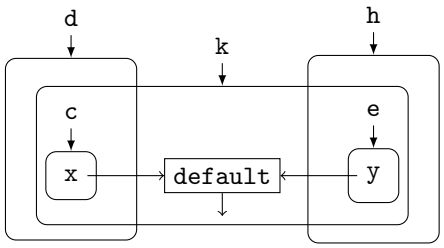


Figure 5.12: The default primitive of signal, as adapted to our graphic representation, joins two flows creating a new clock-domain.

Chapter 6

Analysis of Sampling Networks

Sampling networks permit us to write programs that can be executed synchronously, when all root-clocks are synchronized, asynchronously, when the clock ticks interleave arbitrarily, or quasi-synchronously, when the possible interleavings are bounded.

Drift bounds allow us to describe clocks ranging from total synchrony and complete asynchrony by limiting the maximum and minimum number of ticks of one clock, for every interval of ticks of another clock.

In this chapter, we explore the semantics of sampling networks where the root clocks and input signals are described by to drift bounds. First, we will verify safety properties of such a system by performing a reachability analysis of the state-transition system. Then, we will show how to extract the drift bounds of the output flows of such a system through analysis of the labelled transition system that defines the system. Finally, we show how we can extract drift bounds of the behavior of such a system without constructing the actual state space.

6.1 Verification of a Sampling Network

A safety property is a requirement that something bad, as defined by the property, never happens. The verification of safety properties is important for safety-critical embedded systems where failure could be catastrophic, such as a nuclear power plant. In this section we show how to verify a safety-property of a sampling network in an environment described by drift bounds.

We express safety properties as a reachability problem in of the labelled state-transition system of a sampling network composed with a labelled transition system that models all behaviors allowed by the drift bounds on the root clocks and inputs. The safety property then distinguishes bad states, where safety property has been violated, from good states. Verification of such a safety

property is achieved through a reachability analysis that computes the set of all states that are reachable from the initial state. If the set of reachable states contains any bad state, the system violates the safety property.

Following the example of [HR99] we present the reachability problem as a sampling network that defines a single output that is true, as long as the safety property has not been violated. This approach has the advantage that we may use sampling networks for all aspects of the system.

Satisfaction of the safety-property is checked by a *synchronous observer*, a component in the sampling network that defines a boolean flow that is true as long as the safety-property is satisfied. Multiple observers can be combined to check multiple safety-properties as well as assumptions on the input.

6.1.1 Drift Bound Observer

The drift bound observer is a sampling network component that verifies if two flows satisfy their bounds. It is mainly used to verify that the root-clocks behaves realistically, but can also serve to express assumptions on the inputs or requirements on the outputs.

Figure 6.1 depicts the following upper drift bound observer of the flow `x` with respect to the flow `y`. The simple observer counts the number of instants where `x` is true during the past four instants where `y` is true. If the `x` is true more than twice for the past four, that is previous three and current, instants the output flow of the observer false. Otherwise the output is true. Thus, it implements an observer that verifies that flows `x` and `y` respect a drift bound such that $D^u_{x/y}(4) = 2$.

```
udrift_obs_2_4 (x, y) =
    let cnt = counter (y, x);

        w_1 = 0 -> pre (sample cnt on y);
        w_2 = 0 -> pre w_1;
        w_3 = 0 -> pre w_2;

        sum = sample cnt + sample w_1 + sample w_2 + sample w_3 on x;
        sat = true -> sum <= 2 and pre sat;
    in sat

counter (reset, incr) =
    let cnt = 0 -> (0 if reset else pre cnt) + d;
        d = 1 if incr else 0;
    in cnt
```

The `counter` component counts the number of instants where `incr` is true and resets the counter to zero when `reset` is true. The `drift_obs` component to count the number of events between two consecutive occurrences of y (instances where `y` is true) and stores the counts for the last four occurrences of y in `w_1`,

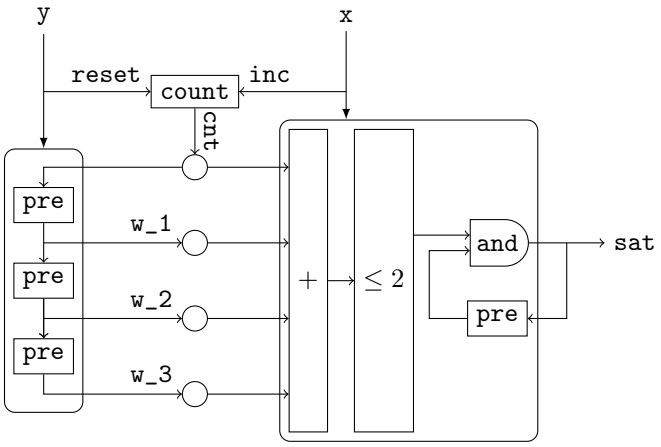


Figure 6.1: Drift bound observer whose output `sat` is true if $X_{x/y}(n+4) - X_{x/y}(n) \leq 2$ for the n-th occurrence of y. It counts the number of occurrences of x in each interval of four consecutive occurrences of y and verifies the total.

`w_2`, and `w_3`. If the total over the last four is greater than four is less than two, the drift bound observer is satisfied. Note that the resulting boolean signal `sat` is clocked by `x` because a violation of the drift bound will occur at the activation of x.

6.1.2 Reachability Analysis of a Watchdog Protocol

Consider the watchdog protocol depicted in Figure 6.2. The watchdog observes an alternating bit to verify if the other component is still active and raises an alarm if it no longer receives changes. Under normal circumstances the system should not raise an alarm, e.g., because the alarm would trigger a irrecoverable emergency shut-down procedure. If the root clocks `p` and `c` do not alternate regularly it is possible that the alarm will be triggered spuriously either because the alternating bit doesn't update on time, or the watchdog doesn't wait long enough for the bit to change.

We extend the watchdog protocol with observers to verify correctness under normal operating conditions. First we add rift bound observers to model assumptions on the root clocks:

- The watchdog cannot oversample the alternating bit. Consequently, the watchdog's clock `d` can tick at most once for every tick of the alternating bit's clock `c`.
- Second, the alternating bit cannot change more than three times for every two samples of the watchdog. Therefore, the clock `c` of the alternating bit may tick at most three times for every two ticks of `d`.

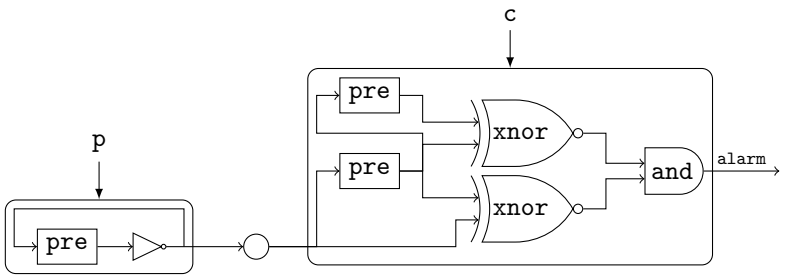


Figure 6.2: A distributed watchdog protocol with an alternating bit. The process on the left sounds the alarm if the alternating bit signal hasn't changed for three instances.

Second, we add an observer that ensure the alarm is not triggered, i.e., the value of alarm is never true. This leads to following sampling network `system` that defines a single output flow that is true, as long as the system behaves correctly.

```
altbit =
    let b = false -> not (pre b);
    in b;

watchdog b_0 =
    let b_1 = pre b_0;
        b_2 = pre b_1;
    in not ((b_0 xor b_1) and (b_1 xor b_2));

always p =
    let x = p -> p and pre x;
    in x;

is_init_2 d = current (true -> true -> pre (pre false) on d)

system (c,d) =
    let bit = altbit on c;
        alarm = watchdog (sample bit) on d;

        real = ldrift_obs_1_1 (c,d) and udrift_obs_3_2(c,d);
        correct = not (current alarm);

    in correct or not always real or is_init_2 d;
```

Let us review the components used in the system.

`altbit` The alternating bit component produces an alternating bit. It is depicted on the right of Figure 6.2.

`watchdog` The watchdog component detects if the alternating bit changes during any of the last three samples. It is depicted on the left of Figure 6.2.

`always` This component defines a flow for a given boolean input flow, that becomes false as soon as its input is false and remains false from that point on. It is used on the output of observers to make sure all states that have been reached by taking a bad or unrealistic transition are marked as bad or unrealistic.

`is_init_2` Creates a stream that is true for two ticks of `d` and remains false after. It is used to allow the system to initialize.

`drift_obs_p_q` The (upper) drift bound observers for a constraint that allows at most p events of one flow per q of the another. They are a variants of the depicted in Figure 6.1 with a window size of q and bound p.

Verification is achieved by constructing the labelled transition system of the observed sampling network and performing a reachability analysis. The reachability analysis consists in the computation of the transitive closure of the transition relation of a finite-transition system. That is, we define the set $reach(\langle S, \Sigma, s_0, \rightarrow\rangle) \subseteq S$ of reachable states of a labelled transition system as follows

$$reach_A = \{s \mid s \in S, s_0 \rightarrow^* s\}$$

where $\rightarrow^* \in S \times S$ denote the transitive closure of transition relation $\rightarrow$.

Constructing the transition system by composition the transition systems of the separate equations yields the reachable state-space depicted in Figure 6.3. The displayed transition diagram does not contain all states, because there would be far too many. In particular, the states with unrealistic behavior — states that can only be reached by violating the drift bounds — have been left out. There should be an outgoing transition for every combination of the clock `c` and `d`, but the missing ones would have violated the drift bounds. Execution starts in the initialization state marked `0|000`, but the system is only behaving correctly once it has left the initialization states. That is, it may (spuriously) raise the alarm during the first two ticks of `d`, even during normal execution. Observe that all reachable states outside of the initialization are indeed correct (have no three equal bits in the watchdog).

With the example we have shown how to verify a safety-property, be it a drift bound or any other property expressed by an observer, of a sampling network driven by clocks described by drift bounds. The main purpose was to introduce drift bound observers and to understand drift bounds in the context of sampling networks and labelled transition systems.

6.2 Extracting Drift Bounds

This section will show how to extract drift bounds of labelled transition systems in general, and the labelled transition systems of sampling networks in particu-

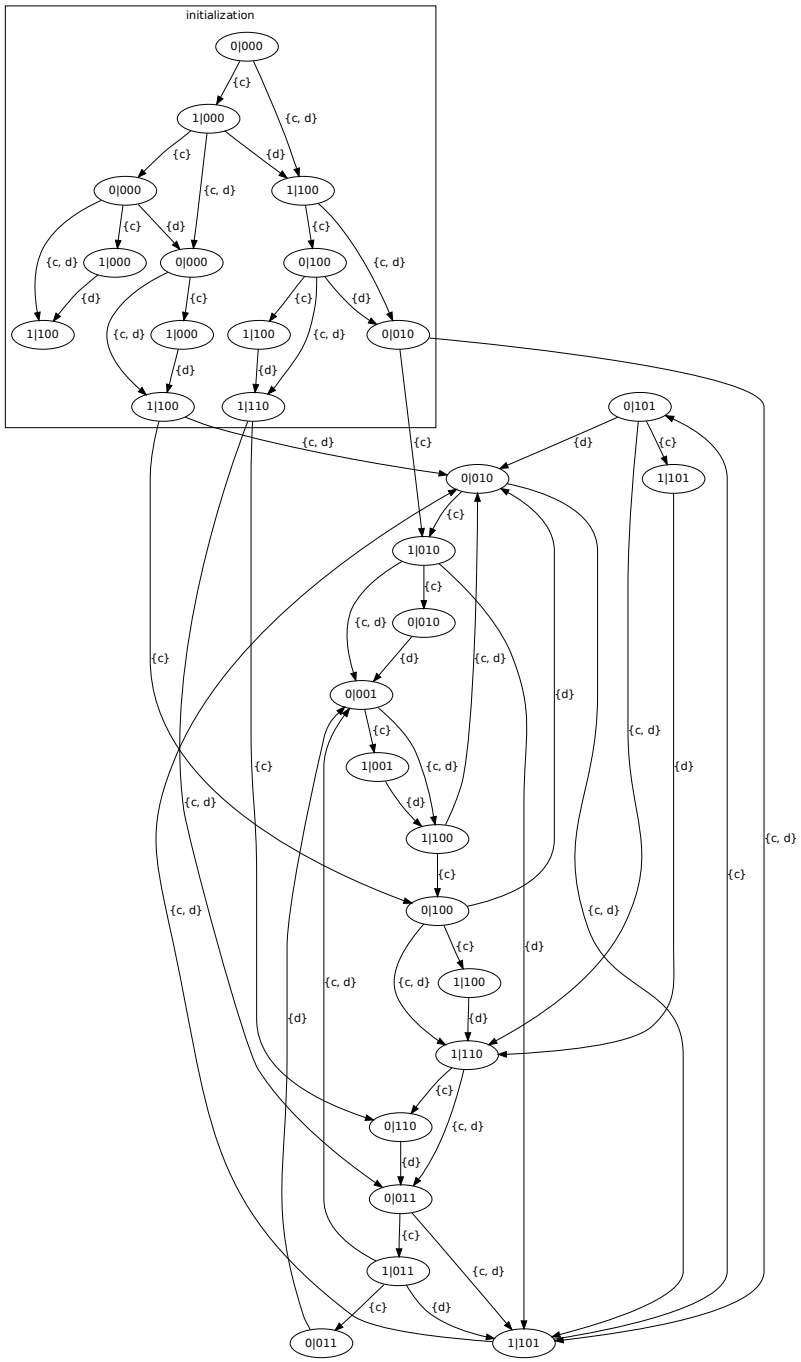


Figure 6.3: Reachable state-space of the watchdog protocol where the drift bounds are satisfied or are in the initialization phase (in the box). States are annotated with the values | `b_0 b_1 b_2` for the flows of the alternating bit (`b`) and the watchdog protocol.

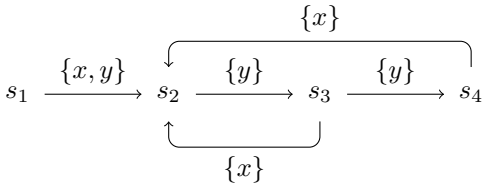


Figure 6.4: A labelled transition system that satisfied a bound $D^u_{y/x}(2) = 2$.

lar. Where the previous section showed how to *verify* (weakly-hard) properties of a system in an environment described by drift-bounds, this section shows how to *infer* drift bounds of signals defined by a sampling network. More precisely, we extract the drift bounds of a sampling network executed in an environment described by drift bounds. That is, we only consider the behavior of a system where the environment — the clock signals that drive execution and inputs — is constrained by drift bounds. For weakly-hard properties the computed drift bounds reveal how frequently it is violated with respect to a reference clock or event.

6.2.1 Drift Bounds of a Transition System

Let us first clarify how signals and their drift bounds relate to transition systems. Realize that a transition system, where transitions are labelled with a set of event names, defines a process of a system with as many signals as there are event names. The process of a transition system consists of all behaviors such that there exists a path from the initial state such that each consecutive transition corresponds to a consecutive time-instant where the signals in the transition label are also present at that time-instant.

Consider the labelled transition system depicted in Figure 6.4. One possible path consist in the following sequence.

$$s_1 \xrightarrow{\{x,y\}} s_2 \xrightarrow{\{y\}} s_3 \xrightarrow{\{y\}} s_4 \xrightarrow{\{x\}} s_2$$

A corresponding behavior is $(x, y) = (\{1, 4\}, \{1, 2, 3\})$. It is constructed by enumerating the transitions and using the transition numbers as time of occurrence.

By assigning different, but strictly increasing, times to the transitions an infinite number of behaviors can be constructed for a single path. Note however, that all such behaviors have the same relative counter functions. This is reflected by the intuition that the relative counter functions describe the interleaving of events.

A drift bound limits the number of possible interleavings of events in a path. For example, an upper drift bound such that $D^u_{x/y}(3) = 2$ dictates that each path segment may have at most two transitions where x is present, per three transitions where y is present. The above path, for example, satisfies that drift bound.

More generally we say that a labelled transition system satisfies a drift bound, if all its behaviors satisfy the drift bound. This means, that the drift bound is satisfied, if there exists no path (from the initial state) with segments that violate the drift bound. The labelled transition system depicted in Figure 6.4, for example, satisfies a drift bound $D^u_{y/x}(2) = 2$. It does not, however, satisfy a bound where $D^u_{x/y}(4) = 2$ as evidenced by following path segment:

$$\ldots \to s_2 \xrightarrow{\{y\}} s_3 \xrightarrow{\{x\}} s_2 \xrightarrow{\{y\}} s_3 \xrightarrow{\{x\}} s_2 \xrightarrow{\{y\}} s_3 \xrightarrow{\{x\}} s_2 \xrightarrow{\{y\}} s_3 \to \ldots$$

The drift bound $D_{x/y}(\Delta)$ of a labelled transition system bounds the number of occurrences of x in all path segments (from any reachable state) that contains exactly Δ occurrences of y and ends in a transition where y occurs.

6.2.2 An Algorithm to Extract Drift Bounds

We now introduce an algorithm that finds the exact drift bound $D_{x/y}$ of the events x and y in of a labelled transition system with states S. That is, an algorithm to find the maximum and minimum number of occurrences of x in a path with Δ occurrences of y and y is present in the last transition.

Algorithm Sketch

The algorithm proceeds in the following steps illustrated in Figure 6.5. For now, we only compute the upper drift bound $D^u_{x/y}$, later we generalize to lower bounds.

Define the weight of a path as the number of occurrences of x in that path. Construct transition system H with states S where the transitions are labelled with weights. The weight is 1 if x is present in the transition and 0 otherwise. The transition system now H contains the weight of all transitions of the labelled transition system where y is absent.

Compute the transitive, reflexive closure H^* of H to obtain the weight of all paths (of an arbitrary number of transitions) free of y transitions. If there are loops (infinitely long paths) where y does not occur but x does, the weight is infinite and we can stop the algorithm because drift is unbounded. Note that it is only necessary to maintain the largest weight of all paths connecting two states.

The upper drift bound $D^u_{x/y}(1)$ for an interval of size one is the maximum weight of all paths that contain a single transition with y in the last transition. To obtain the (maximum) weight of all paths ending in a transition with y, define the weight-labelled transition system K with states S that contains the transitions where y is present. As for H, the transitions of K are labelled with a weight of 1 if x is present and 0 otherwise. By concatenating transitions from H^* with transitions K we obtain the weights of all paths ending in a transition with y. The maximum weight of all paths in the concatenation H^*K gives the upper drift bound $D^u_{x/y}(1)$.

To compute the drift bound for larger intervals, we concatenate transitions of H^*K to obtain the weight of paths with, at first, two occurrences of y, then three, etc.

The analysis for the lower drift bound proceeds in the same fashion, except that we maintain the minimum weight (number of occurrences of x) at each step rather than the maximum weight.

The analysis is restricted to the reachable part of a labelled transition system. This implies that the analysis is preceded by a reachability analysis of the transition system. Without the reachability analysis the result may overestimate the bounds, because it would take into account (segments of) paths that do not come from an initial state.

The Max/Min-Plus Based Algorithm

The algorithm is expressed by matrix computations in the max-plus (for the upper bounds) and min-plus (for the lower bounds) algebras [BCOQ92] on the incidence matrices of transition systems H and K. This not only enables a very concise formulation of the algorithm, but also allows us to use some well-studied algorithms.

Weight-labelled transition systems are transition systems where all transitions are labelled with a number from $\mathbb{N}^\infty$ that we call weight. The incidence matrix of a weight-labelled transition system, henceforth the weight matrix, is a square matrix indexed by states S.

Each element $W_{pq} \in \mathbb{N}^\infty \cup \{\epsilon\}$ of a weight matrix W gives the maximum or minimum weight of all paths from state $p \in S$ to state $q \in S$. For the matrix of maximum weights the elements have the following significance:

- If $W_{pq} \in \mathbb{N}$ there is a path of finite weight W_{pq} from p to q.
- If $W_{pq} = \epsilon$ there is no path from p to q.
- If $W_{pq} = \infty$ there is a path of infinite length.

The transition system H contains the weights of all transitions where y is absent. The transition system K contains the weights of all transitions where y is present. For example, the weight matrices H and K of Figure 6.5 are

$$H = \begin{pmatrix} \epsilon & 1 & \epsilon \\ \epsilon & \epsilon & 0 \\ \epsilon & \epsilon & \epsilon \end{pmatrix} \qquad K = \begin{pmatrix} \epsilon & \epsilon & \epsilon \\ \epsilon & \epsilon & \epsilon \\ 0 & \epsilon & 1 \end{pmatrix}$$

These matrices are elements of the max-plus and min-plus algebras. In these algebras, addition plays the role of multiplication, and the minimum resp. maximum take the role of addition. We extend the addition operator such that $x + \epsilon = \epsilon$ and $x + \infty = \infty$, and extend the maximum such that $\max(x, \epsilon) = x$. In the min-plus algebra $\epsilon = \infty$.

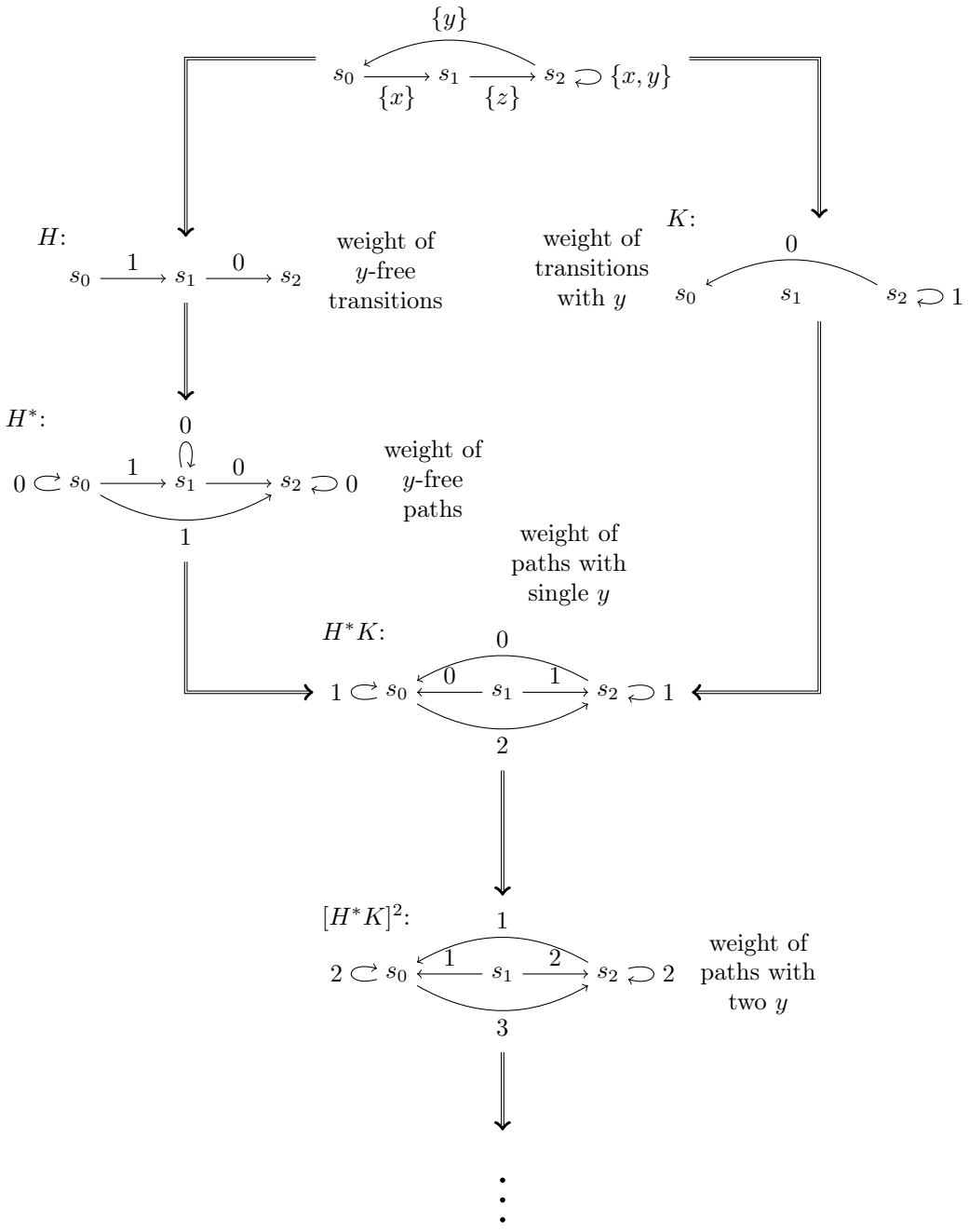


Figure 6.5: The subsequent steps of the drift bound extraction algorithm for the upper drift bound $D^u_{x/y}$ of the example at the top. Each step yields a weight-labelled transition system where the weight w of a path $p \xrightarrow{w} q$ is the maximum number of occurrences of x in a path from p to q.

Matrix multiplication of weight matrices A and B with dimensions $N \times O$ and $O \times M$ respectively, is defined in the max-plus algebra such that

$$[AB]_{ij} = \max_{1 \leq k \leq O} A_{ik} + B_{kj}$$

for all $1 \leq i \leq N$ and $1 \leq j \leq M$ and with the minimum in stead of the maximum for the min-plus algebra.

The concatenation of the paths of two weight-labelled transition systems, where we add the weight of connecting paths and only keep the paths with maximum or minimum weight between any two states, consists in the multiplication of their incidence matrices in the max- and min-plus algebras.

Multiplication also leads to powers of square matrices A of dimension N such that $A^{n+1} = AA^n$ and A^0 is the identity matrix, i.e., for all $1 \leq i, j \leq N$

$$A^0_{ij} = \begin{cases} 0 & \text{if } i = j \\ \epsilon & \text{otherwise} \end{cases}$$

To compute the longest paths (of any length) and find loops (including loops longer than one transition), we concatenate paths of increasing lengths to obtain the transitive closure. For the weight matrices, this amounts to computing higher powers. Each consecutive power of the matrix give the weight (number of transitions containing i) of the path between each pair of states. The H^* closure of H is defined for all $p, q \in S$ as follows

$$H^*_{pq} = \sup_{n \in \mathbb{N}} H^n_{pq}$$

and consists of all longest paths (of any length) between any two pairs of states. For example, the closure of H in Figure 6.5 is as follows

$$H^* = \begin{pmatrix} 0 & 1 & 1 \\ \epsilon & 0 & 0 \\ \epsilon & \epsilon & 0 \end{pmatrix}$$

The closure can be computed using a (modified) Floyd-Warshall [Flo62, ORE] algorithm. Values in the diagonal of H^* can either be zero or infinite, i.e., $H^*_{ss} \in \{0, \infty\}$ for any state $s \in S$. If any of the elements is infinite there is a loop and the drift is unbounded.

The concatenation H^*K gives the weights (maximum number of occurrences of x) for all paths between any two pair of states ending in a transition that contains y. That is, it gives the drift bound for an interval of one for all possible transitions. To obtain the drift of for larger intervals we iteratively concatenate the paths in $[H^*K]$ (adding their weights) to obtain paths of increasing lengths:

$$[H^*K]^1 = \begin{pmatrix} 1 & \epsilon & 2 \\ 0 & \epsilon & 1 \\ 0 & \epsilon & 1 \end{pmatrix} \quad [H^*K]^2 = \begin{pmatrix} 2 & \epsilon & 3 \\ 1 & \epsilon & 2 \\ 1 & \epsilon & 2 \end{pmatrix} \quad [H^*K]^3 = \begin{pmatrix} 3 & \epsilon & 4 \\ 2 & \epsilon & 3 \\ 2 & \epsilon & 3 \end{pmatrix}$$

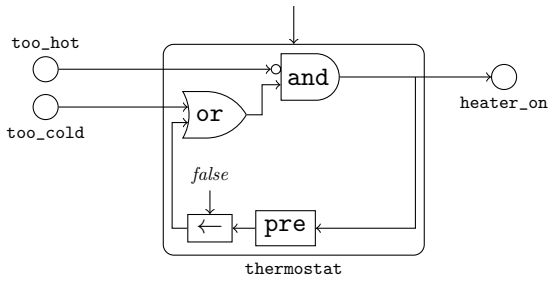


Figure 6.6: Simple thermostat that samples environment temperature to control a heater.

That is, the upper drift bound is computed as the least upper bound of all paths, between any two states, with Δ occurrences of the reference clock y:

$$D^u_{x/y}(\Delta) = \max_{p \in S} \max_{q \in S} [H^* K]^{\Delta}_{pq}$$

The lower drift bounds of the examples are computed similarly, except that computations are performed in the min-plus calculus. Thus, all supremum and maximum operators are replaced by infinimum and minimum operators respectively.

6.2.3 Computing the Drift Bounds of a Thermostat

To illustrate the extraction of drift bounds, we will apply the described algorithm to the following program that models a thermostat and heater in an environment. The thermostat, depicted in Figure 6.6, must control the heater in order to keep temperature within a desired range. We will use the drift bound extraction to determine how many consecutive time units the room will can be too cold or too warm.

```
thermostat (too_cold, too_hot) =
    let heater_on = false -> (not too_hot and pre heater_on) or too_cold;
    in heater_on;

heater command = false -> pre command;

environment heater_state =
    let too_hot = not udrift_obs_4_5 (heater_state, true);
        too_cold = not ldrif_obs_1_5 (heater_state, true);
    in (too_hot, too_cold)

system (c,d) =
    let command = thermostat (sample (too_cold, too_hot)) on c;
        heater_state = heater (sample command) on d;
```

```
        (too_cold, too_hot) = environment (current heater_state);

        realistic = ldrift_obs_1_2 (d, true) and ldrift_obs_1_2 (c, true);

        is_temperate = not (too_hot or too_cold);

    in (always realistic, is_temperate);
```

The system defines two output flows: one that is true as long as the environment has behaved realistically during execution — the clocks have respected their drift bounds — and another that verifies that the temperature is within a given range. The former is used to delineate realistic behavior and the latter is subject of the drift bound extraction. Let us review the components that define the system one-by-one.

`thermostat` The thermostat switches on the heating if it is too cold and keeps it enabled until it becomes too hot and leaves it off until it becomes too cold again.

`heater` The heater executes its command after a delay of one tick, resulting in a flow `heater_state` that is true when the heater is on and false otherwise.

`environment` The environment component models the environments reactions to the heating. We consider the clock of the heater to be a discretization of real time and then consider that it is too warm if the heater was enabled during 4 out of 5 of the last time units, and too cold if the heater was disabled during 4 out of 5 time units. We use the drift bound observers to express these properties.

`system` The composed system with two observers: one that verifies if the clocks have behaved realistically, and another that defines the flow `is_temperate` that is true when it is neither too warm nor too cold.

The environment runs at the global clock's speed whereas the thermostat and heater run on clocks `c` and `d` respectively. The clocks are constrained by the observers such that the thermostat (on `c`) and heater (on `d`) are activated at least once per every two global time steps. The addition of the global clock makes the thermostat system a synchronous system where the sub-clocks `c` and `d` are boolean input signals serving as clocks.

Figure 6.7 shows the reachable state-space of the thermostat. It only shows the realistic behavior, i.e., it only shows the states that can be reached while `realistic` is always true. All transitions correspond with a tick of the root clock and are marked with `c` and `d` when those signals are present.

Note that it is impossible to stay on a path within the colored states indefinitely: eventually the thermostat will start or stop the heater and temperature will be corrected. The lower drift bound of `is_temperate` with respect to its clock (the global root clock) should reflect this because the number of consecutive time instants where `is_temperate` is false (does not occur) should be

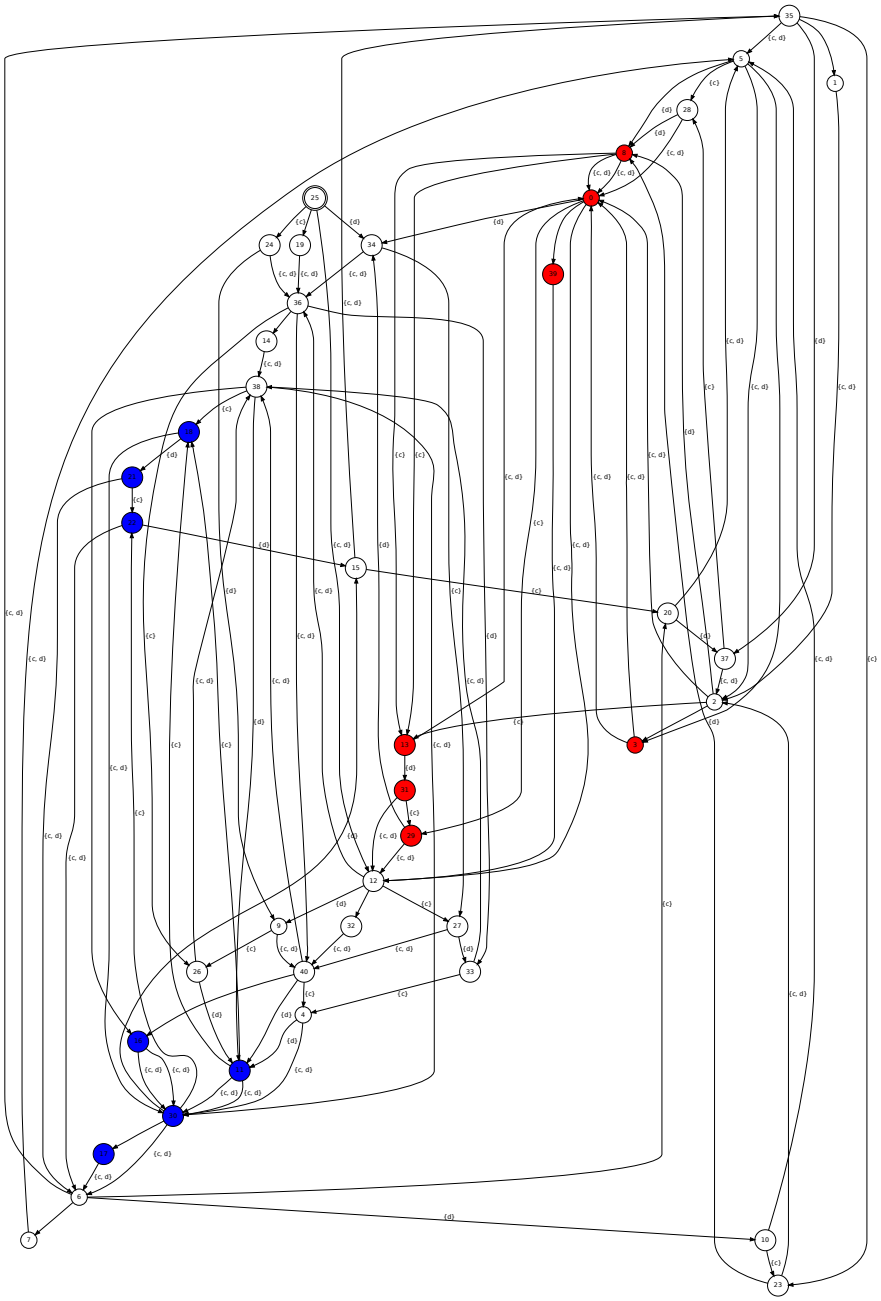


Figure 6.7: Reachable state-space of the thermostat when constrained to realistic behavior. Start state has a double circle, and the red and blue states indicate when temperature is too high or too low respectively.

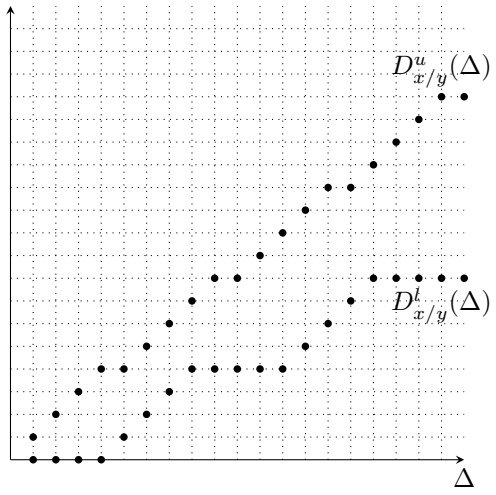


Figure 6.8: Upper and lower drift bounds $D_{t/r}$ where t is the signal of flow `is_temperate` and r is the global root clock signal.

bounded. Figure 6.8 depicts the computed upper and lower drift bounds for `is_temperate`. The lower drift bound shows that temperature will neither be too hot (temperate) after at most four time steps (occurrences of the root clock). The upper drift bound shows that temperature will either be too hot or too cold (not temperate) after at least four time steps.

6.3 Compositional Analysis of Sampling Networks

The precision of the drift bound extraction comes at a great cost: the state-space grows exponentially with every composition, making the analysis infeasible for larger systems. This section proposes a compositional method based on the approach presented in Section 3.4.1. The idea is to partition the system into separate components and compute the abstraction of each component's process when restricted to the drift bounds of the other components' processes.

To this end, we introduce a Galois connection between labelled transition systems, that define the semantics of sampling networks, and drift bounds. The essential operations have already been introduced: the concretization γ of drift bounds with drift bound observers in Section 6.1.1 and the abstraction α of a labelled transition system as the extraction of drift bounds in Section 6.2. The Galois connection justifies the application of the compositional analysis based on abstract interpretation.

6.3.1 The Domain of Languages

Formally, we relate the semantics of sampling networks with processes by connecting the domain of languages of transition systems with the domain of processes and drift bounds. That is, we define the domain of languages $\mathcal{L}$ of transition systems as an abstraction for processes. In addition, we show how drift bounds are indeed an abstraction for the domain of languages. The connections enable us to base the analysis on an iterative fix-point computation (see Section 3.4.1) where components are each modelled by a single mapping over drift bounds. and connect it with the domain of relative counter functions through a Galois connection.

The language of a labelled transition system is defined, as is usual, to consist of the strings that describe all possible paths through the transition system from the initial state. The string consists of the sequence of the labels, the characters, encountered during traversal along a path.

Definition 31 (String)**.** *Let the sequence $s \in \Sigma^*$ denote a string for a labelled transition system system with alphabet Σ. The empty string is denoted ϵ and $(\sigma : s)$ denotes a string where the character $\sigma \in \Sigma$ precedes the string s.*

Consider the labelled transition system depicted in Figure 6.4. The following sequence describes one possible path in the transition system:

$$s_1 \xrightarrow{\{x,y\}} s_2 \xrightarrow{\{y\}} s_3 \xrightarrow{\{y\}} s_4 \xrightarrow{\{x\}} s_2$$

The string of that sequence is $\{x, y\} : \{y\} : \{y\} : \{x\} : \epsilon$.

A string of an labelled transition system effectively describes an interleaving of events, i.e., it describes the order of events in a behavior. We can reconstruct such a behavior by enumerating the transitions to give them a time-stamp and reconstruct the signal of each event as the set of all transition numbers that contain the event in the label. For example, the behavior corresponding to the path and string above is $(x, y) = (\{1, 4\}, \{1, 2, 3\})$. Formally, we define this as the string behavior $\boldsymbol{\tau}^s$ for string s.

Definition 32 (String behavior)**.** *Let $(\tau^s_{x_1}, ..., \tau^s_{x_N}) \in \mathcal{S}^{\mathcal{V}}$ with $\mathcal{V} = \{x_1, ..., x_N\}$ describe the behavior of the string $s \in \wp(\mathcal{V})^*$ of a system with events $\mathcal{V}$ such that for all $x \in \mathcal{V}$*

$$\tau^{\epsilon}_x = \emptyset \text{ and } \tau^{\sigma:s}_x = \{n + 1 \mid n \in \tau^s_x\} \cup \begin{cases} \{0\} & \text{if } x \in \sigma \\ \emptyset & \text{otherwise} \end{cases}$$

The *process* of a labelled transition system consists of all behaviors defined by the alphabet and all possible retimings with a monotonic, bijective *retiming function* that, when applied to all signals of a behavior, stretches or compresses time without changing the interleaving of events.

This is not enough, however, to formally consider drift bounds an abstraction of labelled transition systems because, while both drift bounds and languages are abstractions of processes, they are not related to each other. Therefore, we

define the following Galois connection where relative counter functions are an abstraction of labelled transition systems.

Definition 33 (Concrete of languages ($\mathcal{L} \rightleftharpoons \mathcal{X}$))**.** *Let the domain of languages $\mathcal{L}^{\mathcal{V}}$ for events $\mathcal{V}$ consist of all languages $L \subseteq \wp(\mathcal{V})^*$. The domain of relative counter function matrices is an abstract domain for the domain of languages, i.e., $\mathcal{L}^{\mathcal{V}} \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} \mathcal{X}^{\mathcal{V}}$ defined by the abstraction function*

$$\alpha(L) = \{X \mid X \in \mathcal{X}^{\mathcal{V}}, x, y \in \mathcal{V}, s \in L, X_{x/y} = \chi_{\tau_x^s} \circ \eta_{\tau_y^s}\}$$

In conclusion, the relative counter functions, as well as clock and drift bounds are formally an abstraction of the languages of labelled transition systems and sampling networks are an abstraction of processes.

6.3.2 Abstraction by Concretization and Extraction

With the Galois connection between sampling networks and drift bounds we define the abstract transfer function $[\![x_1 = e_1; ...]\!]^{\|\mathcal{D}}$ for the transition system of a sampling network with equations $x_1 = e_1; ...$ such that

$$[\![x_1 = e_1; ...]\!]^{\|\mathcal{D}}(D) = \alpha([\![x_1 = e_1; ...]\!] \parallel \gamma(D))$$

The labelled transition system defined by a satisfied drift bound observer of Section 6.1.1 realizes the *concretization* γ of a drift bound. We define the concretization of a drift bound $D^u_{x/y}(m) = n$ as follows

$$\gamma(D) = \pi_{\{x,y\}}\left([\![z = \texttt{udrift\_obs\_}m\texttt{\_}n(x,y)]\!] \parallel [\![z = \texttt{true}]\!]\right)$$

The composition with a sampling network that defines z, the output of the drift bound monitor, to be true at all times limits the reachable state space to the states where the drift bound is satisfied. That is, the above describes a labelled transition system with a language that consists of all strings that satisfy the given drift bound.

The extraction of drift bounds presented in Section 6.2 is the *abstraction* of sampling networks, i.e.,

$$[\alpha(\langle S, \wp(\mathcal{V}), s_0, \rightarrow\rangle)]^u_{x/y}(\Delta) = \min_{p,q \in S} [H^* K]^{\Delta}_{pq}$$

where H and K are the weight matrices of (reachable) transitions $\rightarrow$ with resp. without an occurrence of y as described in Section 6.2.

6.3.3 A Compositional Analysis

The main practical consequence of the introduced Galois connection, is the application of the approximation of composed systems, presented in Section 3.4.1. With it, we may approximate the abstraction of the composition of sampling

networks $x_1 = e_1; ...x_n = e_n$ and $x'_1 = e'_1; ...; x'_m = e'_m$ with the fix-point of their abstract transfer functions:

$$\begin{aligned}&\alpha([\![x_1 = e_1; ...; x_n = e_n]\!]^{\mathcal{L}} \parallel [\![x'_1 = e'_1; ...; x'_m = e'_m]\!]^{\mathcal{L}})\\ \sqsubseteq\, &\mathrm{gfp}\left([\![x_1 = e_1; ...; x_n = e_n]\!]^{\parallel\mathcal{D}} \circ [\![x'_1 = e'_1; ...; x'_m = e'_m]\!]^{\parallel\mathcal{D}}\right)\end{aligned}$$

When the concrete composition has a large transition system due to the combinatory explosion of states, the approximation can be used to avoid the construction of the full transition system by analysing only the parts.

Let us illustrate this problem with a simple, synchronous N-place buffer described equations $x_{i+1} = \texttt{pre}\, x_{i-1}\ \texttt{on}\, c$ for $1 \leq i \leq N$. It has one input flow $x_0\ \texttt{on}\, c$ bounded by a drift bound with respect to the clock signal c. The state-space of the fully composed system grows exponentially in the number of places in the buffer. Yet clearly, and this is confirmed by the analysis, $D^u_{x_i/c} = D^u_{x_{i-1}}$ for all $1 \leq i \leq N$[1], so the construction of the state space is unnecessary to obtain the required result. By decomposing the system using the abstract transfer functions, i.e., by computing

$$\left[\mathrm{gfp}\left([\![x_1 = \texttt{pre}\, x_0\ \texttt{on}\, c]\!]^{\parallel\mathcal{D}} \circ ... \circ [\![x_n = \texttt{pre}\, x_{n-1}]\!]^{\parallel\mathcal{D}}\right)\right]_{x_n/c}$$

we obtain the required result, while only computing the composition of a single synchronous delay (`pre`) and the concretization of the drift bounds.

In this case, the decomposition gives a precise result, but in general a decomposition yields an over-approximation, in for sampling networks with cyclical dependencies between flows. The compositional analysis of a sampling network `x = false -> not (pre x) on c` that defines an oscillating flow, for example, will yield no bound on `x` (e.g. $D^u_{x/c}(n) = \infty$ for $n > 0$). Fortunately, the compositional analysis is flexible and we can, for example, choose to decompose only equations involved in a cycle and ignore the others.

6.4 Abstract Interpretation of Sampling Networks

Although the compositional analysis can help us to limit the number of states, the concretization of drift bounds may yield very large transition systems by itself. The resulting explosion of the state-space is especially striking in the case of combinatory components even though the operations are stateless. This section presents an extension where we replace the abstract transfer functions of combinational sampling networks with an approximative, abstract transfer function that operates directly on drift bounds without constructing any labelled transition system.

[1]Note, however, that $D^u_{x_i/c} \neq D^u_{x_{i-1}}$ but $D^u_{x_i/c}(n{+}1) \leq D^u_{x_{i-1}}(n)$, due to the initialization of the buffer elements with the value false.

6.4.1 Drift Bounds of Clocked Signals

Recall that all flows of a well-clocked sampling network are clocked: each signal x is associated with another signal denoted $\hat{x}$. The flow `x` is true at the moments $t \in \mathbb{R}$ when both its clock is present ($t \in \hat{x}$) and its own signal is present ($t \in x$). It is false when only its clock is present.

By construction a flow in a well-clocked sampling network cannot be present when its clock is not, i.e., $x \subseteq \hat{x}$. The clock relation has consequences for the drift between the signal of a flow and its clock signal: between any number of events of x there must be at least an equal number of events in $\hat{x}$ and between any number of events of $\hat{x}$ there can be at most the same number of events. This is formalized by the following lemma.

Lemma 17 (Abstract clock relation)**.** *Let (x, y) denote a behavior such that x `on` y if, and only if, for all n and Δ in $\mathbb{N}^\infty$*

$$X_{x/y}(n + \Delta) - X_{x/y}(n) \leq \Delta \leq X_{y/x}(n + \Delta) - X_{y/x}(n)$$

Proof. To realize that $x \subseteq y$ is implies the inequality, note that at the time of each occurrence of x there must be at least one occurrence of y and, conversely, for each occurrence of y there must be one occurrence of x. □

In the abstract domain the inequality of Lemma 17 implies that, for any processes with behaviors $(x, y) \in \mathcal{S}^2$ where `x on y`

$$D^l_{y/x}(\Delta) \leq \Delta \leq D^u_{x/y}(\Delta)$$

This drift bound is independent of any input or functional aspect of the analysed sampling network. The bounds hold for any signal in a sampling network including input signals and root clocks.

6.4.2 Constant Flow

A constant flow is either true or false at every tick of its clock. Thus, either the stream associated with the constant is equal to its clock (if the constant is true) or empty (if it is false). Hence, the relative counter function of the constant flow with respect to its clock is either a constant zero or the identity function. Let `x = `v` on c` for some constant value $v \in \mathbb{B}$ define a process of all behaviors (x, c) such that for all $n \in \mathbb{N}^\infty$

$$X_{x/c}(n) = \begin{cases} n & \text{if } c \\ 0 & otherwise \end{cases}$$

A constant value of true has no drift with respect to its clock:

$$[\![x = \texttt{true on } c]\!]^{\mathcal{D}}]^l_{x/c}(\Delta) = \Delta = [\![x = \texttt{true on } c]\!]^{\mathcal{D}}]^u_{x/c}(\Delta)$$

for all $\Delta \in \mathbb{N}^\infty$. A constant value of false never increases at all:

$$[\![x = \texttt{false on } c]\!]^{\mathcal{D}}]^l_{x/c}(\Delta) = 0 = [\![x = \texttt{false on } c]\!]^{\mathcal{D}}]^u_{x/c}(\Delta)$$

for all $\Delta \in \mathbb{N}^\infty$. Constant flows are uniquely identified by their drift bounds with respect to their clock: any signal x with clock c with the bounds $D^l_{x/c}(\Delta) = \Delta$ or bound $D^u_{x/c}(\Delta) = 0$ is a constant flow of value true or false respectively.

6.4.3 Abstract Negation

Negation, e.g., the equation `x = not y on c`, defines a signal x that is present on all events in c where y is not present. Therefore the maximum number of times `x` can be true in an interval of Δ ticks of `c` is the difference between Δ and the minimum number of times `y` must be true.

Lemma 18 (Abstract negation)**.** *Let* (x, y, c) *be a behavior such that* `x = not y on c` *and* $(x, y, c) \in \gamma(C, D)$ *then, for any signal* z *and all* $n, \Delta \in \mathbb{N}^\infty$

$$D^l_{c/z}(\Delta) - D^u_{y/z}(\Delta) \leq X_{x/z}(n + \Delta) - X_{x/z}(n) \leq D^u_{c/z}(\Delta) - D^l_{y/z}(\Delta)$$

Proof. Realize that, $x = c \setminus y$ and therefore

$$\chi_x(t) = \chi_c(t) - \chi_y(t)$$

which implies $X_{x/k}(n) = X_{c/k}(n) - X_{y/k}(n)$ for any signal k. That leads, with some arithmetic, to the stated inequalities. □

The abstract negation is defined such that, for all events z and all $\Delta \in \mathbb{N}^\infty$,

$$[\![x = \texttt{not}\ y\ \texttt{on}\ c]\!]^{\|\mathcal{D}}(D)]^u_{x/z}(\Delta) = D^u_{c/z}(\Delta) - D^l_{y/z}(\Delta)$$

Consider, for example, the negation of a constant: `x = not true on c`. As $D^u_{c/c}(\Delta) = \Delta$ and $D^l_{\texttt{true}/c}(\Delta) = \Delta$, the drift $D^u_{x/c}(\Delta) = 0$. The negation `x = not y on c` of a completely unconstrained (apart from the drift bound due to the clock relation) flow `y on c` with bounds

$$D^l_{y/c}(\Delta) = 0 \leq X_{y/c}(n + \Delta) - X_{y/c}(\Delta) \leq \Delta = D^u_{y/c}(\Delta)$$

has the exact same bounds, i.e., $D_{x/c} = D_{y/c}$.

Abstract negation is completely symmetric in the sense that a double negation yields the original bounds. In a way, the abstract negation exchanges the upper and lower bounds of a flow. We will use this fact to define exclusively deal with upper bounds, using upper bounds on negated flows in stead of lower bounds when needed.

6.4.4 Abstract Disjunction

The drift bound of a binary disjunction `x = y or z on c` is derived by considering a best-case scenario, which occurs when signals y and z are true at different time instances in the interval, i.e., the events in y and z never coincide.

Lemma 19 (Abstract disjunction)**.** *Let* (x, y, z, c) *be a behavior such that* `x = y or z on c` *and* $(x, y, z, c) \in \gamma(C, D)$ *then, for any signal* r *and all* $n, \Delta \in \mathbb{N}^\infty$

$$X_{x/r}(n + \Delta) - X_{x/r}(n) \leq D^u_{y/r}(\Delta) + D^u_{z/r}(\Delta)$$

Proof. Realize that $x = y \cap z$, which implies $\chi_x(t) \leq \chi_y(t) + \chi_z(t)$ for all $t \in \mathbb{R}^\infty$ and therefore also for all $\eta_r(n)$ with $n \in \mathbb{N}^\infty$. □

The abstract disjunction is defined such that, for all events v and all $\Delta \in \mathbb{N}^\infty$,

$$[\![x = y \texttt{ or } z \texttt{ on } c]\!]^{\|\mathcal{D}}(D)]^u_{x/v}(\Delta) = D^u_{y/v}(\Delta) + D^l_{z/v}(\Delta)$$

Of course the disjunction x cannot be true more often than its clock is present. Yet, with the above lemma we would conclude that `x = true or true on c` has a bound $D^u_{x/c}(\Delta) = 2\Delta$. Therefore we cannot forget the additional constraint of clocked flows that $D^u_{x/c}(\Delta) \leq \Delta$.

6.4.5 Abstract Conjunction

The abstraction of a binary conjunction `x = y and z on c` is based on a best-case scenario where `y` and `z` are true at exactly the same time-instances (events in x and y occur simultaneously). Logically, the number of time-instances where both occur simultaneously is bounded by the maximum number of events in either x or y for any given interval.

Lemma 20 (Abstract conjunction)**.** *Let* (x, y, z, c) *be a behavior such that* `x = y and z on c` *and* $(x, y, z, c) \in \gamma(C, D)$ *then, for any signal* r *and all* $n, \Delta \in \mathbb{N}^\infty$

$$X_{x/r}(n + \Delta) - X_{x/r}(n) \leq \min(D^u_{y/r}(\Delta),\ D^u_{z/r}(\Delta))$$

Proof. As for Lemma 19 except that $x = y \cup z$ which implies $\chi_x(t) \leq \chi_y(t)$ and $\chi_x(t) \leq \chi_z(t)$ for all $t \in \mathbb{R}^\infty$. □

The abstract conjunction is defined such that, for all events v and all $\Delta \in \mathbb{N}^\infty$,

$$[\![x = y \texttt{ and } z \texttt{ on } c]\!]^{\|\mathcal{D}}(D)]^u_{x/v}(\Delta) = \min(D^u_{y/v}(\Delta), D^l_{z/v}(\Delta))$$

Remark that it is impossible to derive the abstract conjunction from the abstract disjunction via the De Morgan laws (or vice versa). This is because the abstract disjunction only gives an upper drift bound, while the negation depends on the lower bound.

6.4.6 Abstract Boolean Equations

While the introduced operators suffice to represent any boolean operation the composition of abstractions may lead to a loss of precision in some cases, because the boolean disjunction will be pessimistic if its operants can be true simultaneously. For example, `v = (x and y) or (x and z) on c` will yield pessimistic results for any drift bound where $D^u_{y/c} + D^u_{z/c} \not\leq D^u_{x/c}$ and $D^u_{y/c} \leq D^u_{x/c}$ and

$D^u_{z/c} \leq D^u_{x/c}$, because the result cannot be true more often than `x` is true. Rewriting to the equivalent equation `v = x and (y or z) on c` in conjunctive normal form yields a tighter result in the abstraction.

In other words, the decomposition of boolean equations and subsequent abstraction, e.g., to approach the abstraction of $[\![v = (x \texttt{ and } y) \texttt{ or } (y \texttt{ and } z) \texttt{ on } c]\!]$ with the compositional analysis

$$\text{gfp}\left([\![v = v_1 \texttt{ or } v_2 \texttt{ on } c]\!]^{\|\mathcal{D}} \circ [\![v_1 = x \texttt{ and } y \texttt{ on } c]\!]^{\|\mathcal{D}} \circ [\![v_2 = y \texttt{ and } z \texttt{ on } c]\!]^{\|\mathcal{D}}\right)$$

yields an overly pessimistic approximation.

This brings us to the abstraction of logical equations in the conjunctive normal form. Intuitively the conjunctive normal form has the advantage that all elements in a disjunction are independent variables and can interleave arbitrarily and therefore adding their upper drift bounds is not pessimistic.

An expression in conjunctive normal form consists of a conjunction of clauses, where each clause consists of a disjunction of atoms and each atom is either a variable or a negated variable. Following convention, we consider a CNF expression to consist of a set of clauses and, in turn, a clause to be a set of atoms. As such we hold an empty CNF expression (without any clauses) to be a tautology and an empty clause to be unsatisfiable.

Let e denote the expression in conjunctive normal form and e' an equivalent expression, not necessarily in conjunctive normal form, that can be transformed to e using the mutual distributivity of disjunction and conjunction and removal of tautologies. The conjunctive normal form of an expression has the tightest abstraction of all forms of that can be obtained by distributivity of the conjunction and disjunction and the de Morgan laws.

In the abstraction this amounts to distributing addition over the minimum and vice versa. While addition distributes over the minimum ($a + \min(b, c) = \min(a+b, a+c)$), the minimum only weakly distributes over addition ($\min(a, b+c) \leq \min(a, b) + \min(a, c)$).

6.4.7 Analysis of a Resource Sharing Protocol

In this example we analyse a primitive resource sharing system between subcomponents. To negotiate access to the resource a simple priority-based handshake protocol is used. Because the components are too distant from each other to share the same clock domain they communicate by sampling. The clocks are almost synchronous but do drift over large periods. The result is, that the number of states is very large as the observer's windows are large. Therefore we perform the analysis in parts and use abstract interpretation wherever possible. Figure 6.9 depicts the main components of the following program.

```
prio (ack, reqHigh, reqLow) =
    let req = reqHigh or reqLow;
        ackHigh = ack and reqHigh;
        ackLow = ack and not reqHigh and reqLow;
```

```
    in (req, ackHigh, ackLow)

sustain (set,reset) =
    let sig = false -> (set or pre sig) and not reset;
    in sig

system (reqA on c, reqB on d, reqC on d, reqD on e) =
    let (reqX, ackY, ackD) = prio (true, sample reqY, reqD_sust) on e;
        (reqY, ackA, ackZ) = prio (sample ackY, reqA_sust, sample reqZ) on c;
        (reqZ, ackB, ackC) = prio (sample ackZ, reqB_sust, reqC_sust) on d;

        reqA_sust = sustain (reqA, ackA) on c;
        reqB_sust = sustain (reqB, ackB) on d;
        reqC_sust = sustain (reqC, ackC) on d;
        reqD_sust = sustain (reqD, ackD) on e;

        real = udrift_obs_3_20 (reqA, c)
            and ldrift_obs_2_20 (reqA, c)
            and udrift_obs_1_20 (reqB, d)
            and ldrift_obs_1_20 (reqB, d)
            and udrift_obs_1_20 (reqC, d)
            and udrift_obs_1_20 (reqD, e)
            and udrift_obs_21_20 (c, d)
            and ldrift_obs_19_20 (c, d)
            and udrift_obs_21_20 (e, c)
            and ldrift_obs_9_20 (e, c);

        col = (current ackA and current ackB)
            or (current ackA and current ackC)
            or (current ackA and current ackD)
            or (current ackB and current ackC)
            or (current ackB and current ackD)
            or (current ackC and current ackD);

    in (always real, col)
```

The program defines two outputs: one output flow that is true as long as the system has always behaved realistically and one output flow that is true when a collision occurs. The system is said to behave realistically if the clocks respect their drift bounds and the number of resource requests is bounded with respect to the clocks. Let us review the components one-by-one:

`prio` The priority protocol assigns access to one of two parties if they so request and the resource is available. The incoming flow `ack` is true when the resource is available. The incoming flows `reqHigh` and `reqLow` are true when the party with high resp. low priority request access to the resource. The outgoing flow `req` forwards the request from either party to

the resource. The outgoing flows `ackHigh` and `ackLow` are true when the respective parties are granted access. The protocol is compositional in the sense that it can be composed with itself as one of the requesting parties.

`sustain` The sustain component sustains its output signal once it has switched on by the flow `set` and stops when it is reset by flow `reset`. It is used to keep requesting a resource until satisfied.

`system` The composition uses three priority components to share access to a single resource between components `A`, `B`, `C` and `D` (in order of priority).

Because the system is not synchronized, it is possible for two requesting parties to be granted access to the resource at the same time, creating a collision. Such collisions are rare as clocks are nearly synchronous and requests on the resource are also relatively rare. The analysis has to determine how many collisions can occur and if they can occur in bursts.

The analysis starts with a decomposition of the system. First, we separate the stateful components, the sampling primitives and sustain components, from the combinational components. Then, we further separate the combinational components into separate components that each define a single flow with a boolean expression (to be converted to conjunctive normal form later on). We obtain the following decomposition

$$
\begin{aligned}
& [\![\texttt{reqY}' = \texttt{sample}(\texttt{reqY on } c) \texttt{ on } e]\!] \\
\| \; & [\![\texttt{reqX} = \texttt{reqY}' \texttt{ or reqD}_\texttt{s}\texttt{ust on } e]\!] \\
\| \; & [\![\texttt{ackY} = \texttt{true and reqY}' \texttt{ on } e]\!] \\
\| \; & [\![\texttt{ackD} = \texttt{true and (not reqY}') \texttt{ and reqD on } e]\!] \\
\\
\| \; & [\![\texttt{ackY}' = \texttt{sample}(\texttt{ackY on } e) \texttt{ on } c]\!] \\
\| \; & [\![\texttt{reqZ}' = \texttt{sample}(\texttt{reqZ on } d) \texttt{ on } c]\!] \\
\| \; & [\![\texttt{reqY} = \texttt{reqA\_sust or reqZ}' \texttt{ on } c]\!] \\
\| \; & [\![\texttt{ackA} = \texttt{ackY}' \texttt{ and reqA\_sust on } c]\!] \\
\| \; & [\![\texttt{ackZ} = \texttt{ackY}' \texttt{ and (not reqA\_sust) and reqZ}' \texttt{ on } c]\!] \\
\\
\| \; & [\![\texttt{ackZ}' = \texttt{sample}(\texttt{ackZ on } c) \texttt{ on } d]\!] \\
\| \; & [\![\texttt{reqZ} = \texttt{reqB\_sust or reqC\_sust on } d]\!] \\
\| \; & [\![\texttt{ackB} = \texttt{ackZ}' \texttt{ and reqB\_sust on } d]\!] \\
\| \; & [\![\texttt{ackC} = \texttt{ackZ}' \texttt{ and (not reqB\_sust) and reqC\_sust on } d]\!] \\
\\
\| \; & [\![\texttt{reqA\_sust} = \texttt{sustain}(\texttt{reqA}, \texttt{ackA}) \texttt{ on } c]\!] \\
\| \; & [\![\texttt{reqB\_sust} = \texttt{sustain}(\texttt{reqA}, \texttt{ackA}) \texttt{ on } d]\!] \\
\| \; & [\![\texttt{reqC\_sust} = \texttt{sustain}(\texttt{reqA}, \texttt{ackA}) \texttt{ on } d]\!] \\
\| \; & [\![\texttt{reqD\_sust} = \texttt{sustain}(\texttt{reqA}, \texttt{ackA}) \texttt{ on } e]\!]
\end{aligned}
$$

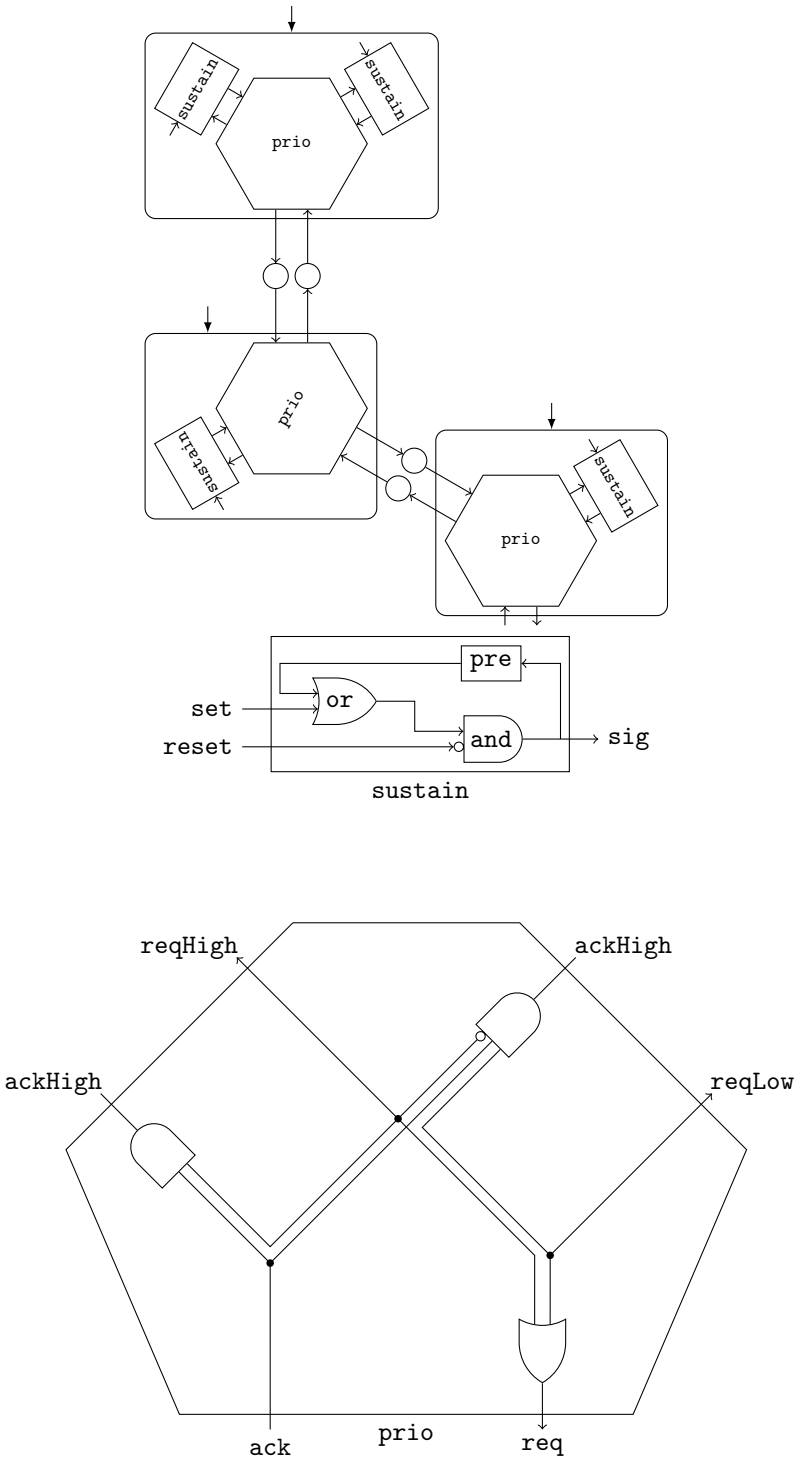


Figure 6.9: Components of the handshake protocol with priority.

The stateful components are treated with the abstract transfer functions introduced in the previous section. That is, the abstract transfer functions that compose the transition system of the component with the concretization of the environment and extracts the drift bounds of the resulting transition system system. The combinational components are converted to conjunctive normal form and represented by the abstract transfer functions that operate directly on the drift bounds. The components' abstract transfer functions are combined with the property mappings (see Section 3.4.2) in a large fix-point computation.

The analysis yields a drift bound for `col` with respect to the root clock that tells us there can be at most 11 collisions per 246 time units and there can be at most 5 consecutive collisions. The example shows that we can decompose systems and analyze the recomposition of their abstractions. While the analysis of labelled transition systems is much more exact, the abstract mappings for combinational circuits are much faster in practice because there is no need to construct the transition system. The result of the analysis, is a conservatively inferred weakly-hard constraint on the number of collisions.

6.5 Conclusion

6.5.1 Related Work

Caspi et al. [CMP01, Cas01] identified the use of quasi-synchronous systems for industrial, distributed embedded systems. In such systems, synchronous programs are composed asynchronously using communication-by-sampling, where independently clocked synchronous programs communicate by reading and writing shared memory. All clocks have the same period with a bounded drift and, consequently, some writes may be overwritten before they are read and some may be read multiple times. They assume the quasi-synchrony hypothesis: a bounded drift such that any pair of clocks can tick at most twice between any two ticks of another. Thus, the number of overwritten and duplicated messages is bounded. The composed system is then verified for its fitness.

A subsequent paper [CMP01] shows that it is sufficient to verify the robustness of synchronous design under distribution. Robustness consists of three system properties: (1) stability, i.e., when the input is unchanging, the output will stabilize; (2) order insensitivity, i.e., any interleaving of activations (triggered by clocks) of different components will lead to the same result; and (3) confluence (of inputs), i.e., the order of changes in the input signal does not affect the result. This method enables a quasi-synchronous, semantics-preserving distribution of robust synchronous systems.

Julien Bertrane continued the verification-based approach in his PhD [Ber08]. He built a tool for the static analysis of quasi-synchronous systems based on abstract interpretation, using a combination of three different, but interacting abstract domains. The analysed systems are described with the synchronous operators of the data-flow language Lustre extended with a non-deterministic, but bounded delay and a sampling operator.

The first domain presented in [Ber05] consists of two kinds of constraints on the value of signals in (continuous) time intervals: one that constrains the signal to have a certain value during the interval and another that requires the signal to take the value at least once during the interval.

In [Ber06] (revisited in [Ber11]) Bertrane adds a local and global changes counting domain that bound the maximum number of changes within either a specific time interval (local) or any time interval of a given size (global). The interaction between local and global changes counting domain, as well as the constraints, improves the precision of the overall analysis. Both in [Ber06] and [Ber11], the analysis proceeds by the computation of the greatest fix-point of the system, proving properties by contradiction if the fix-point has no concrete behavior. The properties themselves can either be expressed directly in the abstract domain (especially in the case of the global changes counting domain) or through a monitor that is added to the verified system.

Wandeler et al. [WT05a] propose an algorithm to derive a quantitative characterization of event-stream generating finite state machines for the purpose of a performance analysis of stream processing systems. Their algorithm that extracts arrival or resource curves is very similar to the algorithm proposed in Section 6.2. The main difference between the two algorithms, due to the relative nature of our drift bounds, is the computation of the transitive and reflexive closure H^*.

In [LPT09] an interface between timed automata and real-time calculus is proposed in order to combine stateful components specified as timed automata with the functional (stateless) stream-processing components. This is achieved by composing the timed-automata of curves, an abstraction of streams similar to drift bounds, with the timed-automaton that defines the component, and then extract the drift bounds of the resulting system. The representation of curves as timed-automata and the extraction of curves from the resulting system are comparable to our concretization and extraction of drift bounds. Apart from the difference between timed-automata and our models, our concretization and abstraction are more general in the sense that we are not constrained to specific forms (e.g. convex) of bounds.

The authors of [AM10] developed a tool that enables the combination of Lustre programs with components in the real-time calculus. The tool translates real-time calculus' curves to an event generator in Lustre and uses existing verification tools for lustre to extract the curves that bound the output. Our drift-bound observer is very similar to their Lustre observers. However, our extraction algorithm extracts the drift bounds directly from the state transition system, whereas they rely on repeated calls of verification tools in a binary-search for the correct bounds. Both methods can yield precise drift bounds.

6.5.2 Discussion

This chapter has presented several important results. First, the verification of safety-properties of sampling networks in an environment constrained by clock bounds through reachability analysis, using drift bound observers. Next, the

inference of drift bounds on the output flow of a sampling network through the construction of weight matrices for the labelled transition system that describes the behavior of the entire system. Finally, a drift bound inference method for sampling networks based on abstract-interpretation where we combine the inference of drift bounds with weight matrices and define specialized abstract mappings for combinational sampling networks that operate directly on drift bounds.

Verification is necessary to validate the design of a safety-critical distributed system. In particular, one needs to ensure that the designed system fulfills its functions even when messages are lost or duplicated due to communication-by-sampling.

The inference of drift bounds can be used for verification of weakly-hard requirements, e.g., if a property may be true at most twice every ten clock ticks, we can infer the drift of that property with respect to the clock to verify the requirement. In this capacity, the main advantage of the inference method over reachability based verification, is the compositionality of the analysis. The inference can, however, also be used during the design phase when the requirements and system parameters such as clock drift are not yet exactly defined, because the inferred drift bounds allow us to evaluate and compare system designs without set limits.

Currently the decomposition of a system is done manually, because the precision of the results of an analysis heavily depends on the chosen decomposition. It should be possible to automate some of the decomposition, for example, by partitioning strongly connected components. Another option would be to start with a full decomposition and gradually compose components until the desired precision has been reached.

Chapter 7

Conclusion

7.1 Recapitulation

We set out to enable the distribution of synchronous system designs by the relaxation of clock constraints of a synchronous program using communication-by-sampling and FIFO channels for (asynchronous) communication across clock domains. Because such a distribution does not preserve the synchronous semantics as messages may be lost or duplicated, we sought to provide methods to analyze the distributed systems.

With the drift bounds both as a measure of relaxation of the clocks and as a way to define weakly-hard requirements, we have shown how to analyze and verify the behavior of such distributed system. Drift bounds have also proven useful to describe the variable availability of resources and the variability of arrival times in streams of messages, leading to an analysis for stream processing systems.

Chapter 2 has shown how the processes of GALS systems can be described with a simple discrete event model and how to extract a relational view with relative counter functions. The relative counter functions provide an interesting view on the events of a GALS system, because they hide real-time, which typically is an uncontrollable (in the sense that we cannot delay or expedite time) or even unobservable (because observations in a GALS occur are quantized by a clock) aspect at runtime, while clearly revealing the interleaving of clock ticks (or of other events). It is, for instance, particularly useful to express communication by sampling as shown in Section 2.3.

Chapter 3 introduced the clock and drift bounds as abstractions of processes. Clock and drift bounds allow high-level reasoning over the processes of GALS systems. Drift bounds in particular have shown to be quite versatile to model (a generalization of) quasi-synchronous clocks as well as weakly-hard requirements. Our bounds generalize similar concepts in literature in a single framework and, by formally introducing the bounds as abstractions with Galois connections, we provide a rigorous framework for further applications and extension.

Chapter 4 proposed a performance analysis for stream-processing systems based on abstract interpretation of stream-processing components. We have shown that our drift and clock bound abstractions allow a more detailed analysis resulting in tighter backlog bounds because our relational abstractions reveal the correlation of events. In addition, the combination of clock and drift bounds allow us to easily model components such as a TDMA resource sharing component.

The sampling networks introduced in Chapter 5 described a programming language to describe GALS boolean data-flow programs by extending a synchronous data-flow language with asynchronous sampling. While the extension is unassuming, the resulting language allows us to easily express asynchronous systems yet remain close to the synchronous approach which was our main goal. Moreover, the sampling primitive is trivial to realize on any architecture that provides memory that is shared by different processes.

Chapter 6 introduced a method for the verification and inference of weakly-hard system invariants in sampling networks where the environment is described by drift bounds. That is, the chapter enables the analysis of distributed systems executed on a platform with quasi-synchronous clocks described by drift bounds and communication-by-sampling. We have shown how to verify properties in such a setting by a classical reachability analysis and how to infer drift bounds on events defined by the sampling network. Because inference of clock bounds can be costly, we suggested different approaches based on the framework of abstractions, to trade precision of the inferred bounds for performance of the analysis. The analyses are applied to examples illustrative of the three classes identified in Section 1.2: a system distributed for robustness (the watchdog protocol in Section 6.1.2), a distributed control system (the thermostat in Section 6.2.3) and a distributed system with weakly-hard constraints (the distributed resource sharing system 6.4.7).

7.2 Implementation

The proposed methods have also been implemented in Python based on a library of operators on ultimately periodic functions, as described in Appendix A, that we developed during the thesis. In contrast to libraries with similar functions, see [BT07, Fid10], is that our library focuses exclusively on functions in the domain of natural numbers rather than positive real numbers. This enables more compact representations and allows an exact implementation of function composition and the pseudo-inverse. In retrospect, it would be possible to modify existing libraries to fit our purposes. Nevertheless, the resulting library is, to our best knowledge, the only library of this nature for Python.

7.3 Discussion

The foundational discrete event model that underlies all of the presented work is no panacea, but we have shown it is nevertheless applicable to many different kinds of systems. The thermostat analyzed in Chapter 6 shows that it can even be used to model systems that are normally modelled with continuous signals, by some simple assumptions that can, once again, be expressed with drift bounds: the room is warm when the heater has been on for more than 2 out of 5 consecutive time units. In any case, there are many possibilities to extend our simple model to include valued discrete events, as is the case in the original tagged-signal model [LSV98], or even non-discrete (continuous) signals such as the ambient temperature over time. The use of abstract domains allows us to extend the model by connecting new abstract domains or new domains that incorporate more detail for which processes are an abstraction.

The choice the relational model provided by the relative counter functions and the abstractions was motivated by the idea that the synchronous components of a GALS system observe its environment at the moments its clock is activated. That is, the relational model provides a view more close to the observations of the digital system. Moreover, we observed that synchronization protocols that run within a clock domain, such as depicted in Figure 1.3 of the Introduction, can only control the relative occurrence of evens. Finally, we showed that it is always possible to introduce at least a discretized notion of real-time as an extra clock signal.

Clock and especially drift have shown to be powerful abstractions for the relevant systems. Drift bounds have turned out to be more useful, because they express a time-invariant abstraction: where clock bounds express constraints on the number of occurrences at each (relative) point in time, the drift bounds allow us to constraint behavior at any point in time. Nevertheless clock and drift bounds complement each other and can be used side-by-side (see Section 3.3.2).

Although drift bounds are very powerful, they often require special monitors to create an additional event on which the bound can be expressed. For example, a bound on the number of changes of a boolean signal — from true to false and vice versa — (a notion similar to the changes counting domain of Bertrane [Ber08]) necessitates an observer that compares the previous value with the current one. That is, the change must be made into an explicit signal that occurs at the time of the change which can then be bounded by drift or clock bounds.

Chapter 6 has shown that the expression of drift bounds as event-labelled transitions is often prohibitively expensive because the state-space grows exponentially with the interval size. The representation of ultimately periodic functions used in the implementation to represent clock and drift bounds (see Appendix A) also grows exponentially in the worst, albeit much less common, case. Approximating drift bounds with convex hulls would enable much compacter and faster implementations, albeit at a loss of precision.

7.4 Perspectives

One of the original goals was to synthesize bounds on the root clocks of a distributed system with communication by sampling such that requirements (expressed by drift bounds on the outputs) are met. A goal shared with [SPS+13] where the authors show how to derive bounds on the number of dropped messages in a distributed control system, leading to a system design with more efficient use of resources while ensuring system correctness. Such a synthesis method would propagate requirements, in the form of drift bounds, backwards through the system from output to inputs and root clocks. As, in general, there are multiple solutions, synthesis must be guided towards a unique solution, e.g., by allowing only select drift bounds on inputs or clocks to be changed. The verification and inference methods for sampling networks presented in this thesis are a necessary step towards synthesis.

We have used a single event model and two abstractions to perform analysis on two types of apparently quite different systems as well as a general analysis for labelled transition systems. One logical next step, would be to analyse heterogeneous systems by combining the analyses of stream-processing systems and sampling networks.

For example, we could model complex resource sharing schemes with sampling networks by interpreting a boolean signal as a resource consumed by a greedy processing component, resulting in a approach similar to [AM10] for network-calculus. The resource sharing system presented in Section 6.4.7 did a step in this direction, except the consumers of the resource were not included in the model.

Another example of the analysis of heterogeneous systems would be to extend sampling networks with FIFO channels for asynchronous communication similar to the buffers of Lucy-N [MP10]. This enables one to use lossless FIFO channels and communication-by-sampling when the most recent information is more important. In such systems, our analysis could be used to determine boundedness of the used buffer-space using the clock bounds. One might also consider bounded FIFO channels with either non-blocking writes (losing messages when the buffer is full) or non-blocking reads (duplicating messages when the buffer is empty) or both to provide for applications where degraded service is tolerated. An extension of our analysis could then be used to verify or infer the provided quality of service.

Finally, there is ample room for improvement in the prototype implementation. In particular the representation of of ultimately periodic drift and clock bounds quickly grows for larger systems. The most promising solution would be to use convex hulls of upper and lower clock and drift bound in the domain of positive rational or real numbers. Convex hulls can be much less precise, e.g., the drift bound $D^u_{i/j}(\Delta) = \lceil 2\Delta/3 \rceil$ that limits the number of events (maximum two) for one interval (interval of three) has a distinct staircase-like shape that is lost by its convex hull. However, convex hulls have very compact representations and allow for very efficient implementations of the used operators (see [BT07, LT01]). Moreover, drift bounds expressed as convex hulls have compact

concretizations representation as labelled transition systems.

Bibliography

[AGA+95] Pascalin Amagbe, Paul Le Guernic, Pascalin Amagbegnon, Loic Besnard, and Paul Leguernic. Implementation of the Data-flow Synchronous Language S IGNAL. 6:163–173, 1995.

[AM10] Karine Altisen and Matthieu Moy. ac2lus: Bringing SMT-Solving and Abstract Interpretation Techniques to Real-Time Calculus through the Synchronous Language Lustre. *2010 22nd Euromicro Conference on Real-Time Systems*, pages 207–216, July 2010.

[BB91] Albert Benveniste and Gerard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[BBL01] Guillem Bernat, Alan Burns, and A Liamosi. Weakly hard real-time systems. *Computers, IEEE Transactions*, 50(4):308–321, 2001.

[BCC+07] Albert Benveniste, Benoit Caillaud, Luca P. Carloni, Paul Caspi, and Alberto L. Sangiovanni-Vincentelli. Composing heterogeneous reactive systems. *ACM Transactions on Computational Logic*, V(N), 2007.

[BCCSV03] Albert Benveniste, L.P. Carloni, Paul Caspi, and Alberto L. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. In *Embedded Software*, pages 35–50. Springer, 2003.

[BCE+03] Albert Benveniste, Paul Caspi, S.a. Edwards, Nicolas Halbwachs, Paul Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[BCG+02] Albert Benveniste, Paul Caspi, P. Guernic, H. Marchand, J.P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In *Embedded Software*, pages 252–265. Springer, 2002.

[BCL99] Albert Benveniste, B. Caillaud, and P Le Guernic. From synchrony to asynchrony. *CONCUR'99 Concurrency Theory*, page 776, 1999.

[BCOQ92] Fancois Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and Linearity, An Algebra for Discrete Event Systems*, volume 52. Wiley, July 1992.

[BELP96] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.

[Ber98] Guillem Bernat. *Specification and Analysis of Weakly Hard Real-Time Systems*. PhD thesis, Universitat de les Illes Balears, 1998.

[Ber05] Julien Bertrane. Static Analysis by Abstract Interpretation of the Quasi-synchronous Composition. In *VMCAI*, pages 97–112, 2005.

[Ber06] Julien Bertrane. Proving the properties of communicating imperfectly-clocked synchronous systems. *Static Analysis*, pages 370–386, 2006.

[Ber08] Julien Bertrane. *Static analysis of communicating imperfectly-clocked synchronous systems using continuous-time abstract domains*. PhD thesis, 2008.

[Ber11] Julien Bertrane. Temporal Abstract Domains. *2011 16th IEEE International Conference on Engineering of Complex Computer Systems*, pages 3–12, April 2011.

[BG92] Gerard Berry and G. Gonthier. The esterel synchronous programming language : design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[BT07] Anne Bouillard and Éric Thierry. An Algorithmic Toolbox for Network Calculus. *DEDS*, 18(1):3–49, October 2007.

[Cas01] Paul Caspi. Embedded control: From asynchrony to synchrony and back. *Lecture Notes in Computer Science*, 2211:80–96, 2001.

[CC77] Patric Cousot and Radia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN*, 1977.

[CC92] P Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 1992.

[CDE+06] Albert Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and Marc Pouzet. N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. In *ACM SIGPLAN Notices*, volume 41, pages 180–193. ACM, 2006.

[Cha84] D.M. Chapiro. *Globally-asynchronous locally-synchronous systems*. PhD thesis, Stanford University, 1984.

[CKT03] Samarjit Chakraborty, S Kunzli, and Lothar Thiele. A general framework for analysing system properties in platform-based embedded system designs. *DATE*, pages 190–195, 2003.

[CMP01] Paul Caspi, Christine Mazuet, and N. Paligot. About the design of distributed control systems: The quasi-synchronous approach. *Computer Safety, Reliability and Security*, (EP 25514):215–226, 2001.

[CMPP08] A. Cohen, L. Mandel, F. Plateau, and Marc Pouzet. Abstraction of clocks in synchronous data-flow systems. *PLS*, pages 237–254, 2008.

[CMSW99] Paul Caspi, Christine Mazuet, Rym Salem, and Daniel Weber. Formal Design of Distributed Control Systems with Lustre. In *Safecomp*, number EP 25514, pages 396–409, 1999.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. LUSTRE: A declarative language for programming synchronous systems. pages 178–188, 1987.

[DP02] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambrige University Press, 2002.

[Fid10] Markus Fidler. Survey of deterministic and stochastic service curve models in the network calculus. *IEEE Communications Surveys & Tutorials*, 12(1):59–86, 2010.

[Flo62] R.W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[HCRP91] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[HHJ+05] R Henia, A Hamann, M Jersak, R. Racu, K Richter, and R Ernst. System level performance analysis – the SymTA/S approach. *CDT*, 152(2):148–166, 2005.

[HKO+93] M.G. Harbour, M.H. Klein, R. Obenza, B. Pollak, and T. Ralya. *A Practitioner's Handbook for Real-Time Analysis*. Kluwer, 1993.

[HP85] D. Harel and A. Pnueli. On the Development of Reactive Systems. *NATO ASI Series, Logics and Models of Concurrent Systems*, F13:477–498, 1985.

[HR99] Nicolas Halbwachs and Pascal Raymond. Validation of synchronous reactive systems: from formal verification to automatic testing. In *Advances in Computing Science—ASIAN'99*, 1999.

[HRR91] N. Halbwachs, Pascal Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Programming Language Implementation and Logic Programming*, volume 3, pages 207–218. Springer, 1991.

[HT07a] W. Haid and Lothar Thiele. Complex task activation schemes in system level performance analysis. In *CODES*, pages 173–178. ACM, 2007.

[HT07b] Kai Huang and Lothar Thiele. Performance analysis of multimedia applications using correlated streams. In *DATE*, pages 912–917. EDA Consortium, 2007.

[HV06] Martijn Hendriks and Marcel Verhoef. Timed automata based analysis of embedded system architectures. In *IPDPS*. IEEE, 2006.

[Kle52] Stephen Cole Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[Kop08] Hermann Kopetz. The Complexity Challenge in Embedded System Design. *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 3–12, May 2008.

[LB87] P. Le Guernic and Albert Benveniste. *Real-Time Synchronous, Data-Flow Programming: The Language" Signal" and Its Mathematical Semantics*. Institut National de Recherche en, en Informatique et en Automatique, 1987.

[LM87] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, June 1987.

[LPT09] Kai Lampka, Simon Perathoner, and Lothar Thiele. Analytic real-time analysis and timed automata: A hybrid method for analyzing embedded real-time systems. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 107–116. ACM, 2009.

[LSV98] E.a. Lee and Alberto L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[LT01] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag, 2001.

[MA10] Matthieu Moy and Karine Altisen. Arrival curves for real-time calculus: the causality problem and its solutions. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 358–372, 2010.

[Mal93] Sharad Malik. Analysis of cyclic combinational circuits. *ICCAD*, 13(7):950–956, 1993.

[MP10] Louis Mandel and Florence Plateau. Lucy-n: a n-Synchronous Extension of Lustre. *Mathematics of Program Construction*, pages 288–309, 2010.

[NNH05] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2005.

[ORE] Geert-jan Olsder, Kees Roos, and Robert-jan Van Egmond. An efficient algorithm for critical circuits and finite eigenvectors in the max-plus algebra. (2):1–9.

[Pet76] C. A. Petri. *Interpretations of net theory*. Gesellschaft für Mathematik und Datenverarbeitung, mbH Bonn, 1976.

[PRT+10] Simon Perathoner, Tobias Rein, Lothar Thiele, Kai Lampka, and Jonas Rox. Modeling structured event streams in system level performance analysis. *LCTES*, 45(4):37, April 2010.

[RE08] Jonas Rox and Rolf Ernst. Modeling Event Stream Hierarchies with Hierarchical Event Models. In *DATE*, pages 492–497. Ieee, March 2008.

[SGL99] Irina M. Smarandache, Thierry Gautier, and Paul Le Guernic. Validation of mixed SIGNAL-ALPHA real-time systems through affine calculus on clock synchronisation constraints. In *FM*, pages 1364–1383, 1999.

[Sif11] Joseph Sifakis. A vision for computer science — the system perspective. *Central European Journal of Computer Science*, 1(1):108–116, March 2011.

[SMB+02] G. Semeraro, G. Magklis, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas, and M.L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pages 29–40, 2002.

[Smi10] Peter Smith. The Galois connection between syntax and semantics. Technical report, University of Cambridge, 2010.

[SPS+13] Damoon Soudbakhsh, Linh T.X. Phan, Oleg Sokolsky, Insup Lee, and Anuradha Annaswamy. Co-design of Control and Platform with Dropped Signals. In *ICCPS*, pages 129–140, 2013.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, June 1955.

[TCN00] Lothar Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *ISCAS*, number March, pages 101–104. IEEE, 2000.

[TGL07] Paul Teehan, Mark Greenstreet, and Guy Lemieux. A Survey and Taxonomy of GALS Design Styles. *IEEE Design & Test of Computers*, 24(5):418–428, September 2007.

[VHR+08] Sriram R. Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, Vasantha Erraguntla, Clark Roberts, Yatin Hoskote, Nitin Borkar, and Shekhar Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, January 2008.

[Wan06] Ernesto Wandeler. *Modular performance analysis and interface-based design for embedded real-time systems*. PhD thesis, ETH Zürich, 2006.

[WT05a] E. Wandeler and L. Thiele. Abstracting functionality for modular performance analysis of hard real-time systems. In *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*, volume 2, pages 697–702. IEEE, 2005.

[WT05b] Ernesto Wandeler and Lothar Thiele. Characterizing workload correlations in multi processor hard real-time systems. In *RTAS*, 2005.

Appendix A

Operations over Non-Decreasing Functions

The principal analysis tool of this thesis are monotonic (non-decreasing) functions over numbers. They are used as counter functions, dater functions, clock and drift bounds, and resource curves.

A.1 Non-Decreasing Functions

Although occasionally functions with a domain or range of real-numbers are used (e.g. when dealing with real-time) this chapter only deals with a representation based on natural numbers, which suffices for our purposes and simplifies the implementation.

Definition 34 (Extended natural numbers ($\mathbb{N}^\infty$))**.** *Let $\mathbb{N}^\infty$ be the set of natural numbers (including 0) extended with ∞, i.e.,*

$$\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$$

The extended set of natural numbers is totally ordered by the usual $\leq$ relation with the least element 0 and greatest element ∞. The extension makes the order complete: any subset of the extended natural numbers has a unique infimum (minimum) and supremum (maximum). Addition of extended natural numbers is defined such that $\infty + n = n + \infty = \infty$ for all $n \in \mathbb{N}^\infty$.

Definition 35 (Non-decreasing function ($\mathcal{F}$))**.** *Let $\mathcal{F}$ be the set of functions $f \in \mathbb{N}^\infty \to \mathbb{N}^\infty$ such that*

- *f is non-decreasing (monotonic) ($n \leq m \implies f(n) \leq f(m)$); and*
- *f goes through the origin ($f(0) = 0$).*

Relative counter functions are represented in $\mathcal{F}$ as functions $f \in \mathcal{F}$ such that $f(n) = 0$ for all $n \leq 0$.

Non-decreasing functions are partially ordered by their point-wise comparison defined by the following extension of the relation $\leq$.

Definition 36 (Partial order of $\mathcal{F}$)**.** *Let f and g be functions in $\mathcal{F}$ then*

$$f \leq g \iff \forall n \in \mathbb{N}^{\infty} : \ f(n) \leq g(n)$$

The partial order of non-decreasing function forms a complete lattice where

- $[\bigvee \mathcal{G}](n) = \sup_{f \in \mathcal{G}} f(n)$ defines the *least upper bound* of $\mathcal{G} \subseteq \mathcal{F}$;
- $[\bigwedge \mathcal{G}](n) = \inf_{f \in \mathcal{G}} f(n)$ defines the *greatest lower bound* of $\mathcal{G} \subseteq \mathcal{F}$;
- $\top(n) = \infty$ for all $n > 0$ is the top (greatest) element in $\mathcal{F}$; and
- $\bot\,(n) = 0$ for all $n \in \mathbb{N}^{\infty}$ is the bottom element.

A.2 Basic Operations

We start with some basic point-wise operations on functions in $\mathcal{F}$ and their properties. Aside from basic properties such as associativity, commutativity and distributivity, we show each operator to be monotonic (order-preserving) and continuous.

A.2.1 Point-Wise Minimum and Maximum

First the point-wise upper and lower bound operators. They differ from the least upper bound and greatest lower bounds only because they are binary operators.

Definition 37 (Point-wise extrema $(\wedge, \vee)$)**.** *Let f and g be functions in $\mathcal{F}$ then*

$$(f \wedge g)(n) = \min(f(n), g(n))$$
$$(f \vee g)(n) = \max(f(n), g(n))$$

The following properties are all trivially inherited from the minimum and maximum operators on natural numbers.

Lemma 21 (Properties of point-wise extrema)**.** *Let $f, g, h \in \mathcal{F}$ denote non-decreasing functions.*

1. *Closed over $\mathcal{F}$: $f \wedge g \in \mathcal{F}$ and $f \vee g \in \mathcal{F}$*
2. *Duality: $f \wedge g = -(-f \vee -g)$*
3. *Idempotent: $f \wedge f = f$ and $f \vee f = f$*
4. *Commutativity: $f \wedge g = g \wedge f$ and $f \vee g = g \vee f$*

5. *Associativity:*
   (a) $(f \wedge g) \wedge h = f \wedge (g \wedge h)$
   (b) $(f \vee g) \vee h = f \vee (g \vee h)$
6. *Zero element:* $f \wedge \perp^{\mathcal{F}} = f$ *and* $f \vee \top^{\mathcal{F}} = f$
7. *Absorbing element:* $f \wedge \top^{F} = \top^{F}$ *and* $f \vee \perp^{F} = \perp^{F}$
8. *Distributivity:*
   (a) $f \wedge (g \vee h) = (f \wedge g) \vee (f \wedge h)$
   (b) $f \vee (g \wedge h) = (f \vee g) \wedge (f \wedge h)$
9. *Order-preserving:* $f \wedge g \leq f' \wedge g'$ *and* $f \vee g \leq f' \vee g'$
   for any $f, f', g, g' \in \mathcal{F}$ *such that* $f \leq f'$ *and* $g \leq g'$

A.2.2 Point-Wise Addition

The point-wise addition of two function consist in adding their results for each element in their domain. The same operators is used for both functions and constants and we may add a constant by writing $f + n$ for some function f in $\mathcal{F}$ and constant n in $\mathbb{N}^{\infty}$.

Definition 38 (Point-wise addition)**.** *Let* f *and* g *be functions in* $\mathcal{F}$ *then*

$$(f + g)(n) = f(n) + g(n)$$

Again, we inherit the properties of the normal addition. Note that $\infty + n = \infty$ for any $n \in \mathbb{N}^{\infty}$.

Lemma 22 (Properties of point-wise addition)**.** *Let* $f, g, h \in \mathcal{F}$ *denote non-decreasing functions.*

1. *Closed over* $\mathcal{F}$*:* $f + g \in \mathcal{F}$
2. *Commutativity:* $f + g = g + f$
3. *Associativity:* $f + (g + h) = (f + g) + h$
4. *Distributivity:*
   (a) $f + (g \vee h) = (f + g) \vee (f + h)$
   (b) $f + (g \wedge h) = (f + g) \wedge (f + h)$
5. *Zero element:* $f + \perp^{F} = f$
6. *Absorbing element:* $f + \top^{F} = \top^{F}$
7. *Order-preserving:* $f + g \leq f' + g'$
   for any $f, f', g, g' \in \mathcal{F}$ *such that* $f \leq f'$ *and* $g \leq g'$

A.2.3 Continuity of Basic Operators

Continuity is important for the application of Kleene's fixpoint theorem (Theorem 4). Operators are only required to be continuous for non-empty chains. Although these results are not new as such we provide the proofs because continuity is neither trivial nor could we find proofs for our operators elsewhere. The following lemma shows that continuity of point-wise operators is inherited from the operators over the natural numbers.

Lemma 23 (Point-wise applications of continuous operators are continuous)**.** *Let $\pi \in \mathbb{N}^\infty \times \mathbb{N}^\infty \to \mathbb{N}^\infty$ denote an operator such that for all $n, m, p, q \in \mathbb{N}^\infty$ and $C \subseteq \mathbb{N}^\infty$ where $C \neq \emptyset$, π is*

1. *limit-preserving: $\pi(0,0) = 0$ and $\pi(\infty,\infty) = \infty$;*
2. *commutative: $\pi(n,m) = \pi(m,n)$;*
3. *monotonic: $n \leq m, p \leq q \implies \pi(n,p) \leq \pi(m,q)$;*
4. *lower semi-continuous with constant: $\sup_{c\in C} \pi(n,c) = \pi(n, \sup_{c\in C})$; and*
5. *upper semi-continuous with constant: $\inf_{c\in C} \pi(n,c) = \pi(n, \inf_{c\in C})$.*

Then the operator $\Pi \in \mathcal{F} \times \mathcal{F} \to \mathcal{F}$, defined by point-wise application of π, i.e., $\Pi(f,g)(n) = \pi(f(n), g(n))$ for all $f, g \in \mathcal{F}$ and $n \in \mathbb{N}^\infty$, is continuous. That is, for all totally ordered, nonempty subsets $C \subseteq \mathcal{F} \times \mathcal{F}$

1. *Π is lower semi-continuous:*

$$\bigvee_{(f,g)\in C} \Pi(f,g) = \Pi\left(\bigvee_{(f,g)\in C} f, \wedge \bigvee_{(f,g)\in C} g\right)$$

2. *Π is upper semi-continuous:*

$$\bigwedge_{(f,g)\in C} \Pi(f,g) = \Pi\left(\bigwedge_{(f,g)\in C} f, \bigwedge_{(f,g)\in C} g\right)$$

Proof. First note that the quantification over the chain can be separated:

$$\bigvee_{(f,g)\in C} \Pi(f,g) = \bigvee_{(f,g')\in C} \bigvee_{(f',g)\in C} \Pi(f,g)$$

for any non-empty C, because on one hand we have

$$\bigvee\{\Pi(f,g) \mid (f,g) \in C\} \leq \bigvee_{(f,g')\in C} \bigvee_{(f',g)\in C} \Pi(f,g)$$

and, on the other hand, we have the reverse inequality, because for any pair $(f, g) \notin C$ such that $(f, g') \in C$ and $(f', g) \in C$, either $(f, g) \leq (f, g')$ (if $(f', g) \leq (f, g')$) or $(f, g) \leq (f', g)$ (if $(f, g') \leq (f', g)$) because C is totally ordered and, by monotonicity of Π, for all such pairs (f, g)

$$\Pi(f, g) \leq \Pi(f \vee f', g \vee g')$$

Then for all $n \in \mathbb{N}^\infty$

$$\begin{aligned}
&[\bigvee_{(f,g')\in C} \bigvee_{(f',g)\in C} \Pi(f, g)](n) \\
&\qquad = \sup_{(f,g')\in C} \sup_{(f',g)\in C} \pi(f(n), g(n)) \\
&\qquad = \sup_{(f,g')\in C} \pi(f(n), \sup_{(f',g)\in C} g(n)) \\
&\qquad = \sup_{(f,g')\in C} \pi(f(n), \sup_{(f',g)\in C} g(n)) \\
&\qquad = [\Pi(\bigvee_{(f,g')\in C} f, \bigvee_{(f',g)\in C} g)](n)
\end{aligned}$$

Upper semi-continuity follows in a similar fashion. □

With this lemma, the point-wise maximum and minimum are trivially shown to be continuous, because the point-wise operations satisfy the listed requirements.

A.3 Function Composition

The composition of functions in $\mathcal{F}$ is defined as a normal function composition.

Definition 39 (Composition ($\circ$))**.** *The composition $f \circ g$ of non-decreasing functions $f, g \in \mathcal{F}$ is defined such that for all $n \in \mathbb{N}^\infty$*

$$(f \circ g)(n) = f(g(n))$$

The properties of function composition are well known.

Lemma 24 (Properties of composition)**.** *Let $f, g, h \in \mathcal{F}$ denote non-decreasing functions.*

1. *Closed over $\mathcal{F}$: $f \circ g \in \mathcal{F}$*
2. *Associativity: $(f \circ g) \circ h = f \circ (g \circ h)$*
3. *Distributivity:*
   (a) $f \circ (g \vee h) = (f \circ g) \vee (f \circ h)$

(b) $(f \vee g) \circ h = (f \circ h) \vee (g \circ h)$

(c) $f \circ (g \wedge h) = (f \circ g) \wedge (f \circ h)$

(d) $(f \wedge g) \circ h = (f \circ h) \wedge (g \circ h)$

4. *Zero element:* $f \circ id = id \circ f = f$
5. *Absorbing elements:* $f \circ \perp^F = \perp^F \circ f = \perp^F$
6. *Order-preserving:* $f \circ g \leq f' \circ g'$
   for any $f, f', g, g' \in \mathcal{F}$ *such that* $f \leq f'$ *and* $g \leq g'$

Proof. Case-by-case

1. Closure: $f(g(0)) = 0$, $f(g(\infty)) = \infty$ and $f \circ g$ is non-decreasing because both f and g are.
2. Associativity follows by expanding the equalities using the definition of function composition.
3. Distributivity is due to the monotonicity of functions in $\mathcal{F}$.
4. Identity is defined to be the zero element of composition, i.e., $id(n) = n$ for all $n \in \mathbb{N}^\infty$.
5. Any function that is constant, i.e., any function $f \in \mathcal{F}$ such that $f(n) = K$ for some $K \in \mathbb{N}^\infty$ is absorbing, which can be seen by expanding definitions.

□

A.3.1 The Pseudo-Inverse

In addition we define the pseudo-inverse for function composition.

Definition 40 (Pseudo-inverse $(\cdot^{-1})$)**.** *The pseudo-inverse* $f^{-1} \in \mathcal{F}$ *for* $f \in \mathcal{F}$ *is defined as*

$$f^{-1}(n) = \inf\{m \mid m \in \mathbb{N}^\infty,\ f(m+1) \geq n\}$$

The pseudo-inverse is derived from the pseudo-inverse of monotonic, but discontinuous functions over real numbers defined, for monotonic functions $g \in \mathbb{R}^\infty \to \mathbb{R}^\infty$:

$$f^{-1}(n) = \inf\{r \mid r \in \mathbb{R}^\infty,\ f(\lceil r \rceil) \geq n\}$$

The pseudo-inverse has many properties reminiscent of the real inverse, but weakened.

Lemma 25 (Properties of pseudo-inverse)**.** *Let* $f, g \in \mathcal{F}$ *denote non-decreasing functions.*

1. *Closed in* $\mathcal{F}$*:* $f^{-1} \in \mathcal{F}$
2. *Pseudo-inverse:*

   (a) $(f^{-1} \circ f)(n) < n$ *for* $n > 0$
   (b) $(f \circ f^{-1}) \leq id$
   (c) $f \circ (f^{-1} + 1) \geq id$

3. *Symmetric:* $[f^{-1}]^{-1} = f$

4. *Distributivity:*

   (a) $[f \wedge g]^{-1} = f^{-1} \vee g^{-1}$
   (b) $[f \vee g]^{-1} = f^{-1} \wedge g^{-1}$

5. *Strictly antitone (order-reversing):*
   $f \leq g \iff f^{-1} \geq g^{-1}$

Proof. Case-by-case

1. Closure: trivially for the extrema $f^{-1}(0) = 0$ and $f^{-1}(\infty) = \infty$ and for $p, q \in \mathbb{N}^\infty$ s.t. $p \leq q$, $\inf\{m \mid m \in \mathbb{N}^\infty,\ f(m+1) \geq p\} \leq \inf\{m \mid m \in \mathbb{N}^\infty,\ f(m+1) \geq q\}$ as $f(m+1) \geq q \geq p$ for all $m \in \mathbb{N}^\infty$.

2. Pseudo-inversion: case-by-case

   (a) By monotonicity of f, i.e.,

$$\begin{aligned}
&(f^{-1}(f(n)) \\
=&\inf\{m \in \mathbb{N}^\infty \mid f(m+1) \geq f(n)\} \\
\leq&\inf\{m \in \mathbb{N}^\infty \mid m+1 \geq n\} \\
<&n
\end{aligned}$$

   (b) Idem dito:

$$\begin{aligned}
&(f(f^{-1}(n)) \\
=&f(\inf\{m \in \mathbb{N}^\infty \mid f(m+1) \geq n\}) \\
\leq&\inf\{f(m) \mid m \in \mathbb{N}^\infty, f(m+1) \geq n\} \\
\leq&\inf\{f(m) \mid m \in \mathbb{N}^\infty, f(m) \geq n\} \\
\leq&n
\end{aligned}$$

   (c)

$$\begin{aligned}
&(f(f^{-1}(n) + 1) \\
=&f(\inf\{m \in \mathbb{N}^\infty \mid f(m+1) \geq n\} + 1) \\
=&\inf\{f(m+1) \mid m \in \mathbb{N}^\infty, f(m+1) \geq n\} \\
\geq&n
\end{aligned}$$

3. Symmetry is shown by proving that $[f^{-1}]^{-1}$ is both larger or equal, and smaller or equal to f (and must therefore be equal). On one hand for all $n \in \mathbb{N}^\infty$

$$\begin{aligned}
&[f^{-1}]^{-1}(n) \\
=&\inf\{m \in \mathbb{N}^\infty \mid f^{-1}(m+1) \geq n\} \\
=&\inf\{m \in \mathbb{N}^\infty \mid m+1 > f(f^{-1}(m+1)) \geq f(n)\} \\
\geq&\inf\{m \in \mathbb{N}^\infty \mid m \geq f(n)\} \\
=&f(n)
\end{aligned}$$

   while at the same time

$$\begin{aligned}
&[f^{-1}]^{-1}(n) \\
=&\inf\{m \mid f^{-1}(m+1) \geq n\} \\
\leq&\inf\{f(m) \mid m \in \mathbb{N}^\infty, f^{-1}(f(m)+1) \geq n\} \\
\leq&\inf\{f(m) \mid m \in \mathbb{N}^\infty, m \geq n\} \\
=&f(n)
\end{aligned}$$

   and therefore $[f^{-1}]^{-1} = f$.

4. Distributivity over point-wise minimum:

$$\begin{aligned}
&[f \wedge g]^{-1}(n) \\
=&\inf\{m \in \mathbb{N}^\infty \mid \min(f(m+1), g(m+1)) \geq n\} \\
=&\inf\{m \in \mathbb{N}^\infty \mid f(m+1) \geq n, g(m+1) \geq n\} \\
=&\inf(\{m \in \mathbb{N}^\infty \mid f(m+1) \geq n\} \cap \{m \in \mathbb{N}^\infty \mid f(m+1) \geq n\}) \\
=&\inf\{m \in \mathbb{N}^\infty \mid f(m+1) \geq n\} \vee \inf\{m \in \mathbb{N}^\infty \mid f(m+1) \geq n\}) \\
=&[f^{-1} \vee g^{-1}](n)
\end{aligned}$$

   Distributivity over point-wise maximum follows analogously.

5. Order-reversing:

$$\begin{aligned}
&f \leq g \\
\implies &\forall n \in \mathbb{N}^\infty :\ f(n) \leq g(n) \\
\implies &\forall m \in \mathbb{N}^\infty :\ f(m+1) \leq g(m+1) \\
\implies &\forall n, m \in \mathbb{N}^\infty :\ n \leq f(m+1) \implies n \leq g(m+1) \\
\implies &\forall n \in \mathbb{N}^\infty :\ \{m \in \mathbb{N}^\infty \mid f(m+1) \geq n\} \subseteq \{m \in \mathbb{N}^\infty \mid g(m+1) \geq n\} \\
\implies &\forall n \in \mathbb{N}^\infty :\ \inf\{m \in \mathbb{N}^\infty \mid f(m+1) \geq n\} \geq \inf\{m \in \mathbb{N}^\infty \mid g(m+1) \geq n\} \\
\implies &\forall n \in \mathbb{N}^\infty :\ f^{-1}(n) \geq g^{-1}(n) \\
\implies &f^{-1} \geq g^{-1}
\end{aligned}$$

□

A.3.2 Continuity of Composition and the Pseudo-Inverse

For completeness, we first show functions in $\mathcal{F}$ to be continuous operators with respect to $\mathbb{N}^\infty$.

Lemma 26 (Continuity of functions in $\mathcal{F}$)**.** *All functions in $\mathcal{F}$ are continuous.*

Proof. Let $C \subseteq \mathbb{N}^\infty$ be a totally ordered nonempty subset, then any $f \in \mathcal{F}$ is

1. upper semi-continuous, because $\inf C = \min C$ and therefore, by monotonicity of f, $\inf_{n\in C} f(n) = f(\inf C)$
2. lower semi-continuous, because if C is finite then $\sup C = \max C$ and therefore, by monotonicity of f, $\sup_{n\in C} f(n) = f(\sup C)$, and if C is infinite then $\sup C = \infty$ and therefore, because f is non-decreasing, either f is constant and $f(\infty) = k = \sup_{n\in C} f(n)$ for some $k \in \mathbb{N}^\infty$, otherwise $f(\infty) = \infty = \sup_{n\in C} f(n)$.

□

Continuity of functions in $\mathcal{F}$ leads to the continuity of composition. Note however, that this is different, than proving the composition of continuous mappings to be continuous (which is also true).

Lemma 27 (Continuity of composition operator)**.** *The composition operator is continuous.*

Proof. Upper semi-continuity follows from the upper semi-continuity of functions in $\mathcal{F}$. That is, for all totally ordered subsets $C \in \mathcal{F} \times \mathcal{F}$ and all $n \in \mathbb{N}^\infty$

$$[\bigwedge_{(f,g)\in C} f \circ g](n) = \inf_{(f,g')\in C} f\left(\inf_{(f',g)\in C} g(n)\right) = [\bigwedge_{(f,g')\in C} f \circ \bigwedge_{(f',g)\in C} g](n)$$

Lower semi-continuity follows similarly. □

A.4 Operators from the Min/Max-Plus Algebra

The max-plus and the related min-plus (also known as the tropical algebra) algebras are defined over natural or real numbers with either the minimum or the maximum in the place of addition and addition in the place of multiplication (see [BCOQ92]). In Section 6.2.2 we quickly introduced these in the context of an algorithm. This section only introduces the convolution in the min-plus and max-plus algebras and their dual, deconvolution operators. These operators have seen many applications in the context of network calculus [LT01]. Our only aim here, is to introduce all used operators. For further information and proofs we refer to [BCOQ92], [LT01], and [BT07].

A.4.1 The Convolution and Deconvolution Operators

Definition 41 (Min-plus convolution ($\otimes$))**.** *The min-plus convolution $f \otimes g$ of functions $f, g \in \mathcal{F}$ is defined such that*

$$(f \otimes g)(n) = \inf\{f(n-\Delta) + g(n) \mid 0 \leq \Delta \leq n\}$$

Lemma 28 (Properties of min-plus convolution)**.** *Let $f, g \in \mathcal{F}$ denote non-decreasing functions and $\mathcal{G} \subseteq \mathcal{F}$ denote a chain.*

1. *Closed over $\mathcal{F}$: $f \otimes g \in \mathcal{F}$*
2. *Idempotent: $f \otimes f = f$ if, and only if,*
   $\forall n, m \in \mathbb{N}^\infty : \ f(n+m) \leq f(n) + f(m)$
3. *Commutativity: $f \otimes g = g \otimes f$*
4. *Associativity: $(f \otimes g) \otimes h = f \otimes (g \otimes h)$*
5. *Distributivity: $f \otimes (g \wedge h) = (f \otimes g) \wedge (f \otimes h)$*
6. *Order-preserving: $f \otimes g \leq f' \otimes g'$*
   for any $f, f', g, g' \in \mathcal{F}$ such that $f \leq f'$ and $g \leq g'$

The deconvolution operator is the dual of the convolution, in the sense that $f \leq g \otimes h$ if, and only if, $f \oslash g \leq h$.

Definition 42 (Min-plus deconvolution ($\oslash$))**.** *The min-plus deconvolution $f \oslash g$ of functions $f, g \in \mathcal{F}$ is defined such that*

$$(f \oslash g)(\Delta) = \sup\{f(n+\Delta) - g(n) \mid n \in \mathbb{N}^\infty\}$$

The deconvolution is not closed over $\mathcal{F}$, but this poses little problems in practice.

Lemma 29 (Properties of min-plus deconvolution)**.** *Let $f, g, h \in \mathcal{F}$ denote monotonic functions then*

1. *Closure: if $f \geq g$ then $f \oslash g \in \mathcal{F}$*
2. *Duality with convolution: $f \leq g \otimes h \iff f \oslash g \otimes h$*
3. *Distributivity:*
   - *over $\vee$: $(f \vee g) \oslash h = (f \oslash h) \vee (g \oslash h)$*
   - *over $\wedge$: $f \oslash (g \wedge h) = (f \oslash g) \vee (f \oslash h)$*
   - *over $\otimes$: $(f \otimes g) \oslash g \leq f \otimes (g \oslash g)$*
4. *Composition: $(f \oslash g) \oslash h = f \oslash (g \otimes h)$*
5. *Order-preserving: $f \otimes g \leq f' \otimes g'$ for any $f, f', g, g' \in \mathcal{F}$ such that $f \leq f'$ and $g \geq g'$*

A.4.2 Sub-additive Closure

Drift-bounds are sub-additive operators (see Lemma 11). Sub-additivity can be stated in a number of different ways.

Lemma 30 (Sub-additivity)**.** *The following statements are equivalent:*

1. $f(n+m) \leq f(n) + f(m)$ *for all* $n, m \in \mathbb{N}^\infty$
2. $f \otimes f = f$
3. $f \oslash f = f$

The sub-additive closure of $f \in \mathcal{F}$ yields the greatest sub-additive function in $\mathcal{F}$ less than or equal to f. It is defined by the fix-point of convolution.

Definition 43 (Sub-additive closere (f^*))**.** *The sub-additive closure* f^* *is defined such that*

$$f^* = \top^F \wedge f \wedge (f \otimes f) \wedge (f \otimes f \otimes f) \wedge ... = \bigwedge_{n \in \mathbb{N}} f^n$$

Lemma 31 (Properties of sub-additive closure)**.** *Let* $f, g \in \mathcal{F}$ *denote monotonic functions then*

1. *Sub-additivity:* $f^* \otimes f^* = f^*$
2. *Sub-additive closure:* $g \leq f,\ g \otimes g = g \implies g \leq f^*$
3. *Distributivity:* $[f \wedge g]^* = [f \otimes g]^* = f^* \otimes g^*$

A.4.3 The Dual Max-Plus Operators

The max-plus operators can be defined as a dual (with negation as the inversion) of the min-plus operators (although not in the domain of $\mathcal{F}$). As such, they are have very similar properties. We only define the operators, see [LT01], [Wan06] and [BT07] for more information.

Definition 44 (Max-plus convolution ($\overline{\otimes}$))**.** *The max-plus convolution* $f \overline{\otimes} g$ *of functions* $f, g \in \mathcal{F}$ *is defined such that*

$$(f \overline{\otimes} g)(n) = \sup\{f(n-\Delta) + g(n) \mid 0 \leq \Delta \leq n\}$$

Definition 45 (Max-plus deconvolution ($\overline{\oslash}$))**.** *The max-plus deconvolution* $f \overline{\oslash} g$ *of functions* $f, g \in \mathcal{F}$ *is defined such that*

$$(f \overline{\oslash} g)(\Delta) = \inf\{f(n+\Delta) - g(n) \mid n \in \mathbb{N}^\infty\}$$

Definition 46 (Super-additive closere ($f^{\overline{*}}$))**.** *The super-additive closure* f^* *is defined such that*

$$f^{\overline{*}} = \bot^F \vee f \vee f \overline{\otimes} f \vee f \overline{\otimes} f \overline{\otimes} f \vee ...$$