A Synchronous Approach to Quasi-Periodic Systems

Guillaume Baudart

THÈSE DE DOCTORAT
de l'Université de recherche Paris Sciences Lettres
PSL Research University

Préparée à l'École normale supérieure

A Synchronous Approach to Quasi-Periodic Systems

Une Approche Synchrone des Systèmes Quasi-Périodiques

ED 386 SCIENCES MATHÉMATIQUES DE PARIS CENTRE
Spécialité INFORMATIQUE

COMPOSITION DU JURY :

M. Albert Benveniste
Inria Rennes
Co-encadrant

M. Timothy Bourke
Inria Paris
Encadrant

M. Nicolas Halbwachs
Vérimag
Rapporteur

M. Marc Pouzet
École normale supérieure et UPMC
Directeur de thèse

M. Xavier Rival
Inria Paris
Président

M. Alberto Sangiovanni-Vincentelli
University of California Berkeley
Examinateur

M. Cesare Tinelli
University of Iowa
Rapporteur

Soutenue le 13 Mars 2017
par **Guillaume BAUDART**

Dirigée par Marc POUZET

ENS ÉCOLE NORMALE SUPÉRIEURE | PSL RESEARCH UNIVERSITY PARIS

Pour Lucile,
la plus belle personne du monde.

Merci !

I would like to warmly thank Nicolas Halbwachs and Cesare Tinelli. I admire their work, and it is a great honor for me that they both accepted to review this manuscript. I am also very grateful to Xavier Rival and Alberto Sangiovanni-Vincentelli for their participation to my committee.

Je voudrais remercier Marc Pouzet qui m'a accueilli en stage de master sur un sujet étrange, un peu en dehors des thématiques habituelles de l'équipe. Cette rencontre m'a permis de découvrir dans un cadre idéal les langages synchrones qui me fascinent encore aujourd'hui. J'ai ensuite eu la chance de pouvoir poursuivre en thèse sous sa direction pendant trois années incroyablement enrichissantes. L'élégance et l'originalité de ses travaux ont été une grande source d'inspiration pour les développements présentés dans cette thèse. Je voudrais également le remercier pour sa disponibilité et ses conseils. J'ai beaucoup apprécié nos longues séances de travail qu'il arrivait à inclure dans son agenda chargé.

Timothy Bourke a été un encadrant exceptionnel. Il a su admirablement composer avec mon caractère riche et contrasté tout au long de ma thèse. J'ai adoré nos séances de rédaction à quatre mains où nous avons pu débattre pendant des heures de la place d'un mot, d'une virgule, ou même d'un simple espace. Merci Tim pour ton incroyable disponibilité. Merci d'avoir toujours été à l'écoute pendant ces trois années. J'espère que nous aurons encore de nombreuses occasions de nous disputer.

Au début de ma thèse, j'ai contacté Albert Benveniste pour discuter d'un de ses articles paru des années plus tôt. Cette discussion a été le point de départ d'une collaboration à qui je dois tous les développements sur les LTTAs. Grâce à Albert, j'ai également eu la chance d'entrer en contact avec des industriels qui ont grandement contribué à éclaircir ma vision (jusqu'alors un peu théorique) des systèmes embarqués. Je voudrais le remercier pour ses conseils, sa bienveillance, son ouverture d'esprit, et son impressionnante créativité.

À la fin de ma thèse j'ai eu la chance de pouvoir effectuer un stage à IBM Research. Merci à toute l'équipe du projet CloudLens: Louis Mandel, Olivier Tardieu, et Mandana Vaziri. J'ai adoré ces quelques mois de stage et vous y êtes pour beaucoup. Merci Louis pour tous ces bons moments passé à New-York. Merci Jérôme pour Harlem et les chats.

Merci à tous les membres de l'équipe Parkas pour tous ces repas et pauses cafés partagés. Merci aux vénérables anciens Léonard Gérard, Adrien Guatto[1], et Cédric Pasteur. Merci

[1]Adrien a été un énorme soutien sur le plan scientifique et humain tout au long de ma thèse. Je lui dois beaucoup et j'espère que nous aurons encore de nombreuses occasions de discuter d'obscurs points techniques à la terrasse de cafés, d'un côté ou de l'autre de l'océan.

Louis pour ReactiveML et tout ce qui va avec. Merci Ulysse, co-bureau exemplaire, pour les ravioles.

J'ai eu la chance de travailler avec un personnel administratif aussi redoutablement efficace que sympathique. Merci à Anna Bednarik, Lise-Marie Bivard, Isabelle Delais, Joëlle Isnard, Sophie Jaudon, Valérie Mongiat, et Assia Saadi.

Merci à tous mes amis: Thibault qui a trop souvent subi mes divagations, Pierre (Félicitation !) qui est toujours là après tant d'années, Antoine le nouveau Prince de Mineapolis, Mathilde et ses cailloux, Benjamin la fripouille, Paulin et Anaïs et leur bonne humeur communicative, Xavier colloc légendaire, les atiamiens Camille et Quentin, Élise et Tanguy (et Sandro), Morgan et Salomé (et Andréa), Agathe et Clément, Maxence, Marguerite et Paul, Agnès. J'espère tous vous revoir bientôt de l'autre côté de l'atlantique.

J'ai une pensée spéciale pour mes parents, Pôpa et Môman, et mes frères Clément, Nicolas, Vincent, et Joseph, qui me soutiennent depuis toujours (même quand je leur ai annoncé que je ferais de l'informatique). Merci aussi à toute la famille Nême/Debray qui m'a si gentiment adopté il y a maintenant quelques années.

Enfin merci Lucile de rendre la vie si agréable, amusante, et fascinante.

Abstract

In this thesis we study embedded controllers implemented as sets of unsynchronized periodic processes. Each process activates quasi-periodically, that is, periodically with bounded jitter, and communicates with bounded transmission delays. Such reactive systems, termed *quasi-periodic*, exist as soon as two periodic processes are connected together. In the distributed systems literature they are also known as synchronous real-time models. We focus on techniques for the design and analysis of such systems without imposing a global clock synchronization.

Synchronous languages were introduced as domain specific languages for the design of reactive systems. They offer an ideal framework to program, analyze, and verify quasi-periodic systems. Based on a synchronous approach, this thesis makes contributions to the treatment of quasi-periodic systems along three themes: verification, implementation, and simulation.

Verification: The *quasi-synchronous abstraction* is a discrete abstraction proposed by Paul Caspi for model checking safety properties of quasi-periodic systems. We show that this abstraction is not sound in general and give necessary and sufficient conditions on both the static communication graph of the application and the real-time characteristics of the architecture to recover soundness. We then generalize these results to multirate systems.

Implementation: *Loosely time-triggered architectures* are protocols designed to ensure the correct execution of an application running on a quasi-periodic system. We propose a unified framework that encompasses both the application and the protocol controllers. This framework allows us to simplify existing protocols, propose optimized versions, and give new correctness proofs. We instantiate our framework with a protocol based on clock synchronization to compare the performance of the two approaches.

Simulation: Quasi-periodic systems are but one example of timed systems involving real-time characteristics and tolerances. For such nondeterministic models, we propose a *symbolic simulation* scheme inspired by model checking techniques for timed automata. We show how to compile a model mixing nondeterministic continuous-time and discrete-time dynamics into a discrete program manipulating sets of possible values. Each trace of the resulting program captures a set of possible executions of the source program.

Keywords Embedded systems; Synchronous real-time distributed systems; Synchronous languages; Quasi-synchronous abstraction; Loosely time-triggered architectures; Symbolic simulation.

Résumé

Cette thèse traite de systèmes embarqués contrôlés par un ensemble de processus périodiques non synchronisés. Chaque processus est activé quasi-périodiquement, c'est-à-dire périodiquement avec une gigue bornée. Les délais de communication sont également bornés. De tels systèmes réactifs, appelés *quasi-périodiques*, apparaissent dès que l'on branche ensemble deux processus périodiques. Dans la littérature, ils sont parfois qualifiés de systèmes distribués temps-réels synchrones. Nous nous intéressons aux techniques de conception et d'analyse de ces systèmes qui n'imposent pas de synchronisation globale.

Les langages synchrones ont été introduits pour faciliter la conception des systèmes réactifs. Ils offrent un cadre privilégié pour programmer, analyser, et vérifier des systèmes quasi-périodiques. En s'appuyant sur une approche synchrone, les contributions de cette thèse s'organisent selon trois thématiques: vérification, implémentation, et simulation des systèmes quasi-périodiques.

Vérification: *L'abstraction quasi-synchrone* est une abstraction discrète proposée par Paul Caspi pour vérifier des propriétés de sûreté des systèmes quasi-périodiques. Nous démontrons que cette abstraction est en général incorrecte et nous donnons des conditions nécessaires et suffisantes sur le graphe de communication et les caractéristiques temps-réels de l'architecture pour assurer sa correction. Ces résultats sont ensuite généralisés aux systèmes multi-périodiques.

Implémentation: Les *LTTAs* sont des protocoles conçus pour assurer l'exécution correcte d'une application sur un système quasi-périodique. Nous proposons d'étudier les LTTA dans un cadre synchrone unifié qui englobe l'application et les contrôleurs introduits par les protocoles. Cette approche nous permet de simplifier les protocoles existants, de proposer des versions optimisées, et de donner de nouvelles preuves de correction. Nous présentons également dans le même cadre un protocole fondé sur une synchronisation d'horloge pour comparer les performances des deux approches.

Simulation: Un système quasi-périodique est un exemple de modèle faisant intervenir des caractéristiques temps-réels et des tolérances. Pour ce type de modèle non déterministe, nous proposons une *simulation symbolique*, inspirée des techniques de vérification des automates temporisés. Nous montrons comment compiler un modèle mêlant des composantes temps-réels non déterministes et des contrôleurs discrets en un programme discret qui manipule des ensembles de valeurs. Chaque trace du programme résultant capture un ensemble d'exécutions possibles du programme source.

Mot-clés Systèmes embarqués; Systèmes distribués temps-réels synchrones; Langages synchrones; Abstraction quasi-synchrone; Architectures LTTA; Simulation symbolique.

Contents

Chapter 1

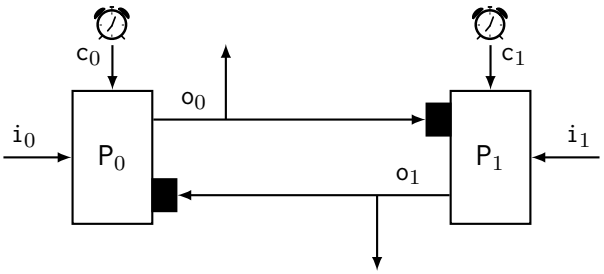
Introduction

A quasi-periodic system exists as soon as two periodic processes are connected together. They are everywhere: flight control systems in aircraft, control loops in factories or power plants, and distributed control in vehicles. Figure 1.1 illustrates a two-node quasi-periodic system and the code of the control loop executed by each process.

Each process P_k is triggered by a periodic clock c_k. The code of the process is executed inside an infinite loop. A new iteration starts at each tick of the clock c_k. Before the loop, the internal state s_k is initialized. At each iteration of the loop, a process reads the inputs i_k from the environment and the output o_{1-k} of the other process to update its state and produce the output o_k. The function *step* is a placeholder for the body of the code executed by the process. This function is typically compiled from specifications written in high-level languages.

Since a process is triggered by local clocks, it only has a local notion of time. Without explicit synchronization their intercommunication is subject to sampling artifacts. One possibility for avoiding them is to rely on a clock synchronization protocol as in the Time-Triggered Architecture (TTA) [KB03]. It is then possible to treat a quasi-periodic system as if all the processes were triggered by a single global clock. Perhaps surprisingly, it is also possible to design and analyze such systems without imposing a global synchronization.

This approach originates in the work of Paul Caspi who noticed, while consulting at Airbus, that for historical reasons, 'in the domain of critical control systems, the use of clock synchronization is not so frequent' [Cas00, §1]. Instead, Airbus engineers relied on




```
/* Process P_k */

initialize(s_k);
while true do
  o_k, s_k := step(s_k, i_k, o_{1-k});
  wait(c_k);
done
```

Figure 1.1: A two-node quasi-periodic system. The two processes are triggered by their local clocks and execute the control loop shown on the right.

a set of design *good practices* that Paul Caspi started to formalize in what he called the *quasi-synchronous approach to distributed systems*. As already noted by Caspi in the original report, the so called 'cooking book' [Cas00]:

> This raises the question of the techniques used in these systems for simulating, validating, and implementing fault tolerance without clock synchronization, and for the well-foundedness of these techniques which equip, up to now, some of the most safety critical control systems ever designed.

The work in this thesis began in an attempt to precisely understand Caspi's report. It led us to consider quasi-synchronous systems from three related points of view:

- **Verification:** How to precisely model and verify safety properties, that is, checking that nothing bad ever happens, of a system implemented as a set of unsynchronized processes?
- **Implementation:** How to ensure the correct execution of an embedded application running on a quasi-periodic system despite the absence of synchronization?
- **Simulation:** How to capture the possible behaviors of the system with simulations before implementation?

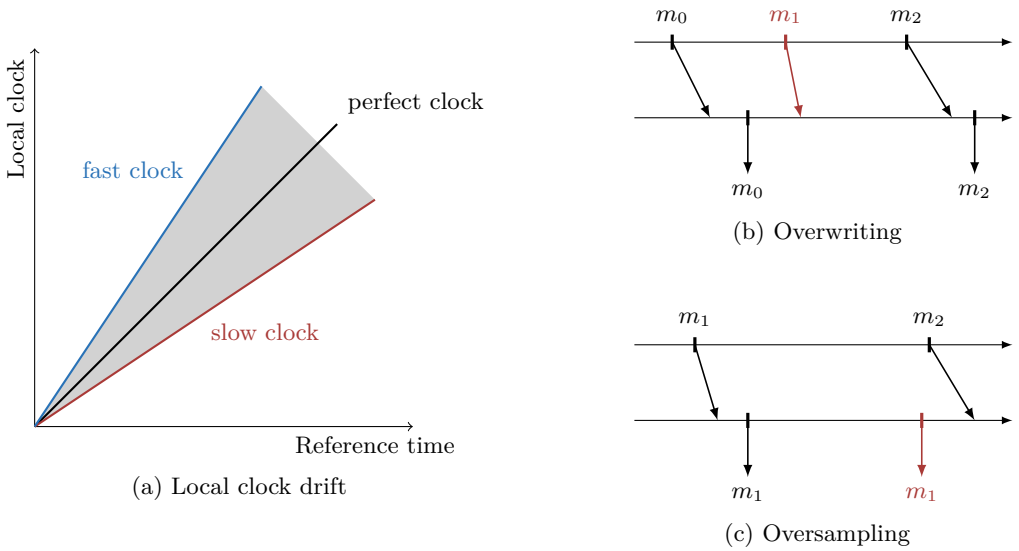
1.1 Quasi-periodic systems

Designing controllers running on distributed architectures is notoriously difficult. The quasi-synchronous approach thus focuses on particular architectures characterized by the following assumptions.

Quasi-periodicity Each process is periodically activated by its own local clock. These clocks are imperfect—subject to jitter—and unsynchronized. However, in the context of embedded systems the system must be predictable. This is especially the case for critical embedded systems, for instance, the fly-by-wire system found in aircraft We thus assume that each process executes *quasi-periodically*, that is, the delay between two successive activations is bounded by known constants. As shown in figure 1.2a, this assumption does not prevent clock values from drifting arbitrarily apart during execution. For instance, one process can always execute as quickly as possible while another executes as slowly as possible.

Reliable transmissions For the same reasons, we focus on systems with a reliable communication network: transmission delays are always bounded with known constants and there is no loss or reordering of messages during transmission.

Blackboard communication Since processes are not synchronized, it is almost never the case that a message is received by a process precisely when it activates. Processes thus communicate using *blackboards*: messages are stored in local memories that are only sampled when the receiver activates, no message queues are allowed and a message written in the memory remains there until it is overwritten by a newer one. This communication scheme is often used as a robust alternative to bounded buffers which require control mechanisms



(a) Local clock drift

(b) Overwriting

(c) Oversampling

Figure 1.2: Quasi-periodic systems. During execution local clock may drift apart (a). Blackboard communications are subject to loss (b) or duplication of data (c) depending on the last received value when a process is activated.

ensuring that buffer overflows cannot occur. Blackboards avoid these control dependencies between processes at the cost of introducing sampling artifacts. For instance, data may be lost—a value can be overwritten before being read (figure 1.2b)—or duplicated—a process may twice sample the same value (figure 1.2c).

These assumptions are very general and not difficult to satisfy. They thus potentially apply to many systems. Embedded controllers implemented on these *quasi-periodic* architectures are termed *quasi-periodic systems* and are the central focus of this thesis.

1.2 A synchronous approach

Synchronous languages [BCE+03] were introduced in the late 80s as *domain specific languages* for the design of reactive systems. A synchronous program executes in a succession of discrete steps. A programmer writes high level specifications in the form of stream functions specifying variable values at each step. This approach is reminiscent of block diagrams, a popular notation to describe control systems.

Specific compilation techniques for synchronous languages exist to generate efficient code for embedded controllers. Compilers produce imperative code that can be executed in a control loop like the one presented in figure 1.1. In addition, dedicated formal verification tools have been developed to check program properties. One example is the language Lustre [CPHP87] which is the backbone of the industrial language and compiler Scade.[1] Scade is now routinely used to program embedded controllers in many critical applications.

[1] `www.esterel-technologies.com/products/scade-suite`

In this thesis, we focus on quasi-periodic systems where multiple synchronous programs (one for each process) execute in parallel but are not synchronized. We propose to reason about the entire system in a synchronous framework. We can thus benefit from the mathematically precise semantics offered by synchronous languages to reason about the global system, and rely on well-understood compilation techniques to generate the embedded code of processes. Compared to other modeling techniques, this approach reduces the gap between the model and the executable code.

The main question is then: *How to model the asynchronous behavior of quasi-periodic systems in a synchronous framework?* In particular, when modeling and simulating a complete quasi-periodic system, the timing constraints of the underlying architecture are of paramount importance. We thus look for extensions of the synchronous approach that allow the inclusion of real-time characteristics in the model.

Abstracting real-time specifications as discrete predicates A classic solution consists in abstracting from the real-time behavior of the architecture using discrete-time predicates like: 'A process never activates twice between two activations of another'. Both the discrete-time controllers and the predicates capturing their real-time behavior can be expressed as synchronous programs. This allows the use of the existing verification tools for synchronous programs to check safety properties of the model. For instance, we can check that the system never reaches an *alarm* state.

The main limitation of this approach is that we must ensure that the discrete-time abstraction captures all possible behaviors of the real-time system. Otherwise, one cannot assume that safety properties that hold for the discrete model also hold for the real system. Designing sound discrete abstractions is complex and error prone.

Models mixing continuous and discrete time Another approach is to rely on tools that allow the modeling of systems with complex interactions between discrete-time and continuous-time dynamics. For instance, Simulink/Stateflow[2] or Ptolemy.[3]

In this thesis, we choose to model systems using Zélus,[4] a synchronous language extended with continuous time that is developed in our team. Zélus is a conservative extension of a Lustre-like synchronous language to which it adds the ability to define continuous-time dynamics by means of ordinary differential equations and state events. In Zélus it is possible to design and implement discrete-time controllers and model their physical environment in the very same language.

1.3 Contributions

This thesis makes contributions to the treatment of quasi-periodic systems along three themes: verification, implementation, and simulation. In each, using a synchronous approach, we clarify existing work and propose new developments.

Verification The quasi-synchronous abstraction [Cas00] is a discrete abstraction proposed by Paul Caspi for model-checking the safety properties of quasi-periodic systems. Logical steps

[2] `http://mathworks.com/products/simulink.html`
[3] `http://ptolemy.eecs.berkeley.edu/`
[4] `http://zelus.di.ens.fr/`

account for transmission delays and the quasi-periodic behavior of processes is captured by the following condition: no process may be activated more than twice between two successive activations of any other. We show that the abstraction is not sound for general systems of more than two nodes. Interestingly, the main difficulty comes from the modeling of transmissions as *unit delays*, that is, as single logical steps of the discrete model. We introduce the notion of *unitary discretization* that characterizes traces for which transmission delays can be safely abstracted as unit delays. This notion links the causality induced by communications in the real-time traces with the causalities expressible in the discrete model. We are then able to give necessary and sufficient conditions on both the static communication graph of the application and the real-time characteristics of the architecture to recover soundness. These results generalize naturally to multirate systems where each process is characterized by its own activation period. The quasi-synchronous abstraction thus becomes n/m-quasi-synchrony which states that a process cannot activate more than n times between m activations of another.

Implementation Introduced in 2002 [BCLG$^+$02], Loosely Time-Triggered Architectures or LTTAs are protocols designed to ensure the correct execution of an application running on a quasi-periodic architecture despite the sampling artifacts introduced by communications between unsynchronized processes. Noticing that protocol controllers are also synchronous programs that can be compiled together with application code, we propose a unified synchronous framework that encompasses both the application and the protocol controllers. This framework can be instantiated with any of the LTTA protocols: *back-pressure* [TPB$^+$08] and *time-based* [CB08]. We thereby give executable specifications of the protocols written in the kind of languages typically used to program embedded controllers. We show that both of these protocols can be expressed as two-state automata. Additionally, we present new correctness proofs of the protocols, give a simpler version of the time-based protocol and show that it requires broadcast communication. Based on this remark, we propose optimized protocols for system using broadcast communication. Finally, we instantiate our framework with a simple protocol based on clock synchronization to compare its performance with the LTTA approach. This shows that there is no gain in performance when using the LTTA protocols. However, the LTTA protocols are simple to implement and remain a lightweight alternative to clock synchronization. They add the minimum amount of control to ensure the correct execution of an embedded application.

Simulation The specifications of quasi-periodic systems involve real-time characteristics and tolerances: the bounds on the activation periods and the transmission delays. The continuous dynamics of such systems are limited to *timers* that measure time elapsing but also involve nondeterminism. For this kind of system we propose a symbolic simulation scheme—inspired by model checking techniques for timed automata [AD94]—where multiple executions are captured in a single discrete trace and nondeterminism is controlled by the user. Each step is characterized by a set of possible values for the timers and a set of enabled actions. Simulation advances when the user chooses a transition. Starting from a small synchronous language extended with timers and nondeterministic constructs, we show how to adapt typing and modular compilation techniques developed for Zélus to generate discrete synchronous code for symbolic simulation.

1.4 Organization

The body of this thesis comprises two introductory chapters and three distinct technical chapters. Related work is described and discussed throughout.

Chapter 2 The first chapter is a brief introduction to Zélus, a synchronous language reminiscent of Lustre [CPHP87] extended with continuous time. The chapter aims to familiarize the reader with the syntax and key features of the language that is used in the rest of the thesis. In particular we show how discrete components activate on the emissions of signals produced by continuous components modeling a physical environment. As an illustration of the modeling possibilities offered by Zélus, this chapter concludes with a complete model of an old fashioned clock.

Chapter 3 The second chapter introduces quasi-periodic architectures. We precisely define these architectures and describe the sampling artifacts introduced by communications between unsynchronized nodes. Then we present a synchronous model of quasi-periodic architectures written in the discrete part of Zélus. The link between discrete and real time is realized through input signals for the clocks of the nodes and their delayed versions that model non-instantaneous transmissions. Using the continuous part of Zélus we then show how to implement continuous components producing these signals according to the real-time characteristics of the architecture. This gives a complete executable specification for quasi-periodic systems that can be used for testing and simulation. Our modeling approach is not specific to Zélus and we show how to adapt it to other modeling languages.

Chapter 4 The first technical chapter focuses on the quasi-synchronous abstraction proposed by Paul Caspi. We first show how this abstraction allows the models presented in chapter 3 to be simplified. Then, we prove that the abstraction is not sound in general and give necessary and sufficient conditions to recover soundness. Finally, we extend our results to multirate systems. This chapter is based on the article [BBP16].

Chapter 5 The second technical chapter is dedicated to loosely time-triggered architectures. We first show how the model of chapter 3 can be refined to capture *middleware* that ensures the correct execution of the embedded application. This synchronous framework is instantiated with the two historical LTTA protocols (back-pressure and time-based) an optimized protocol for systems using broadcast communication, and a simple protocol based on clock synchronization. For each of these protocols we give a correctness proof and its theoretical worst-case performance. Using the Zélus model described in chapter 3, we simulate all protocols under various parameter values to compare their performance. This chapter is based on the articles [BBB15] and [BBB16].

Chapter 6 The last technical chapter presents a symbolic simulation scheme for systems involving nondeterministic real-time specifications. We motivate our approach with the example of a simple quasi-periodic system. Then we present a kernel synchronous language extended with *timers* to measure time elapsing and nondeterministic constructs and show how to adapt the typing and compilation of Zélus to generate discrete code for symbolic simulation.

Chapter 7 The last chapter concludes this thesis with a summary of our results and presents possibilities for future research.

The source code presented throughout the thesis can be compiled and executed in version 1.2.3 of Zélus.[5] This code can be downloaded from:

`http://guillaume.baudart.eu/thesis`.

[5] `http://zelus.di.ens.fr/download.html`

Related publications

[BBP16] Guillaume Baudart, Timothy Bourke, and Marc Pouzet. Soundness of the quasi-synchronous abstraction. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 9–16, USA, October 2016

[BBB16] Guillaume Baudart, Albert Benveniste, and Timothy Bourke. Loosely Time-Triggered Architectures: Improvements and comparison. *Transactions on Embedded Computing Systems*, 15(4):71:1–71:26, August 2016 Extended journal version of [BBB15]

[BBB15] Guillaume Baudart, Albert Benveniste, and Timothy Bourke. Loosely Time-Triggered Architectures: Improvements and comparisons. In *International Conference on Embedded Software (EMSOFT)*, pages 85–94, The Netherlands, October 2015. Best paper nominee

[BBBC14] Guillaume Baudart, Albert Benveniste, Anne Bouillard, and Paul Caspi. A unifying view of Loosely Time-Triggered Architectures. Technical Report RR-8494, INRIA, March 2014. Corrected version of [BBC10]

Chapter 2

A Brief Introduction to Zélus

In this thesis, we make a significant effort to illustrate our contributions with actual code. This discipline forces a high degree of precision in the models. Additionally, this approach gives both formal models and executable specifications that can be simulated.

Although the main contributions of this thesis are not specific to any programming language, we present all the examples in a single language developed in our team: Zélus [BP13]. We choose Zélus for three main reasons:

1. Zélus is a synchronous language in the spirit of Lustre [CPHP87] and Scade. We can thus program the quasi-periodic processes in the kind of programming language typically used to implement embedded applications.
2. The language provides constructs to mix discrete controllers and continuous-time dynamics expressed as Ordinary Differential Equations (ODE). We are thus able in chapter 3 to precisely model the real-time specifications of a quasi-periodic system and program the discrete processes in a single, precisely defined, and executable language.
3. Compilation of the continuous components is based on source-to-source translations into discrete functions with additional inputs and outputs to handle continuous computations. We show in chapter 6 how to adapt this compilation technique to the symbolic simulation of nondeterministic real-time systems.

The compilation of a Zélus program mixing discrete and continuous components produces code that must be executed in interaction with a numeric solver. During the simulation, the solver computes an approximation of the continuous-time expressions defined with ODEs. Compared to the control loop presented in figure 1.1 where discrete steps are triggered by an external clock, in Zélus discrete computations are triggered when the solver detects particular events: *zero-crossings* of continuous-time expressions.

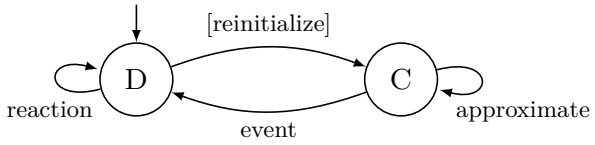


Figure 2.1: [BP13] The basic structure of a Zélus simulation.

As illustrated in figure 2.1, a simulation alternates between two phases. In phase D, discrete components are executed and physical time does not progress. When the computation terminates, the program enters phase C and integration in the numeric solver begins. In phase C, the numeric solver approximates the evolution of continuous-time variables to detect and locate zero-crossing events. The integration stops when such an event has been detected. The program then returns to phase D and the next discrete step is triggered.

A Zélus program is compiled to statically scheduled sequential OCaml[1] code. The semantics of the language, based on non-standard analysis, is described in [BBCP12]. Details on the compilation process can be found in [BBCP11a, BBCP11b, BCP+15].

Outline This chapter is a *Zélus survival kit.*[2] It aims to familiarize the reader with the basic syntax and the key features of the language that we use in the following chapters. We illustrate in section 2.1 the discrete part of the language inherited from Lustre. Then in section 2.2 we show how to express continuous dynamics using ODEs, and how to activate discrete components on events generated by continuous ones. Finally in section 2.3 we illustrate this hybrid modeling approach on a complete example: a model of an old-fashioned clock.

2.1 A synchronous language ...

Zélus is a synchronous language reminiscent of Lustre [CPHP87] and borrows constructs from Lucid Synchrone [Pou06]. A program is a set of equation defining *streams* of values. Time proceeds by discrete logical steps, and at each step, a program computes the value of each stream depending on its inputs and possibly previously computed values.

For instance, the following function returns the stream of natural numbers starting from an initial value `v`:

```
let node nat(v) = cpt where
  rec cpt = v → pre (cpt + 1)
```

The keyword `node` indicates a discrete stream function. The only equation uses the initialization operator → and the non initialized *unit delay* `pre`. It defines the variable `cpt` as follows: $\forall n \in \mathbb{N}$,

$$
\begin{aligned}
cpt_n &= (v \rightarrow \texttt{pre}\ (cpt + 1))_n && \text{(Definition)}\\
&= \mathit{if}\ n = 0\ \mathit{then}\ v_0\ \mathit{else}\ (\texttt{pre}\ (cpt + 1))_n && \text{(Initialization)}\\
&= \mathit{if}\ n = 0\ \mathit{then}\ v_0\ \mathit{else}\ (cpt + 1)_{n-1} && \text{(Unit delay)}\\
&= \mathit{if}\ n = 0\ \mathit{then}\ v_0\ \mathit{else}\ cpt_{n-1} + 1 && \text{(Simplification)}
\end{aligned}
$$

The execution of a program can be represented as a timeline, called a *chronogram*, showing the sequence of values taken by its streams at each step. For instance, applying the `nat` function to the constant stream of 10's yields the execution:

`v`	10	10	10	10	10	10	10	10	...
`pre cpt`	*nil*	10	11	12	13	14	15	16	...
`cpt`	10	11	12	13	14	15	16	17	...

[1] `http://ocaml.org`

[2] More details can be found at `http://zelus.di.ens.fr`.

The delay operator `pre` has an unspecified value (denoted *nil*) at the first instant. The compiler performs an *initialization check* to ensure that the behavior of a program never depends on the value *nil*. Alternatively, the initialized delay operator `fby` combines initialization and delay operators: x `fby` $y = x \rightarrow$ `pre` y.

When given `nat`, the Zélus compiler returns the following type signature:

val nat: int $\xrightarrow{\text{D}}$ *int*

The '$\xrightarrow{\text{D}}$' indicates a discrete stream function. The compiler infers that `nat` produces values of type *int* from an input of type *int*.

Operators—like `+`, `*`, or the logical `and`—and constants—like `1`, `true`, or `4.2`—are lifted to streams of values.

`1`	1	1	1	1	1	1	1	1	...
`true`	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	...
`1+2`	3	3	3	3	3	3	3	3	...

For some functions, like arithmetic operators, the output at an instant only depends on the inputs at the same instant. These functions are termed *combinatorial*.

```
let average(x, y) = (x + y) / 2
```

val average: int $\times$ *int* $\xrightarrow{\text{A}}$ *int*

The '$\xrightarrow{\text{A}}$' in the type signature stands for *any* and indicates a combinatorial function. Such functions can be used in any context: discrete or continuous (see section 2.2).

Valued signals Compared to Lustre, Signal [GGBM91], or Lucid Synchrone, Zélus does not have explicit *clocks* to indicate the presence or absence of a value. Instead, the language has *valued signals*, built and accessed through the constructions `emit` and `present`, to model sporadic activations. Consider, for instance, the following program:

```
let node positive(i) = s where
  rec present i(v) → do emit s = (v > 0) done
```

val positive: int signal $\xrightarrow{\text{D}}$ *bool signal*

Whenever signal `i` is present, its value is bound to `v` and signal `s` is emitted with value `(v > 0)`. A signal is absent if not explicitly emitted.

Memories A signal can be used to update a *memory*, declared by the keyword `init`. The value of a memory is maintained between two updates. The following example returns the sum of the values received on signal `i`.

```
let node cumul(i) = o where
  rec init o = 0
  and present i(v) → do o = last o + v done
```

val cumul: int signal $\xrightarrow{\text{D}}$ *int*

The operator `last(.)` refers to the value of a memory at the last update.

i			10		2			30	...
last o	0	0	0	10	10	12	12	12	...
o	0	0	10	10	12	12	12	42	...

Automaton Complicated behaviors are often best described as automata whose defining equations at an instant are mode-dependent. An automaton is a collection of states and transitions. There are two kinds of transitions: *weak* (`until`) and *strong* (`unless`).

Consider the following example.

```
let node edge_strong(x) = o where
  rec automaton
      | Wait → do o = false unless (x = 0) then Found
      | Found → do o = true done
```

val edge_strong: int $\xrightarrow{\mathrm{D}}$ *bool*

Starting in state `Wait`, the output `o` is defined by the equation `o = false` while the condition `(x = 0)` is *false*. At the instant that this condition is *true*, `Found` becomes the active state and the output is thereafter defined by the equation `o = true`.

x	3	1	2	0	−1	0	...
o	*false*	*false*	*false*	*true*	*true*	*true*	...

Weak transitions introduce a delay between the instant when a transition guard becomes *true* and the instant when the mode changes. Consider the same automaton written with weak transitions.

```
let node edge_weak(x) = o where
  rec automaton
      | Wait → do o = false until (x = 0) then Found
      | Found → do o = true done
```

val edge_weak: int $\xrightarrow{\mathrm{D}}$ *bool*

Applied to the same input stream, we get:

x	3	1	2	0	−1	0	...
o	*false*	*false*	*false*	*false*	*true*	*true*	...

2.2 ... extended with continuous time

Zélus combines two models of time: *discrete* and *continuous*. Continuous-time functions are introduced by the keyword `hybrid` and continuous dynamics are expressed with ordinary differential equations.

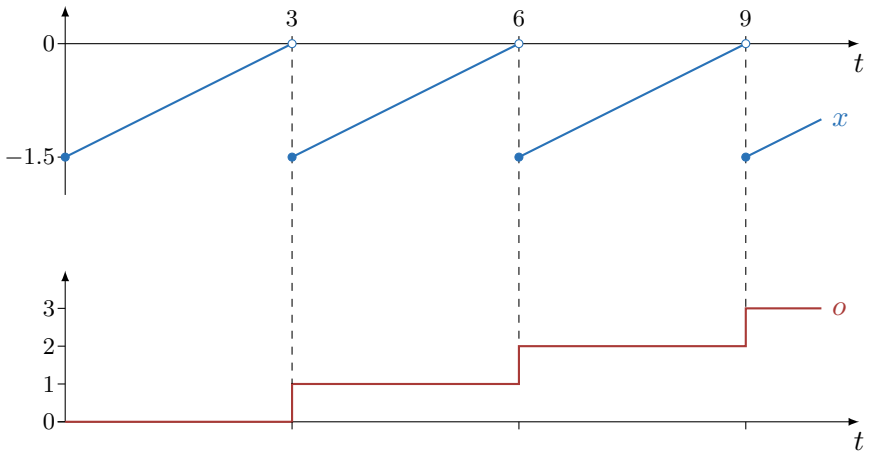


Figure 2.2: Simulation trace of the function `sawtooth(0.5, -1.5)`.

Consider the following Zélus program that implements the initial value problem shown on the right. The continuous-time variable `x` is defined by an ODE and an initial condition.

```
let hybrid affine(a, b) = x where
  rec der x = a init b
```

val affine: float $\times$ *float* $\xrightarrow{C}$ *float*

$$\begin{cases} \dot{x}(t) = a \\ x(0) = b \end{cases}$$

The '$\xrightarrow{C}$' in the type signature indicates a continuous time function. The keyword `der` defines `x` by its derivative and initial value. The ideal value of `x` produced by `affine` is:

$$x(t) = b + \int_0^t a \, dx$$

During execution this value is approximated by a numeric solver.

Zero-crossings Discrete computations are triggered when the solver detects particular events: zero-crossings of continuous-time variables. In Zélus, (rising) zero-crossings are monitored using the `up` operator. For instance, the following program triggers the `nat` function of section 2.1 on the zero-crossings of a sawtooth signal.

```
let hybrid sawtooth(a, b) = o where
  rec init o = 0
  and der x = a init b reset z → b
  and z = up(x)
  and present z → do o = nat(1) done
```

val sawtooth: float $\times$ *float* $\xrightarrow{C}$ *int*

A memory `o` is initialized with value 0. At zero-crossing instants—when the value of `x` computed by the numeric solver passes through zero from a negative value to a positive one—`x` is reset to `b` and the new value of `o` is computed. Otherwise the last computed value is maintained. Figure 2.2 shows a simulation trace of this program.

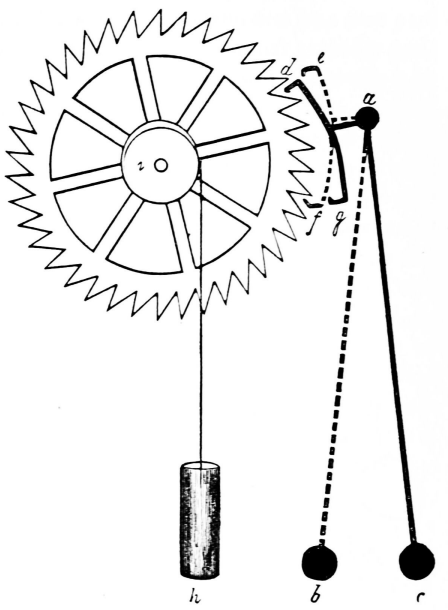


Figure 2.3: [Mat86] Gravity clock escapement mechanism aided by weight.

2.3 A complete example: the Zélus clock

In this section we present a complete example: the Zélus clock. First we show how to implement a physical model of an old-fashioned pendulum clock in Zélus. The clock generates a regular signal `tick` that we use to trigger a discrete stopwatch. This example illustrates how to mix the continuous dynamic of the pendulum clock with the discrete control of the stopwatch.

A pendulum clock

Figure 2.3 illustrate the simplest form of clock work or *movement*. A cog-wheel turns on a pin by the force of the suspended weight. The position of the weight h is a function of its initial position h_0, and, without any additional control, follows a simple ODE.

```
let hybrid weight(h0) = h where
  rec der hd = -.g init 0.0
  and der h = hd init h0
```

val weight: float $\xrightarrow{C}$ *float*

$$\begin{cases} \ddot{h}(t) = -g \\ \dot{h}(0) = 0 \\ h(0) = h_0 \end{cases}$$

where g is the acceleration of gravity ($g \approx 9.8\, m.s^{-2}$ on Earth) and `+.`, `-.`, `*.`, `/.` denote floating-point operations. The hands of the clock are attached to the wheel (or, more probably, another wheel coupled with the main one) and move with the weight.

The *escapement* mechanism controls the fall of the weight. It comprises two elements: a *pendulum* and an *anchor* attached to the pendulum and swinging with it. Assume that the pendulum starts in the position r (marked in black) in figure 2.3. In this position, the anchor blocks the wheel. When the pendulum starts swinging to the position b (marked by the dotted line), the anchor moves, releasing the wheel which is then dragged down by the weight. But before it moves too far, the anchor catches the wheel and stops the movement of the weight.

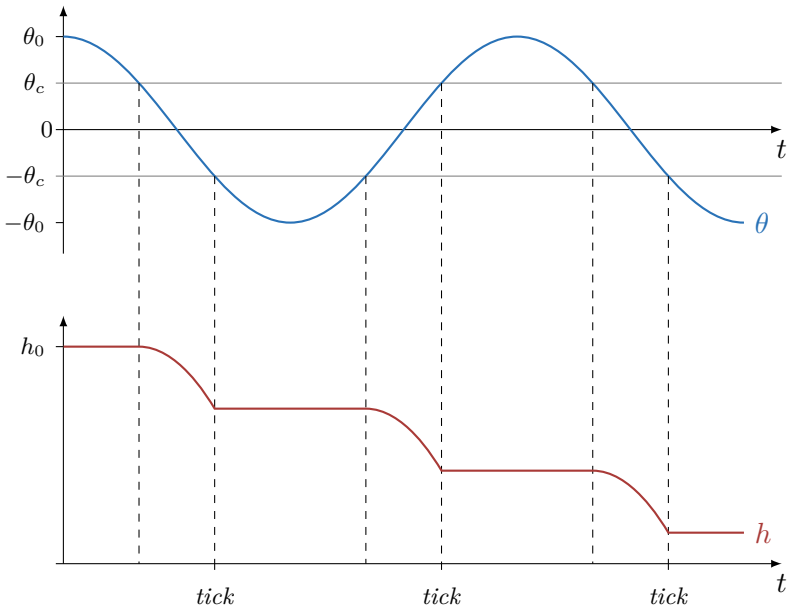


Figure 2.4: Simulation trace of the function `clock`.

The discontinuous movement of the wheel produces a regular signal that can be used to measure time elapsing (the ticks of the clock). In other words, the escapement mechanism discretizes the continuous fall of the weight. In an old-fashioned watch, the weight is replaced by a circular spring, and the pendulum by a small oscillating spring-mass system, but the principle is the same.

The position of the pendulum is a function of its (constant) length l and initial angle θ_0, following the well-known pendulum equation.

```
let hybrid pendulum(theta0) = theta where
  rec der td =  -. (g /. l) *. sin (theta) init 0.0
  and der theta = td init theta0
```

val pendulum: float $\xrightarrow{C}$ *float*

$$\begin{cases} \ddot{\theta}(t) = -(g/l)\sin(\theta(t)) \\ \dot{\theta}(0) = 0 \\ \theta(0) = \theta_0 \end{cases}$$

The escapement mechanism blocks the movement of the weight when the absolute value of the angle $|\theta|$ is greater than a value θ_c, characteristic of the anchor. This behavior is implemented in Zélus by the following two-state automaton.

```
let hybrid clock(h0, theta0) = tick where
  rec theta = pendulum(l, theta0)
  and automaton
      | Block(hi) → do h = hi
          until up(thetac -. abs(theta)) then Move(h)
      | Move(hi) → do h = weight(hi)
          until up(abs(theta) -. thetac) then do emit tick in Block(h)
      init Block(h0)
```

val clock: float $\times$ *float* $\xrightarrow{C}$ *unit signal*

The state `Block` is parametrized by the position of the weight. This position is maintained until $|\theta|$ (`abs(theta)`) crosses the value θ_c. Then the escapement frees the weight and the clock enters state `Move`. The state `Move` is also parametrized by the initial position of the weight. In this state, the wheel is free and the movement of the weight is controlled by the function `weight`. When $|\theta|$ (`abs(theta)`) crosses the value θ_c the escapement blocks the wheel (producing the characteristic *tick* sound of the clock). The clock emits a signal `tick` and goes back to state `Block` with the current position of the weight `h` (`do emit tick in Block(h)`). This is a simple example of actions triggered on the transitions of an automaton. The automaton starts in state `Block` (`init Block(h0)`) with the initial position of the weight h_0. Figure 2.4 shows a simulation trace of this program.

A discrete stopwatch

A simple stopwatch can be implemented with the hierarchical automaton illustrated in figure 2.5. The user controls the stopwatch with two buttons: `toggle` to start and stop the stopwatch, and `restart` to reinitialize the controller. The output is the number of clock ticks emitted when the stopwatch is running. The automaton of figure 2.5 is readily programmed in Zélus.

```
let node stopwatch(toggle, restart) = t where
  rec automaton
      | Main →
          do automaton
             | Idle → do t = 0
                 until toggle() then Run(0)
             | Run(ti) → do t = ti fby (t + 1)
                 until toggle() then Stop(t)
             | Stop(ti) → do t = ti
                 until toggle() then Run(t)
             end
          until restart() then Main
```

val stopwatch : unit signal × *unit signal* $\xrightarrow{\mathrm{D}}$ *int*

Initially, in the `Idle` state, the output of the stopwatch `t` is always 0. The user can activate the stopwatch by pressing the `toggle` button. The stopwatch then enters the parametrized state `Run` with value 0. In state `Run`, the output is incremented by one at every instant starting from the value of the parameter `ti`. In this state, it is possible to pause the stopwatch by pressing the `toggle` button. The stopwatch then enters the parametrized state `Stop` where the value of the output is maintained until the user presses the `toggle` button to restart the counter. In any state, it is possible to press a `restart` button to go back to the `Idle` state, thus reinitializing the stopwatch.

The complete model

The last thing to do is to link the stopwatch controller to the ticks of the pendulum clock. Since the user can choose to press the buttons `toggle` and `restart` at any moment, the ticks of the clock (signal `tick`) and the events *button pressed* are not synchronized a priori. We thus start by defining a global discrete clock to trigger the stopwatch by taking the union of the emissions of all these signals. In Zélus we write `present tick() | toggle() | restart()`. Then, by definition, at each step of the base clock, signals `toggle` and `restart` are either absent or

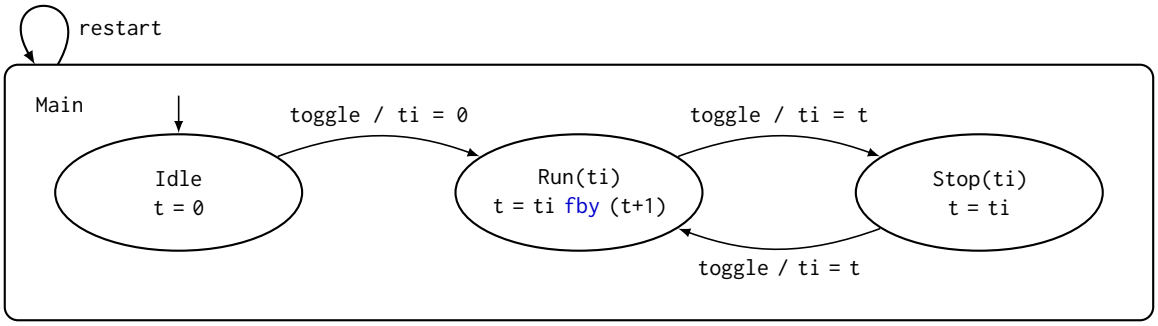


Figure 2.5: A simple stopwatch controller.

present. These continuous signals can thus be treated as discrete signals and used as input for the discrete stopwatch controller.

The complete Zélus model is then:

```
let hybrid chrono(toggle, restart) = t where
  rec init t = 0
  and tick = clock(h0, theta0)
  and present tick() | toggle() | restart() → do t = stopwatch(toggle, restart) done
```

val chrono: unit signal $\times$ *unit signal* $\xrightarrow{C}$ *int*

A memory `t` is initialized with value 0. Signal `tick` is produced by the physical model of the pendulum clock `clock`. At each step of the base clock, the new value t is computed by the discrete controller `stopwatch`.

We finally have a complete model of a simple old-fashioned chronometer (in the void, since we do not consider energy loss in our simple model). A more faithful physical model would integrate energy loss and mechanisms to ensure isochronous oscillations of the pendulum [Huy73].

2.4 Conclusion

In this chapter we presented Zélus, a synchronous language extended with continuous time, that we chose for the implementations presented in this thesis. In Zélus it is possible to write discrete controllers in a language reminiscent of Lustre, and to model the continuous dynamics of the environment with ODEs.

The compilation of a Zélus program produces code that must be executed in interaction with a numeric solver. During an execution, the solver approximates the continuous-time variables defined with ODEs and monitors particular events: zero-crossings of continuous-time variables. Such events can be used to trigger discrete computations.

We presented the syntax and key features of Zélus on elementary examples. Then we illustrated the modeling possibilities offered by Zélus with a physical model of an old-fashioned clock coupled with a discrete stopwatch.

Using Zélus, we are able in chapter 3 to propose a discrete model of a quasi-periodic architecture and to link it with its real-time characteristics.

Chapter 3

Quasi-Periodic Architectures

Introduced in [Cas00], the *quasi-synchronous approach* is a set of techniques for building distributed control systems. It is a formalization of practices that Paul Caspi observed while consulting in the 1990s at Airbus, where engineers were deploying Lustre/SCADE [HCRP91] designs onto networks of non-synchronized nodes communicating via shared memories with bounded transmission delays.

In contrast to the *Time-Triggered Architecture* (TTA) [Kop11], the quasi-synchronous approach does not rely on clock synchronization. Processors execute periodically with the same nominal period and communicate via a reliable network, but the activation periods of the processors and communication delays are subject to jitter. We call this a *quasi-periodic architecture*. This is a classic model, also called *synchronous real-time model* in the distributed systems community [ADLS94, Cri96]. Such systems arise naturally as soon as two or more microcontrollers running periodic tasks are interconnected. They are common in aerospace, power generation, and railway systems.

This chapter presents the quasi-periodic architectures that are central to this thesis. The aim of this chapter is to provide a global framework to describe embedded applications running on such architectures that can adapted and refined in the following chapters. We adopt a classic approach of modeling distributed systems using a discrete synchronous formalism and give a complete discrete model of the architecture: computing nodes, communication media, and delayed transmission.

Traditionally, the link between the discrete mode and real time is 'pushed outside' the model: real-time constraints are modeled with additional inputs—like activation signals or regular clock signals—and the model does not formally provide any information on how to produce these inputs. We, however, implement our model in Zélus—introduced in chapter 2—which allows us to express both the discrete synchronous model and the real-time constraints of the architecture in an executable language.

Outline In section 3.1 we define the concept of a quasi-periodic architecture. Running an application on such an architecture introduces sampling artifacts that are described in section 3.2. Then, in section 3.3, we show how to model quasi-periodic architectures in a discrete synchronous formalism. Using Zélus, we link this discrete model to the real-time characteristics of the architecture in section 3.4. Finally, in section 3.5, we show how to adapt our approach to other modeling tools, namely, Ptolemy and Simulink.

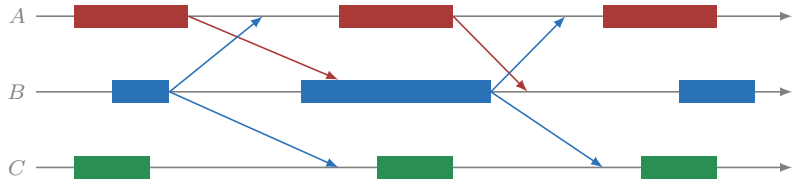


Figure 3.1: Example of a real-time trace of a quasi-periodic architecture with three processors. Rectangles represent tasks and arrows denote message transmissions. Note the jitter both on node activation periods and transmission delays.

3.1 Definition

In this section we give an abstract model of quasi-periodic architectures. The goal is to account for all sources of timing nondeterminism—hardware, operating system, communication network—in one simple model.

Traces of distributed systems are typically described as a set of tasks and transmissions. Figure 3.1 shows an example of such a trace. In our model, we assume that individual processes are synchronous: reactions triggered by a local clock execute in zero time (atomically with respect to the local environment). The only events are thus processor activations and message transmissions.

To abstract from the execution time, the easiest solution—illustrated in figure 3.2—is to consider instantaneous activations and capture the execution time as part of the communication delay. This abstraction imposes that a processor cannot receive a message during the execution of a task. For instance, in figure 3.1, the first message sent by processor A is only read by processor B at its third activation. It is, however, safe to assume that a message received exactly when a consumer activates can be read immediately.

For each processor, an execution is now an infinite sequence of instantaneous activations triggered by a local clock.

Definition 3.1 (Quasi-periodic architecture)**.** *A* quasi-periodic architecture *is a finite set of processors, or nodes $\mathcal{N}$, where*

1. *every node $n \in \mathcal{N}$ executes almost periodically, that is, the actual time between any two successive activations $T \in \mathbb{R}$ may vary between known bounds during an execution,*

$$0 \leq T_{\min} \leq T \leq T_{\max}, \tag{RP}$$

2. *values are transmitted between processes with a delay $\tau \in \mathbb{R}$, bounded by $\tau_{\min}$ and $\tau_{\max}$,*

$$0 \leq \tau_{\min} \leq \tau \leq \tau_{\max}. \tag{RT}$$

A quasi-periodic system can also be characterized by its nominal period $T_{\text{nom}} = (T_{\min} + T_{\max})/2$ and maximum jitter ε, where $T_{\min} = T_{\text{nom}} - \varepsilon$ and $T_{\max} = T_{\text{nom}} + \varepsilon$, and similarly for the transmission delay.

We assume without loss of generality that all nodes start executing at $t = 0$. Initial phase differences between nodes can be modeled by a succession of *mute* activations before the actual start of the system. The margins encompass all sources of divergence between nominal and actual values, including relative clock jitter, interrupt latencies, and scheduling delays.

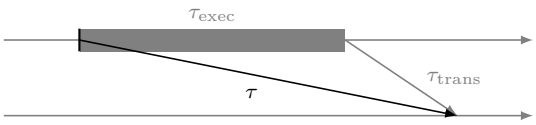


Figure 3.2: We abstract a task as an instantaneous activation and a communication delay τ. The communication delay τ encompasses both the execution time τ_{exec} and the transmission delay τ_{trans}: $\tau = \tau_{\text{exec}} + \tau_{\text{trans}}$.

3.2 Communication by Sampling

Messages sent by nodes are stored at receivers in local memories which are updated atomically. A producer can send the same message to several receivers. A memory is only sampled when the corresponding node is activated by its local clock. There is no synchronization between producers and receivers since local clocks are unsynchronized. This communication model is called *Communication by Sampling* (CbS) [BCDN$^+$07].

To illustrate this communication scheme one imagine a sleepy student in a class room. Periodically, the student wakes up and reads the blackboard. Then he falls asleep until he wakes up and reads the board again. In this example, there is no synchronization between the teacher and the student. During a nap, information written on the board may or may not change. Due to this analogy, this simple communication scheme is sometimes called *blackboard communication* [Ber89, §3].

Finally we assume a reliable communication network, that is, the network guarantees message delivery and preserves message order. If a message m_1 is sent before m_2 then m_2 is never received before m_1. This is necessarily the case when $\tau_{\max} \leq T_{\min} + \tau_{\min}$. Otherwise, with a lossless network, it is always possible to number messages and stall those that arrive out of sequence.

Value duplication and loss

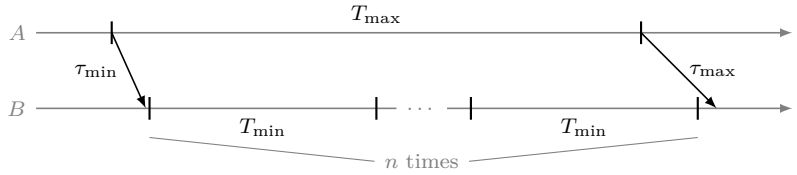
The lack of synchronization means that successive variable values may be duplicated or lost. For instance, if a consumer of a variable is activated twice between the arrivals of two successive messages from the producer, it will *oversample* the buffered value. On the other hand, if two messages of the producer are received between two activations of the consumer, the second value *overwrites* the first which is then never read.

These effects occur for any non-zero jitter ε, regardless of how small. The timing bounds of definition 3.1 mean, however that the maximum numbers of consecutive oversamplings and overwritings are functions of the bounds on node periods and transmission delays ($\forall x \in \mathbb{R}$, $\lceil x \rceil$ denotes the smallest integer i such that $x \leq i$).

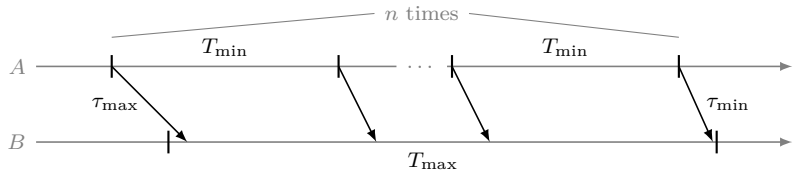
Property 3.1. *Given a pair of nodes executing and communicating according to definition 3.1, the maximum number of consecutive oversamplings and overwritings is*

$$n_{os} = n_{ow} = \left\lceil \frac{T_{\max} + \tau_{\max} - \tau_{\min}}{T_{\min}} \right\rceil - 1. \tag{3.1}$$

Proof. Consider a pair of nodes A and B with B receiving messages from A. In the best case, illustrated in figure 3.3a, a message sent by A at time t arrives in B's shared memory at $t + \tau_{\min}$. Then if A runs as slowly as possible the next message is sent at $t + T_{\max}$ and



(a) Each execution of B samples the first message sent by A.



(b) All messages sent by A are overwritten by the last one.

Figure 3.3: Witnesses for the maximum number of oversamplings (a) and overwritings (b)

arrives in B's shared memory at worst at $t + T_{\max} + \tau_{\max}$. The maximal delay between two successive arrivals is thus

$$T_{\max} + \tau_{\max} - \tau_{\min}.$$

At best, B is activated every $T_{\min}$. The maximum number of executions n of B is thus:

$$nT_{\min} < T_{\max} + \tau_{\max} - \tau_{\min}.$$

Each execution of B that occurs between the two arrivals samples the last received value. The maximum number of oversamplings $n_{os} = n - 1$ is thus given by equation (3.1).

The proof for the number of consecutive overwritings, illustrated in figure 3.3b, is similar. □

Property 3.1 implies that data loss can be prevented by activating a consumer more frequently than the corresponding producer. This can be achieved by introducing *mute* activations of the receiver at the cost of higher oversampling. Quasi-periodic architectures involving such producer-consumer pairs are studied in [BCLG+02].

Quasi-periodic architectures are a natural fit for continuous control applications where the error due to sampling artifacts can be computed and compensated for. On the other hand, discrete control systems, like state machines, are generally intolerant to data duplication and loss.

Signal combinations

There is another obstacle to implementing discrete applications on a quasi-periodic architecture: naively combining variables can give results that diverge from the reference semantics. Consider the classic example of figure 3.4 [Cas00, CB08, BBC10]. A node C reads two boolean inputs a and b, produced by nodes A and B, respectively, and computes the conjunction, $c = a \wedge b$. The first two lines of figure 3.4 show the content of the local memories of node C corresponding to variables a and b.

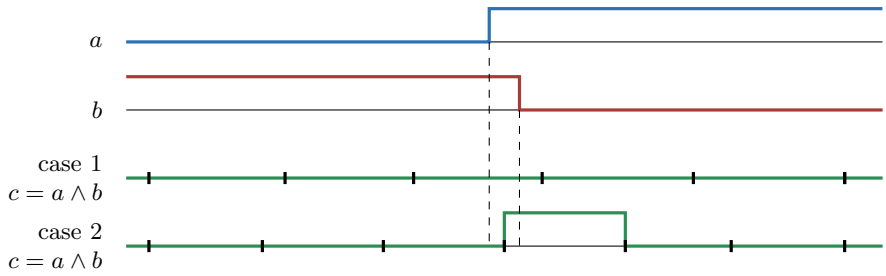


Figure 3.4: The effect of sampling on signal combinations

Suppose that a is *false* for three activations of A before becoming *true* and b is *true* for three activations of B before becoming *false*. In a synchronous semantics, with simultaneous activations of A, B, and C, node C should return *false* at each activation. But, as figure 3.4 shows, since nodes are not synchronized, it is probable that the values of a and b do not change at exactly the same time. There is thus a small interval where both a and b are *true*.

If node C does not sample the content of the memories during this interval, the output is always *false*, as expected (case 1). On the other hand, if node C does sample the content of the memories during this interval, the output c is set to *true* for one period of C (case 2).

In other words, the outputs of this simple application depend on the relative activations of nodes A, B, and C. This phenomenon cannot be avoided by changing the frequency of node activations.

Buffers We consider here one-place buffers, but the previous remarks can be generalized to bounded buffers of arbitrary size. Since nodes are not synchronized it is impossible to ignore the case of producer/consumer pairs where the producer runs as quickly as possible, that is every $T_{\min}$, and the consumer runs as slow as possible, that is every $T_{\max}$. As soon as the nodes are not perfectly synchronous, that is, $T_{\min} < T_{\max}$, the difference between the cumulative tick counts of nodes can diverge without bound. Without any additional synchronization mechanism it is thus impossible to bound the size of the buffers while ensuring that no data will be lost or oversampled.

Properties of quasi-periodic architectures such as the maximal number of lost values (property 3.1), message inversions, and end-to-end message latency have been verified with the proof assistant PVS [LS14]. Ad-hoc abstract domains have also been designed for the static analysis of quasi-periodic architectures [Ber08]. These domains can be used to check quantitative properties such as *How many times a value changed during a time interval* or *Two redundant computing units must agree more than half of the time*. These works are based on the real-time model of definition 3.1. We propose to reason about these quasi-periodic architectures in a discrete-time model. In the following we present the discrete model, and show how Zélus allows us to relate the discrete-time with the real-time characteristics of the architecture.

3.3 Discrete model

One of the central ideas in the original quasi-synchronous approach is to replace a model with detailed timing behavior by a discrete model [Cas00, §3]. An embedded application running

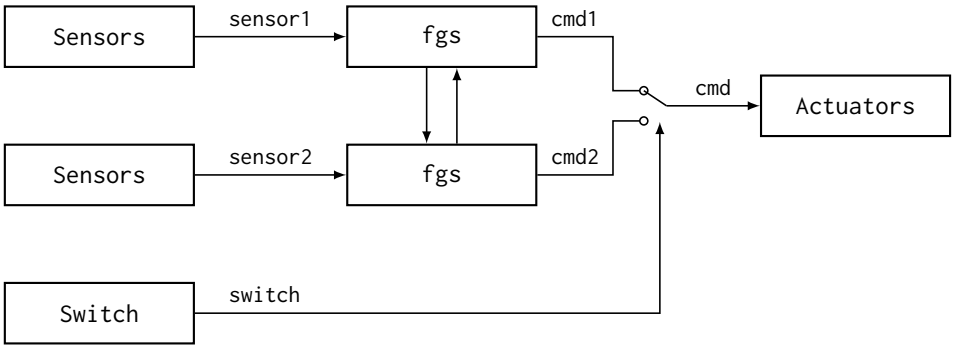


Figure 3.5: Flight guidance system. Only one FGS component, the *pilot* side is active. The crew can switch from one component to the other using a *transfer switch*.

on a quasi-periodic architecture can then be described in the language used to program the application: Scade, Lustre, or (the discrete part of) Zélus. This is a classic approach to architecture modeling using synchronous languages [BS01, HB02].

Illustration: a flight guidance system

Consider the simple embedded application (from [MBT+15]) shown in figure 3.5. A *flight guidance system* (FGS) is a part of an aircraft controller. Using data obtained from airplane sensors, the FGS periodically generates yaw, pitch, and roll commands to adapt the trajectory of the plane. For reason of fault tolerance, it is often implemented as two redundant modules.

Most of the time, only one FGS component, the *pilot side*, is active. During flight, the crew can switch from one component to the other using a *transfer switch*. The two components share information to avoid glitches when transferring control from one to the other.

At this level, the controller is a purely discrete application where all elements execute synchronously. For example, in Zélus we write:

```
let node controller(sensor1, sensor2, switch) = cmd where
  rec cmd1 = fgs(sensor1, cmd2)
  and cmd2 = fgs(sensor2, cmd1)
  and cmd = if switch then cmd1 else cmd2
```

val controller: data $\times$ *data* $\times$ *bool* $\xrightarrow{\mathrm{D}}$ *cmd*

At each step, using the input from a sensor and the command emitted by the other component each FGS generates a new command. The controller discards one of these commands depending on the state of the transfer switch.

To model the deployment of the FGS application on a two node quasi-periodic architecture, we introduce additional inputs: activations signals to model the quasi-periodic clocks and transmissions delays. Figure 3.6 illustrates the resulting model. In Zélus, we use signals to model variables that are not necessarily defined at each logical step: the activation signals, and node outputs. In Zélus we write:

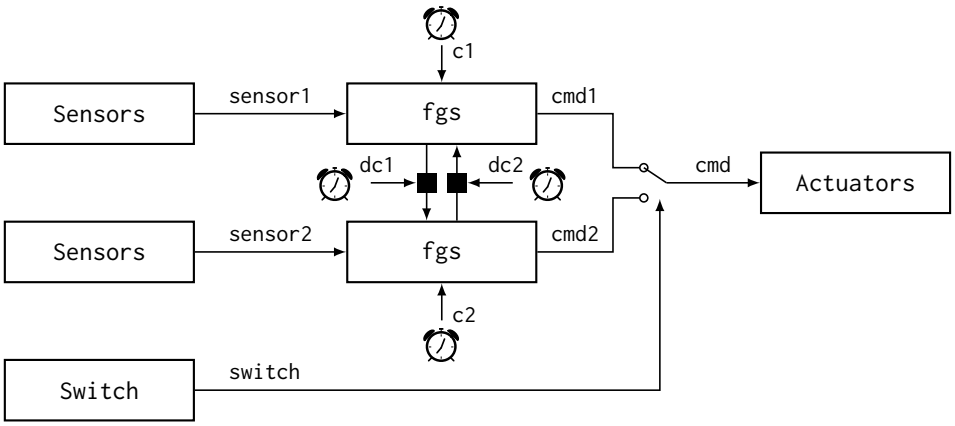


Figure 3.6: A quasi-periodic model of the FGS example of figure 3.5. Logical clocks model asynchronous activations. Symbols ■ denote *blackboard* communication.

```
let node qp_controller((c1, c2, dc1, dc2), (sensor1, sensor2, switch)) = cmd where
  rec present c1() → do emit cmd1 = fgs(sensor1, mcmd2) done
  and present c2() → do emit cmd2 = fgs(sensor2, mcmd1) done
  and mcmd1 = link(c1, dc1, cmd1, idle)
  and mcmd2 = link(c2, dc2, cmd2, idle)
  and cmd = if switch then cmd1 else cmd2
```

val qp_controller:

$$(\textit{unit signal} \times \textit{unit signal} \times \textit{unit signal} \times \textit{unit signal}) \times (\textit{data} \times \textit{data} \times \textit{bool}) \xrightarrow{\mathrm{D}} \textit{cmd signal}$$

Compared to the previous synchronous controller, there are four additional inputs (the small clocks in figure 3.6). Signals `c1` and `c2` denote the quasi-periodic clocks of the nodes, `dc1` and `dc2` their delayed versions that model transmission delays (one for each communication channel). The union of these signals gives a base notion of logical instant or step: the global clock. It allows us to model the rest of the architecture in a discrete synchronous framework.

A quasi-periodic node is activated at each tick of its local clock `c`. In Zélus we write:

```
present c1() → do emit cmd1 = fgs(sensor1, mcmd2) done
```

When the unit signal `c1` is emitted, the first node executes one step of the `fgs` application using the input from the sensor, `sensor1`, and the content of the communication link, `mcmd2`. The result is emitted on signal `cmd1`.

The two communication links `mcmd1` and `mcmd2` allow information sharing between the two `fgs` despite their asynchronous activations. These links, initialized with a constant value `idle` of type *data*, are controlled by the clocks of the nodes, `c1` and `c2`, and their delayed versions, `dc1` and `dc2`.

Modeling links

A link, shown in figure 3.7, models communication by sampling between two nodes. Since nodes are not synchronized, the output of a link must be defined at each logical step. Received values are thus stored in a memory.

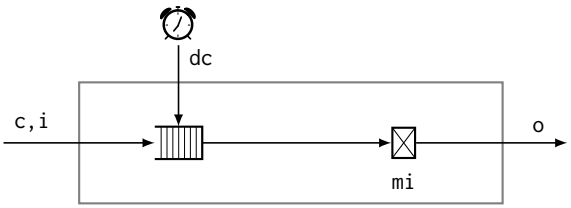
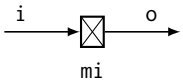


Figure 3.7: Schema of a communication link modeling delayed transmission between nodes. The striped box represents a FIFO queue.

```
let node mem(i, mi) = o where
  rec init m = mi
  and present i(v) → do m = v done
  and o = last m
```



val mem: α *signal* $\times$ $\alpha \xrightarrow{D} \alpha$

The keyword `init` initializes a memory, that is, a variable defined at each activation of the node, and `last(m)` is its previous value. Each time the input signal `i` is emitted, `m` is updated with the new received value `v`. By returning `last m`, we ensure that the output does not depend instantaneously on the inputs.

Delayed communications are modeled using a *first in first out* (FIFO) queue that can be accessed with the following imported functions:

`empty` denotes an empty queue.

`is_empty(q)` tests if a queue `q` is empty.

`size(q)` returns the size of `q`.

`front(q)` returns the oldest value in `q`.

`back(q)` returns the newest value in `q`.

`enqueue(q, x)` pushes a value `x` into `q`.

`dequeue(q)` removes the oldest value from `q`.

The complete Zélus interface, that is, the type signatures of the imported functions, is:

val empty: α *queue*
val is_empty: α *queue* $\xrightarrow{A}$ *bool*
val size: α *queue* $\xrightarrow{A}$ *int*
val front: α *queue* $\xrightarrow{A}$ α
val back: α *queue* $\xrightarrow{A}$ α
val enqueue: α *queue* $\times$ $\alpha \xrightarrow{A}$ α *queue*
val dequeue: α *queue* $\xrightarrow{A}$ α *queue*

To model a communication channel, a queue is triggered by the input signal `i` and the delayed sender clock `dc` that models transmission delays. Messages in transmission are stored in the queue and emitted when the transmission delay elapses, that is, when clock `dc` ticks.

```
let node channel(dc, i) = o where
  rec init q = empty
  and present
      | dc() & i(v) → do q = enqueue(dequeue(last q), v) done
      | i(v) → do q = enqueue (last q, v) done
      | dc() → do q = dequeue (last q) done
  and present dc() → do emit o = front(last q) done
```

val channel: unit signal $\times$ α *signal* $\xrightarrow{\text{D}}$ α *signal*

The first `present` block maintains the queue `q`. Each new message `v` received on signal `i` is added to the queue: `q = enqueue(last q, v)`. When a transmission delay elapses, that is, each time clock `dc` ticks, the first pending message is removed from the queue: `q = dequeue(last q)`. If both signals `i` and `dc` are emitted at the same time we combine the previous behaviors. In parallel, the second `present` block emits the first pending message on signal `o` when a transmission elapses.

i		1	2		3			4	...
dc				•		•	•		...
q	[]	[1]	[2;1]	[2]	[3;2]	[3]	[]	[4]	...
o				1		2	3		...

In this model, we assume that the clock `dc` models delayed transmission, that is, `dc` is a delayed version of signal `i`. This assumption guarantees that the queue is never empty when `dc` ticks. However, it may be the case that a node does not send a message at each activation of its clock. In this case, `dc` depends on conditions computed locally by the sender node. This implies that the activation clock of the complete model (the conjunction of all activation signals: `c1`, `c2`, `dc1`, and `dc2`) depends on conditions computed by one of its components. But our goal is to maintain a clear separation between global inputs modeling the real-time characteristics of the architecture (the activation signals) and the discrete logic of our model.

A classic solution is to keep a delayed version of all activations of a sender node, and use a special value to denote the *absence* of a message. When a node is activated but does not produce a message, an empty message is sent. We use an option type to denote the presence or absence of a value. A value of type α *option* is either `None` or `Some(v)` where `v` is a value of type α.

```
type α option = None | Some of α
```

The expression `get(x)` returns the value `v` when `x = Some(v)`.

val get: α *option* $\xrightarrow{\text{A}}$ α

A more complete model for the links thus takes three inputs instead of two: the clock of the sender node `c`, its delayed version `dc`, and messages sent on signal `i` whose emissions are a subset of the activations of `c`. Signal `s` is emitted with value `Some(v)` when a value `v` is received on signal `i`, and with value `None` if the sender is activated but does not send a message (when `c` is present but not `i`). The rest of the code follows the structure of the previous model with signal `s` as input instead of `i`.

```
let node channel(c, dc, i) = o where
  rec init q = empty
  and trans = present (is_empty (last q)) → false else front(last q) ≠ None
  and present
      | i(v) → do emit s = Some(v) done
      | c() → do emit s = None done
  and present
      | dc() & s(v) → do q = enqueue(dequeue(last q), v) done
      | s(v) → do q = enqueue (last q, v) done
      | dc() → do q = dequeue (last q) done
  and present dc() & trans → do emit o = get(front (last q)) done
```

val channel : unit signal $\times$ *unit signal* $\times$ α *signal* $\xrightarrow{\mathrm{D}}$ α *signal*

Signal `o` is only emitted when the queue is not empty and when the oldest element of the queue is an actual message, that is, when variable `trans` is set to *true*. The `present` construct in the definition of `trans` ensures that we only check the front of a non-empty queue.

c		•	•		•			•	•	...
i		1			2			3		...
dc				•		•	•		•	...
s		Some(1)	None		Some(2)			Some(3)	None	...
q	[]	[1]	[None, 1]	[None]	[2, None]	[2]	[]	[3]	[None]	...
trans	*false*	*false*	*true*	*true*	*false*	*false*	*true*	*false*	*true*	...
o				1			2		3	...

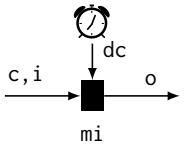
Bounded FIFO Using the quasi-periodic nature of the architecture, it is possible to bound the size of the FIFO that models messages in transmission. At worst, a first message is as slow as possible and arrives in the memory after a delay $\tau_{\max}$. At best, the producer then sends messages every $T_{\min}$. The maximum number of messages in transmission is thus:

$$n_{trans} = \left\lceil \frac{\tau_{\max}}{T_{\min}} \right\rceil .$$

Finally, we can combine a `channel` and a `memory` to model the complete link of figure 3.7:

```
let node link(c, dc, i, mi) = o where
  rec s = channel(c, dc, i)
  and o = mem(s, mi)
```

val link: unit signal $\times$ *unit signal* $\times$ α *signal* $\times$ α $\xrightarrow{\mathrm{D}}$ α



When a message is sent on signal `i`, it goes through the channel and, after the transmission delay (modeled by the delayed clock `dc`) is stored in a memory. New messages overwrite previous memory values. The memory contents are output by the link. Note that the memory `mem` imposes a unit delay between the input `i` and the output `o` that precludes instantaneous transmission.

3.4 Real-time model

The discrete model of section 3.3 does not make any assumptions on how and when clock signals and their delayed versions are produced. We can now complete our Zélus model with continuous components that simulate the real-time behavior of the quasi-periodic architecture. The discrete model can be simulated and verified using discrete language tools, and the complete Zélus program can be executed to simulate real-time traces.

For instance, the following Zélus program simulates possible executions of the FGS example:

```
let hybrid rt_controller(sensor1, sensor2, switch) = cmd where
  rec c1 = metro(t_min, t_max)
  and dc1 = delay(c1, tau_min, tau_max)
  and c2 = metro(t_min, t_max)
  and dc2 = delay(c2, tau_min, tau_max)
  and present c1() | dc1() | c2() | dc2() → do emit g done
  and present g() → do cmd = qp_controller ((c1, c2, dc1, dc2),
                                            (sensor1, sensor2, switch)) done
```

val rt_controller: data × data × bool $\xrightarrow{C}$ *cmd signal*

Activation clocks `c1` and `c2`, and their delayed versions `dc1` and `dc2` are produced by the two continuous-time functions `metro` and `delay`. Signal `g` is the union of all these signals. It is used as a base clock to activate the discrete model `qp_controller`.

By the definition of `g`, the activations of clocks `c1`, `c2`, `dc1`, and `dc2` are subsets of the activations of `g`. When the signal `g` is present, the clocks are either present or absent. Since the discrete model is activated at each emission of `g`, these clocks can be treated as discrete signals and used as inputs for the discrete model.

We now show how to implement functions `metro` and `delay` in Zélus to produce random simulations of a quasi-periodic system.

Quasi-periodic clocks and delays

Let us first consider a quasi-periodic clock that triggers the activation of a node according to equation (RP) of definition 3.1. Such a clock can be simulated in Zélus using a timer, a simple ODE $\dot{t} = 1$, initialized to an arbitrary value between $-T_{\min}$ and $-T_{\max}$, and similarly reinitialized whenever t reaches 0. To produce random simulations, we express an arbitrary delay by making a random choice.

```
let arbitrary(l, u) = l +. Random.float (u -. l)
```

Then, the model for a quasi-periodic clock is

```
let hybrid metro(t_min, t_max) = c where
  rec der t = 1.0 init -. arbitrary (t_min, t_max)
      reset z → -. arbitrary (t_min, t_max)
  and z = up(t)
  and present (init) | z → do emit c done
```

val metro: float × float $\xrightarrow{C}$ *unit signal*

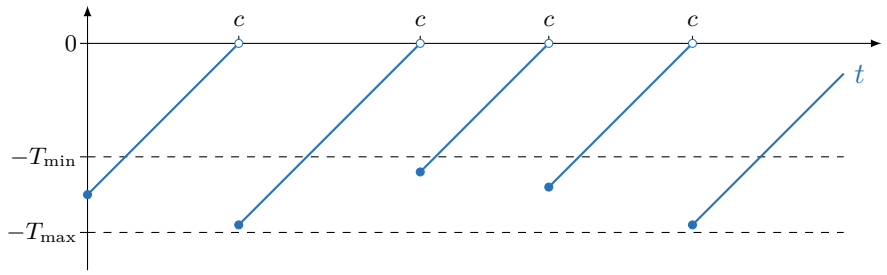


Figure 3.8: Simulation trace of the function `metro`.

The variable `t` increases with slope `1.0`. At execution start time (`init`) and zero-crossing instants signal `c` is emitted and `t` is reinitialized. Figure 3.8 illustrates this behavior.

Similarly, the constraint on transmission delays from equation (RT) of definition 3.1 is modeled by delaying the discrete signal corresponding to the sender's clock. Figure 3.9 illustrates the expected behavior. A simple Zélus model is:

```
let hybrid delay(c, tau_min, tau_max) = dc where
  rec der t = 1.0 init 0.0 reset c() → -. arbitrary (tau_min, tau_max)
  and present up(t) → do emit dc done
```

val delay: unit signal × *float* × *float* $\xrightarrow{C}$ *unit signal*

The function `delay` takes a clock `c` as input. When `c` ticks, the timer `t` is reinitialized to an arbitrary value between $-\tau_{\min}$ and $-\tau_{\max}$ corresponding to the transmission delay. Then, when the delay has elapsed, that is, when a zero-crossing is detected, a signal `dc` for the delayed clock is emitted.

While capturing the idea of delayed activation, the previous model is over-simplified. In particular, it does not allow for simultaneous ongoing transmissions, that is, it mandates $\tau_{\max} < T_{\min}$. The full version must queue ongoing transmissions.

```
let hybrid delay(c, tau_min, tau_max) = dc where
  rec init q = empty
  and der time = 1.0 init 0.0
  and der t = 1.0 init 0.0
      reset z on (size(q) > 0) | c() on (size(q) = 1) → time -. front(q)
  and z = up (t)
  and present
      | c() & z → do q = add_horizon(dequeue(last q), time, tau_min, tau_max) done
      | c() → do q = add_horizon(last q, time, tau_min, tau_max) done
      | z → do q = dequeue(last q) done
  and present z → do emit dc done
```

val delay: unit signal × *float* × *float* $\xrightarrow{C}$ *unit signal*

The queuing mechanism is similar to the one used in the discrete model (section 3.3). The first present block maintains the queue. At each activation of `c` we add a new horizon to the queue `q`, that is, the date of the next delayed activation (timer `time` is used as a wall-clock and refers to the current date). The timer `t` is reinitialized with the remaining delay until the next horizon in two cases:

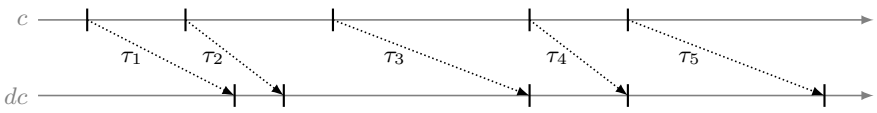


Figure 3.9: Illustration of function `delay`. Delays $\tau_1, \tau_2, \ldots$ are bounded by $\tau_{\min}$ and $\tau_{\max}$. We assume that there is no message inversion or loss.

1. If we reach a horizon while there are still pending horizons in the queue (`size(q) > 0`).
2. When `c` ticks while the queue is empty (in that case `size(q) = 1`, since the first `present` block adds the new horizon to the queue when `c` ticks).

Signal `dc` is emitted each time a zero-crossing is detected. To ensure that there is no message inversion or loss in our model, we force the function that generates new horizons to be strictly increasing.

```
let add_horizon(q, time, v_min, v_max) =
  let d = arbitrary(time +. v_min, time +. v_max) in
  if not (is_empty(q)) then enqueue(q, max(back(q) +. eps_delay, d))
  else enqueue(q, d)
```

val add_horizon: float list $\times$ *float* $\times$ *float* $\times$ *float* $\xrightarrow{\text{A}}$ *float list*

If the queue is not empty, the new horizon is at least the last enqueued value `back(q)` plus a small constant `eps_delay`.

Horizons Relying on a numerical solver to monitor the zero-crossings of simple timers is clearly overkill. The Zélus runtime uses *horizons* that allow to directly specify the date of the next event, thus bypassing the numerical solver. The use of horizons would not significantly change the structure of the functions `metro` and `delay`—the `delay` operator would still require a queuing mechanism. But in our particular case where all ODEs are timers, this mechanism is more robust and precise than monitoring zero-crossings. In the current version of Zélus, users can specify ultimately periodic horizons with floating-point constants. For instance,

```
present (period 1.0 (2.0)) → do emit tick done
```

emits the signal `tick` after 1.0 unit (of simulated time) and then periodically every 2.0 units. Unfortunately, horizons with arbitrary expressions are not yet available to users.

3.5 Other modeling tools

In this section we show how our approach can be adapted to other modeling tools for hybrid systems, namely Ptolemy[1] [Pto14] and Simulink.[2] Ptolemy offers a framework for the design and simulation of embedded systems with a focus on combining different *models of computation*, for instance, discrete and continuous time. To illustrate the parallel with Zélus, we give a complete model of the FGS example. Then we briefly show how to adapt our approach to Simulink a widely-used industrial tool for modeling hybrid systems.

[1] `http://ptolemy.eecs.berkeley.edu/`
[2] `http://www.mathworks.com/products/simulink`

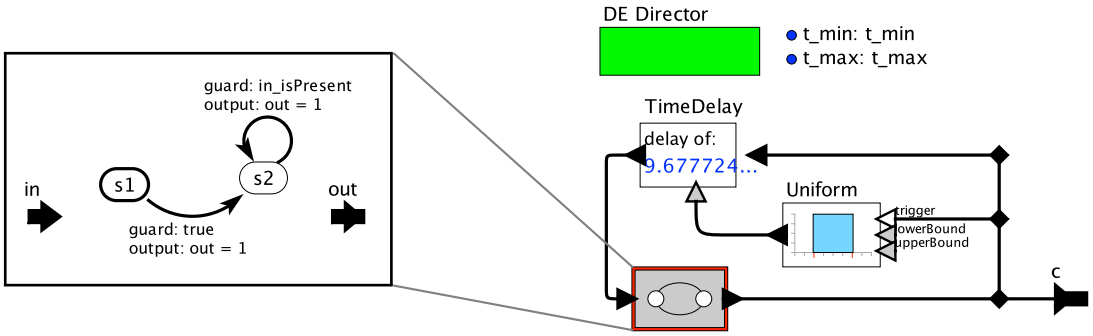


Figure 3.10: Ptolemy models of a quasi-periodic clock (`metro`).

Comparison with Ptolemy

The semantics of a Ptolemy model is determined by its *director*. A director exists for each model of computation. Some directors give a discrete semantics a model, like synchronous dataflow (SDF) and finite state machine (FSM); others allow the design of continuous time models, namely continuous time (CT) and discrete event (DE). We use the discrete-event (DE) director for continuous-time modeling, and the synchronous-reactive (SR) director for the discrete parts.

Modeling quasi-periodic clocks and delays The Ptolemy continuous-time (CT) director allows describing continuous dynamics using ODEs. We could thus reproduce the Zélus model in Ptolemy, modeling clocks with simple timers ($\dot{t} = 1$) reset to arbitrary values at zero-crossing instants. But Ptolemy also offers a discrete-events (DE) director dedicated to the modeling of 'timed, discrete interaction between concurrent actors' [Pto14, §7]. Actors produce events associated to time stamps. These time stamps define a global ordering. A DE model executes by processing events in order of increasing time stamps. Dedicated actors delay and produce events. This director is well adapted to our setting where a discrete model is triggered by events produced by quasi-periodic clocks and delays.

Figure 3.10 shows a DE model of a quasi-periodic clock. Events—the ticks of the quasi-periodic clock—are produced by a simple automaton following the scheme presented in [Pto14, §7.5.1]. The guard of the transition from the initial state `s1` is always *true*, causing the automaton to send an event at execution start time. Then the automaton is in state `s2` and sends an event whenever an input is detected (guard `in_isPresent`). Each event produced by the automaton is delayed by an arbitrary value between $T_{\min}$ and $T_{\max}$ (produced by the `Uniform` actor) and triggers the next transition of the automaton.

The model of the delayed version of the quasi-synchronous clock, shown in figure 3.11, relies on a dedicated Ptolemy operator: the `TimeDelay` actor. Each event is delayed by an arbitrary delay between $\tau_{\min}$ and $\tau_{\max}$ (produced by the `Uniform` actor). This actor allows multiple pending transmissions by queuing the delays.[3] It is the Ptolemy equivalent of the queuing mechanism used in our Zélus `delay` model in section 3.4.

[3] 'This actor keeps a local FIFO queue to store all received but not produced inputs'. Documentation of the `TimeDelay` actor, Ptolemy II, version 10.0.1 (December 17, 2014).

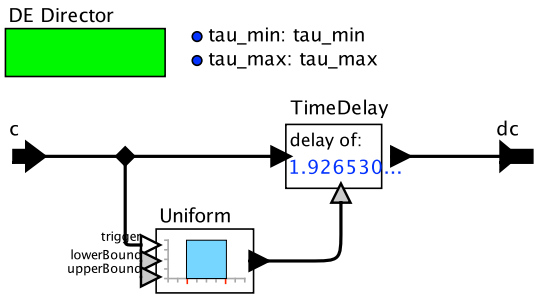


Figure 3.11: Ptolemy model of transmission delays (`delay`).

Discrete-events in Zélus This model shows that we only need a single *timed delay* operator that delays a flow of events. For comparison, we now show that the same implementation works in Zélus. Using the `delay` operator of section 3.4 we can give an alternative implementation of the quasi-periodic clock `metro` using the simple feedback loop of the Ptolemy model.

```
let hybrid metro(t_min, t_max) = c where
  rec present (init) | s() → do emit c done
  and s = delay(c, t_min, t_max)
```

val metro: float × *float* $\xrightarrow{C}$ *unit signal*

The first equation is a compact expression of the automaton used in the Ptolemy model. It emits an event `c` at execution start time (`init`), or each time signal `s` is present. Signal `s` is the delayed version of the output `c` produced by the `delay` function of section 3.4.

A generic `delay` operator could be added as a Zélus primitive construct and coupled with the horizon mechanism described above to bypass the numerical solver. Beyond modeling quasi-periodic architectures, this would allow the modeling of timed discrete components in Zélus without requiring a numerical solver.

The FGS example in Ptolemy The synchronous reactive director (SR) is inspired by dataflow synchronous languages like Lustre and Signal [Pto14, §5]. An execution is a sequence of logical steps, and there is no notion of real-time. The Lustre operators—`pre`, `when`, and `current`—are available, and it is also possible to write hierarchical automata. The models presented in section 3.3 can thus be directly translated into SR actors.

Ptolemy offers the possibility to combine multiple models of computation. It is thus possible to activate a discrete SR model on events produced by a continuous DE model. The complete Ptolemy model of the FGS example is given in figure 3.12. The model is parametrized by the timing characteristics of the architecture: $T_{\min}$, $T_{\max}$, $\tau_{\min}$, and $\tau_{\max}$. In addition to the global inputs `sensor1`, `sensor2`, and `switch`, the `FGS` actor takes as inputs the clock signals, `c1` and `c2`, generated by the `metro` actors, and their delayed versions, `dc1` and `dc2`, generated by the `delay` actors. The `FGS` actor is triggered whenever an event is detected on one of its input ports.

The two `fgs` components are modeled by *enabled composite* actors [Pto14, §5.2.4], which are only triggered when the *enable* port is set to *true*, that is, when signals `c1` or `c2` are present. The ? actor is used to test the presence of an input signal. It is similar to the Zélus `present . → do . done` construct.

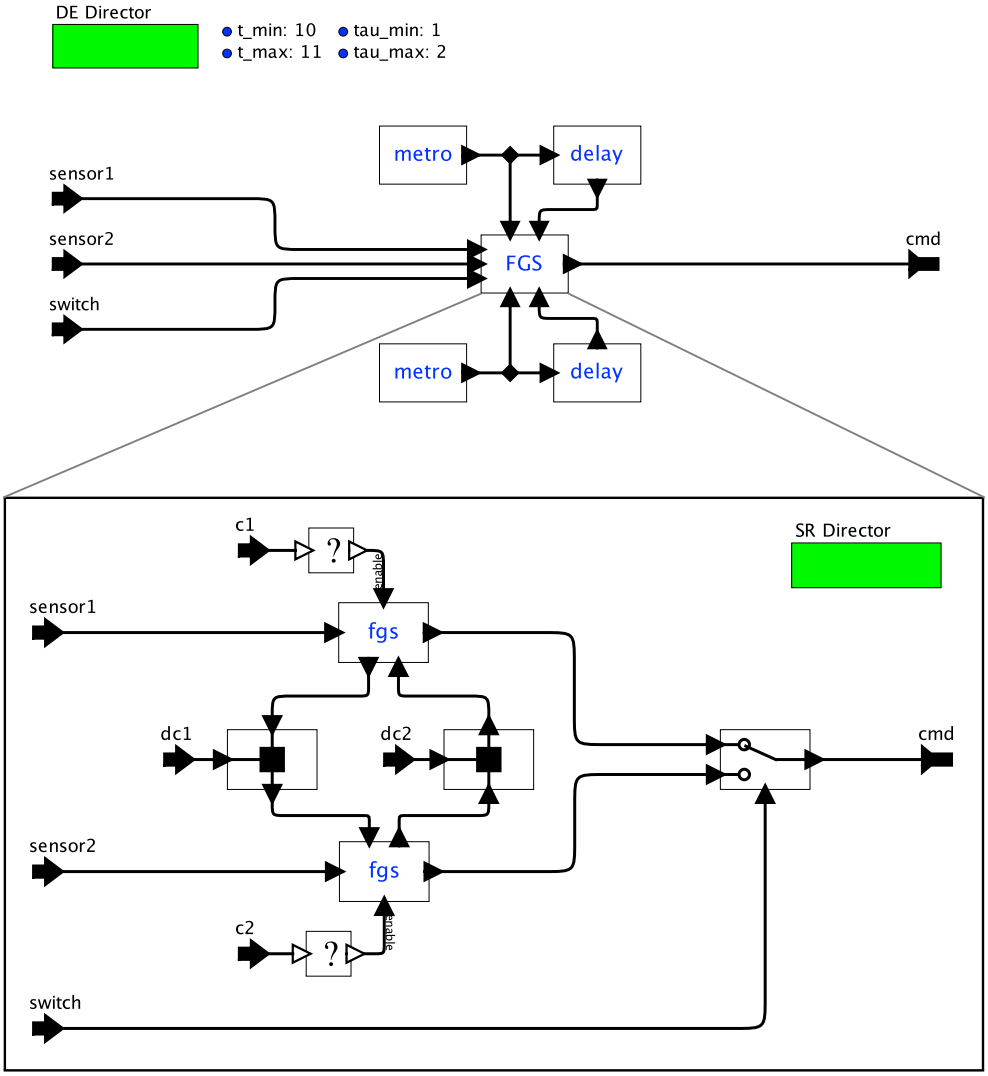


Figure 3.12: The Ptolemy model of the FGS example. The continuous DE model on top activates the discrete SR model: `FGS`. Note the similarity between the discrete model (below) and figure 3.6 (icons of the blackboards and the switch are customized to highlight this proximity).

Comparison with Simulink

Matlab/Simulink[4] is a widely-used industrial tool for modeling hybrid systems. The close link between synchronous languages and the discrete part of Simulink has been extensively studied [CCM+03, SSC+04, TSCC05]. It is thus clear that the discrete model of section 3.3 can be adapted to Simulink.

This discrete model can then be triggered on timed events using Simulink *triggered blocks*. These events can be generated by a continuous component following the Zélus implementation: simple timers ($\dot{t} = 1.0$) reset to an arbitrary value between known bounds. As in Zélus, zero-crossings of the continuous signals can then be monitored and used to activate triggered blocks containing discrete code.

3.6 Bibliographic notes

In the distributed systems literature, quasi-periodic architectures are known as the *synchronous real-time model* [ADLS94, Cri96]. The spectrum of formal models for distributed systems runs from completely synchronous (definition 3.1) to completely asynchronous [Lyn96]. The completely synchronous model makes the strongest timing assumptions: bounded execution time and bounded transmission delays, though they are not unreasonable for embedded systems. In this setting, it is possible to simulate round-based applications and solve problems like consensus and leader election even in the presence of failures [Pon91, PS92].

Partially synchronous models

The impossibility of consensus in the asynchronous model [FLP85] and the desire to treat more general systems than the synchronous model motivates the study of *partially synchronous* models [Lyn96, Part III]. There are models with bounds on transmission delays and the relative speeds of processes, and these bounds are not necessarily known or may only hold eventually [DLS88]. The Archimedean model [Vit84] bounds the relative speed of processes and the ratio between transmission delay and minimal computing step time. In the *θ-model* [WS09] bounds are not given on transmissions but rather on the ratio of the longest and shortest end-to-end delays of messages simultaneously in transit. The *Finite Average Response* time model [FSS05] only assumes a lower bound on activations and a finite average response time for transmissions. Timing assumptions may also be allowed to vary across different communication links [ADGFT04].

Other partially synchronous models address unreliable architectures. For instance, bounds on transmission delays and successive executions may occasionally be violated [Cri96, CF99] or where a system is asynchronous except for a synchronous *timely computing base* [VC02].

We treat the standard synchronous distributed systems model and our model has nothing to do with recovering possibility results or determining algorithmic complexity in a partially synchronous mode. We study a different question: how to program a discrete-time model of such an architecture and how to relate activations of this discrete model to the real-time characteristics of the architecture.

[4] `http://www.mathworks.com/products/simulink`

Discrete models of asynchronous systems

Since [Mil83] it is well known that asynchronous systems can be simulated in a synchronous model with sporadic activations and memories. This idea has been formalized and applied with synchronous languages [HB02, HM06]. This approach was earlier used to show that a multiclock Esterel program can be simulated in pure Esterel [BS01, §3.2]. Also, based on this idea, modeling tools for asynchronous architectures were developed with the synchronous language Signal [GG03a, GG03b].

The originality of our approach is that we use Zélus, a synchronous languages extended with continuous time. We can thus take advantage of the synchronous formalism to describe a complete discrete model of quasi-periodic systems, but also express real-time constraints of the architecture in the same executable language.

3.7 Conclusion

In this chapter we introduced the quasi-periodic architecture. Nodes are activated by unsynchronized local clocks and communicate with bounded transmission delay. Importantly, we explained how executing an application on such a distributed architecture introduces artifacts—duplication, loss of data and unintended signal combinations.

Then we adopted the classic approach of modeling distributed systems using a discrete synchronous formalism. We use Zélus to link this discrete model to the real-time characteristics of the architecture (bounds on activation periods and transmission delays). This discrete model is the starting point of chapter 4 where we show how to abstract the inputs representing real-time constraints to give a purely discrete model. In chapter 5 we show how to refine the model of the nodes to capture a *middleware* that ensures the correct execution of the embedded application despite the sampling artifacts introduced by the architecture.

For our implementation, we chose Zélus, but we argue that our approach can be adapted for Ptolemy and Simulink, two alternative modeling tools for systems that mix continuous-time and discrete-time dynamics.

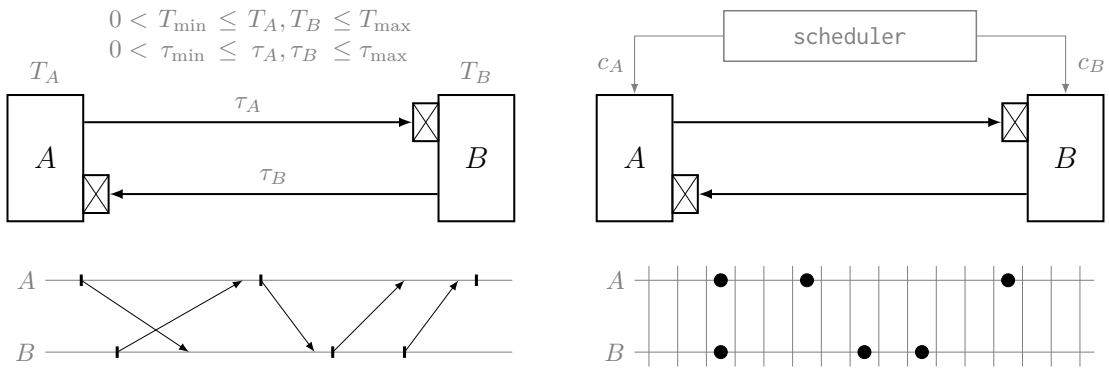
Chapter 4

The Quasi-Synchronous Abstraction

We showed in chapter 3 how to model a quasi-periodic architecture in a discrete synchronous framework. However, this model still depends on the input signals that model activation clocks and transmission delays. The quasi-synchronous abstraction [Cas00, §3] is a purely discrete abstraction introduced by Paul Caspi for model-checking safety properties of applications running on quasi-periodic architectures. The main idea is to abstract from the continuous-time signals that activate our discrete model with discrete predicates while capturing the essence of the quasi-periodic architecture (bounds on relative execution speed for instance). This model can then be used to check safety properties using dedicated discrete model-checking tools like Lustre/Lesar [HLR92], or Kind2 [CMST16].

The simplicity of the abstraction is appealing: the only events are node activations; logical steps account for transmission delays; and no node may be activated more than twice between two successive activations of any other.

Figure 4.1 gives an overview of the quasi-synchronous approach. On the left is a real-time model comprising two nodes, A and B, communicating through network links. Nodes and links are annotated with the timing bounds on executions—$T_{\min}$ and $T_{\max}$—and transmission delays—$\tau_{\min}$ and $\tau_{\max}$— characteristic of quasi-periodic architectures (definition 3.1 page 34).



(a) Real-time model (RT)

(b) Discrete-time model (DT)

Figure 4.1: Soundness: A property φ true for the discrete-time model must also hold for the real-time model, $RT \models \varphi \iff DT \models \varphi$.

Underneath is an example trace showing node activations and corresponding message transmissions. On the right is a discrete-time abstraction in which timing parameters are replaced by a discrete program called `scheduler` that overapproximates their effect by controlling node activations. Importantly, message transmissions are modeled by a single logical step.

The ultimate aim is to verify properties of the real-time model in the simpler discrete-time model. The essential property is that every sequence of states that occurs in the real-time model can also occur in the discrete-time model. This implies in particular that the state of the nodes does not reference real time. Such an association guarantees soundness: all safety properties of the discrete-time model also hold for the real-time one.

Since changes in state are directly related to received messages, we focus on traces without modeling node and network states explicitly. This means that a discrete model is a valid abstraction if every real-time trace has a discrete-time counterpart.

Contributions

This chapter presents the first main contribution of this thesis. We formalize the relation between the real-time model presented in chapter 3 and the quasi-synchronous abstraction by introducing the notion of *unitary discretization*. We can then precisely characterize systems for which the quasi-synchronous abstraction is sound. The main difficulties come from the modeling of transmission delays as unit-delays, that is one logical step of the discrete model. Surprisingly, we show that this modeling of transmission delays is not sound for general systems of more than two nodes and give necessary and sufficient conditions on both communication topologies and timing parameters to recover soundness.

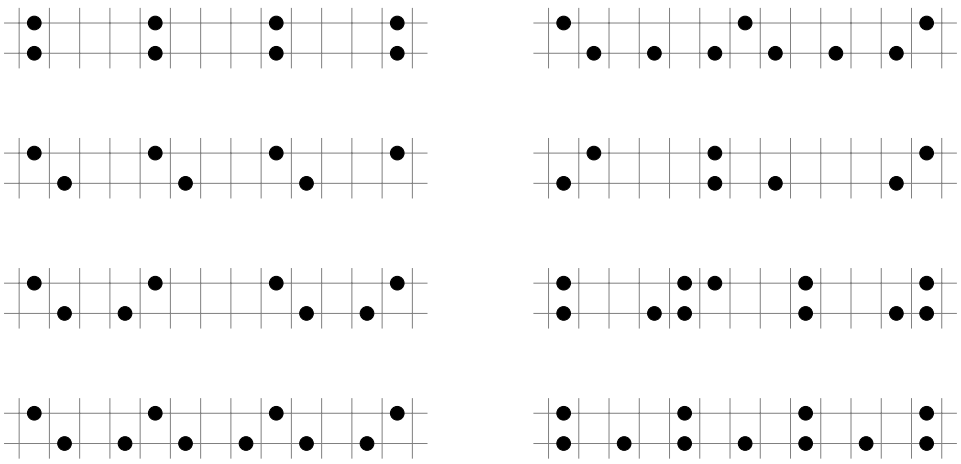
Based on theses results, we give the exact application conditions of the quasi-synchronous abstraction and extend our results to multirate systems where each node is characterized by its own activation period.

Overview In section 4.1, we define the quasi-synchronous abstraction and show how it can be used to develop a discrete model of a quasi-periodic architecture. Then, in section 4.2, we formalize real-time traces and their induced causality relation. The notion of unitary discretization presented in section 4.3 links this causality relation with the causalities expressible in the discrete model. It is quite constraining due to the modeling of communications as unit delays, but still allows for the treatment of practically relevant systems as we show in section 4.4. Based on these results, we define precisely when the quasi-synchronous abstraction can be applied to a quasi-periodic architecture. Finally, we show in section 4.5 how to extend these results to multirate systems where each node is characterized by a nominal period.

4.1 A discrete abstraction

Consider the discrete model of the quasi-periodic architecture described in section 3.3 page 37. Time is ignored altogether and node activations and communication delays are modeled with logical clocks. The model comprises two types of event: node activations where nodes can send messages to other nodes, and message receptions.

From this model, one can build a simple discrete abstraction where node activations and message receptions may be interleaved arbitrarily. This abstraction captures all possible real-time traces but also has drawbacks. Namely, it is too general: many properties that



(a) Quasi-synchronous (b) Not quasi-synchronous

Figure 4.2: Examples of quasi-synchronous and not quasi-synchronous traces.

hold in the real-time model, like property 3.1 page 35 (bounding the number of successive losses and duplications), cannot be shown in the discrete one. Furthermore, the many possible interleavings complicate model-checking the discrete-time model.

A finer abstraction was proposed by Caspi for nodes that execute 'almost periodically', that is, $T_{\min} \approx T_{\max}$. He realized that the interleavings of systems satisfying equation (RP) of definition 3.1 can be constrained [Cas01, §3.2]:

> It is not the case that a component process executes more than twice between two successive executions of another process.

Furthermore, he observed that when transmission delays are 'significantly shorter than the periods of [node activations]' they can be modeled by unit delays in the discrete-time model, but that 'if longer transmission delays are needed, modeling should be more complex' [Cas00, §3.2.1].

A unit delay models the fact that a message sent at one logical instant is received at the next one. There is thus no need to explicitly model message receptions. For instance the two first events of figure 4.1b occur in the same instant. The first message sent by B is thus only received at the second activation of A. On the other hand, the message sent at the second activation of A is received before the second activation of B since the latter occurs strictly after the former.

These observations allow us to abstract from the timing details of the real-time model of definition 3.1 to give a nondeterministic and discrete-time model of systems termed *quasi-synchronous*. The quasi-synchronous model aims at reducing the state-space of the discrete model in two ways:

1. by limiting the interleavings of node activations and,
2. by simplifying message transmission modeling.

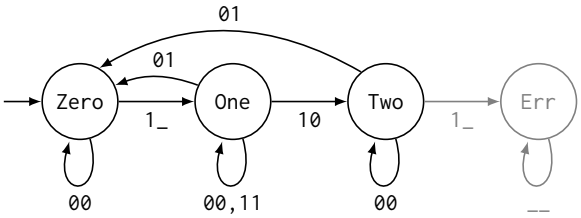


Figure 4.3: Automaton checking the quasi-synchronous condition: *no more than two ticks of* `c1` *between two ticks of* `c2`. Labels denote clock activations: `01` means, for instance, that `c2` ticks alone. The predicate is only *true* in the black region.

Definition 4.1 (Quasi-synchronous model)**.** *A* quasi-synchronous model *comprises a scheduler and finite set of nodes $\mathcal{N}$. The scheduler is connected to each node by a discrete clock signal. It activates the nodes nondeterministically but ensures that no pair of clock signals (c_A, c_B), for a pair of communicating nodes $A, B \in \mathcal{N}$, ever contains the subsequence*

$$\begin{bmatrix}1\\\_\end{bmatrix} \cdot \begin{bmatrix}0\\0\end{bmatrix}^* \cdot \begin{bmatrix}1\\0\end{bmatrix} \cdot \begin{bmatrix}0\\0\end{bmatrix}^* \cdot \begin{bmatrix}1\\\_\end{bmatrix},$$

where 1 *indicates an activation,* 0 *means no activation, and* _ *means either of the two. Nodes communicate through unit delays activated at every scheduler tick.*

This restriction on subsequences of pairs of clock signals [Cas00, §3.2.2] expresses formally the constraint quoted beforehand. The forbidden subsequence involves at least three activations of one node (A) between two successive activations of another (B). Figure 4.2 shows several examples of quasi-synchronous traces.

Valid sequences can be tested using the automaton illustrated in figure 4.3. This automaton checks that there is never more than two ticks of `c1` between two ticks of `c2`. If `c1` is emitted three times in a row, the automaton enters state `Err` where the output `ok` is set to *false*.

The automaton of figure 4.3 is readily programmed in Zélus:

```
let node check_qs(c1, c2) = ok where
  rec automaton
      | Zero → do ok = true unless c1() then One
      | One  → do ok = true unless c1() & c2() then One
                              else c1() then Two
                              else c2() then Zero
      | Two  → do ok = true unless c1() then Err
                              else c2() then Zero
      | Err  → do ok = false done
```

val check_qs: unit signal × *unit signal* $\xrightarrow{\mathrm{D}}$ *bool*

We use strong transitions (`unless`), so that `ok` is immediately set to *false* when an input violates the predicate.

This predicate is a typical example of a *synchronous observer* [HLR93]. It can be used to constrain the clocks of a quasi-synchronous system using, for instance, assertions in Lustre/Lesar [HLR92], or assumptions in Kind2 [CMST16]. The quasi-synchronous abstraction thus allows properties of the complete distributed embedded system to be checked with tools usually used for the application itself.

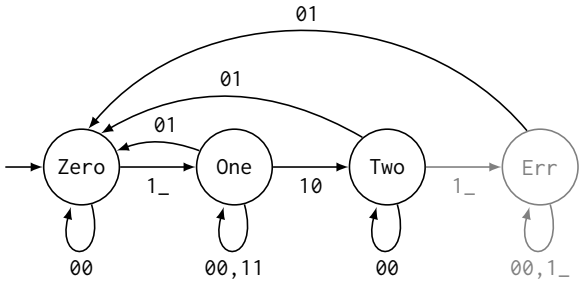


Figure 4.4: An automaton filtering parts of a trace that respect the quasi-synchronous condition. The filtering condition is only *true* in the black region.

Oracle and scheduler

Another approach, formalized in [HM06] is to rely on an oracle to produce nondeterministic inputs. In our case, an oracle Ω produces arbitrary input signals `c1` and `c2`. We then filter these inputs according to the predicate `check_qs`. We thus replaced timing assumptions on the architecture by a purely discrete scheduler. This scheduler can be used for simulation or verification of the overall system. This is the approach illustrated in figure 4.1b.

However, we first need to adapt our predicate `check_qs`. The automaton of figure 4.3 can only accept or reject traces. If we filter the output of Ω using this predicate, most of the traces will eventually stop.

Another idea is to add transitions to escape the error state `Err` when possible. Instead of rejecting traces that are not quasi-synchronous, we ignore the parts of the traces that violate the predicate. For instance, if signal `c1` is emitted three times in a row or more, we wait in state `Err` and ignore subsequent emissions of `c1` until signal `c2` is emitted.

The following predicate implements this behavior in Zélus:

```
let node filter_qs(c1, c2) = ok where
  rec automaton
      | Zero → do ok = true unless c1() then One
      | One  → do ok = true unless c1() & c2() then One
                              else c1() then Two
                              else c2() then Zero
      | Two  → do ok = true unless c1() then Err
                              else c2() then Zero
      | Err → do ok = false unless c2() then Zero
```

val filter_qs: unit signal × *unit signal* $\xrightarrow{\mathrm{D}}$ *bool*

Figure 4.4 shows a graphical representation of this automaton. In the error state `Err` the output `ok` is set to *false*. We can escape this error state whenever an emission of `c2` alone is detected. Using `filter_qs`, the Zélus code of the complete scheduler is then:

```
let node sch_qs(c1, c2) = c1', c2' where
  rec ok = filter_qs(c1, c2) && filter_qs(c2, c1)
  and present c1() & ok → do emit c1' done
  and present c2() & ok → do emit c2' done
```

val sch_qs: unit signal × *unit signal* $\xrightarrow{\mathrm{D}}$ *unit signal* × *unit signal*

We filter emissions of signals `c1` and `c2` that respect the predicate `filter_qs` of figure 4.4 for both pairs `(c1, c2)` and `(c2, c1)`.

A simpler discrete model

Compared to the discrete model presented in section 3.3, transmission delays are now modeled by unit delays. Hence, we can discard the FIFO triggered by a delayed clock `dc` that we used to model delayed communication (figure 3.7 page 40). Delayed communication is now directly implemented by the function `mem` ($\boxtimes$) of section 3.3 which already delays communication by one logical step. The rest of the model remains the same.

By removing all the input clocks used to model delayed transmission, and further constraining node activations, the quasi-synchronous abstraction drastically reduces the state-space of the model. This simplified model has been used for simulation and model-checking safety properties of embedded applications running on a quasi-periodic architecture [HM06, JHR08, BMY+14, MBT+15], such as the FGS example of figure 3.6.

An abstraction is sound to check safety properties (that is, nothing bad ever happens) if it is an overapproximation of all possible traces of the real system. The main question is then: *Is the quasi-synchronous abstraction sound?* In other words, does this simplified abstraction still capture all the possible traces of the system? The answer to these questions is the main contribution of this chapter. Despite the fact that this abstraction has been used in the literature, we show in the following that it is not sound for general systems of more than two nodes. We precisely characterize systems for which the abstraction is sound and give the necessary and sufficient conditions on the static communication graph of the application and the timing characteristic of the architecture to recover soundness.

4.2 Traces and causality

We first define a formal model for reasoning about real-time models of quasi-periodic architectures and their discretization. It has two components: (real-time) traces and their induced causality relations. In the following, we fix an arbitrary quasi-periodic architecture with nodes $\mathcal{N}$ and parameters $T_{\min}$, $T_{\max}$, $\tau_{\min}$, and $\tau_{\max}$ that satisfy definition 3.1. We formalize pairs of sending and receiving nodes using a *communicates-with* relation, written $\rightrightarrows$, between the nodes of a real-time model. This relation is not necessarily symmetric, $A \rightrightarrows B$ need not imply $B \rightrightarrows A$, but it must be reflexive ($A \rightrightarrows A$).

Definition 4.2 (Trace)**.** *A (real-time)* trace $\mathcal{E} = \{A_i \mid A \in \mathcal{N} \wedge i \in \mathbb{N}\}$ *is a set of* activation events *and two functions:*

- $t(A_i)$*, the date of event* A_i *with respect to an ideal reference clock, and*
- $\tau(A_i, B)$*, the transmission delay of the message sent at* A_i *to a node* B*.*

Both $t(A_i)$ *and* $\tau(A_i, B)$ *are non-negative reals that satisfy the constraints of definition 3.1, namely if* $A \rightrightarrows B$*,*

$$\begin{array}{rcccl} 0 \leq & T_{\min} & \leq t(A_{i+1}) - t(A_i) & \leq & T_{\max}, \text{ and} \\ 0 \leq & \tau_{\min} & \leq \quad \tau(A_i, B) \quad & \leq & \tau_{\max}. \end{array}$$

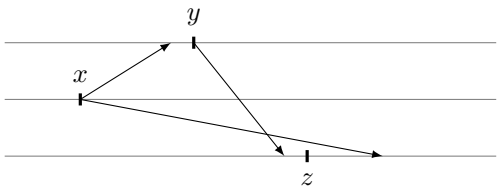


Figure 4.5: Example trace where $x \not\to z$, but $x \to y \to z$. We do not close the $\to$ relation by transitivity. Therefore $x \to y$ means that y occurs strictly after the reception of the message sent at x.

The causality relation between events within a given trace is essentially the *happened before* relation of Lamport [Lam78]. Unlike Lamport, however, we do not explicitly model message reception. A message is received if the next execution of the receiver occurs after the corresponding transmission delay.

Definition 4.3 (Happened before)**.** *For a trace $\mathcal{E}$, let $\to$ be the smallest relation on activation events that satisfies*

(*local*) *If $i < j$ then $A_i \to A_j$, and*

(*received*) *If $A \rightrightarrows B$ and $t(A_i) + \tau(A_i, B) \leq t(B_j)$ then $A_i \to B_j$.*

Activations at a single node are totally ordered (*local*); an activation at one node happened before an activation at another node when a message sent at the former is received before the latter (*received*).

Compared to Lamport, we do not close the $\to$ relation by transitivity. Hence $A_i \to B_j$ means that the message sent by node A at A_i is received by B strictly before B_j. Otherwise figure 4.5 shows an example with three events x, y, and z, where the message sent by B to C at x is not received at z even though $x \to y \to z$. However we still have a weaker form of *local transitivity*, that is, if $A_i \to B_j$ we have for all $k \geq j$, $A_i \to B_k$. The same technique is used elsewhere [RS11, definition 1].

4.3 Unitary discretization

We now address the central question of relating the real-time and discrete-time models. The problem is essentially one of correctly discretizing real-time traces.

The originality of the quasi-synchronous abstraction is to avoid modeling explicit transmissions. In the discrete model, the only events are node activations, the discretization gives a total order on events, and transmissions can be rephrased in terms of precedence: if an event x occurs strictly before another event y, the message sent at x is received before y. However, this simple idea is quite constraining and we show in the following that, even when transmission delays are 'significantly shorter than the periods of the [node activations]' [Cas00, §3.2.1] some traces cannot be discretized.

To illustrate the constraints introduced by modeling communication as unit delays, consider the example trace in figure 4.6. Unfortunately classic sampling techniques based on absolute time stamps inevitably split one of the transmission arrows. In this example, the message sent at x is received before y, x must thus occur strictly before y in the discrete model. But

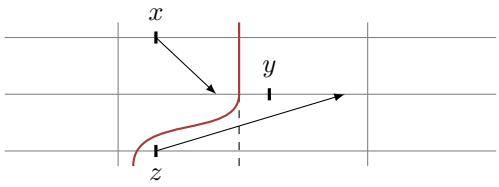


Figure 4.6: Unitary discretization. To capture communication as unit delays, x and y must be in two different instants, but z cannot be placed strictly before y.

separating these two events by a straight line (dashed line) split the transmission arrow of message z. In this discretization, y occurs strictly after z but the message sent at z is not received before y. This violates the principle that a single logical step accounts for message transmission. To capture message transmission with unit delays, we must bend the fences between logical steps (red line), thus loosing the direct link between real and discrete time.

If node A sends messages to node B, the most general approach is to ensure that when an event A_i happens before an event B_j in the real-time trace ($A_i \to B_j$), A_i occurs before B_j in the discrete-time trace and vice versa. We call such a discretization a *unitary discretization*. Figure 4.7 shows an example trace for a three-node system and a possible unitary discretization.

Definition 4.4 (Unitary discretization)**.** *A function $f : \mathcal{E} \to \mathbb{N}$ that assigns each event in a (real-time) trace $\mathcal{E}$ to a logical instant of a corresponding discrete trace is a* unitary discretization *if for all $A_i, B_j \in \mathcal{E}$,*

$$A_i \to B_j \iff (f(A_i) < f(B_j) \text{ and } A \rightrightarrows B)\,. \tag{UD}$$

Discretizing a quasi-periodic architecture satisfying definition 3.1 to a model of the form given in definition 4.1 amounts to finding a unitary discretization for *each* of its (real-time) traces. In other words, a system can be verified using the quasi-synchronous abstraction if all possible traces can be unitary discretized.

Definition 4.5 (Unitary discretizable)**.** *A quasi-periodic architecture is* unitary discretizable *if for each of its possible traces there exists a unitary discretization.*

The forward implication: $A_i \to B_j \implies (f(A_i) < f(B_j)$ and $A \rightrightarrows B)$ comes from the fact that the $\to$ relation induces a partial order on events. Completing this relation to a total order gives a discretization that respects the causality of the real-time model [Lam78].

The backward direction of the equivalence $A_i \to B_j \impliedby (f(A_i) < f(B_j)$ and $A \rightrightarrows B)$ imposes that, for a pair of communicating nodes $A \rightrightarrows B$, if an event A_i occurs strictly before an event B_j in the discrete-time model, that is, $f(A_i) < f(B_j)$, it is either because B_j is a later activation of the same node as A_i ($A = B$ and $j > i$), or because B_j occurs strictly after the receipt of the message sent at A_i.

The relation between the real-time model and the discrete abstraction is not based on sampling with a regular or irregular metric, but rather on respecting the causality of events induced by the communications. A unitary discretization links the causality of events in the real-time model to the causality implicit in the discrete-time model. It is the communication through unit delays on a common clock that tightly links the two causality relations.

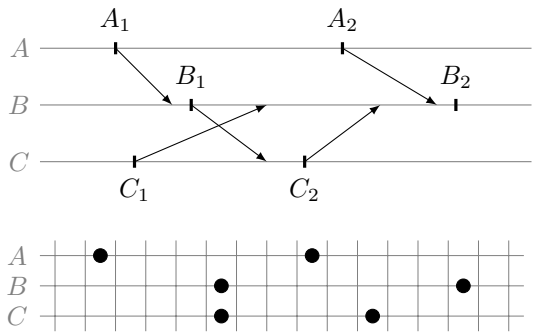


Figure 4.7: A trace (above) and a possible unitary discretization.

In distributed systems terminology, condition UD is called *strong consistency* [RS96]. The problem of finding a unitary discretization is thus equivalent to the problem of finding a strongly consistent scalar clock. Raynal and Singhal report in their survey [RS96] that this is not possible in general, that is, there is no scalar clock function f that satisfies UD. This was already noted by Lamport in his original paper: 'We cannot expect the converse condition to hold as well [...]' [Lam78, p.560].

We now show that general systems of more than two nodes are not unitary discretizable and formulate necessary and sufficient conditions on the (static) $\rightrightarrows$ relation and on the timing characteristics of the real-time model to guarantee the existence of a unitary discretization.

Trace graph

An intermediate step is to characterize traces for which there is no unitary discretization. From definition 4.4 we can deduce the following constraints.

Property 4.1. *If f is a unitary discretization for a trace, for a pair of nodes $A \rightrightarrows B$ we have*

$$A_i \to B_j \implies f(A_i) < f(B_j), \text{ and } A_i \not\to B_j \implies f(A_i) \geq f(B_j).$$

Proof. The first implication is a direct consequence of the definition of a unitary discretization. The second one follows by contraposition. If $f(A_i) < f(B_j)$, and since $A \rightrightarrows B$, we have $A_i \to B_j$ by the definition of f. □

For a particular trace, the two constraints of property 4.1 can be gathered in a weighted graph: the *trace graph*.

Definition 4.6 (Trace graph)**.** *Given a trace $\mathcal{E}$, its directed, weighted* trace graph $\mathcal{G}$ *has as vertices $\{A_i \mid A \in \mathcal{N} \wedge i \in \mathbb{N}\}$ and as edges the smallest relations that satisfy*

1. *If $A_i \to B_j$ then $A_i \xrightarrow{1} B_j$.*
2. *If $A \rightrightarrows B$ and $A_i \not\to B_j$ then $B_j \xrightarrow{0} A_i$.*

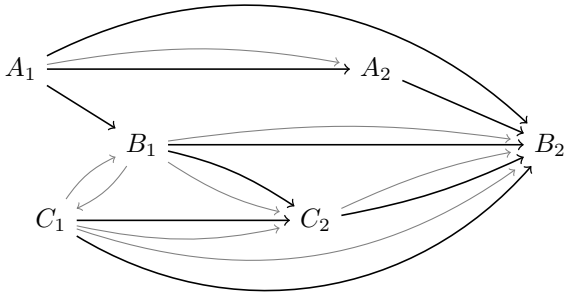
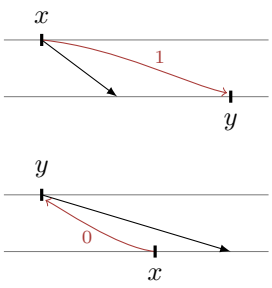


Figure 4.8: The trace graph of the example trace in figure 4.7. Black arrows denote $x \xrightarrow{1} y$, constraint $f(x) < f(y)$. Gray arrows denote $x \xrightarrow{0} y$, constraint $f(x) \leq f(y)$.



An example trace graph is shown in figure 4.8. Edges labeled with one ($x \xrightarrow{1} y$) represent the constraints $f(x) < f(y)$. Each such edge indicates that the source activation must come before the destination activation in a unitary discretization, that is, the value of f, from source to destination, must increase by at least one. Edges labeled with zeros ($x \xrightarrow{0} y$) represent the constraints $f(x) \leq f(y)$. Each such edge indicates that the source activation cannot be placed before the destination activation in a unitary discretization, that is, the value of f, from source to destination, must be the same or larger. A path through several activations defines their relative ordering in all possible unitary discretizations.

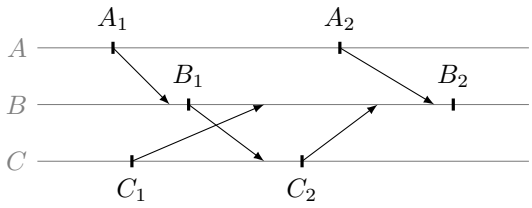
In the example of figure 4.8 constraints $C_1 \xrightarrow{0} B_1$, and $B_1 \xrightarrow{0} C_1$ impose $f(B_1) = f(C_1)$ in all possible unitary discretizations. On the other hand there is no constraint on the relative placement of C_2 and A_2. Even though A_2 occurs after C_2 in the real-time trace, a unitary discretization where A_2 occurs strictly before C_2, as in figure 4.7 is valid. Communication are correctly modeled as unit delay in the discrete-time trace. This example shows that there is, in general, no unique unitary discretization for a given trace.

The satisfaction of the required constraints, or the impossibility of satisfying them, can now be phrased in terms of cycles in the graph. A cycle comprising only $\xrightarrow{0}$'s is acceptable: its activations are all assigned the same discrete slot (for example, B_1 and C_1 in figure 4.8). Any cycle containing a $\xrightarrow{1}$ represents a set of unsatisfiable constraints: one of the events would have to be placed in two different slots to satisfy the constraints.

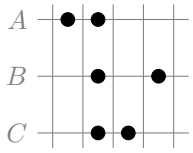
Lemma 4.1 ($\exists\text{UD} \iff \overline{\exists\text{PC}}$)**.** *For a trace $\mathcal{E}$, there is a unitary discretization* ($\exists$UD) *if and only if there is no cycle of positive weight in the corresponding trace graph $\mathcal{G}$* ($\overline{\exists\text{PC}}$)*.*

Proof. We show that $\exists\text{UD} \implies \overline{\exists\text{PC}}$ by contraposition. Assume there is a cycle of positive weight. By the construction of $\mathcal{G}$ there is an event x such that, for any unitary discretization function, $f(x) < f(x)$, which is impossible.

We now show that $\overline{\exists\text{PC}} \implies \exists\text{UD}$. If there are no cycles of positive weight, we may define a function f that maps each event x to the weight of the longest path in $\mathcal{G}$ that leads to x. By construction, $A_i \to B_j \implies f(A_i) < f(B_j)$, which is the forward implication of UD (definition 4.4). The other direction of UD follows by contraposition. Assume $A_i \not\to B_j$. If $A \rightrightarrows B$, we have $B_j \xrightarrow{0} A_i$ and thus, by the definition of f, that $f(B_j) \leq f(A_i)$. This gives $\neg(f(A_i) < f(B_j))$ as required. The other case, $A \not\rightrightarrows B$, is trivial. □



(a) Real-time trace



(b) Canonical unitary discretization

Figure 4.9: The example trace of figure 4.7 and its canonical unitary discretization based on the trace graph of figure 4.8.

The unitary discretization described in the proof above is the most concise one and can be expressed as

$$f(x) = \max\left(\{f(y) + 1 \mid y \xrightarrow{1} x\} \cup \{f(z) \mid z \xrightarrow{0} x\} \cup \{0\}\right).$$

This function gives a notion of *canonical* unitary discretization. Figure 4.9 shows the canonical discretization of the example trace of figure 4.7 based on the trace graph presented in figure 4.8. Other unitary discretizations, like the one of figure 4.7, can be obtained by stretching this discretization, that is, by inserting *mute* instants where no node is activated, or by separating events that are not constrained by each other, like B_1 and A_2 in the example of figure 4.7.

Finally, the following property shows that problematic cycles, that is cycles of positive weight, can be reduced to a form where subsequent activations of a node are grouped together.

Property 4.2. *If a trace graph has a cycle of positive weight, then it has a cycle of positive weight of the form:*

$$A^+ \xrightarrow{b_0} B^+ \xrightarrow{b_1} C^+ \xrightarrow{b_2} \ldots \xrightarrow{b_n} A^+$$

where nodes $A, B, C, \ldots$ are pairwise distinct.

The notation N^+ denote a series of subsequent activations and $\xrightarrow{b_i}$ is used as a generic notation for either $\xrightarrow{1}$ or $\xrightarrow{0}$. Remark that if there is a transition $N_i \xrightarrow{0} N_j$ in a block of activation N^+ we also have $N_i \xrightarrow{1} N_j$ by definition 4.6. To simplify the proofs we assume in the following that a block N^+ only contains $\xrightarrow{1}$ edges.

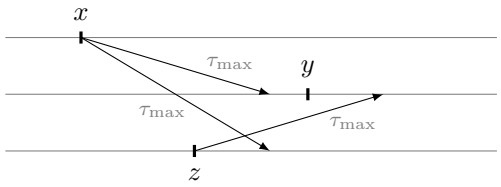
Proof. Consider a cycle of positive weight c that is not of the form:

$$A^+ \xrightarrow{b_0} B^+ \xrightarrow{b_1} C^+ \xrightarrow{b_2} \ldots \xrightarrow{b_n} A^+$$

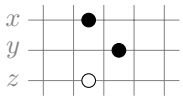
where nodes $A, B, C, \ldots$ are pairwise distinct. Then there exist two nodes $P \neq Q$, such that

$$c = e_0 \xrightarrow{b'_0} \ldots \xrightarrow{b'_k} P_i \xrightarrow{b'_l} Q_k \xrightarrow{b'_m} \ldots \xrightarrow{b'_n} P_j \xrightarrow{b'_p} \ldots \xrightarrow{b'_q} e_0$$

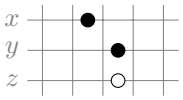
Now i and j can be related in three different ways. For each there is another cycle of positive weight where P_i and P_j are regrouped into a block of successive activations P^+ ($P_{i \to j}$ denotes the successive activations of P, $P_i \to P_{i+1} \to \ldots \to P_j$).



(a) Real-time trace.



(b) $z \to y$.



(c) $x \to z$.

Figure 4.10: A real-time trace that is not unitary discretizable ($x \xrightarrow{1} y \xrightarrow{0} z \xrightarrow{0} x$) and that may occur whenever $\tau_{\max} > 0$.

- If $i < j$, we take $e_0 \xrightarrow{b'_0} \dots \xrightarrow{b'_k} P_{i \to j} \xrightarrow{b'_p} \dots e_0$.
- If $i > j$, we take $P_i \xrightarrow{b'_l} Q_k \xrightarrow{b'_m} \dots \xrightarrow{b'_n} P_{j \to i}$.
- If $i = j$, one of the following two cycles has a positive weight:
  - $e_0 \xrightarrow{b'_0} \dots \xrightarrow{b'_k} P_i \xrightarrow{b'_p} \dots \xrightarrow{b'_q} e_0$, or
  - $P_i \xrightarrow{b'_l} Q_k \xrightarrow{b'_m} \dots \xrightarrow{b'_n} P_i$.

By iterating this result, we obtain a cycle of positive weight of the form:

$$A^+ \xrightarrow{b_0} B^+ \xrightarrow{b_1} C^+ \xrightarrow{b_2} \dots \xrightarrow{b_n} A^+$$

where nodes $A, B, C, \dots$ are pairwise distinct. □

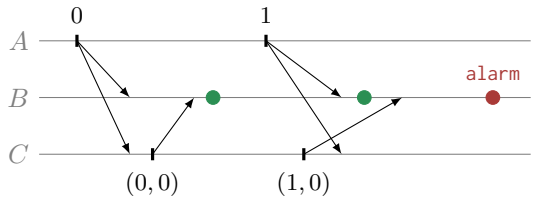
Discretizing general systems

One might expect that real-time models are unitary discretizable if the transmission delays are 'significantly shorter' than the period of the nodes, that is $\tau_{\max} \ll T_{\min}$. Unfortunately this is not the case.

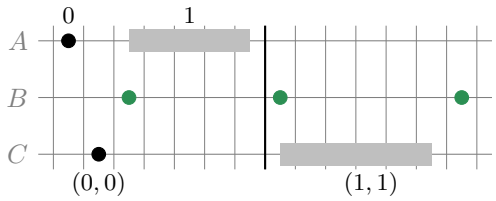
Theorem 4.1 (No general unitary discretization)**.** *General real-time models with three or more nodes communicating non-instantaneously are not unitary discretizable.*

Proof. If $\tau_{\max} > 0$, figure 4.10 shows a trace with a cycle of positive weight, $x \xrightarrow{1} y \xrightarrow{0} z \xrightarrow{0} x$, for which there is no unitary discretization (lemma 4.1). □

Figure 4.10 shows the two possible canonical discretizations of the counterexample. In figure 4.10b the message sent at z should have been received at y ($z \to y$); whereas in figure 4.10c the message sent at x should have been received at z ($x \to z$). Neither correctly abstracts the real-time trace of figure 4.10a.



(a) Real-time trace.



(b) Discrete contraints.

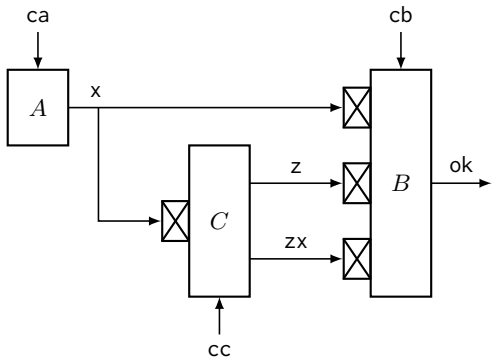
Figure 4.11: A real-time trace that raises an alarm but that is not captured by the discrete abstraction where transmission are modeled with unit delays.

Is it that bad? Consider the following application.

```
let node a() = x where
  rec x = 0 fby (x + 1)

let node c(x) = z, zx where
  rec z = 0 fby (z+1)
  and zx = x

let node b(x, z, zx) = ok where
  rec cx = not (x > (0 fby x))
  and cz = (z > (0 fby z))
  and czx = (x > zx)
  and ok = not (cx && cz && czx)
```



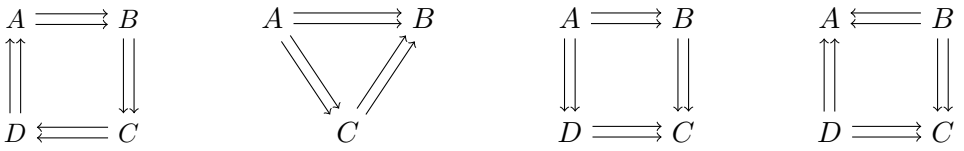
Each node is triggered by its local clock. Node A outputs a variable x, incremented at each activation. Similarly, node C increments a variable z at each activation, but also outputs zx, the last received value from A. Using variables x, z, and zx, node B computes a simple boolean condition.

Condition cx is *true* if no message was received from A since the last activation of B. Condition cz is *true* if a message from C was received since the last activation of B. Condition czx is *true* if the value of x received from A is fresher than the value of x received via C

Assume that the previous application triggers an alarm when the output of B is set to *false*. Can we check, using the quasi-synchronous abstraction, that this application running on a quasi-periodic architecture never raises an alarm?

Based on the counterexample of figure 4.10, figure 4.11a shows an example of a real-time trace where an alarm is raised at the third activation of node B. In contrast, figure 4.11b shows that the alarm cannot be raised in the discrete model.

To raise an alarm, B must receive a message from A between its first and second activations. In the discrete model, an activation of A must thus be placed after the first activation of B (or simultaneously since there is no instantaneous communication), and strictly before the second activation of B, that is, in the gray rectangle on line A. In addition B must also receive a message from C between its second and third activations. The activation of C must thus be placed after the second activation of B, and strictly before the third one, that is, in the gray rectangle on line C. Finally, the value zx sent by C must be outdated, which means that C did not received the message sent by A when it sent its last message. But, wherever the activations of A and C are placed in the gray areas, they are always separated by at least



(a) cycle (b) u-cycle (c) b-cycle (d) b-cycle

Figure 4.12: Cycle, u-cycle, and (balanced) b-cycle.

one logical step. Hence, this last condition cannot be satisfied in the discrete model where transmission are modeled by unit delays.

In other words, even though an alarm can be raised by the real-time system, it is possible to prove using the quasi-synchronous abstraction that the application is safe. There are real-time traces that are not captured by the abstraction, hence this abstraction is, in this particular case, not sound.

Remark that the problem originates from the modeling of transmissions as unit delays. It is independent of whether or not nodes activations are constrained as in definition 4.1 In other words, this problem also occurs in a completely asynchronous model where transmission are modeled as unit delays.

Recovering Soundness

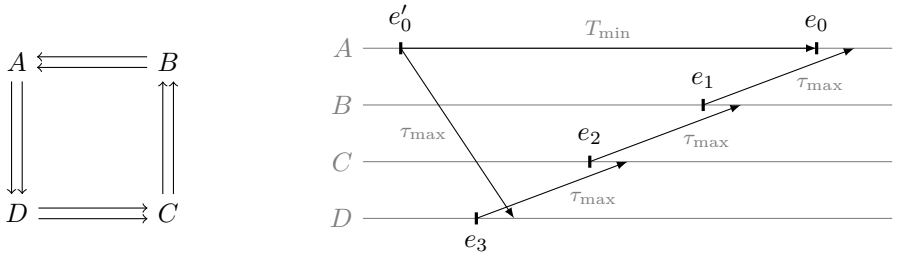
The counterexample of figure 4.11 shows that when three nodes communicate such that $A \rightrightarrows B \leftleftarrows C \leftleftarrows A$, there is at least one trace with no unitary discretization. Problematic cycles in traces can be prevented either by constraining the timing parameters of the model or by restricting communication graphs: forbidding $A \rightrightarrows B$ removes $A_i \xrightarrow{1} B_j$ and $B_j \xrightarrow{0} A_i$, for all i and j, in associated trace graphs (if $A \neq B$). To recover soundness, we propose conditions that preclude cycles of positive weight in all possible traces and thus guarantee the existence of unitary discretizations.

A *u-cycle* is an elementary cycle in the undirected communication graph, that is, the graph obtained from the communication graph by abstracting from the direction of the edges. A balanced u-cycle—or b-cycle—has the same number of edges in both directions. Figure 4.12 shows three examples of u-cycles, the two rightmost ones are also b-cycles. In the following $\mathcal{C}$, $u\mathcal{C}$, and $b\mathcal{C}$ denote the sets of cycles, u-cycles, and balanced u-cycles.

Theorem 4.2. *Let L_c be the size of the longest elementary cycle in the communication graph. A quasi-periodic architecture (definition 3.1) is unitary discretizable if and only if, the three following conditions hold:*

1. *All u-cycles of the communication graph are cycles, or balanced u-cycles, or $\tau_{\max} = 0$.*
2. *There is no balanced u-cycle in the communication graph or $\tau_{\min} = \tau_{\max}$.*
3. *There is no cycle in the communication graph, or*

$$T_{\min} \geq L_c \tau_{\max}. \tag{CD}$$



(a) Cycle

(b) $e'_0 \xrightarrow{1} e_0 \xrightarrow{0} e_1 \xrightarrow{0} e_2 \xrightarrow{0} e_3 \xrightarrow{0} e'_0$

Figure 4.13: A cycle of positive weight based on a cycle of the communication graph.

The term, *communication graph* refers to communication at the application level. The physical network may have an arbitrary topology if the communications of the application respect the conditions of theorem 4.2.

In simpler terms, theorem 4.2 states that communication topologies containing u-cycles that are neither cycles nor b-cycle are only permissible if communication is perfectly instantaneous. Cycles can be allowed by imposing the additional constraint CD and balanced u-cycles can be allowed by imposing $\tau_{\min} = \tau_{\max}$, that is forbidding jitter on transmission delays.

Remark that theorem 4.1 is a particular case of theorem 4.2. Without assumptions on the communication graph there could be a u-cycle that is neither a cycle nor a balanced u-cycle.

Proof. The proof is by contraposition in both directions. Using lemma 4.1 we also have $\overline{\exists \mathrm{UD}} \iff \exists \mathrm{PC}$. Therefore we will prove the following result which is logically equivalent to theorem 4.2.

$$\exists \mathrm{PC} \iff \left| \begin{array}{ll} \exists c \in \mathcal{C} \text{ and } \overline{\mathrm{CD}}, \text{ or,} & (\mathrm{C}_1) \\ \exists c \in b\mathcal{C} \text{ and } \tau_{\min} < \tau_{\max}, \text{ or,} & (\mathrm{C}_2) \\ \exists c \in u\mathcal{C} \setminus (\mathcal{C} \cup b\mathcal{C}) \text{ and } \tau_{\max} > 0. & (\mathrm{C}_3) \end{array} \right.$$

$\mathbf{C_1}$ or $\mathbf{C_2}$ or $\mathbf{C_3} \implies \exists\mathbf{PC}$

We show that each condition C_1, C_2, or C_3, allows the construction of a trace that contains a cycle of positive weight.

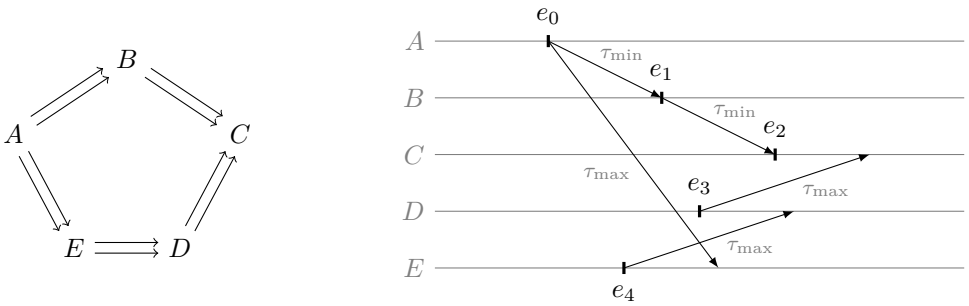
$\mathbf{C_1} \implies \exists\mathbf{PC}$ Assume that

$$\exists c \in \mathcal{C} \text{ and } T_{\min} < L_c \tau_{\max} \qquad (\mathrm{C}_1)$$

Consider one of the longest cycles of the communication graph: $N_0 \leftleftarrows N_1 \leftleftarrows \ldots \leftleftarrows N_{L_c} \leftleftarrows N_0$. We define $\varepsilon = (L_c\tau_{\max} - T_{\min})/L_c > 0$ and $\mathcal{E}$ a trace where for all events e the transmission delay is as long as possible, $\forall e \in \mathcal{E},\ \tau(e, \_) = \tau_{\max}$, and

$$\begin{aligned} t(e'_0) &= 0 \\ t(e_0) &= T_{\min} \\ t(e_{i+1}) &= t(e_i) - (\tau_{\max} - \varepsilon), \qquad \forall 0 \leq i \leq L_c \end{aligned}$$

with $N_{L_c+1} = N_0$.



(a) u-cycle

(b) $e_0 \xrightarrow{1} e_1 \xrightarrow{1} e_2 \xrightarrow{0} e_3 \xrightarrow{0} e_4 \xrightarrow{0} e_0$

Figure 4.14: A cycle of positive weight based on a u-cycle of the communication graph.

We thus have $t(e_{L_c+1}) = t(e'_0) + T_{\min} - L_c(\tau_{\max} - \varepsilon) = t(e'_0)$, that is, $e_{L_c+1} = e'_0$ and we have $\forall 0 \leq i \leq L_c$

$$N_{i+1} \rightrightarrows N_i$$
$$t(e_i) < t(e_{i+1}) + \tau(e_{i+1}, N_i).$$

Hence $e'_0 \xrightarrow{1} e_0$ and $\forall 0 \leq i \leq L_c$ we have $e_i \xrightarrow{0} e_{i+1}$. This is a cycle of weight 1. Figure 4.13 gives an example of such a trace for four nodes.

C_2 or C_3 $\implies$ $\exists$PC Suppose

$$\exists c \in b\mathcal{C} \text{ and } \tau_{\min} < \tau_{\max}, \text{ or,} \tag{C_2}$$
$$\exists c \in u\mathcal{C} \setminus (\mathcal{C} \cup b\mathcal{C}) \text{ and } \tau_{\max} > 0. \tag{C_3}$$

In both cases c is a chain of nodes $N_0, \ldots, N_n, N_0$. Let p be the number of $\rightrightarrows$ edges, and q the number of $\leftrightarrows$ edges. One can assume without loss of generality that $q \geq p > 0$. Note that $p > 0$, otherwise c would be a cycle, contradicting the assumptions $c \in b\mathcal{C}$ or $c \in u\mathcal{C} \setminus (\mathcal{C} \cup b\mathcal{C})$.

Let $\varepsilon = (q\tau_{\max} - p\tau_{\min})/q$. In both cases, $\varepsilon > 0$. Indeed if $c \in b\mathcal{C}$ we would have $p = q$ but also $0 \leq \tau_{\min} < \tau_{\max}$, and, conversely, if $c \in u\mathcal{C} \setminus (\mathcal{C} \cup b\mathcal{C})$ we would have $q > p$ and $\tau_{\max} > 0$. Finally, let $\mathcal{E}$ be a trace where $t(e_0) = 0$ and $\forall 0 \leq i \leq n$,

$$N_i \rightrightarrows N_{i+1} \implies \begin{cases} t(e_{i+1}) = t(e_i) + \tau_{\min} \\ \tau(e_i, N_{i+1}) = \tau_{\min} \end{cases}$$

$$N_i \leftrightarrows N_{i+1} \implies \begin{cases} t(e_{i+1}) = t(e_i) - (\tau_{\max} - \varepsilon) \\ \tau(e_{i+1}, N_i) = \tau_{\max}, \end{cases}$$

with $N_{n+1} = N_0$. We thus have $t(e_{n+1}) = t(e_0) + p\tau_{\min} - q(\tau_{\max} - \varepsilon) = t(e_0)$, that is, $e_{n+1} = e_0$ and $\forall 0 \leq i \leq n$

$$N_i \rightrightarrows N_{i+1} \implies t(e_{i+1}) \geq t(e_i) + \tau(e_i, N_{i+1}), \text{ and}$$
$$N_i \leftrightarrows N_{i+1} \implies t(e_i) < t(e_{i+1}) + \tau(e_{i+1}, N_i).$$

Hence

$$N_i \rightrightarrows N_{i+1} \implies e_i \xrightarrow{1} e_{i+1}, \text{ and}$$
$$N_i \leftrightarrows N_{i+1} \implies e_i \xrightarrow{0} e_{i+1}.$$

This is a cycle of weight $p > 0$. Figure 4.14 gives an example of such a trace for five nodes.

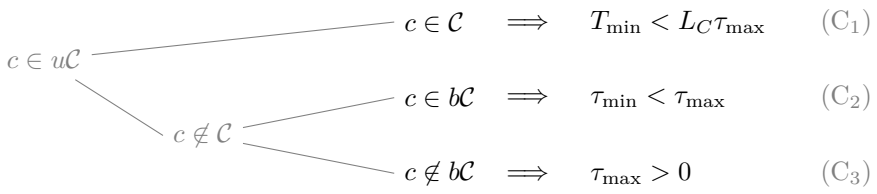


Figure 4.15: Proof scheme for $\exists \text{PC} \implies C_1$ or C_2 or C_3. There are three possible topologies for the u-cycle c, each one implies one of the conditions C_1, C_2, or C_3.

$\exists$PC $\implies$ C_1 or C_2 or C_3

Suppose that there exists a trace that contains a cycle of positive weight. By property 4.2, there exists a trace with a cycle of positive weight of the form:

$$x = A^+ \xrightarrow{b_0} B^+ \xrightarrow{b_1} C^+ \xrightarrow{b_2} \dots \xrightarrow{b_n} A^+$$

where the nodes $A, B, C, \dots$ are pairwise distinct. For two nodes A and B, by property 4.2, a transition $A_i \xrightarrow{0} B_j$ corresponds to a communication channel $A \leftleftarrows B$, and a transition $A_i \xrightarrow{1} B_j$ corresponds to a communication channel in the opposite direction, $A \rightrightarrows B$, with the possibility that $A = B$ for activations of the same node. Since the nodes of x are pairwise distinct, the sequence of nodes $c = A, B, C \dots, A$ forms a u-cycle in the communication graph.

We define $x = e_0 \xrightarrow{b_0} e_1 \xrightarrow{b_1} \dots \xrightarrow{b_n} e_n \xrightarrow{b_{n+1}} e_0$ to refer to the particular activation e_k, and N_k as the corresponding nodes. Depending on the nature of c there are three cases to consider (figure 4.15).

Cycle $c \in \mathcal{C}$ Note that according to definition 3.1 we have $x \xrightarrow{1} y \implies t(x) < t(y)$. This implies that a cycle of positive weight cannot contain only $\xrightarrow{1}$ edges. Furthermore if a cycle of positive weight is based on a cycle of the communication graph, an edge $\xrightarrow{1}$ can only reflect subsequent activations of the same node. Otherwise $\xrightarrow{0}$ and $\xrightarrow{1}$ edges correspond to communications in opposite directions and c cannot be a cycle.

Let p be the number of $\xrightarrow{1}$ edges and q the number of $\xrightarrow{0}$ edges in trace x. We have by definition of $\xrightarrow{1}$ and $\xrightarrow{0}$ that

$$\begin{aligned} e_i \xrightarrow{1} e_{i+1} \implies t(e_{i+1}) &\geq t(e_i) + T_i^{N_i} \\ &\geq t(e_i) + T_{\min}, \text{ and} \\ e_i \xrightarrow{0} e_{i+1} \implies t(e_{i+1}) &> t(e_i) - \tau(e_{i+1}, N_i) \\ &> t(e_i) - \tau_{\max}. \end{aligned}$$

where $T_i^N = t(N_{i+1}) - t(N_i)$ denotes the delay between the ith and $(i+1)$th activations of node N.

Following the cycle we have $t(e_0) + pT_{\min} - q\tau_{\max} < t(e_0)$ and hence $q\tau_{\max} > pT_{\min}$. By definition, L_c is the maximum length of a cycle thus $L_c \geq q$ and x is a cycle of positive weight, that is, $p \geq 1$. Hence,

$$L_c \, \tau_{\max} \geq q\tau_{\max} > pT_{\min} \geq T_{\min}$$

which violates condition CD. Therefore C_1 holds.

Balanced u-cycle $c \in b\mathcal{C}$ Let p be the number of edges $e_i \xrightarrow{1} e_{i+1}$ in x such that $N_i \neq N_{i+1}$ (message transmissions), r the number of edges $e_i \xrightarrow{1} e_{i+1}$ such that $N_i = N_{i+1}$ (subsequent activations of the same node, denoted here by $\xrightarrow{1}_N$), and q the number of edges $\xrightarrow{0}$. By definition of $\xrightarrow{1}$, $\xrightarrow{1}_N$, and $\xrightarrow{0}$, we have

$$\begin{aligned} e_i \xrightarrow{1} e_{i+1} \implies t(e_{i+1}) &\geq t(e_i) + \tau(e_i, N_{i+1}) \\ &\geq t(e_i) + \tau_{\min}, \end{aligned}$$

$$\begin{aligned} e_i \xrightarrow{1}_N e_{i+1} \implies t(e_{i+1}) &\geq t(e_i) + T_i^{N_i} \\ &\geq t(e_i) + T_{\min}, \text{ and} \end{aligned}$$

$$\begin{aligned} e_i \xrightarrow{0} e_{i+1} \implies t(e_{i+1}) &> t(e_i) - \tau(e_{i+1}, N_i) \\ &> t(e_i) - \tau_{\max}. \end{aligned}$$

Along the cycle x: $t(e_0) + p\tau_{\min} + rT_{\min} - q\tau_{\max} < t(e_0)$. Since $c \in b\mathcal{C}$, we also have $p = q$, and thus $p(\tau_{\min} - \tau_{\max}) + rT_{\min} < 0$ which imposes $\tau_{\min} < \tau_{\max}$. Therefore C_2 holds.

General u-cycle $c \in u\mathcal{C} \setminus (\mathcal{C} \cup b\mathcal{C})$ With the notation of the previous paragraph, we also have $t(e_0) + p\tau_{\min} + rT_{\min} - q\tau_{\max} < t(e_0)$, that is, $p\tau_{\min} + rT_{\min} - q\tau_{\max} < 0$. Since $\tau_{\min}, T_{\min} \geq 0$ and $p, q, r \geq 0$ this implies $\tau_{\max} > 0$. Hence C_3 holds. □

2-node systems

Corollary 4.1 (2-nodes unitary discretization)**.** *A real-time model satisfying definition 3.1 with two nodes can be unitary discretized if and only if*

$$T_{\min} \geq 2\tau_{\max}. \tag{2D}$$

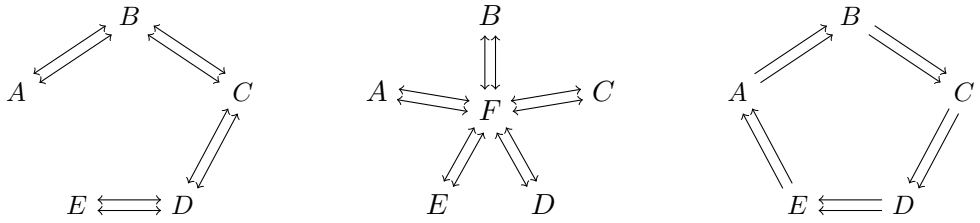
Proof. This is a direct consequence of theorem 4.2: for systems of two nodes, $L_c = 2$ and CD becomes $T_{\min} \geq 2\tau_{\max}$. □

Two-node models were the focus of the original work on the quasi-synchronous approach [Cas00] and they are relevant in practice [HM06, JHR08]. This result is coherent with Caspi's requirement that transmission delays be 'significantly shorter than the periods of [node activations]' [Cas00, §3.2.1].

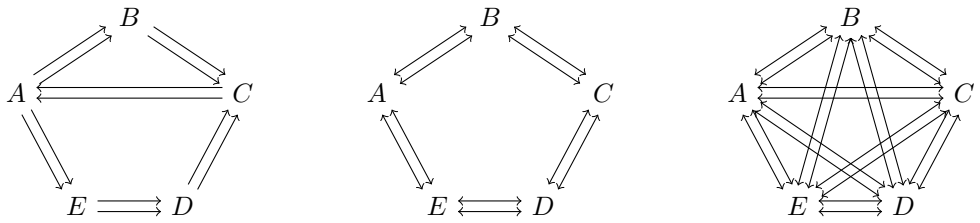
More generally, figure 4.16 shows some common network topologies and the associated real-time constraints to ensure that the model is unitary discretizable. The second line shows topologies containing unbalanced u-cycle, thus requiring instantaneous communication to be unitary discretizable. Note, in particular that combining two valid topologies can lead to a problematic communication network. For instance, figure 4.16d is a combination of two cycles, but the resulting outermost path forms a u-cycle of five nodes. Recall the we consider communications at the application level. The physical network may have an arbitrary topology.

4.4 Quasi-synchronous systems

We now apply the preceding definitions and results on unitary discretizations to precisely describe when the quasi-synchronous model can be applied to a quasi-periodic architecture.



(a) Line: $T_{\min} \geq 2\tau_{\max}$ (b) Star: $T_{\min} \geq 2\tau_{\max}$ (c) Ring: $T_{\min} \geq 5\tau_{\max}$



(d) Mesh: $\tau_{\max} = 0$ (e) Double ring: $\tau_{\max} = 0$ (f) Clique: $\tau_{\max} = 0$

Figure 4.16: Common examples of network topologies and the associated constraints to ensure that the system is unitary discretizable.

A discrete-time model is termed quasi-synchronous if [Cas01, §3.2]

> It is not the case that a component process executes more than twice between two successive executions of another process.

Since any given node only detects the activations of another by receiving the corresponding messages, the quasi-synchronous condition corresponds to two constraints. For any node,

1. there are no more than two activations between two message receptions, and
2. there are no more than two message receptions between two activations.

This definition can be formalized using unitary discretizations.

Definition 4.7 (Quasi-Synchronous Model)**.** *A real-time model is quasi-synchronous if, for every trace $\mathcal{E}$,*

1. *it has a unitary discretization f, and*
2. *for nodes $A \leftleftarrows B$, there are no i and j such that*

$$\begin{aligned} &f(B_j) < f(A_i) < f(A_{i+2}) \leq f(B_{j+1}) \text{ or,} \\ &f(A_j) \leq f(B_i) < f(B_{i+2}) < f(A_{j+1}). \end{aligned} \tag{QS}$$

This definition expresses the two aspects of quasi-synchrony: communications as logical unit delays, and constraints on interleavings of node activations.

Using the unitary discretization of definition 4.7, the activation instants of a node A can be represented by a boolean clock: $c_A(i) = 1$ iff $f(A_j) = i$ for some j. Then definition 4.7 reflects the fact that the pair of clocks (c_A, c_B) associated to nodes A and B never contains either of the subsequences

$$\begin{bmatrix}1\\0\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}1\\0\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}1\\\_\end{bmatrix} \quad \text{or} \quad \begin{bmatrix}\_\\1\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}0\\1\end{bmatrix}\cdot\begin{bmatrix}0\\0\end{bmatrix}^*\cdot\begin{bmatrix}0\\1\end{bmatrix},$$

where _ means either 1 of 0 as before. Automata that check these regular expressions can be implemented following the technique presented in section 4.1, or more concisely, using a counter.

```
let node check_qs_1(c1, c2) = ok where
  rec init p = 0
  and init ok = true
  and present
      | c1() on (p = 2) → do ok = false done
      | c2() → do p = 0 done
      | c1() → do p = last p + 1 done
```

val check_qs_1: unit signal × *unit signal* $\xrightarrow{\mathrm{D}}$ *bool*

Variable `p` counts the number of activations of `c1` with a reset at each emission of `c2`. The first condition of definition 4.7 is violated if `c1` is activated while `p = 2`.

The second condition can be checked using the same technique.

```
let node check_qs_2(c1, c2) = ok where
  rec init p = 0
  and init ok = true
  and present
      | c1() & c2() → do p = 1 done
      | c2() on (p = 2) → do ok = false done
      | c1() → do p = 0 done
      | c2() → do p = last p + 1 done
```

val check_qs_2: unit signal × *unit signal* $\xrightarrow{\mathrm{D}}$ *bool*

Compared to `check_qs_1`, the roles of `c1` and `c2` are reversed and the counter is also reinitialized to the value 1 if both `c1` and `c2` are emitted during the same instant. In Lustre, similar predicates can be generated from the two regular expressions using the Reglo tool [Ray96].

Condition QS is less constraining than definition 4.1 page 54. That definition, proposed by Caspi, has the advantage of only forbidding a single symmetric subsequence, but the link with node interleavings is obscured. In fact, the property below shows that it is violated in any real-time system with unidirectional communications ($A \leftleftarrows B$ but $A \not\rightleftarrows B$) that is not perfectly synchronous. So, while definition 4.7 does not directly translate definition 4.1, we argue that it more faithfully describes quasi-synchronous systems in terms of node interleavings.

Property 4.3. *A pair of (real-time) nodes A and B where $A \leftleftarrows B$ but $A \not\rightleftarrows B$ cannot be quasi-synchronous in the sense of definition 4.1 if $T_{\min} + \tau_{\min} < T_{\max} + \tau_{\max}$.*

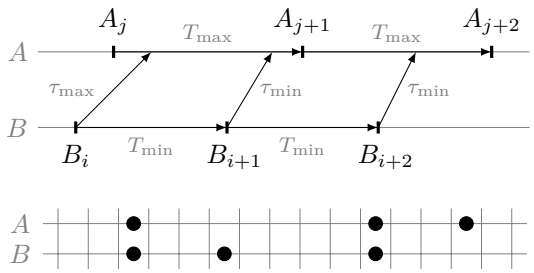


Figure 4.17: A trace (above) and a possible discretization that violates definition 4.1.

Proof. If $T_{\min} + \tau_{\min} < T_{\max} + \tau_{\max}$, figure 4.17 shows an execution trace where

$$A_j \xrightarrow{0} B_i \xrightarrow{1} B_{i+1} \xrightarrow{1} A_{j+1} \xrightarrow{0} B_{i+2}.$$

A discretization f with $f(A_j) = f(B_i)$ and $f(A_{j+1}) = f(B_{i+2})$ is a valid unitary discretization that violates the condition of definition 4.1. □

While definition 4.7 conveys the essence of quasi-synchrony, its conditions are rather abstract. The following theorem incorporates the results of sections 4.2 and 4.3 to state concrete requirements on real-time parameters and communication topologies.

Theorem 4.3. *A quasi-synchronous architecture (definition 4.1) is quasi-synchronous (condition QS) if and only if,*

1. *the conditions of theorem 4.2 hold, and*
2. *the following condition holds,*

$$2T_{\min} + \tau_{\min} \geq T_{\max} + \tau_{\max}. \tag{QT}$$

Proof. Consider a pair of communicating nodes A and B where $A \leftleftarrows B$. The first condition ensures that the system is unitary discretizable. To show that QT $\implies$ QS, assume that condition QS does not hold, that is, there exist i and j such that

$$\begin{aligned} &f(B_j) < f(A_i) < f(A_{i+2}) \leq f(B_{j+1}) \text{ or,} \\ &f(A_j) \leq f(B_i) < f(B_{i+2}) < f(A_{j+1}). \end{aligned}$$

In the first case we have $B_j \to A_i$ and $B_{j+1} \not\to A_{i+2}$. Then, from definition 4.3,

$$\begin{aligned} t(A_i) &\geq t(B_j) + \tau(B_j, A) \\ t(A_{i+2}) &< t(B_{j+1}) + \tau(B_{j+1}, A) \\ t(B_{j+1}) &= t(B_j) + T_j^B \\ t(A_{i+2}) &= t(A_i) + T_i^A + T_{i+1}^A. \end{aligned}$$

From the definition of a quasi-periodic trace (definition 4.2) we obtain:

$$\begin{aligned} t(A_i) &\geq t(B_j) + \tau_{\min} \\ t(A_{i+2}) &< t(B_{j+1}) + \tau_{\max} \\ t(B_{j+1}) &\leq t(B_j) + T_{\max} \\ t(A_{i+2}) &\geq t(A_i) + 2T_{\min}. \end{aligned}$$

Giving

$$\begin{aligned} t(B_j) + \tau_{\min} + 2T_{\min} &\leq t(A_i) + 2T_{\min} \\ &\leq t(A_{i+2}) \\ &< t(B_{j+1}) + \tau_{\max} \\ &\leq t(B_j) + T_{\max} + \tau_{\max}. \end{aligned}$$

Hence $2T_{\min} + \tau_{\min} < \tau_{\max} + T_{\max}$, that is $\overline{\text{QT}}$.

The second case is similar. We have $B_i \not\to A_j$ and $B_{i+2} \to A_{j+1}$. Then, from definition 4.3,

$$\begin{array}{rcl} t(A_j) & < & t(B_i) + \tau(B_i, A) \\ t(A_{j+1}) & \geq & t(B_{i+2}) + \tau(B_{i+2}, A) \\ t(A_{j+1}) & = & t(A_j) + T_j^A \\ t(B_{i+2}) & = & t(B_i) + T_i^B + T_{i+1}^B. \end{array}$$

From the definition of a quasi-periodic trace (definition 4.2) we obtain:

$$\begin{array}{rcl} t(A_j) & < & t(B_i) + \tau_{\max} \\ t(A_{j+1}) & \geq & t(B_{i+2}) + \tau_{\min} \\ t(A_{j+1}) & \leq & t(A_j) + T_{\max} \\ t(B_{i+2}) & \geq & t(B_i) + 2T_{\min}. \end{array}$$

Giving

$$\begin{aligned} t(A_j) + 2T_{\min} + \tau_{\min} &< t(B_i) + \tau_{\max} + 2T_{\min} + \tau_{\min} \\ &\leq t(B_{i+2}) + \tau_{\min} + \tau_{\max} \\ &\leq t(A_{j+1}) + \tau_{\max} \\ &\leq t(A_j) + T_{\max} + \tau_{\max}. \end{aligned}$$

Hence we have $2T_{\min} + \tau_{\min} < \tau_{\max} + T_{\max}$, that is $\overline{\text{QT}}$.

On the other hand, if condition QT does not hold: $2T_{\min} + \tau_{\min} < T_{\max} + \tau_{\max}$, figure 4.18 shows a trace where

$$B_j \xrightarrow{1} A_i \xrightarrow{1} A_{i+1} \xrightarrow{1} \ldots \xrightarrow{1} A_{i+n} \xrightarrow{0} B_{j+m-1}.$$

Then a discretization f such that $f(B_j) < f(A_i)$ and $f(A_{i+2}) = f(B_{j+1})$ is a valid unitary discretization which violates condition QS.

□

Theorem 4.3 states precisely when the quasi-synchronous abstraction is sound. If a real-time system satisfies the given constraints on (logical) topology and timing, then the quasi-synchronous abstraction can be used to formally verify its properties.

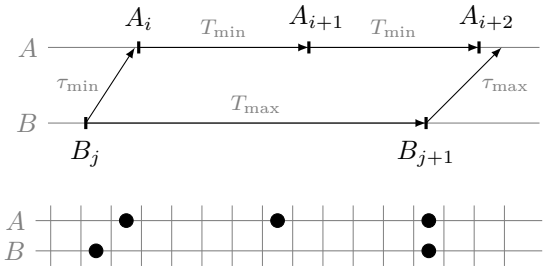


Figure 4.18: Witness for QS $\implies$ QT.

Oversamplings and overwrites

A classic property of the quasi-synchronous abstraction is the existence of bounds on the numbers of successive overwrites n_{ow} (message losses) and oversamplings n_{os} (message duplications).

Property 4.4 (Overwrites, oversamplings)**.** *For a quasi-synchronous system satisfying the conditions of theorem 4.3 a consumer misses at most one message between two successive activations, and reads the same message at most twice.*

Proof. From property 3.1 we have

$$n_{os} = n_{ow} = \left\lceil \frac{T_{\max} + \tau_{\max} - \tau_{\min}}{T_{\min}} \right\rceil - 1.$$

For quasi-synchronous systems, that is quasi-periodic architectures satisfying the conditions of theorem 4.3, we also have

$$2T_{\min} + \tau_{\min} \geq T_{\max} + \tau_{\max}.$$

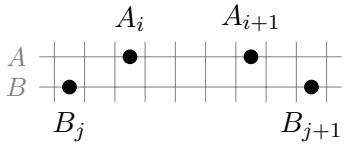
Combining these two results we obtain:

$$n_{os} = n_{ow} = \left\lceil \frac{T_{\max} + \tau_{\max} - \tau_{\min}}{T_{\min}} \right\rceil - 1 \leq \left\lceil \frac{2T_{\min} + \tau_{\min} - \tau_{\min}}{T_{\min}} \right\rceil - 1 \leq 1.$$

□

Alternatively, since we consider quasi-synchronous systems that satisfy the conditions of theorem 4.3, we can safely apply the quasi-synchronous abstraction. Property 4.4 is then a direct consequence of definition 4.7.

Proof. The worst case is:



For the maximum number of overwrites, the first message sent at A_i is overwritten by the message sent at A_{i+1} which is received by B at B_{j+1}. Symmetrically, for the maximum number of oversamplings, activation A_{i+1} oversamples the value sent by B at B_j which is already received by A at A_i. □

4.5 Multirate systems

In this section we extend our results to multirate systems where each node is characterized by its own nominal period. We thus adapt both the real-time model of definition 3.1 and the discrete model of definition 4.7 to reflect these individual activation rates.

Multirate real-time model

We still assume global bounds on the transmission delay, but now each node is characterized by its own nominal period. Definition 3.1 then becomes:

Definition 4.8 (Multirate system)**.** *A multirate system is a finite set of nodes $\mathcal{N}$, where for each node $N \in \mathcal{N}$, the delay T^N between two successive activations is bounded.*

$$0 \leq T^N_{\min} \leq T^N \leq T^N_{\max}.$$

Values are transmitted between nodes with a delay $\tau \in \mathbb{R}$, bounded by $\tau_{\min}$ and $\tau_{\max}$.

$$0 \leq \tau_{\min} \leq \tau \leq \tau_{\max}.$$

Since the proof of theorem 4.2 is mostly based on the bounds on transmission delays, this generalization requires very few changes to the results of section 4.3. Theorem 4.2 becomes

Theorem 4.4. *A multirate system is unitary discretizable if and only if,*

1. *all u-cycles of the communication graph are cycles or balanced u-cycles, or $\tau_{\max} = 0$,*
2. *there is no balanced u-cycle in the communication graph, or $\tau_{\min} = \tau_{\max}$,*
3. *there is no cycle in the communication graph, or for all cycles c*

$$T^c_{\min} \geq L_c \tau_{\max} \tag{CD}$$

where $L_c = \text{size}(c)$, and $T^c_{\min} = \min\{T^N_{\min} \mid N \in c\}$.

Compared to theorem 4.2, the only difference is the last condition where a global bound on the size of all cycles L_c is replaced by a condition that must hold for all cycles of the communication graph.

Proof. The proof is similar to that of theorem 4.2. The only difference is the treatment of cycles (condition C_1). The rest of the proof remains the same. For multirate systems, condition C_1 becomes:

$$\exists c \in \mathcal{C} \text{ such that } T^c_{\min} < L_c \tau_{\max}. \tag{C_1}$$

Assume that C_1 holds. Let c be such a cycle,

$$c = N_0 \leftleftarrows N_1 \leftleftarrows \ldots \leftleftarrows N_{L_c} \leftleftarrows N_0,$$

where N_0 is the node with the smallest lower bound in c, $T^{N_0}_{\min} = T^c_{\min}$. Following the proof of theorem 4.2, we show that it is possible to build a trace with a cycle of positive weight based

on c. Let $\varepsilon = (L_c\tau_{\max} - T_{\min})/L_c > 0$ and $\mathcal{E}$ be a trace where for all events e_i the transmission delay is as long as possible, $\forall e \in \mathcal{E},\ \tau(e, \_) = \tau_{\max}$, and

$$\begin{aligned} t(e'_0) &= 0 \\ t(e_0) &= T^{N_0}_{\min} = T^c_{\min} \\ t(e_{i+1}) &= t(e_i) - (\tau_{\max} - \varepsilon), \qquad \forall 0 \leq i \leq L_c \end{aligned}$$

with $N_{L_c+1} = N_0$. We thus have $t(e_{L_c+1}) = t(e'_0) + T^c_{\min} - L_c(\tau_{\max} - \varepsilon) = t(e'_0)$, that is, $e_{L_c+1} = e'_0$ and we have $\forall 0 \leq i \leq L_c$,

$$\begin{aligned} &N_{i+1} \rightrightarrows N_i \\ &t(e_i) < t(e_{i+1}) + \tau(e_{i+1}, N_i). \end{aligned}$$

which is a cycle of weight 1.

Conversely, suppose there exists a cycle of positive weight of the form

$$x = A^+ \xrightarrow{b_0} B^+ \xrightarrow{b_1} C^+ \xrightarrow{b_2} \ldots \xrightarrow{b_n} A^+,$$

such that the sequence $c = A, B, C \ldots, A$ is a cycle.

Let p be the number of $\xrightarrow{1}$ edges and q the number of $\xrightarrow{0}$ edges in trace x. If a cycle of positive weight is based on a cycle of the communication graph, an edge $\xrightarrow{1}$ can only reflect subsequent activations of the same node (property 4.2). Hence $L_c = \text{size}(c) = q$. We have by definition of $\xrightarrow{1}$ and $\xrightarrow{0}$, that

$$\begin{aligned} e_i \xrightarrow{1} e_{i+1} \implies t(e_{i+1}) &\geq t(e_i) + T^{N_i}_i \\ &\geq t(e_i) + T^{N_i}_{\min} \\ &\geq t(e_i) + T^c_{\min}, \text{ and} \\ e_i \xrightarrow{0} e_{i+1} \implies t(e_{i+1}) &> t(e_i) - \tau(e_{i+1}, N_i) \\ &\geq t(e_i) - \tau_{\max}. \end{aligned}$$

Following the cycle we have $t(e_0) + pT^c_{\min} - q\tau_{\max} < t(e_0)$ and hence $q\tau_{\max} > pT_{\min}$. Since x is a cycle of positive weight, $p \geq 1$. Hence,

$$L_c\tau_{\max} = q\tau_{\max} > pT^c_{\min} \geq T^c_{\min}$$

which violates condition CD. Therefore C_1 holds. □

n/m-quasi-synchrony

We now apply the preceding definitions and results on unitary discretizations to generalize the quasi-synchronous model to multirate systems after the work of [SG12]. More recently a similar generalization appeared in [MBT+15].

A discrete-time model is termed n/m-quasi-synchronous if there are no more than n activations of one node between m successive activations of another. This natural generalization of the quasi-synchronous abstraction allows the expression of arbitrary rational constraints on the relative activation rates of the nodes. It is thus well suited for multirate systems.

As in section 4.4, since any given node only detects the activations of another by receiving the corresponding messages, the quasi-synchronous condition corresponds to two constraints.

For any two nodes,

1. there are no more than n activations between m message receptions, and
2. there are no more than n message receptions between m activations.

This definition can also be formalized using unitary discretizations.

Definition 4.9 (n/m-quasi-synchronous model)**.** *A real-time model is n/m-quasi-synchronous with $n \geq m > 1$ if, for every trace t,*

1. *it has a unitary discretization f, and*
2. *for nodes $A \leftleftarrows B$, there is no i and j such that*

$$\begin{aligned} &f(B_j) < f(A_i) < \cdots < f(A_{i+n}) \leq f(B_{j+m-1}) \text{ or,} \\ &f(A_j) \leq f(B_i) < \cdots < f(B_{i+n}) < f(A_{j+m-1}). \end{aligned} \tag{MS}$$

Following the techniques used in section 4.4 for predicates `check_qs`, we can test if a discrete system is n/m-quasi-synchronous. We use a circular buffer to store the number of activations of a clock `c1` between m activations of another clock `c2`. If the buffer is already full, pushing a new value overwrites the oldest value in the buffer.

```
let node check_nmqs_1(n, m, c1, c2) = ok where
  rec init b = buffer(m - 2)
  and init p = 0
  and init ok = true
  and present c2() → do b = push(last p, last b) done
  and present
      | c1() on (sum(b) + p = n) → do ok = false done
      | c2() → do p = 0 done
      | c1() → do p = last p + 1 done
```

val check_nmqs_1: int × int × unit signal × unit signal $\xrightarrow{\mathrm{D}}$ *bool*

The variable `p` counts the number of activations of `c1` between two successive activations of `c2`. When `c2` is emitted, we push the value of `p` into the buffer `b` and reset its value. Since the buffer `b` is of size $m-2$, the sum of its values corresponds to the number of activations of `c1` between the last $m-1$ ticks of `c2`. The number of activations of `c1` during the last m ticks of `c2` is then `sum(b) + p`. The first condition of definition 4.9 is violated if `c1` is emitted while `sum(b) + p = n`. The second condition can be checked using the same technique.

```
let node check_nmqs_2(n, m, c1, c2) = ok where
  rec init b = buffer(m - 2)
  and init p = 0
  and init ok = true
  and present c1() → do b = push(last p, last b) done
  and present
      | c1() & c2() → do p = 1 done
      | c2() on (sum(b) + p = n) → do ok = false done
      | c1() → do p = 0 done
      | c2() → do p = last p + 1 done
```

val check_nmqs_2: int × int × unit signal × unit signal $\xrightarrow{\mathrm{D}}$ *bool*

Like the second condition of definition 4.7, the roles of `c1` and `c2` are reversed and the counter is also reinitialized to the value 1 if both `c1` and `c2` are emitted during the same instant.

Remark that the quasi-synchronous abstraction of definition 4.7 page 69 is a particular case of definition 4.9 with $n = m = 2$. In that case, the buffers of the `check_nmqs` predicates are of size zero. They are then functionally equivalent to the two `check_qs` predicates presented in section 4.4.

The following theorem is the multirate counterpart of theorem 4.3.

Theorem 4.5. *A multirate system (definition 4.8) is n/m-quasi-synchronous (condition MS) if and only if,*

1. *the conditions of theorem 4.4 hold, and*
2. *for any pair of nodes $A \rightrightarrows B$, the following conditions hold,*

$$nT^A_{\min} + \tau_{\min} \geq (m-1)T^B_{\max} + \tau_{\max}, \text{ and} \qquad (\mathrm{MT}_1)$$
$$nT^B_{\min} + \tau_{\min} \geq (m-1)T^A_{\max} + \tau_{\max}. \qquad (\mathrm{MT}_2)$$

MT_1 ensures that there are no more that n activations of A between m message receptions from B, and MT_2 ensures that there are no more than n message receptions from B between m activations of A.

Proof. The proof resembles that of theorem 4.3. For $n \geq m > 1$, consider a pair of communicating nodes $A \leftleftarrows B$. To show that $\mathrm{MT}_1 \wedge \mathrm{MT}_2 \implies \mathrm{MS}$, assume that condition MS does not hold, then there exist i and j such that

$$f(B_j) < f(A_i) < \cdots < f(A_{i+n}) \leq f(B_{j+m-1}) \text{ or,}$$
$$f(A_j) \leq f(B_i) < \cdots < f(B_{i+n}) < f(A_{j+m-1}).$$

In the first case we have $B_j \to A_i$ and $B_{j+m-1} \not\to A_{i+n}$. Then, from the definition of multirate systems (definition 4.8) we obtain:

$$\begin{aligned} t(A_i) &\geq t(B_j) + \tau_{\min} \\ t(A_{i+n}) &\leq t(B_{j+m-1}) + \tau_{\max} \\ t(B_{j+m-1}) &\leq t(B_j) + (m-1)T^B_{\max} \\ t(A_{i+n}) &\geq t(A_i) + nT^A_{\min}. \end{aligned}$$

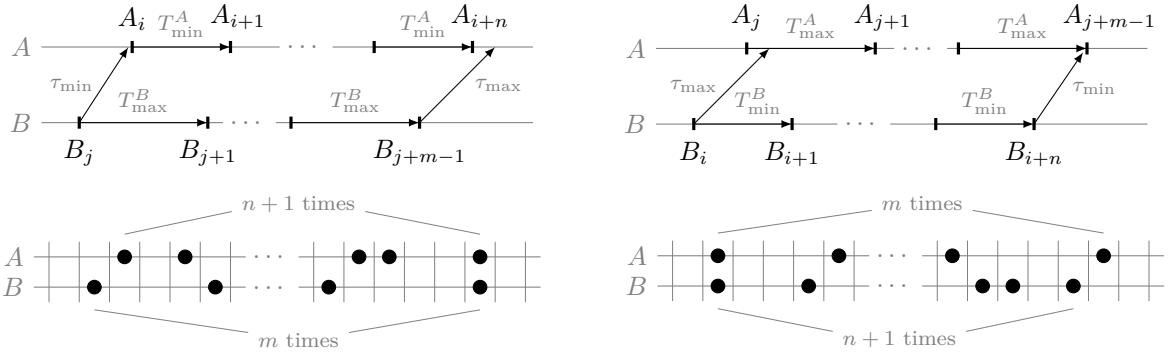
Giving

$$\begin{aligned} t(B_j) + \tau_{\min} + nT^A_{\min} &\leq t(A_i) + nT^A_{\min} \\ &\leq t(A_{i+n}) \\ &< t(B_{j+m-1}) + \tau_{\max} \\ &\leq t(B_j) + (m-1)T^B_{\max} + \tau_{\max}. \end{aligned}$$

Hence we have $nT^A_{\min} + \tau_{\min} < \tau_{\max} + (m-1)T^B_{\max}$, that is $\overline{\mathrm{MT}_1}$.

The second case is similar. We have $B_i \not\to A_j$ and $B_{i+n} \to A_{j+m-1}$. Then, from the definition of multirate systems (definition 4.8), we obtain:

$$\begin{aligned} t(A_j) &\leq t(B_i) + \tau_{\max} \\ t(A_{j+m-1}) &\geq t(B_{i+n}) + \tau_{\min} \\ t(A_{j+m-1}) &\leq t(A_j) + (m-1)T^A_{\max} \\ t(B_{i+n}) &\geq t(B_i) + nT^B_{\min}. \end{aligned}$$



(a) QS $\implies$ MT_1 (b) QS $\implies$ MT_2

Figure 4.19: Witness for MS $\implies$ (MT_1 and MT_2) and the associated unitary discretizations.

Giving

$$\begin{aligned} t(A_j) + nT^B_{\min} + \tau_{\min} &< t(B_i) + \tau_{\max} + nT^B_{\min} + \tau_{\min} \\ &\leq t(B_{i+n}) + \tau_{\min} + \tau_{\max} \\ &\leq t(A_{j+m-1}) + \tau_{\max} \\ &\leq t(A_j) + (m-1)T^A_{\max} + \tau_{\max}. \end{aligned}$$

Hence we have $nT^B_{\min} + \tau_{\min} < \tau_{\max} + (m-1)T^A_{\max}$, that is $\overline{\mathrm{MT}_2}$.

Conversely, if either of the conditions MT_1 or MT_2 does not hold, figure 4.19 shows traces and valid unitary discretizations that violate MS.

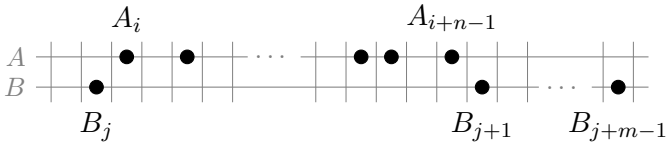
□

Oversamplings and overwrites

Finally we can generalize property 4.4 for n/m-quasi-synchronous systems.

Property 4.5 (Overwrites, oversamplings)**.** *The maximum number of successive overwrites or oversamplings in an n/m-quasi-synchronous system satisfying definition 4.9 is $n-1$.*

Proof. The worst acceptable case is



For the maximum number of overwrites, the $n-1$ messages sent between A_i and A_{i+n-2} are overwritten by the message sent at A_{i+n-1} which is received by B at B_{j+1}. Symmetrically, for the maximum number of oversamplings, the $n-1$ activations of A between A_{i+1} and A_{i+n-1} oversample the value sent by B at B_j which is already received by A at A_i. □

4.6 Bibliographic notes

Most existing work on the quasi-synchronous abstraction either assumes instantaneous communication [BMY+14, MBT+15] or takes the discrete model as given and applies it directly to model and analyze systems [HM06, JHR08, SG12]. We seek to clarify the original definitions [Cas00] and to precisely define the relation between the real-time and discrete-time models. This leads to the understanding of discretization in terms of causality and the restrictions on process intercommunications and timing which are the central contributions of this chapter.

Our work is complementary to the development of abstract domains to statically analyze synchronous real-time systems [Ber08], and to the verification of properties like maximal lost messages, message inversions, and message latency, in an interactive theorem prover [LS14, LGS15].

Logical clocks

As already mentioned in section 4.3, the existence of a unitary discretization is equivalent to the problem of finding a strongly consistent scalar clock. As this is not possible in general [Lam78, RS96], researchers have sought more powerful mechanisms, like vector clocks [Mat89] and matrix clocks [FM82], for capturing the causalities of events. These mechanisms do not resolve the problem posed in this chapter, since the modeling of transmissions as unit delays and the activations of processes on boolean streams require the total ordering given by a global scalar clock: a synchronous modeling of an asynchronous system.

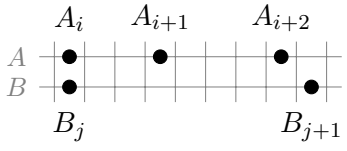
Other interpretations of quasi-synchrony

The interpretation of Caspi's condition sometimes differs from definition 4.1 or definition 4.7. A weaker alternative used in [HM06, MBT+15] is to only forbid traces where a clock ticks *alone* more than twice between two successive activations of another.

Compared to definition 4.7, this condition has the advantage of forbidding a single symmetric subsequence, and unlike definition 4.1, this alternative does not require a perfectly synchronous architecture in the case of unidirectional communications (property 4.3).

This model is weaker than definition 4.7, that is, a valid trace for definition 4.7 is also a valid trace for this alternative definition. Hence, properties that hold in the new model also hold for a system satisfying definition 4.7. But the converse is not true, and in particular, with this alternative definition, we lose the close link between quasi-synchrony and the maximum number of lost or duplicate values.

The worst acceptable case is



Messages sent at A_i and A_{i+1} are received strictly after B_j and are overwritten by the message sent at A_{i+2} which is received by B at B_{j+1}. This trace shows that two messages can be lost, while there are no more than two ticks of one clock *alone* between two ticks of another.

In any case, alternative constraints on activation interleavings do not avoid the main limitation of the quasi-synchronous abstraction, which results from modeling transmissions as unit delays. Any quasi-synchronous system must thus respect the constraints imposed by theorem 4.2 on communication topologies and timing characteristics of the architecture.

n-synchrony

The *n-synchronous* model is another discrete model that relaxes synchronization between nodes using bounded buffers. In an n-synchronous system, unlike in a quasi-synchronous system, the difference of cumulative node activation counts is bounded [CDE$^+$06]. Given the activation conditions of all the nodes (or an overapproximation), a type system has been proposed to statically compute the size of the communication buffers [Pla10]. Compared to an n-synchronous system, in a quasi-synchronous system a node can consistently run twice as fast as another. It is thus impossible to statically bound the size of communication buffers.

Using the terminology introduced in [SG12], n-synchrony bounds the difference between two clocks (*clock bounds*) while quasi-synchrony bounds the difference between the derivative of these two clocks (*drift bounds*).

To avoid buffer overflows, n-synchronous systems require a form of *loose synchronization* between nodes. The relation between a similar model and real-time distributed architectures has recently been studied [DSQ$^+$15]. They assume instantaneous communications thus avoiding the difficulty of the unitary discretization. In this case, theorem 4.2 states that there are no forbidden topologies. Our result on unitary discretizable systems (theorem 4.2) remains valid in an n-synchronous context.

The n-synchronous model has been used to model latency introduced by wires on a system on chip [MPP11]. Compared to a quasi-periodic architecture, the system on chip is triggered by a global clock, and the link between real-time and discrete-time models is straightforward. In this case it is correct to model transmissions with unit delays.

4.7 Conclusion

In this chapter we focused on the *quasi-synchronous abstraction*, a discrete abstraction proposed by P. Caspi for analyzing embedded applications running on quasi-periodic architectures. This abstraction proposes two mechanisms to constrain the state-space of the models:

1. modeling transmissions as unit delays, and
2. constraining interleavings of node activations.

We showed that the first condition of the abstraction is problematic: the model is not sound for general systems of more than two nodes. Then we introduced the notion of *unitary discretization* to characterize traces for which there exists a discretization where transmissions can be modeled as unit delays. We showed that constraints on the discretization function can be gathered in a weighted graph. The existence of a unitary discretization can then be rephrased in terms of cycles in the corresponding graph. Not only does reasoning in the trace graph permitted intuitive proofs, it also lead directly to the conditions on the underlying static communication graph and timing parameters that are necessary and sufficient to recover soundness. In other words, only a precise class of practically-relevant distributed control

systems can be verified without resorting to timed formalisms and tools, and by modeling message transmissions as unit delays.

Building on these results we gave the exact application conditions of the complete quasi-synchronous abstraction. We then illustrated the quasi-synchronous approach with a classic property of quasi-periodic architectures: bounding the maximum number of successive message losses or duplications.

Finally we showed how to generalize the quasi-synchronous approach to multirate systems where each node is characterized by its own nominal period. The condition on interleavings of node activations can then be extended to express arbitrary rational constraints on the relative activation rates of the nodes (n/m-quasi-synchrony).

For a class of quasi-periodic architectures that we precisely characterized, it is thus possible to check safety properties in a purely discrete model. This model capture the essence of the architecture and in particular is subject to the sampling artifacts described in section 3.2. In the next chapter we show how to prevent these artifacts by adding a layer of *middleware* in the code of the nodes to ensure the semantics preservation of a synchronous application running on a quasi-periodic architecture.

Chapter 5

Loosely Time-Triggered Architectures

We show in chapters 3 and 4 how to develop discrete models of a quasi-periodic architecture. However, the architecture is still subject to the sampling artifacts described in section 3.2 (properties 4.4 and 4.5). These artifacts may be acceptable for robust controllers—a PID (proportional–integral–derivative) controller for instance—but are clearly harmful for discrete logic where sampling artifacts can induce incorrect results. One approach to correctly implement a synchronous specification on a quasi-synchronous architecture is to add a layer of *middleware* to eliminate or compensate for sampling artifacts.

One possibility is to rely on a clock synchronization protocol as in the time-triggered architecture (TTA) [Kop11]. Another is to use less constraining protocols as in the Loosely Time-Triggered Architecture (LTTA) [BCLG$^+$02, BCDN$^+$07, TPB$^+$08, CB08, BBC10]. They are simple to implement and involve little additional network communication. They thus remain an interesting alternative despite the undeniable advantages of solutions based on clock synchronization (like straightforward coordination, determinism, and traceability).

Contributions

This chapter present the second main contribution of this thesis. We first show how the discrete model of section 3.3 can be refined to capture the middleware in the modeling of the nodes. Indeed, protocol controllers are also synchronous programs: they can be compiled together with application code. This framework is then instantiated with the two LTTA protocols described in the literature: *back-pressure* [TPB$^+$08] and *time-based* [CB08]. The back-pressure protocol is based on acknowledging the receipt of messages. While efficient, it introduces control dependencies. The time-based protocol is based on a waiting mechanism. It is less efficient but allows controllers to operate more independently.

This idea of unifying the LTTA protocols in a single framework started with [BBC10] where both protocols are expressed as timed Petri nets. Although this formalism helped to derive some theoretical results, such as the worst-case throughput of the protocols, it is not an implementation and the development of such models is complex and error prone. In fact, we started this work by correcting the timed Petri net model [BBBC14]. Our approach gives both a unified formal framework and executable specifications.

Not only do we clarify the models and reasoning presented in the literature—the proofs of the semantics preservation for the back-pressure and the time-based protocols (theorems 5.1 and 5.2) are new—, but we give a simpler version of the time-based protocol and show

that broadcast communication is required. We then propose an optimized protocol—called round-based LTTA—for systems using broadcast communication. Finally, modern clock synchronization protocols are now cost-effective and precise [Kop11, LEWM05, Mil06, CDE+12], raising the question: *Is there really any need for the LTTA protocols?* We thus instantiate our framework one more time with a simple protocol based on clock synchronization and compare it to the LTTA protocols. Results shows that there is no gain in performance when using the LTTA protocols. However, we show that LTTA protocols are simple to implement and remain a lightweight alternative to solution based on clock synchronization.

Outline In section 5.1 we recall the fundamentals of synchronous applications. Then, in section 5.2, we present a general framework based on the model presented in chapter 3 for modeling LTTA protocols. This framework is instantiated with the different LTTA protocols: back-pressure (section 5.3), time-based (section 5.4), and round-based (section 5.5); and a controller based on clock synchronization (section 5.6). Finally, in section 5.7, using our executable framework, we simulate the protocols and compare their performances.

5.1 Synchronous applications

This chapter addresses the deployment of a *synchronous application* onto a quasi-periodic architecture. By synchronous application, we mean a composition of communicating Mealy machines communicating through unit delays. The question of generating such a form from a high-level language like Lustre/SCADE, Signal, Esterel, or the discrete part of Simulink is not addressed here. Still, this composition of communicating Mealy machines can be written in any of these languages.

In the synchronous model, machines are executed in lockstep. But as our intent is to distribute each machine onto its own network node, we must show that a desynchronized execution yields the same overall input/output relation as the reference semantics. The aim of this section is to describe the activation model and the related requirements on communications, and thereby the form of, and the constraints on program distribution. The desynchronized executions we consider are still idealized—reproducing them on systems satisfying definition 3.1 is the subject of sections 5.3 to 5.5.

Definition 5.1 (Mealy machine)**.** *A Mealy machine m is a tuple $\langle s_{\text{init}}, I, O, F\rangle$, where s_{init} is an initial state, I is a set of input variables, O is a set of output variables, and F is a transition function mapping a state and input values to the next state and output values:*

$$F \ : \ \mathcal{S} \times \mathcal{V}^I \to \mathcal{S} \times \mathcal{V}^O$$

where $\mathcal{S}$ is the domain of state values and $\mathcal{V}$ is the domain of variable values.

In the following $\mathcal{X}^\infty = \mathcal{X}^* \cup \mathcal{X}^\omega$ denotes the set of possibly finite streams over elements of the set $\mathcal{X}$. A Mealy machine $m = \langle s_{\text{init}}, I, O, F\rangle$ defines a stream function.

$$[\![m]\!] \ : \ (\mathcal{V}^I)^\infty \to (\mathcal{V}^O)^\infty$$

generated by repeated firings of the transition function from the initial state:

$$\begin{aligned} &s(0) = s_{\text{init}} \\ &s(n),\ o(n) = F(s(n-1), i(n)) \ \forall n > 0. \end{aligned}$$

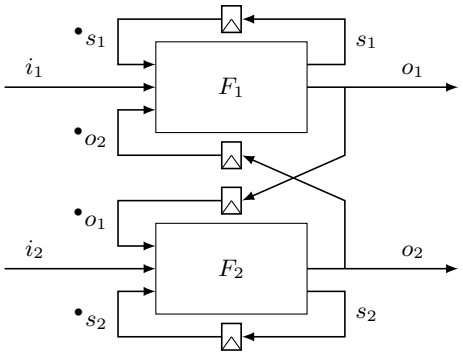


Figure 5.1: Moore-composition of two mealy machines. The ⍁ symbols denote unit delays.

The fact that the outputs of Mealy machines may depend instantaneously on their inputs makes both composition [MR01] and distribution over a network problematic (see [Gir05] for a survey). An alternative is to only consider a *Moore-style* composition of Mealy machines: outputs may be instantaneous but communications between machines must be delayed. A machine must wait one step before consuming a value sent by another machine. This choice precludes the separation of subprograms that communicate instantaneously, but it increases node independence and permits simpler protocols. Figure 5.1 illustrates the Moore-composition of two Mealy machines.

For a variable x, let $^\bullet x$ denote its delayed counterpart (for $n > 0$, $^\bullet x(n) = x(n-1)$). Similarly, let $^\bullet X = \{^\bullet x \mid x \in X\}$. Now, a set of machines $\{m_1,\ m_2,\ \ldots,\ m_p\}$ can be composed to form a *system* $N = m_1 \parallel m_2 \parallel \ldots \parallel m_p$. The corresponding Mealy machine $N = \langle s_{\text{init}}, I, O, F_N \rangle$ is defined by

$$
\begin{aligned}
&I = I_1 \cup \cdots \cup I_p \setminus {}^\bullet O,\\
&O = O_1 \cup \cdots \cup O_p,\\
&s_{\text{init}} = (s_{\text{init}_1}, \ldots, s_{\text{init}_p}, \mathit{nil}, \ldots, \mathit{nil})\\
&F_N(({}^\bullet S, {}^\bullet O),\ I) = ((S, O),\ O)
\end{aligned}
$$

where $S = (s_1, \ldots, s_p)$ and $(s_i, o_i) = F_i(^\bullet s_i, i_i)$. The actual inputs of the global Mealy machine are the inputs of all machines m_i that are not delayed versions of variables produced by other machines. At each step a delayed version of the output of machines m_i, initialized with *nil*, is stored into the state of the global Mealy machine. The notation used to define F_N describes the shuffling of input, output, and delayed variables.

The composition is well defined if the following conditions hold: for all $m_i \neq m_j$,

$$I_i \cap O_j = \emptyset, \tag{5.1}$$

$$O_i \cap O_j = \emptyset, \text{ and} \tag{5.2}$$

$$I_i \setminus {}^\bullet O \cap I_j \setminus {}^\bullet O = \emptyset, \tag{5.3}$$

Equation (5.1) states that no machine ever directly depends on the output of another. Equation (5.2) imposes that a variable is only defined by one machine. Finally, equation (5.3) states that an input from the environment is only consumed by a single machine. Otherwise, it would require synchronization among consumers to avoid nondeterminism. Additionally, since the delayed outputs are initially undefined, the composition is only well defined when the F_i do not depend on them at the initial instant.

Semantics

In the synchronous model, all processes run in lock-step, that is, executing one step of N executes one step of each m_i. Execution order does not matter since no node ever directly depends on the output of another. Thus, at each step, all inputs are consumed simultaneously to immediately produce all outputs. The *Kahn semantics* [Kah74] proposes an alternative model where each machine is considered a function from a tuple of input streams to a tuple of output streams (the variables effectively become unbounded queues). Synchronization between distinct components of tuples and between the activations of elements in a composition are no longer required. The semantics of a program is defined by the sequence of values at each variable:

$$[\![m]\!]^K \ : \ (\mathcal{V}^\infty)^I \to (\mathcal{V}^\infty)^O.$$

Property 5.1. *For Mealy machines, composed as described above, the synchronous semantics and the Kahn semantics are equivalent*[1]

$$[\![m]\!] \approx [\![m]\!]^K.$$

Proof. We write $x :: xs \in \mathcal{V}^\infty$ to represent a stream of values, where $x \in \mathcal{V}$ is the first value of the stream, and $xs \in \mathcal{V}^\infty$ denotes the rest of the stream. Let us first prove for n-tuples of finite or infinite streams *of the same length* that $(\mathcal{V}^n)^\infty \approx (\mathcal{V}^\infty)^n$. We define:

$$\begin{aligned}
&F : (\mathcal{V}^n)^\infty \to (\mathcal{V}^\infty)^n \\
&F(x_1, \ldots, x_n) :: (xs_1, \ldots, xs_n) = (x_1 :: xs_1, \ldots, x_n :: xs_n) \\
&G : (\mathcal{V}^\infty)^n \to (\mathcal{V}^n)^\infty \\
&G(x_1 :: xs_1, \ldots, x_n :: xs_n) = (x_1, \ldots, x_n) :: (xs_1, \ldots, xs_n).
\end{aligned}$$

By construction, streams $x_1 :: xs_1, \ldots, x_n :: xs_n$ all have the same length. Hence, $F \circ G = Id$ and $G \circ F = Id$. This isomorphism can be lifted naturally to functions and we obtain $(\mathcal{V}^I)^\infty \to (\mathcal{V}^O)^\infty \approx (\mathcal{V}^\infty)^I \to (\mathcal{V}^\infty)^O$ for streams of the same length.

Mealy machines always consume and produce streams of the same length since the execution of a Mealy machine consumes all inputs at each step and produces all outputs. The two semantics are thus equivalent. □

The overall idea is to take a synchronous application that has been arranged into a Moore-composition of Mealy machines $N = m_1 \,||\, m_2 \,||\, \ldots \,||\, m_p$, so that each machine m_i can be placed on a distinct network node. If the transmission and consumption of values respects the Kahn semantics, then the network correctly implements the application. Since we do not permit instantaneous dependencies between variables computed at different nodes, a variable x computed at one node may only be accessed at another node through a *unit delay*, that is, a delay of one logical step. In this way we do not need to *microschedule* node activations according to their inter-dependencies.

Note that communications follow the model of section 3.3 page 37 which already forbids instantaneous communication. Unit delays are thus part of the communication process and need not to be included in the implementation of the nodes.

[1] $A \approx B$ means here that A and B are isomorphic.

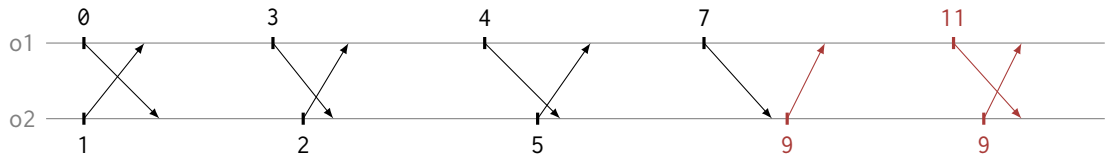


Figure 5.2: Execution trace of the program qp_app that violates the Kahn semantics of the embedded application.

5.2 General framework

We now consider the implementation of a synchronous application S of p Mealy machines communicating through unit delays on a quasi-periodic architecture with p nodes.

This task is trivial if the underlying nodes and network are *completely synchronous*, that is, $T_{\min} = T_{\max} \geq \tau_{\max}$ and with all elements initialized simultaneously. One simply compiles each machine and assigns it to a node. At each tick, all the machines compute simultaneously and send values to be buffered at consumers for use at the next tick. The synchronous semantics of an application is preserved directly.

Consider for instance the following two-node application where each Mealy machine only depends on the output of the other.

```
let node m1(po2) = 0 → (po2 + 2)
let node m2(po1) = 1 → (po1 + 2)
```

```
let node app() = o1, o2 where
  rec o1 = m1(po2)
  and o2 = m2(po1)
  and po1 = pre o1
  and po2 = pre o2
```

o1	0	3	4	7	8	11	12	15	...
o2	1	2	5	6	9	10	13	14	...

Our goal is to compute o1 and o2 on two distinct nodes. The unit delays are abstracted by the fact that nodes do not require an input for the first activation and then execute with the last received value from the other node. Delay operators are thus replaced by communication links.

The deployment of our application over a quasi-periodic architecture can then be modelled following the technique presented in section 3.3. A complete model in Zélus is

```
let node qp_app(c1, dc1, c2, dc2) = o1, o2 where
  rec present c1() → do emit o1 = m1(po2) done
  and present c2() → do emit o2 = m2(po1) done
  and po1 = link(c1, dc1, o1, -1)
  and po2 = link(c2, dc2, o2, -1)
```

where signals c1 and c2 model nodes activations and their delayed versions dc1 and dc2 model communication delays.

On a quasi-periodic architecture, node activations are not synchronized and we must confront the sampling artifacts described in section 3.2: duplication, loss of data, and unintended signal combination. Figure 5.2 show an example trace of the program qp_app that violates the Kahn semantics of the embedded application app.

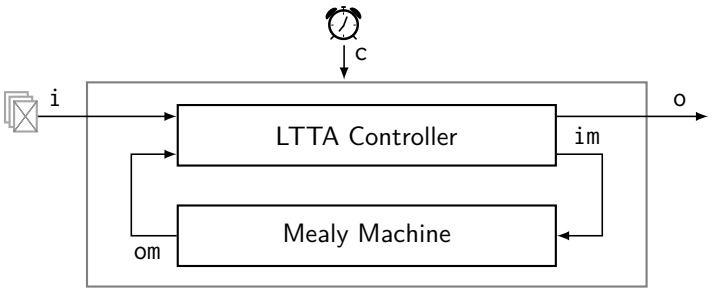


Figure 5.3: Schema of an LTTA node: At instants determined by the protocol, the controller samples a list of inputs to triggers the embedded machine, and controls the publication of the output. Symbols ⊠ are implemented by the `mem` function defined in section 3.3 page 37.

We thus introduce a layer of middleware between application and architecture. An LTTA is the combination of a quasi-periodic architecture with a protocol that preserves the semantics of synchronous applications. We denote the implementation of an application S on a quasi-periodic architecture as LTTA(S).

LTTA controllers

An LTTA node is formed by composing a Mealy machine with a controller that determines when to execute the machine and when to send outputs to other nodes. The basic idea comes from the *shell wrappers* of LID [CMSV01, CSV02]. The schema is shown in figure 5.3 and is modeled in Zélus as:

```
let node ltta_node(i) = o where
  rec (o, im) = ltta_controller(i, om)
  and present im(v) → do emit om = machine(v) done
```

val ltta_node : α *list* $\xrightarrow{\mathrm{D}}$ β *signal*

The `ltta_controller` node is instantiated with one of the controllers described in the following sections. At instants determined by the protocol, the controller samples a list of inputs from incoming LTTA links `i` and passes them on `im` to trigger the machine, which produces output `om` (which may be a tuple). The value of `om` is then sent on outgoing LTTA links `o` when the protocol allows.

The function of the controller is to preserve the semantics of the global synchronous application by choosing:

1. when to execute the machine (emission of signal `im`), and,
2. when to send the resulting outputs (emission of signal `o`).

All the protocols presented in the following ensure that before sending a new value, the previous one has been read by all consumers.

Fresh values

The LTTA controllers must detect when a fresh write is received in an attached shared memory even when the same value is sent consecutively. An *alternating bit* protocol suffices for this

task since the controllers ensure that no values are missed.

```
type α msg = {data: α; alt: bool}

let node alternate(i) = o where
  rec present i(v) → local flag in
        do flag = true → not (pre flag)
        and emit o = {data = v; alt = flag} done
```

val alternate: α *signal* $\xrightarrow{\mathrm{D}}$ α *msg signal*

A message, that is, a value of type α `msg`, is a record with two fields: `data` and `alt`. Function `alternate` turns the inputs received on signal `i` into messages. The value of the boolean variable `flag` is paired with each new value `v` received on signal `i` to form a message. Its value alternates between *true* and *false* at each emission of `i`. This simple protocol logic is readily incorporated into the link model of section 3.3.

```
let node ltta_link(c, dc, i, mi) = o where
  rec s = channel(c, dc, i)
  and o = mem(alternate(s), {data = mi; alt = false})
```

val ltta_link: unit signal $\times$ *unit signal* $\times$ α *signal* $\times$ α $\xrightarrow{\mathrm{D}}$ α *msg*

An alternating bit is associated to each new value stored in the memory.

Within a controller, the freshness of an incoming value can now be detected and signaled:

```
let node fresh(i, r, st) = o where
  rec init m = st
  and present r(_) → do m = i.alt done
  and o = (i.alt ≠ last m)
```

val fresh: α *msg* $\times$ β *signal* $\times$ *bool* $\xrightarrow{\mathrm{D}}$ *bool*

Variable `m` stores the alternating bit associated with the last read value. It is updated at each new read signaled by an emission on signal `r`. A fresh value is detected when the current value of the alternating bit differs from the one stored in `m`, that is, when `i.alt` $\neq$ `last m`. The boolean flag `st` states whether or not the initial value is considered as fresh.

A complete example

We can now write a complete model of the deployment of our simple application on a quasi-periodic architecture. Each Mealy machine is controlled by an LTTA controller:

```
let node ltta_m1(po2) = o where
  rec (o, im) = ltta_controller(po2, om)
  and present im(v) → do emit om = m1(v) done

let node ltta_m2(po1) = o where
  rec (o, im) = ltta_controller(po1, om)
  and present im(v) → do emit om = m2(v) done
```

The complete model link these two LTTA nodes with communication links.

```
let node ltta_app(c1, dc1, c2, dc2) = o1, o2 where
  rec present c1() → do o1 = ltta_m1(po2) done
  and present c2() → do o2 = ltta_m2(po1) done
  and po1 = ltta_link(c1, dc1, o1)
  and po2 = ltta_link(c2, dc2, o2)
```

The goal is now to design LTTA protocols, that is, implementations of `ltta_controller`, that ensure the preservation of the Kahn semantics of the embedded application. In other words for each LTTA protocol P we must show that:

$$[\![\mathrm{LTTA}_{\mathrm{P}}(S)]\!]^K = [\![S]\!]^K.$$

We now present the LTTA protocols. There are two historical proposals, one based on back-pressure (back-pressure LTTA), and another based on time (time-based LTTA); and two optimizations for networks using broadcast communication (round-based LTTA).

5.3 Back-pressure LTTA

The back-pressure protocol [TPB+08] is inspired by *elastic circuits* [CK07, CKLS06] where a consumer node must acknowledge each value read by writing to a *back pressure* link [Car06] connected to the producer. This mechanism allows executing a synchronous application on an asynchronous architecture while preserving the Kahn semantics. In an elastic circuit, nodes are triggered as soon as all their inputs are available. This does not work for LTTA nodes since they are triggered by local clocks, so a *skipping* mechanism was introduced in [TPB+08] and included in later Petri net formalizations [BBC10, BBBC14].

For each link from a node A to a node B, we introduce a back-pressure link from B to A. This link is called `a` (acknowledge) at B and `ra` (receive acknowledge) at A. The controller, shown in figure 5.4, is readily programmed in Zélus:

```
let node bp_controller(i, ra, om, mi) = (o, a, im) where
  rec m = mem(om, mi)
  and automaton
      | Wait →
          do (* skip *)
          unless all_inputs_fresh then
            do emit im = data(i) and emit a in Ready
      | Ready →
          do (* skip *)
          unless all_acks_fresh then
            do emit o = m in Wait

  and all_inputs_fresh = forall_fresh(i, im, true)
  and all_acks_fresh = forall_fresh(ra, o, false)
```

val bp_controller: α *msg list* $\times$ *unit msg list* $\times$ γ *signal* $\times$ $\gamma \xrightarrow{\mathrm{D}}$ γ *signal* $\times$ *unit signal* $\times$ α *list signal*

The controller automaton has two states. It starts in `Wait` and skips at each tick until fresh values have been received on all inputs. It then triggers the machine (`data(.)` accesses the `data`

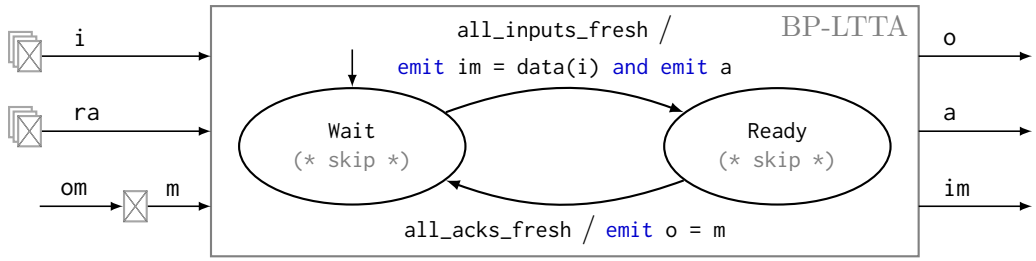


Figure 5.4: The back-pressure LTTA controller. The additional inputs `ra` are acknowledgments from consumers. The additional output `a` is for acknowledging producers.

field of the `msg` structure), stores the result in a local memory `m`, sends an acknowledgment to the producer, and transitions immediately to `Ready`. The controller skips in `Ready` until acknowledgments have been received from all consumers indicating that they have consumed the most recently sent outputs. It then sends the outputs from the last activation of the machine and returns to `Wait`.

The freshness of the inputs since the last execution of the machine is tested by a conjunction of `fresh` nodes (`forall_fresh(i, im, true)`). The controller also tests whether fresh acknowledgments have been received from all consumers since the last emission of the output signal `o`.

Initially there are no fresh acknowledgements since controllers start in the `Wait` state. However, we can safely assume that the initial values of the inputs `i` are fresh since the embedded machines do not require this value to compute the first step.

Remark 5.1. The composition of a back-pressure controller and a Mealy machine to form an LTTA node is well defined. Indeed, the dependency graph of the controller is:

$$\texttt{im} \leftarrow \texttt{i} \quad \texttt{a} \leftarrow \texttt{i} \quad \texttt{o} \leftarrow \texttt{ra} \quad \texttt{o} \leftarrow \texttt{m}.$$

Since the communication with the embedded machine adds the dependency $\texttt{om} \leftarrow \texttt{im}$, the composition of the two machines is free of cycles and therefore well defined.

Preservation of Semantics

This result was first proved in [TPB+08] for networks of nodes communicating through buffers of arbitrary size. Another proof is given in [BBC10,BBBC14] based on the relation with elastic circuits. We give here a new straightforward proof based on the following *liveness* property.

Property 5.2. *Let $t(E_k^N)$ be the date of the kth execution of the embedded machine of a node N. For $k > 0$, and for any node N, we have:*

$$t(E_k^N) \leq 2(\tau_{\max} + T_{\max})(k-1).$$

Proof. This property is shown by induction on k.

Initialization Since all nodes start at $t = 0$ and since they can execute immediately without having received values from other nodes, we have for all nodes N, $t(E_1^N) = 0$.

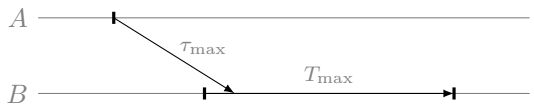


Figure 5.5: The worst case transmission delay on a quasi-periodic architecture is $T_{\max} + \tau_{\max}$.

Induction Assume the property holds up to and including k. At worst, the last node executes and sends an acknowledgment at $t = 2(\tau_{\max} + T_{\max})(k-1)$. The last acknowledgment is thus received at worst $\tau_{\max}$ later, just after a tick of a receiver's clock. Therefore the receiver does not detect the message until $t + \tau_{\max} + T_{\max}$. The worst-case transmission delay on a quasi-periodic architecture, illustrated in figure 5.5, is thus $T_{\max} + \tau_{\max}$. The latest kth publication then occurs at $t + \tau_{\max} + T_{\max}$. Symmetrically this publication is detected at worst $\tau_{\max} + T_{\max}$ later. Hence the $(k+1)$th execution occurs at $t + 2(\tau_{\max} + T_{\max})$, that is, at $2(\tau_{\max} + T_{\max})k$. □

Consequently, in the absence of crashes, nodes never block, which is enough to ensure the preservation of semantics.

Theorem 5.1 ([TPB+08, BBC10])**.** *Implementing a synchronous application S over a quasi-periodic architecture (definition 3.1 page 34) with back-pressure controllers preserves the Kahn semantics of the application:*

$$[\![LTTA_{\text{BP}}(S)]\!]^K = [\![S]\!]^K.$$

Proof. Back-pressure controllers ensure that nodes always sample fresh values from the memories (guard `all_inputs_fresh`) and never overwrite a value that has not yet been read (guard `all_acks_fresh`). Since property 5.2 ensures that nodes will always execute another step, the Kahn semantics of the application is preserved. □

Performance Bounds

Property 5.2 can be used for the worst-case performance analysis of back-pressure LTTA nodes.

Theorem 5.2 ([BBC10])**.** *The worst case throughput of a back-pressure LTTA node is*

$$\lambda_{\text{BP}} = 1/2(T_{\max} + \tau_{\max}).$$

Proof. This result follows from property 5.2. In the worst case, the delay between two successive executions of a node is $2(T_{\max} + \tau_{\max})$. □

5.4 Time-based LTTA

The time-based LTTA protocol realizes a synchronous execution on a quasi-periodic architecture by alternating *send* and *execute* phases across all nodes. Each node maintains a local countdown whose initial value is tuned for the timing characteristics of the architecture so that, when the countdown elapses, it is safe to execute the machine or publish its results.

A first version of the time-based LTTA protocol was introduced in [Cas00]. The protocol was formalized as a Mealy machine with five states in [CB08] and a simplified version was modeled with Petri nets in [BBC10, BBBC14]. We propose an even simpler version that can be expressed as a two-states automaton, formalize it in Zélus, and prove its correctness.

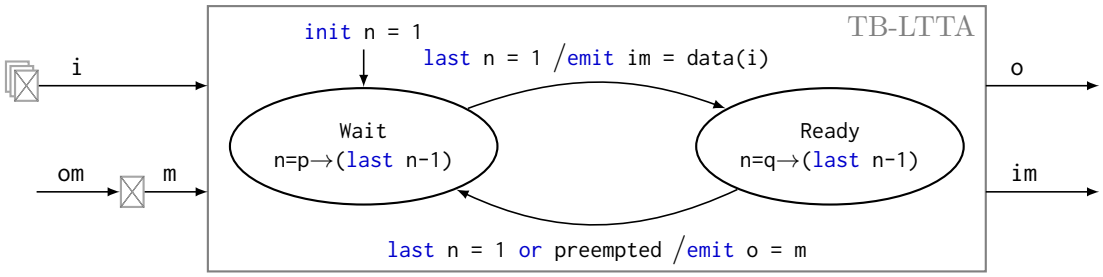


Figure 5.6: The time-based LTTA controller. A counter `n` is decremented in each state initialized with value `p` in state `Wait` and `q` in state `Ready`; `preempted` indicates that a fresh value was received on some input.

Unlike the back-pressure protocol, the time-based protocol requires *broadcast communication* and acknowledgment values are not sent when inputs are sampled.

Assumption 5.1 (Broadcast Communication)**.** *All variable updates must be visible to all nodes and each node must update at least one variable.*

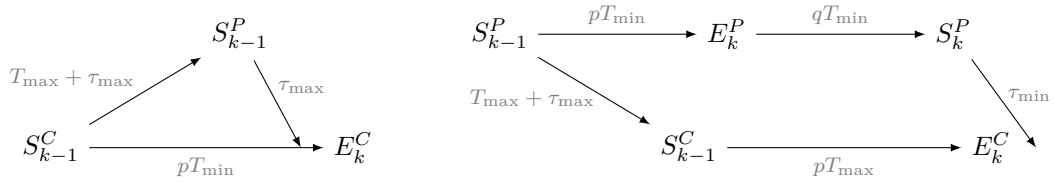
The controller for the time-based protocol is shown in figure 5.6, for parameters `p` and `q`.

```
let node tb_controller(i, om, mi) = (o, im) where
  rec m = mem(om, mi)
  and init n = 1
  and automaton
      | Wait →
          do n = p → (last n - 1)
          unless (last n = 1) then
            do emit im = data(i) in Ready
      | Ready →
          do n = q → (last n - 1)
          unless ((last n = 1) or preempted) then
            do emit o = m in Wait

  and preempted = exists_fresh(i, im, true)
```

val tb_controller: α *msg list* $\times$ β *signal* $\times$ $\beta \xrightarrow{\mathrm{D}}$ β *signal* $\times$ α *list signal*

The controller automaton has two states. Initially, it passes via `Wait`, emits the signal `im` with the value of the input memory `i` and thereby *executes* the machine, stores the result in the local memory `m`, and enters `Ready`. In `Ready`, the equation `n = q → (last n - 1)` initializes a counter `n` with the value `q` and decrements it at each subsequent tick of the clock `c`. At the instant when the `Ready` counter would become zero, that is, when the previous value `last n` is one, the controller passes directly into the `Wait` state, resets the counter to `p`, and *sends* the previously computed outputs from the memory `m` to `o`. It may happen, however, that the local clock is much slower than those of other nodes. In this case, a fresh value from any node, `exists_fresh(i, im)`, preempts the normal countdown and triggers the transition to `Wait` and the associated writing of outputs (`exists_fresh` is essentially a disjunction of `fresh` nodes). The `Wait` state counts down from `p` to give all inputs enough time to arrive before the machine is retriggered.



(a) Property 5.3: $S^P_{k-1} \prec E^C_k$.

(b) Property 5.4: $E^C_k \prec S^P_k$.

Figure 5.7: Explanation of the proofs of properties 5.3 and 5.4.

Basically, nodes slow down by counting to accommodate the unsynchronized activations of other nodes and message transmission delays, but accelerate when they detect a new message.

Remark 5.2. The composition of a time-based controller and a Mealy machine to form an LTTA node is always well defined. The proof is similar to that of remark 5.1. The dependency graph of a node is:

$$\mathtt{n} \leftarrow \mathtt{i} \quad \mathtt{o} \leftarrow \mathtt{i} \quad \mathtt{o} \leftarrow \mathtt{m} \quad \mathtt{om} \leftarrow \mathtt{im} \quad \mathtt{im} \leftarrow \mathtt{i}.$$

It has no cyclic dependencies.

Preservation of the semantics

The time-based protocol only preserves the Kahn semantics of the application if the countdown values p and q are correctly chosen. Similar results can be found in [CB08, BBC10, BBBC14] for previous versions of the protocol.

Theorem 5.3. *The Kahn semantics of a synchronous application S implemented on a quasi-periodic architecture (definition 3.1) with broadcast communication (assumption 5.1) using time-based controllers is preserved,*

$$[\![LTTA_{\text{TB}}(S)]\!]^K = [\![S]\!]^K$$

provided that both

$$p > \frac{2\tau_{\max} + T_{\max}}{T_{\min}} \tag{5.4}$$

$$q > \frac{\tau_{\max} - \tau_{\min} + (p+1)T_{\max}}{T_{\min}} - p. \tag{5.5}$$

Proof. The theorem follows from two properties which together imply that the kth execution of a node samples the $(k-1)$th values of its producers. Since nodes communicate through unit delays, the Kahn semantics is preserved.

Property 5.3 ($S^P_{k-1} \prec E^C_k$)**.** *For $k > 0$, the $(k-1)$*th *sending of a producer is received at its consumers before their respective k*th *executions.*

Property 5.4 ($E^C_k \prec S^P_k$)**.** *For $k > 0$, the k*th *execution of a consumer occurs before the k*th *sending from any of its producers is received.*

The properties are shown by induction on k.

Initialization Nodes start at $t = 0$ and execute immediately (E_1^C) without having to receive values from other nodes. The slowest possible consumer first executes at $pT_{\max}$. On the other hand, the smallest delay before the first send of any producer arrives at the consumer is $pT_{\min} + qT_{\min} + \tau_{\min}$ (countdowns in `Wait` and `Ready` with the shortest possible ticks for the first node to publish). From equations (5.4) and (5.5) we then have

$$(p+q)T_{\min} + \tau_{\min} > \tau_{\max} + (p+1)T_{\max} > pT_{\max},$$

which guarantees that the consumer executes before the reception of the new value.

Induction Assume that the properties hold up to and including $k-1$. The proofs proceed by considering the worst-case scenarios illustrated in Figure 5.7.

For property 5.3, if the kth execution of a consumer E_k^C occurs at time t then its $(k-1)$th sending S_{k-1}^C must have occurred at or before $t - pT_{\min}$ (countdown in `Wait` with the shortest possible ticks). This sending is detected by any node at worst $T_{\max} + \tau_{\max}$ later, which causes a producer in the `Ready` state to send (a producer in the `Wait` state has already done so), with the value arriving at the consumer at most $\tau_{\max}$ later. Equation (5.4) guarantees that this happens before the consumer executes. If node C was not the first to send the $(k-1)$th value, S_{k-1}^P would have occured even earlier.

For property 5.4, if the kth execution of a consumer E_k^C occurs at time t then its $(k-1)$th sending S_{k-1}^C cannot have occurred before $t - pT_{\max}$ (countdown in `Wait` with the longest possible ticks). The first send by a producer in the $(k-1)$th round S_{k-1}^P cannot occur before $t - pT_{\max} - (T_{\max} + \tau_{\max})$, since any send preempts the consumer in `Ready` at worst after a delay of $T_{\max} + \tau_{\max}$. Since the smallest delay before the subsequent kth send of any producer arrives at the consumer is $pT_{\min} + qT_{\min} + \tau_{\min}$ (countdowns in `Wait` and `Ready` with the shortest possible ticks for the first node to publish), equation (5.5) guarantees that the kth execution of the consumer occurs beforehand. □

Broadcast Communication The time-based protocol does not wait for acknowledgments from all receivers but rather sends a new value as soon as it detects a publication from another node. Controllers thus operate more independently, but broadcast communication is necessary. Otherwise, consider the scenario of figure 5.8 obtained by adding a third node N to the scenario in figure 5.7b such that it communicates with node P but not node C. Now, P may be preempted in the `Ready` state one tick after E_k^P causing it to send a message that arrives at C at $S_{k-1}^P + (p+1)T_{\min} + \tau_{\min}$. Since node C would not be preempted by N but only by P, in the worst case E_k^C occurs $(p+1)T_{\max} + \tau_{\max}$ after S_{k-1}^P. Property 5.4 would then require the impossible condition

$$(p+1)T_{\min} + \tau_{\min} > (p+1)T_{\max} + \tau_{\max}.$$

Global synchronization Properties 5.3 and 5.4 imply strictly more than the preservation of the Kahn semantics of an application.

Corollary 5.1. *The time-based controller ensures a strict alternation between* execute *and* send *phases throughout the architecture.*

Proof. Since the time-based protocol requires broadcast communication, each node is a producer and consumer for all others. Therefore, properties 5.3 and 5.4 impose a strict alternation between *execute* and *send* phases. □

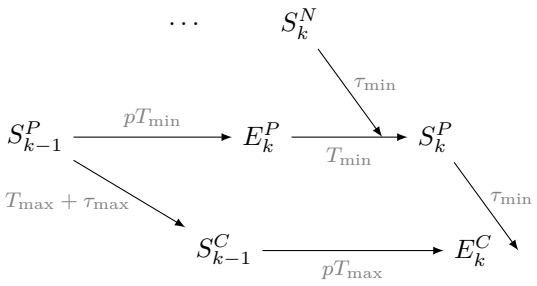


Figure 5.8: Behavior of the time-based protocol without broadcast communication. Node N preempts node P but not node C. Then node P preempts node C.

Performance bounds

Optimal performance requires minimal values for p and q:

$$p^* = \left\lfloor \frac{2\tau_{\max} + T_{\max}}{T_{\min}} \right\rfloor + 1$$
$$q^* = \left\lfloor \frac{\tau_{\max} - \tau_{\min} + (p+1)T_{\max}}{T_{\min}} - p \right\rfloor + 1$$

where $\forall x \in \mathbb{R}$, $\lfloor x \rfloor$ denotes the greatest integer i such that $i \leq x$.

Theorem 5.4. *The worst-case throughput of a time-based LTTA node is:*

$$\lambda_{\text{TB}} = 1/(p^* + q^*)T_{\max}.$$

Proof. The slowest possible node spends $p^*T_{\max}$ in `Wait` and $q^*T_{\max}$ in `Ready`. □

Note that this case only occurs if all nodes are perfectly synchronous and run as slowly as possible. Otherwise, slow nodes would be preempted by the fastest one, thus improving the overall throughput. To give a rough comparison with theorem 5.2, note that we have $p, q \geq 2$ thus, in any case $\lambda_{\text{TB}} \leq 1/4T_{\max}$. A more detailed comparison can be found in section 5.7.

5.5 Round-based LTTA

Compared to the back-pressure protocol, the time-based protocol forces a global synchronization of the architecture. But running the back-pressure protocol under the same broadcast assumption (assumption 5.1) also induces such strict alternations since every node must wait for all others to execute before sending a new value. However, when all nodes communicate by broadcast, there are simpler and more efficient alternatives. We propose two optimizations for these particular networks.

The idea of the round-based controller is to force a node to wait for messages from all other nodes before computing and sending a new value. Nodes together perform rounds of execution. Unfortunately, at the start of a round, a value sent from a faster node may be received at a slower one and overwrite the last received value before the latter executes. A simple solution, based on the synchronous network model [Lyn96, Chapter 2], is to introduce separate communication and execution phases. In this case, we could simply execute each

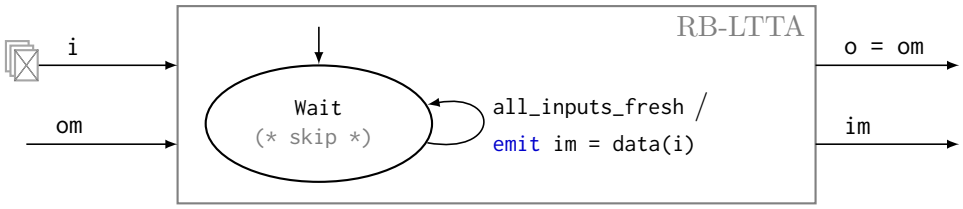


Figure 5.9: The round-based LTTA controller. Acknowledgment is no longer required. When all inputs are detected, the controller triggers the embedded machine and directly sends the output om to other nodes.

application every two rounds. But since lock-step execution ensures that no node can execute more than twice between two activations of any other, it is enough to communicate via buffers of size two. This ensures that messages are never overwritten even if nodes execute the application and directly send the output at every activation. Acknowledgments are no longer required. The Zélus code of the controller shown in figure 5.9 is:

```
let node rb_controller(i, om) = (o, im) where
  rec automaton
      | Wait →
          do (* skip *)
          unless all_inputs_fresh then
            do emit im = data(i) in Wait

  and all_inputs_fresh = forall_fresh(i, im, true)
  and o = om
```

val rb_controller: α *msg list* $\times$ β *signal* $\xrightarrow{\mathrm{D}}$ β *signal* $\times$ α *list signal*

Compared to the back-pressure and time-based protocols, a local memory is not required to store the result of the embedded Mealy machine since the machine's output is immediately sent to other nodes.

Remark 5.3. The composition of a round-based controller and a Mealy machine to form an LTTA node is always well defined. The proof is again similar to that of remark 5.1. The dependency graph of a node is:

$$\mathtt{o} \leftarrow \mathtt{om} \quad \mathtt{om} \leftarrow \mathtt{im} \quad \mathtt{im} \leftarrow \mathtt{i}.$$

It has no cyclic dependencies.

Preservation of the semantics For systems using broadcast communication (assumption 5.1), round-based controllers induce a synchronous execution throughout the entire system thus ensuring the preservation of the Kahn semantics. All nodes execute at approximately the same time.

Performance bounds Compared to nodes controlled by the back-pressure protocol, round-based nodes can be twice as fast since they immediately send the output of the embedded machine at each step.

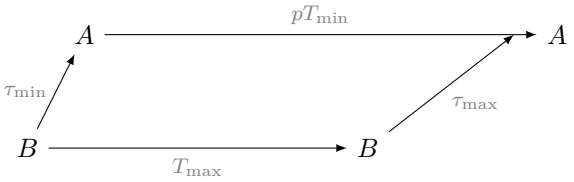


Figure 5.10: Explanation of the proof of property 5.5.

Theorem 5.5. *The worst case throughput of a round-based LTTA node is:*

$$\lambda_{\text{RB}} = 1/(T_{\max} + \tau_{\max}).$$

Proof. Suppose that the last execution of the $(k-1)$th round occurs at time t. At worst, a node detects this last publication and sends its message at $t + \tau_{\max} + T_{\max}$. The last execution of the kth round thus occurs $\tau_{\max} + T_{\max}$ after the last execution of the previous round. □

Timeout

Like the back-pressure protocol, the round-based protocol uses blocking communication. If a node crashes, the entire application stops. To avoid such problems, a classic idea is to add timeouts [ADLS94] and to run a crash detector together with the round-based controller on each node. When a controller executes a step of the application, it knows which other nodes are still functioning, since it has received messages from them, and which have crashed. It can continue to compute using the values last received from crashed nodes.

At each activation, nodes broadcast a *heartbeat* message to signal that they are still active. Every node A maintains a counter initialized to a value p for each other node. The counter corresponding to a node B is reset to its initial value whenever a *heartbeat* message is received from B. The following property ensures that when the counter reaches zero, node A can conclude that B has crashed. We call this last protocol *timed round-based* LTTA.

Property 5.5 ([ADLS94])**.** *For all nodes A, the counter associated to another node B can only reach zero if B crashed, provided that:*

$$p > \frac{\tau_{\max} - \tau_{\min} + T_{\max}}{T_{\min}} \tag{5.6}$$

Proof. The proof involves considering the worst case scenario illustrated in figure 5.10. Each time a node B executes, it sends a *heartbeat* message to A. The maximum difference between the times of two consecutive sends is $T_{\max}$. In the worst case, A receives the first message after the shortest possible delay $\tau_{\min}$ and the second after the longest possible delay $\tau_{\max}$. If A runs as fast as possible the counter reaches zero $pT_{\min}$ after the reception of the first message. Hence the condition $\tau_{\min} + pT_{\min} > \tau_{\max} + T_{\max}$ suffices to ensure that the counter only reaches zero if node B has crashed. □

The Zélus code for the timeout mechanism is:

```
let node timeout(i_live) = (n ≤ 0) where
  rec reset n = p fby (n - 1) every i_live
```

val timeout: bool $\xrightarrow{\text{D}}$ *bool*

There is one additional boolean input `i_live` for each node. It indicates if a *heartbeat* message has been received since the last activation. This input can be modeled by a simple boolean memory set by the delayed clock of the producer `dcs` and reset by the clock of the receiver `cr`.

```
let node live(dcs, cr) = o where
  rec init o = false
  and present
      | dcs() → do o = true done
      | cr() → do o = false done
```

val live: unit signal × *unit signal* $\xrightarrow{\text{D}}$ *bool*

A node executes a step of the application if for every other node it has either received a fresh message or detected a crash. In our model, we need only replace the implementation of `fresh(i, r, st)` (section 5.2) with:

```
let node timed_fresh(i, i_live, r, st) =
  fresh(i, r, st) or timeout(i_live)
```

val timed_fresh: α *msg list* × *bool* × β *signal* × *bool* $\xrightarrow{\text{D}}$ *bool*

Performance bounds In the absence of crashes the timeout mechanism has no influence on the behavior of nodes (property 5.5) and the timed round-based protocol coincides with the round-based one. Otherwise the minimal value for the initial value p is:

$$p^* = \left\lfloor \frac{\tau_{\max} - \tau_{\min} + T_{\max}}{T_{\min}} \right\rfloor + 1.$$

When one or more nodes crash, active nodes wait at worst $p^*T_{\max}$ before detecting the problem and only then execute a step of the application and send the corresponding message. The delay between two successive rounds is thus bounded by $p^*T_{\max} + \tau_{\max}$.

Since every node broadcasts a message at every step, the timeout mechanism has a high message complexity. An alternative is to send a *heartbeat* message only once every k steps and to adjust the initial value of the counters appropriately. The worst case delay between two successive rounds increases accordingly.

5.6 Clock synchronization

The LTTA protocols are designed to accommodate the loose timing of node activations in a quasi-periodic architecture. But modern clock synchronization protocols are cost-effective and precise: the Network Time Protocol (NTP) [Mil06] and True-Time (TT) [CDE+12] provide millisecond accuracies across the Internet, the Precise Time Protocol (PTP) [LEWM05] and the Time-Triggered Protocol (TTP) [Kop11, Chapter 8] provide sub-microsecond accuracies at smaller scales. With synchronized clocks, the completely synchronous scheme outlined at the start of section 5.2 becomes feasible, raising the question: *is there really any need for the LTTA protocols?*

To respond to this question we recall the basics of one of the most efficient clock synchronization schemes: *central master synchronization*. Then we work from well-known principles [Kop11, Chapter 3] to build a globally synchronous system. Finally we compare the result with the two LTTA protocols and their round-based counterparts in section 5.7.

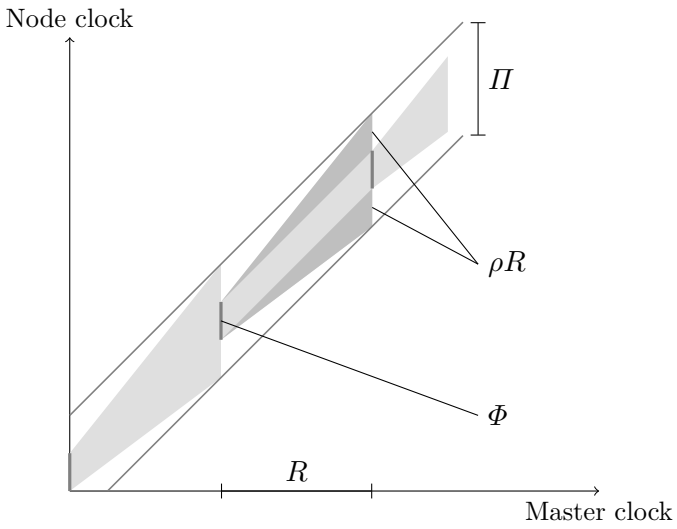


Figure 5.11: [Kop11, Figure 3.10] Central master synchronization: a node's clock stays within the entire shaded area. R denotes the resynchronization interval, Φ the offset after resynchronization, ρ the drift rate between two clocks, and Π the precision of the protocol.

Central master synchronization

In central master synchronization, a distinguished node, the *central master*, periodically sends the value of its local time to all other nodes. In Zélus the code of the master node is the following:

```
let node master() = o where
  rec o = 0.0 fby (o +. t_nom)
```

val master: unit $\xrightarrow{\text{D}}$ *float*

where `t_nom` is the constant T_{nom}. When other nodes, called *slave* nodes, receive this message, they correct their local time reference according to the sent value and the transmission delay. Otherwise a slave's local time reference is incremented by the nominal period T_{nom} at every activation. In Zélus we write:

```
let node slave(gt) = t where
  rec t = if (gt > 0.0 fby gt) then gt +. tau_nom
          else 0.0 fby (t +. t_nom)
```

val slave: float $\xrightarrow{\text{D}}$ *float*

where `gt` is the value received from the master, and `tau_nom` is the constant τ_{nom}. This synchronization scheme is illustrated in figure 5.11.

For the quasi-periodic architecture, and assuming the central master is directly connected to all other nodes, the maximum difference between local time references immediately after resynchronization depends on the difference between the slowest and the fastest message transmissions between the central master and slaves:

$$\Phi = \tau_{\max} + T_{\max} - \tau_{\min}.$$

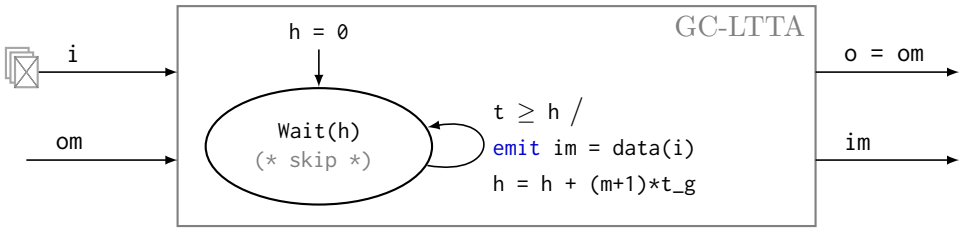


Figure 5.12: The global-clock controller. When the local time `t` reach the horizon `h`, the controller triggers the embedded machine, directly sends the output `om` to other nodes, and computes the next horizon.

The delay between successive resynchronizations R is equal, at best, to the master's activation period. Between synchronizations, a node clock may drift from the master clock. The maximum drift rate ρ is, in our case,

$$\rho = \frac{T_{\max}}{T_{\text{nom}}} - 1 = \frac{T_{\max} - T_{\min}}{T_{\max} + T_{\min}} = \varepsilon.$$

The optimal precision of clock synchronization is then the maximal accumulated divergence between two node clocks during the resynchronization interval, that is,

$$\Pi = \Phi + 2\rho R.$$

Global clock protocol

A global notion of time can be realized by subsampling the local clock ticks of nodes provided the period of the global clock T_g is greater than the precision of the synchronization, that is, $T_g > \Pi$. This assumption is called the *reasonableness condition* in [Kop11, Chapter 3, §3.2.1]. On any given node, the nth tick of the global clock occurs as soon as the local reference time is greater than nT_g. These particular ticks of the local clocks are called *macroticks*. Under the reasonableness condition the delay between nodes activations that occur at the same macrotick is less than Π. Activating nodes on each of their macroticks thus naturally imposes a synchronous execution of the architecture. Then, as for the round-based protocols, communication through two-place buffers suffices to ensure that messages are never incorrectly overwritten.

Finally, the transmission delay may prevent a value sent at the kth macrotick from arriving before the $(k+1)$th macrotick begins. From the maximum transmission delay, we can calculate the number of macroticks m that a node must wait to sample a new value with certainty:

$$m = \left\lceil \frac{\tau_{\max}}{T_g} \right\rceil + 2.$$

At worst a slow node sends a message T_g after the beginning of a round. On the other hand a fast node receives a message at best T_g before the end of the next round. Between those two events we wait $\lceil \tau_{\max}/T_g \rceil$ macroticks to ensure that a message sent with the worst possible transmission delay is received.

This means that the Kahn semantics of an application is preserved if nodes execute one step every m macroticks and communicate through buffers of size two. This gives a throughput of

$$\lambda_{\text{GC}} = 1/mT_g. \tag{5.7}$$

We refer to this simple scheme as the *global-clock* protocol. The Zélus code of the controllers illustrated in figure 5.12 is then:

```
let node gc_controller(gt, i, om) = (o, im) where
  rec automaton
      | Wait(h) →
          do (* skip *)
          unless (t ≥ h) then
            do emit im = data(i) in Wait(h +. (m +.1.)*.t_g)
      init Wait(0.0)

  and t = slave(gt)
  and o = om
```

val gc_controller: float $\times$ α *msg list* $\times$ β *signal* $\xrightarrow{\mathrm{D}}$ β *signal* $\times$ α *list signal*

Remark 5.4. The composition of a global-clock controller and a Mealy machine is always well defined. The proof is again similar to that of remark 5.1. The dependency graph of a node is:

$$\texttt{o} \leftarrow \texttt{om} \quad \texttt{om} \leftarrow \texttt{im} \quad \texttt{im} \leftarrow \texttt{i} \quad \texttt{im} \leftarrow \texttt{t} \quad \texttt{t} \leftarrow \texttt{gt}.$$

It has no cyclic dependencies.

5.7 Comparative evaluation

In this section we compare the performances of the LTTA protocols (back-pressure, time-based, and round-based) with our simple global-clock protocol. We analyze three different classes of architecture: slower nodes/faster communication, comparable nodes and communication, faster nodes/slower communication. In each class, we consider different jitter values (ε) applied to both the nominal period (T_{nom}) and transmission delay (τ_{nom}).

Each of the protocols entails some overhead in application execution time compared to an ideal scheme where $T_{\text{min}} = T_{\text{max}}$ and $\tau_{\text{min}} = \tau_{\text{max}}$. The results of this section are thus presented in terms of the slowdown relative to the ideal case—a synchronous architecture with fixed period T_{nom} and transmission delay τ_{nom}. The slowdown is the relative application speed for a given architecture and protocol: 1.0 indicates the same speed as an ideal system; 2.0 means twice as slow.

We instantiate in table 5.1 the worst-case throughputs of the protocols—theorems 5.2, 5.4 and 5.5 and equation (5.7). In addition, we used the complete framework described in chapter 3 to simulate the protocols. The embedded application is the simple two-node example described at the end of section 5.2 page 89 in which we instantiate the `ltta_controller` with each protocol: back-pressure, time-based, round-based, and global-clock. Our example may seem too simple for a proper evaluation but it has two main advantages:

1. the execution time at each step is negligible, and,
2. broadcast communication and point-to-point communication are in this case equivalent. We can thus compare the protocols with exactly the same settings.

This two communicating nodes example is also reminiscent of the FGS controller from chapter 3.

$T_{\text{nom}}/\tau_{\text{nom}}$	ε	BP	TB	RB/TRB	GC
100.	1%	2.0	4.0	1.0	3.1
	5%	2.1	4.2	1.1	3.5
	15%	2.3	5.7	1.2	4.5
1.00	1%	4.0	6.1	2.0	3.2
	5%	4.2	6.3	2.1	3.8
	15%	4.6	10.3	2.3	5.4
0.01	1%	2.0	2.2	1.0	1.1
	5%	2.1	2.7	1.1	1.3
	15%	2.3	4.7	1.2	1.9

Table 5.1: Relative worst-case slowdowns for the different protocols: back-pressure (BP) and time-based (TB); the optimizations round-based (RB) and timed round-based (TRB); and global clock (GC), compared to an ideal synchronous execution.

For each architecture, we measure the length of the first 1000 execution steps and compare it to the ideal case to compute the associated slowdown. Simulation results are shown in figure 5.13. The shaded area around the mean curves corresponds to the standard deviations.

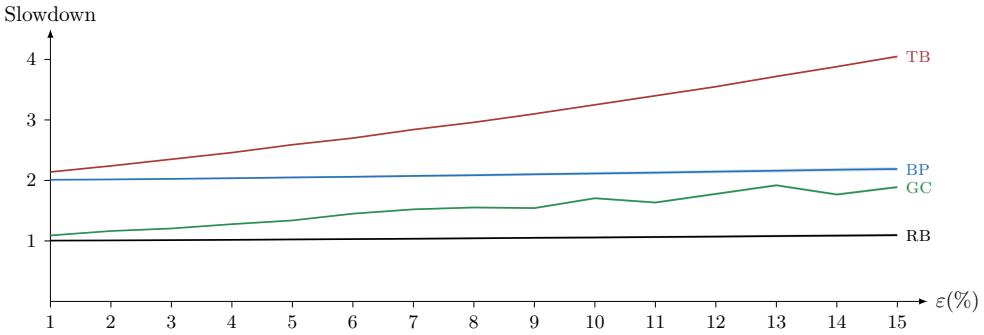
Although these evaluations cannot replace experiments on real hardware, we think that they give a good idea of the worst-case and average performances of the protocols. We can thus give some possible answers to one of the main question of this chapter: *Is there really any need for the LTTA protocols?*

Discussion

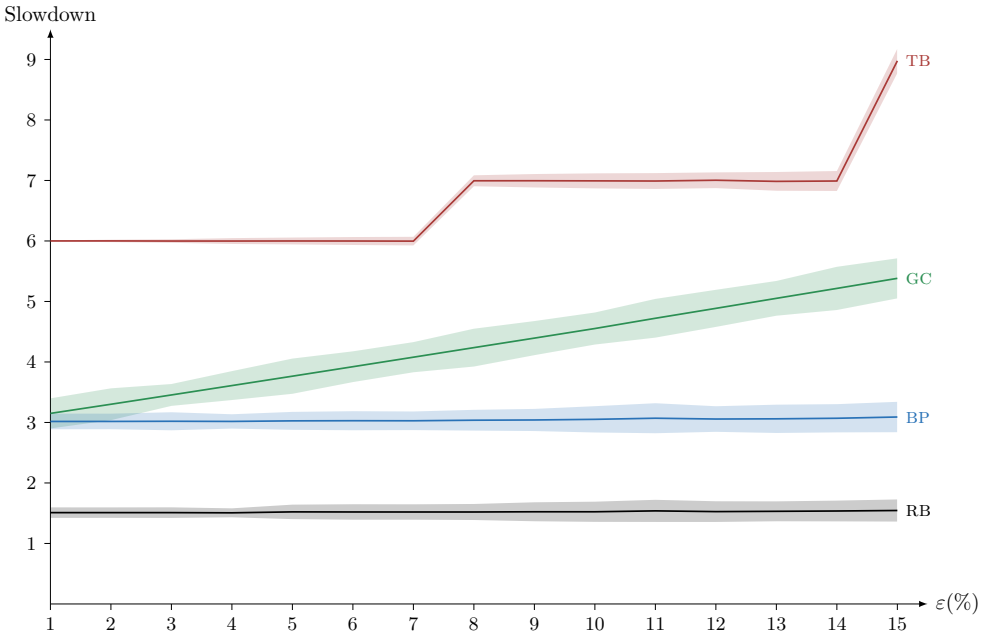
Both in the worst-case and simulation results, the round-based protocol shows the best performances, that is, it is almost as efficient as the ideal execution scheme. Interestingly, when the transmission delay is much greater than the activation period of the nodes, the round-based protocol benefits from the jitter of the system and converges to a *pipelined* execution mode, illustrated in figure 5.14, with two messages simultaneously in transmission. To take this artifact into account in figure 5.13, we clustered execution steps by two before computing the means and standard deviations. This pipelined execution mode is impossible with the historical LTTA protocols where execution and communication phases cannot overlap.

Unsurprisingly, the round-based protocol is always twice as fast as the back-pressure protocol. The back-pressure controller separates execution and communication phases, thus requiring twice as much time between rounds.

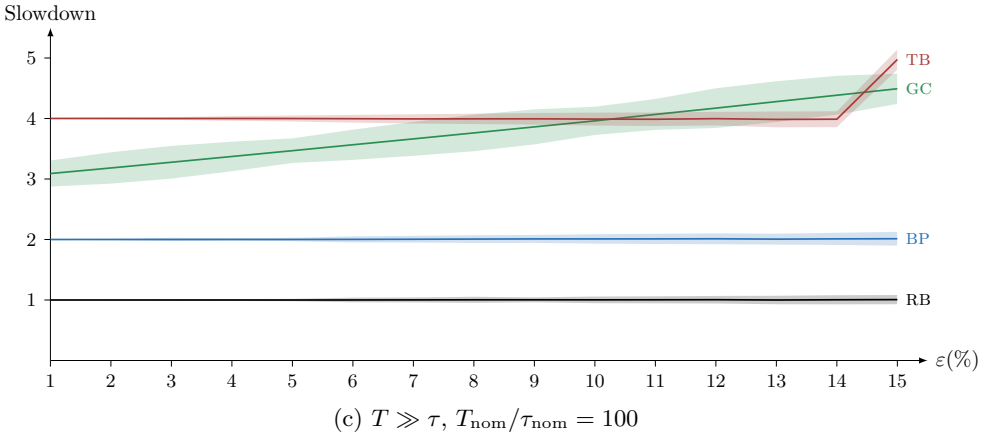
The global-clock performs best when the activation period is much less than the transmission delay. In this case, the cost of clock synchronization is negligible. Conversely, when the activation period is much greater than the transmission delay, the overhead due to clock synchronization becomes significant and protocols that do not require it perform best. The standard deviation of the global-clock protocol increases with the activation period of the node because it also corresponds to the resynchronization interval: the master sends a resynchronization message at each of its activations. Note, though, that we consider a simplified and optimistic case; realistic distributed clock synchronization algorithms will have higher overhead.



(a) $T \ll \tau$, $T_{\text{nom}}/\tau_{\text{nom}} = 0.01$



(b) $T \approx \tau$, $T_{\text{nom}}/\tau_{\text{nom}} = 1$



(c) $T \gg \tau$, $T_{\text{nom}}/\tau_{\text{nom}} = 100$

Figure 5.13: Slowdown factor compared to a synchronous architecture for a simple two-node application (smaller is better). Shaded areas around the mean curves correspond to the standard deviations.

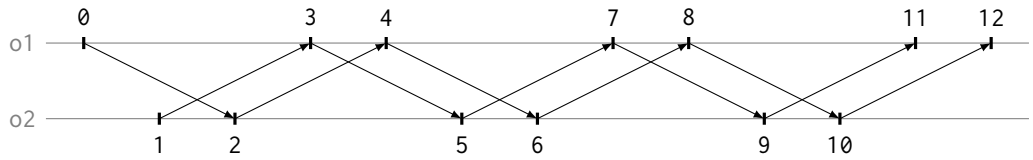


Figure 5.14: Pipelined mode of the round-based protocol for architectures with $T_{\text{nom}} \ll \tau_{\text{nom}}$.

Finally, the time-based protocol is almost always outperformed by the other protocols and is especially sensitive to jitter: its performance decreases rapidly as jitter increases. The 'broken line' aspect of the mean curve is due to the presence of integer parts in parameters p and q. Note that this protocol is still a plausible alternative to clock synchronization when the activation period is much greater than the transmission delay.

The relatively high values obtained for architectures where $T_{\text{nom}} \approx \tau_{\text{nom}}$ can be explained by the fact that the jitter affects both the activation period and the transmission delay. In that case the jitter is twice as significant as for other architectures where either T_{nom} or τ_{nom} is dominates.

Simulation results show that the round-based protocol and back-pressure protocol perform, on average, as well as if they were executed on an architecture where $T_{\text{min}} = T_{\text{max}} = T_{\text{nom}}$ and $\tau_{\text{max}} = \tau_{\text{min}} = \tau_{\text{nom}}$. These protocols are designed to execute as soon as possible, that is, as soon as a new value is available. They can thus compensate 'slow' rounds with 'fast' ones. This compensation mechanism also explains why even though the worst-case slowdown of these two protocols increases with jitter, the average slowdown remains stable even for architectures with significant jitter (15%).

On the other hand, the simulation results of the time-based protocol and the global-clock protocol remain very close to their theoretical values. These protocols are *pessimistic*: rather than waiting for messages from all other nodes, they wait long enough ensure that all messages have been received.

These results show that LTTA protocols remain viable alternatives to clock synchronization when the activation period of the node is significantly slower than the transmission delay, especially for architecture with significant jitter.

Fault tolerance

The back-pressure and round-based protocols rely on blocking communication. If a node crashes, the entire system stops. Therefore fault tolerance mechanisms must be implemented in the middleware (for instance, resurrection mechanisms). On the other hand, the time-based, timed round-based, and global-clock protocols use timing mechanisms. If a node crashes, active nodes continue computing using the values last sent by the crashed node. This behavior allows fault tolerance mechanisms to be implemented in the application.

That being said, the global-clock protocol is vulnerable to a crash of the master node. Robust alternatives exist but entail additional overhead.

Finally we only consider fail-stop crashes. Fault-tolerance in the general case with omission or byzantine failures is a complex problem that requires more sophisticated protocols (with voters, self-checking, agreement protocols, clique avoidance, and node reintegration) [KB03]. The LTTA protocols aim only to provide a lighter alternative for less demanding systems.

Buffer size

A natural question is: *can we optimize the throughput of an application by increasing the size of its communication buffers?* Unfortunately, this is not possible without additional assumptions on the application.

Consider the example of section 5.2 page 87. At each round, each node waits for a message from the other to compute the next value. In this case, increasing the size of the communication buffers is useless since nodes always wait for each other. On the other hand, consider an application that is only feed-forward (for instance, a series of filters without feedback). In that case, nodes can produce values as fast as possible while the communication buffers are not full. Increasing the buffer sizes thus increases the global throughput of the application.

LTTA protocols are a middleware proposal, and are thus *agnostic* to the embedded application. The aim is to ensure the preservation of the semantics of any synchronous application. Protocols are thus as conservative as possible and force nodes to wait for each other at each execution step. Hence, if communication and execution phases are separated, buffers never contain more than one value, and—as shown in section 5.5—if these two phases are merged, buffers contain at most two values.

It is possible to adapt the protocols to allow *pipelined* executions, thus maximizing the throughput of the embedded application, but this requires exploiting knowledge about the embedded application—like the communication topology or the number of logical delays between nodes—in the middleware. This extension has been studied for the back-pressure protocol in [TPB+08]

5.8 Conclusion

We have presented the LTTA protocols in a unified, executable framework. LTTA protocols aim to ensure the preservation of the semantics of a synchronous application running on a quasi-periodic architecture.

We showed how the framework presented in section 3.3 can be refined to incorporate the LTTA protocol in the modeling of the nodes. We instantiated this framework with the two historical LTTA protocols: back-pressure, and time-based LTTA. For each of these protocols we gave new and elementary proofs of the preservation of the semantics based on the real-time characteristics of the architecture, and a theoretical result on the worst-case throughput. Our approach allowed us to highlight the close proximity between the two historical protocols. Both rely on a two-mode execution scheme: *wait* for new inputs; and *delay* the sending of the next value. We also showed that broadcast communication is required to implement the time-based protocol and then gave optimized versions of the protocols for systems using broadcast communication. Finally, for comparison we instantiated our framework with a simple and efficient protocol based on clock synchronization.

Our approach gives both a precise description of the implementation of synchronous applications on a quasi-periodic architecture, and also permits the direct compilation of protocol controllers together with application code. Theoretical and simulation results showed that LTTA protocols are competitive for jittery architectures where the transmission delay is not significant relative to node periods. In addition, LTTA protocols are simple to implement: a node needs only listen and wait and can be implemented as a one- or two-state automaton. They thus remain a lightweight alternative to clock synchronization.

Our simulation results are obtained from the timing bounds characterizing the architecture by generating random values for transmission delays and activations period. This technique gives an idea of the average behavior of the protocols but does not reflect how they react in limit cases, for instance when one node is always as fast as possible while another remains as slow as possible. Rather than multiplying test cases, we present in the next chapter an alternative symbolic simulation scheme where one simulation trace capture a set of possible executions.

Chapter 6

Symbolic Simulation

Timing characteristics are of paramount importance for modeling quasi-periodic systems. In chapter 4 we showed one way to abstract from these real-time characteristics to reason about a purely discrete model. However, designing such abstractions is complex and error prone. We showed, for instance, that the quasi-synchronous abstraction is only applicable to a limited class of quasi-periodic systems. Another approach, presented in chapter 3, is to use a modeling tool like Zélus to express the continuous-time dynamics of an architecture. In this setting, one way of treating nondeterminism is to pick values at random; a simulation trace then captures one possible execution chosen at random among a continuous set of possibilities. We used this approach to simulate the LTTA protocols in section 5.7.

Quasi-periodic systems are but one example of a large class of embedded systems whose specifications involve real-time characteristics and tolerances, like $1\,minute$ or $250 \pm 10\,ms$. Other examples include a pacemaker [Bos07, Table 7 page 34], or a micro-printer [BS09]. In this chapter we focus on such nondeterministic timed systems where the dynamics of continuous components is limited to *timers*, that is, variables evolving with slope 1 to measure time elapsing, but may also involve tolerances.

We propose an alternative simulation scheme inspired by model-checking techniques for timed automata [AD94] where a trace captures a set of executions. A real-time model is translated into a discrete program where nondeterminism is controlled by the user. Each step is characterized by a set of timer values and a set of enabled actions. The user chooses among the enabled actions to trigger discrete computations which return a new set of timer values. This changes nondeterminism on a continuous set of timer values into a nondeterministic choice from a finite set of transitions. The user interacts with the program by choosing a path among all possible executions.

Contributions

This chapter presents the last contribution of this thesis (not yet published). We present ZSy: a subset of Zélus where the continuous components only involve timers extended with nondeterministic constructs to express *guards* on the emission of signals, and *invariants*. We propose a symbolic simulation scheme for programs written in ZSy where the user controls time elapsing via distinguished `wait` transitions.

In Zélus, discrete computations are triggered by zero-crossings on continuous-time expressions. In our setting, we replace these events with signal emissions guarded by nondeterministic

conditions on timers. We show how to adapt the type system of Zélus that distinguishes discrete computations from continuous ones.

We adapt the modular source-to-source compilation of Zélus to produce discrete code for symbolic simulation. Guarded signals are turned into additional inputs controlled by the user, and continuous functions return additional outputs for the set of timer values and the set of enabled actions (`wait` or firing guards). Compilation produces discrete code that can be compiled with the Zélus compiler and executed.

Outline We start in section 6.1 by motivating our approach with a ZSy model of quasi-periodic architectures. Related work is described and discussed in section 6.2. Sets of timer values are represented and manipulated using difference-bound matrices presented in section 6.3. The syntax and key features of ZSy are explained in section 6.4. We detail the type system in section 6.5 and the source-to-source compilation pass in section 6.6. In section 6.7 we discuss how to extend ZSy with valued signals and automata and detail a complete example: the train gate controller.

6.1 Motivation

Consider the example of a quasi-periodic clock that emits a signal `c`. The specification of such a clock can be decomposed into three statements:

1. a timer `t` is reset at each emission of `c`,
2. `c` can be emitted as soon the timer reaches the value T_{min}, and,
3. the value of the timer must never exceed T_{max}.

Zélus allows a timer to be simulated with a simple ODE: $\dot{t} = 1$. Nondeterminism can be added to the language by importing an `arbitrary` function. The variable t is initialized to an arbitrary value between $-T_{\text{min}}$ and $-T_{\text{max}}$, and similarly reinitialized whenever t reaches 0. Signal `c` is emitted on the zero-crossings of t.

```
let hybrid metro(t_min, t_max) = c where
  rec der t = 1.0 init -. arbitrary (t_min, t_max)
      reset z → -. arbitrary (t_min, t_max)
  and z = up(t)
  and present z → do emit c done
```

One way of treating nondeterminism is to implement the `arbitrary` function with a random generator. This is the approach used in chapters 3 and 5 to simulate the behavior of the LTTA protocols (section 5.7 page 102).

This approach 'pushes' nondeterminism outside the model and forces the programmer to make explicit implementation choices that are not part of the specification. For instance, in the previous model arbitrary values are computed on the zero-crossings of t. Besides, this technique is not modular. Adding constraints on t, like another invariant, would require adapting the parameters of the `arbitrary` function. In this chapter, we propose to integrate timing nondeterminism more directly into the language by using dedicated constructs.

The following model introduces the two main elements for expressing nondeterminism: *guards* characterized by an *activation zone* that express a possibility and *invariants* which express an obligation.

```
let hybrid metro(t_min, t_max) = c where
  rec timer t init 0 reset c() → 0
  and emit c when {t ≥ t_min}
  and always {t ≤ t_max}
```

As in Zélus, the keyword `hybrid` declares a continuous-time component. The first equation, `timer t init 0 reset c() → 0`, declares a timer: a variable t such that $\dot{t} = 1$, with initial value 0 and that is reset to 0 at each emission of `c`. We replace `der t = 1.0` with `timer t` to make it clear that we focus on timed systems where continuous dynamics can only be expressed with timers. The guard `emit c when {t ≥ t_min}` states that signal `c` *may* be emitted if $t \geq T_{\text{min}}$. The invariant `always {t ≤ t_max}` states that the value of t *must* never exceed T_{max}. We use braces to differentiate constraints from boolean conditions.

Note that it is possible to write unsatisfiable constraints, for example, by combining two contradictory invariants. For instance, `always {t < 2}` and `always {t > 2}`. If the program reaches a state with such constraints the simulation becomes stuck.

A model of a simple two-node architecture can be obtained by instantiating the function `metro` twice. For simplicity, we assume instantaneous communication and omit the modeling of transmission delays.

```
let hybrid archi(t_min, t_max) = c1, c2 where
  rec c1 = metro(t_min, t_max)
  and c2 = metro(t_min, t_max)
```

The simulation traces of such systems comprise two kinds of events: time elapsing and discrete transitions triggered by signal emissions. Figure 6.1 shows a possible execution trace of the two-node architecture with $T_{\text{min}} = 3$ and $T_{\text{max}} = 5$. Variables t_1 and t_2 denote the values of the two timers, one for each quasi-periodic clock.

Symbolic traces

For the kind of systems we consider, that is, nondeterministic timed discrete-event systems, an execution is a sequence of discrete events (here, clock ticks). Rather than simulating one concrete trace that assigns a precise date to each event, we propose an alternative simulation scheme that focuses on the ordering of events. This is already the approach of chapter 4 where possible interleavings of events are abstracted with discrete predicates.

A symbolic simulation trace is a discrete-time execution where each discrete step is associated with a *zone*, that is, a set of timer values. A zone is obtained from an initial set of timer values, the *initial zone*, by letting time elapse until the next change in the set of enabled guards. All guards with an activation zone that overlaps the current zone are enabled. Each zone is thus characterized by a set of enabled guards.

At each step the user chooses among a set of possible transitions. A transition means either waiting for a change in the set of enabled guards or firing enabled guards.

wait If the user chooses the `wait` transition, we compute the new zone by letting time elapse until the next change in the set of enabled guards (if allowed by the invariants).

guards Otherwise, firing guards triggers discrete computations, possibly resets some timers, and returns a new initial zone. The new zone is obtained by letting time elapse from this initial zone until the next change in the set of enabled guards.

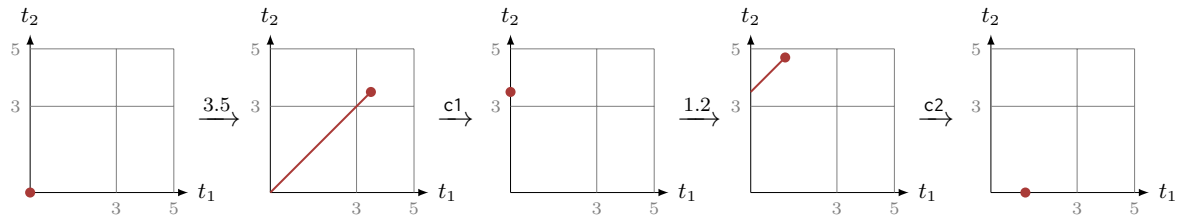


Figure 6.1: A concrete simulation trace of `archi(3,5)`: t_1 and t_2 denote the values of the two timers, one for each quasi-periodic clock.

Zones capture nondeterminism in timer values but not nondeterminism in the ordering of events, which is externalized in input signals similarly to the clocks of the quasi-periodic model of section 3.3. Continuous nondeterminism is replaced by discrete nondeterminism on a finite set of possible choices. Events can be produced by the user, for instance in an interactive mode, or by a discrete program coupled with the simulation function. In other words, signals emitted by the guards are turned into inputs of the simulation.

The symbolic trace corresponding to the simulation of figure 6.1 is presented in figure 6.2.

1. The simulation starts in the initial position $\{t_1 = t_2 = 0\}$ where no guards are enabled. The first zone is obtained from this initial position by letting time elapse until one or more guards become enabled, that is, when $\{t_1 = t_2 = 3\}$. In this first zone the user has no other choice than the `wait` transition.
2. At $t_1 = t_2 = 3$ both guards are enabled and the new zone is obtained by letting time elapse as long as permitted by the invariants, that is, until both timers reach 5. In this zone, the user can choose `c1` or `c2` but the `wait` transition is no longer enabled.
3. The user triggers `c1` which resets t_1 to 0. The new initial zone is $\{t_1 = 0 \wedge 3 \leq t_2 \leq 5\}$ where the user can only choose `c2`. The new zone is obtained by letting time elapse from this initial zone until we reach the limit fixed by the invariants, that is, when t_2 reaches 5.
4. The user triggers `c2` which resets t_2 to 0. The new initial zone is $\{t_2 = 0 \wedge 0 \leq t_1 \leq 2\}$ where no guards are enabled. The new zone is obtained by letting time elapse until the next change in the set of enabled guards, that is, when t_1 reaches 3.

The symbolic trace of figure 6.2 captures all possible executions where `c1` is activated before `c2`. It shows that a single execution can test many particular executions including those involving limit conditions.

It is possible that during an execution, the program reaches a state where the constraints are unsatisfiable, or where no transitions (neither `wait` nor firing guards) are enabled. In that case the simulation is stuck. This situation is only possible if the specifications of the system introduce *deadlocks*.

The symbolic representation of timers is a classic technique to compute the set of reachable transitions for model-checking timed automata. Our proposal is inspired by these techniques, but with a fundamental difference: there is a notion of time elapsing. Our simulation scheme only proposes enabled guards in the current zone, not all reachable guards, and allows the user to control time elapsing using `wait` transitions. The user thus has a clear view of the succession of possible transitions as time elapses.

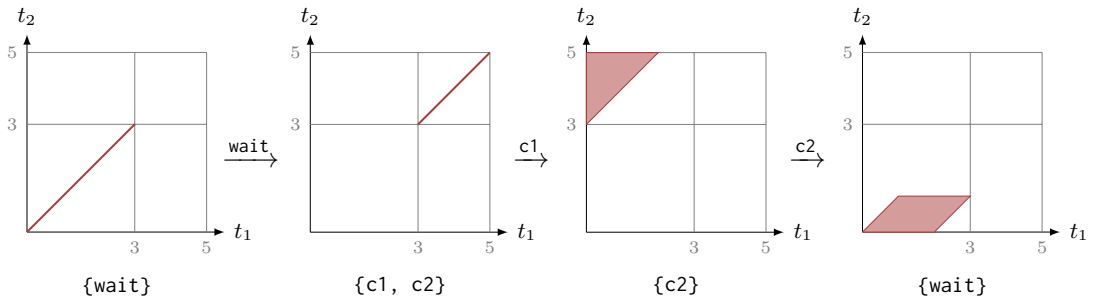


Figure 6.2: A symbolic simulation trace of `archi(3,5)`. Each step corresponds to a set of timer values and a set of enabled transitions (below).

6.2 Related work

The related work can be divided into three categories: verification of timed and hybrid systems, nondeterminism in synchronous languages, and the treatment of continuous-time in synchronous languages.

Verification of timed and hybrid systems

Uppaal[1] [BDL06] is a model checker for systems expressed as networks of communicating timed automata. The passage of time is modelled by abstract timers, called *clocks*, states can be characterized by invariants and transitions are guarded by nondeterministic conditions on clocks. The model checker is based on the symbolic representation of reachable states using *Difference-Bound Matrices* (DBMs). Uppaal was mainly designed for model-checking, but the system is also equipped with a simulator where a user controls the firing of guards to navigate through symbolic states. This simulator was a great source of inspiration for our work. DBMs allow the efficient representation and manipulation of symbolic states and suffice to express the timing dynamics that interest us. We also model nondeterministic transitions using inputs controlled by a user. Our approach differs in two ways.

1. Our simulation scheme is based on computing a succession of zones. At each step the user can only choose from the set of enabled guards and the passage of time that changes this set is controlled by `wait` transitions. The Uppaal simulator, on the other hand, proposes all reachable guards. There is no explicit control of time elapsing and the succession of possibilities is not presented one-by-one.
2. As Uppaal is oriented toward model checking, the language is quite constrained and every component, discrete and continuous, must be expressed as a timed automaton. We present a source-to-source compilation where a program mixing discrete and continuous components is translated into a purely discrete function with additional inputs and outputs for symbolic simulation. The user can thus use all control structures offered by Zélus, like hierarchical automata. These control structures are tedious to define using only communicating timed automata and their implementation is error prone. We provide a means of simulating programs written in a language dedicated to embedded systems. Efficient compilation schemes exists for this language and are used in practice.

[1] `http://www.uppaal.org/`

There has been significant work on the verification of hybrid systems usually expressed as hybrid automata (see [Alu11] for a survey). Except for systems where the allowed dynamics are severely restricted, like timed automata, it is known that the safety verification problem is undecidable [ACH+95]. Verification of hybrid systems thus proceeds by overapproximation of the reachable states and focuses on restricted classes of hybrid systems like *linear hybrid systems* [HPR94] or more recently *piecewise affine hybrid systems* [FLGD+11].

Our proposal is different: continuous components are basically timed automata known to be closed under both discrete transitions (signal emissions, mode changes, resets) and continuous evolution (time elapsing). We can thus propose a symbolic representation of the states of the system without overapproximation. In other words, a symbolic trace captures a set of valid concrete simulations. Besides, we focus on simulation and thus do not worry about state space explosion which is a concern for model-checkers. States are computed during execution, and exploration is driven by user choices.

Constraints in synchronous languages

Several approaches propose to add nondeterministic constraints to synchronous languages. Lutin [RJ13,RRJ08a,RRJ08b] is a language to design test scenarios for Lustre programs. Users write nondeterministic specifications for the inputs of a program which can also depend on its outputs. At each step, an input is thus characterized by a set of possible values. The system chooses a value at random to start the next step. The output is a concrete simulation trace not a symbolic one. These ideas were recently implemented in the Argosim Stimulus tools.[2]

Yo-yo [Mau96, Gar02] is a tool for the symbolic simulation of discrete dataflow programs extended with nondeterministic relations between variables. At each step the simulator returns a symbolic state that represents the set of possible values for each variable. In the same vein, it is possible to define variables by relations with other variables in Signal [BGJ91] with the compiler checking that the resulting program is deterministic before producing code.

These approaches all import nondeterministic constraints into the synchronous model where time is a sequence of discrete steps. The user still writes a discrete program, even if it may involve nondeterminism on some variables. We, on the other hand, add nondeterminism to continuous components. The result of the compilation is also a discrete function, but the discretization of time depends on the nondeterministic constraints of the model.

Continuous-time and synchronous languages

Synchronous languages are based on a discrete notion of time. Physical time is typically handled as an external event, for instance the reception of a regular signal `second`. However, real-time characteristics of the system are sometimes critical to verify safety properties (the correctness of the time-based LTTA protocol is a good example). Lots of work has thus focused on verifying the real-time properties of synchronous programs.

Argos [Mar92] has been extended with timeouts and watchdogs [JMO93]. The semantics of the models is expressed in terms of timed automata that can then be verified using the Kronos model checker [Yov97]. Taxys [MAS06] allows the annotation of Esterel programs with real-time characteristics. The annotated program is translated into a timed automaton that can also be analyzed using Kronos. Quartz [LS02] is an Esterel-like language for which real-time verification has been proposed. The semantics of a Quartz program can be expressed in terms of

[2] `http://argosim.com/`

timed Kripke structures to allow the formal verification of timing properties. Halbwachs showed how to apply abstract interpretation techniques to verify safety properties of synchronous models involving multiform timing characteristics linked by linear relations [Hal93].

We focus on a different problem: the symbolic simulation of real-time models involving nondeterminism. Instead of adding real-time assumptions to a discrete program, we start from a continuous-time model and show how an execution can be discretized in a succession of discrete steps where each step corresponds to a set of timer values.

Finally, we already presented Zélus [BP13] and Ptolemy [Pto14] that allow the user to write models that mix discrete control and continuous dynamics expressed as ODEs. Compared to our approach, these languages focus on the simulation of deterministic models.

6.3 Difference-Bound Matrices

Difference-bound matrices [Dil90,Yov96,Ben02] are a well-known data structure for representing and manipulating zones. They are, for instance, used in Uppaal. DBMs are simple to implement and form a closed set with respect to both discrete transitions (mode changes, resets, intersections), and continuous evolution (time elapsing), thus allowing a symbolic representation of zones without overapproximation.

Let $\mathcal{T} = \{t_i\}_{0 \leq i \leq n}$ be a set of timer variables, with the convention that $t_0 = 0$. DBMs can only be used to represent *difference constraints* of the form $t_i \sim n$ or $t_i - t_j \sim n$ where $n \in \mathbb{N}$, and $\sim \in \{<, \leq, \geq, >\}$. Any such constraint can be rewritten into the form $t_i - t_j \preceq n$ where $\preceq \in \{<, \leq\}$ and $n \in \mathbb{Z}$:

$$\begin{aligned} t_i \preceq n &\longmapsto t_i - t_0 \preceq n, \\ t_i - t_j \preceq n &\longmapsto t_i - t_j \preceq n, \\ t_i - t_j \succeq n &\longmapsto t_j - t_i \preceq -n, \\ t_i \succeq n &\longmapsto t_0 - t_i \preceq -n. \end{aligned}$$

Difference constraints can be gathered into a $|\mathcal{T}| \times |\mathcal{T}|$ matrix where each coefficient represents a bound on the difference between two timers (which explains the term *difference-bound matrices*).

To compute the DBM representation of a set of constraints $\mathcal{C}$, each timer is assigned to one row and one column of the matrix. The row stores upper bounds on the difference between the timer and all other timers, the column is used for lower bounds.

The coefficients C_{ij} of the matrix are computed as follows:

1. For each constraint $t_i - t_j \preceq n \in \mathcal{C}$ we have $C_{ij} = (n, \preceq)$.
2. If a clock difference $t_i - t_j$ is unbounded we have $C_{ij} = (\infty, <)$.
3. We complete the matrix assuming that all clocks are positive $t_0 - t_i \leq 0$: $C_{0i} = (0, \leq)$; and the difference between a timer and itself is always 0, $t_i - t_i \leq 0$: $C_{ii} = (0, \leq)$.

Figure 6.3 shows an example of a set of constraints and its representation as a DBM.

Comparison Bounds are pairs that can be compared with the lexicographic order where $<$ is smaller than $\leq$ and $\forall n \in \mathbb{Z},\ n < \infty$:

$$(n_1, \preceq_1) < (n_2, \preceq_2) \quad \Longleftrightarrow \quad n_1 < n_2 \text{ or } (n_1 = n_2 \text{ and } \preceq_1 = <).$$

$$\left\{\begin{array}{c} t_1 < 20 \\ 6 \leq t_2 \\ 5 < t_3 \leq 12 \\ 4 \leq t_1 - t_2 \leq 8 \end{array}\right\} \qquad \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \left[\begin{array}{cccc} (0,\leq) & (0,\leq) & (-6,\leq) & (-5,<) \\ (20,<) & (0,\leq) & (8,\leq) & (\infty,<) \\ (\infty,<) & (-4,\leq) & (0,\leq) & (\infty,<) \\ (12,\leq) & (\infty,<) & (\infty,<) & (0,\leq) \end{array}\right] \end{array}$$

Figure 6.3: Example of a set of constraints (left) and a corresponding DBM (right).

Canonical form

Several different DBMs may represent the same set of constraints. For instance, on the example of figure 6.3, combining constraints $4 \leq t_1 - t_2 \leq 8$ and $6 \leq t_2$ we obtain $10 \leq t_1$. This constraint can be added to the set of constraints to obtain another DBM representing the same zone.

Fortunately, there is a canonical form for each set of constraints. A DBM can be interpreted as the adjacency matrix of a weighted graph where vertices are timers t_i, and the weight on an edge between two vertices t_i and t_j is a bound on their difference. The path with the minimal weight between two vertices t_i and t_j in the graph corresponds to the tightest bound on the difference $t_i - t_j$ that is deducible from the initial set of constraints. A canonical DBM can thus be computed using a shortest path algorithm like the Floyd-Warshall algorithm [Roy59, Flo62] with a complexity in $\mathcal{O}(|\mathcal{T}|^3)$ (cubic in the number of timers):

for $k = 0$ *to* $|\mathcal{T}| - 1$ *do*
  for $i = 0$ *to* $|\mathcal{T}| - 1$ *do*
    for $j = 0$ *to* $|\mathcal{T}| - 1$ *do*
      $C_{ij} = \mathtt{min}(C_{ij}, C_{ik} + C_{kj})$
    end
  end
end

Operations on DBMs

We now present the elementary operations on DBMs that will be useful for symbolic simulation. We assume that all DBMs are in canonical form even if that means running the normalization algorithm after an operation. There are clever alternative implementations of operations on DBMs that do not require renormalization [Ben02, §3.2], but we only present the basic principles. A graphical illustration of the DBM operations is shown in figure 6.4.

In the following we make no distinction between a zone $\mathcal{C}$ and its DBM representation C. We also assume the same dimension for all matrices: the total number of timers $|\mathcal{T}|$. It is always possible to add more timers, that is, to increase the dimension of the DBMs, without adding constraints.

is_empty(C) A zone is empty if the set of constraints contains a contradiction, that is, an upper bound smaller than a lower bound. These inconsistencies can be detected during normalization. In that case, by convention, we set coefficient C_{00} to $(-1, <)$: a zone C is empty if $C_{00} = (-1, <)$.

relax(C, t_i) Given a zone C, this operation removes all constraints on a timer t_i. All coefficients on row t_i and column t_i of C are set to $(\infty, <)$ except C_{0i}, the lower bound, set to $(0, \leq)$.

inter(C, D) The intersection I of two zones C and D is given by the minimum of each pair of coefficients: $\forall 0 \leq i, j < |\mathcal{T}|,\ I_{ij} = \min(C_{ij}, D_{ij})$.

reset(C, t_i, v) In a zone C, it is possible to reset a timer t_i to a value v by relaxing all constraints on t_i, *relax*(C, t_i), and setting both the upper and lower bound of t_i to the value v, that is, $C_{i0} = (v, \leq)$ and $C_{0i} = (-v, \leq)$.

up(C) From an initial zone C the zone obtained by letting time elapse indefinitely is obtained from C by setting all upper bounds, the first column of the matrix, to $(\infty, <)$.

Computing zones

Our simulation scheme is based on detecting changes in the set of enabled guards. We thus need to compute the distance between an initial zone and the guard activation zones.

Each guard G divides the state-space of timer values into three zones: before activation, during activation, and after activation. A guard is enabled whenever the intersection between its activation zone and the current zone is not empty. Given an initial zone Z, we can compute two distances for each guard G, $d_{\text{in}}(Z, G)$ the maximum distance before activation becomes possible, and $d_{\text{out}}(Z, G)$ the distance before deactivation. Figure 6.5 illustrates these two distances.

Distances Like difference bounds, a distance is a pair $(d, \preceq)$. The relation $\preceq$ specifies whether the limit is strict or not. Consider, for instance, a guard with activation zone $\{3 < t < 5\}$ and the initial zone $\{t = 0\}$.

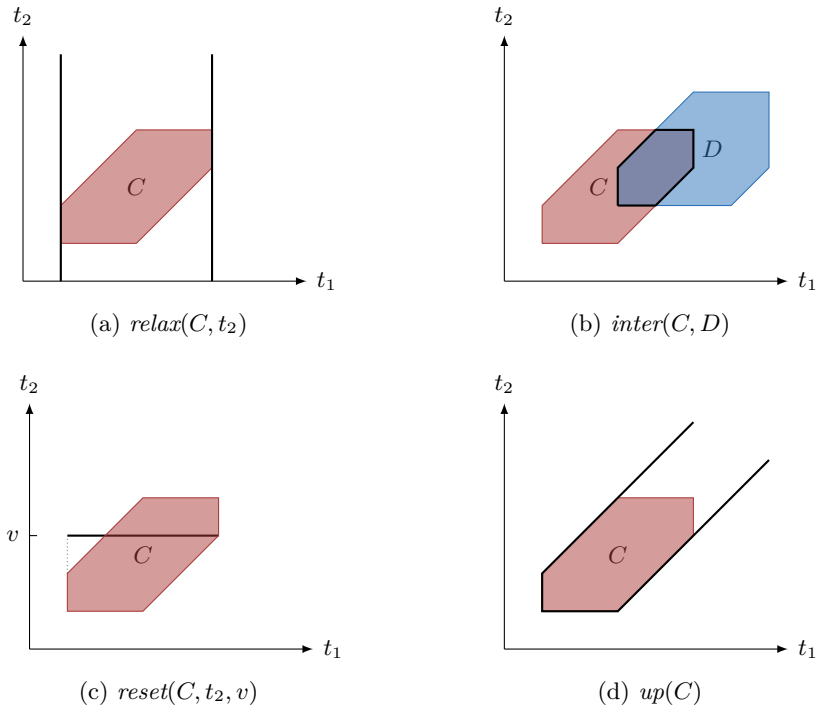
The distance before activation is $(3, <)$ which means that the guard is only enabled strictly after $t = 3$. The strictness of the limit is given by the relation of the lower bounds of the guard: a change occurs as soon as the guard is enabled. The distance before activation is obtained by comparing the upper bounds of the initial zone, DBM Z, with the lower bounds of the guard, DBM G (argmin is the index of the minimal value in a set).

$$d_{\text{in}}(Z, G) = (g_j - z_j, \preceq_j) \text{ with } j = \underset{1 \leq i \leq |\mathcal{T}|}{\text{argmin}}\{g_i - z_i\}$$
$$\text{where } \forall 1 \leq i < |\mathcal{T}|,\ G_{0i} = (-g_i, \preceq_i) \text{ and } Z_{i0} = (z_i, \_)$$

On the other hand, the distance before deactivation is $(5, \leq)$ because the guard is disabled when $t = 5$. This time, the nature of the limit corresponds to the opposite relation of the upper bound of the guard: a change only occurs when we leave the activation zone. The distance before deactivation is obtained by comparing the lower bounds of the initial zone with the upper bounds of the guards.

$$d_{\text{out}}(Z, G) = (g_j - z_j, \overline{\preceq_j}) \text{ with } j = \underset{1 \leq i \leq |\mathcal{T}|}{\text{argmin}}\{g_i - z_i\}$$
$$\text{where } \forall 1 \leq i < |\mathcal{T}|,\ G_{i0} = (g_i, \preceq_i) \text{ and } Z_{0i} = (-z_i, \_)$$

where $\overline{\preceq}$ denotes the opposite relation: $\overline{\leq} = <$ and $\overline{<} = \leq$.



(a) $relax(C, t_2)$ (b) $inter(C, D)$

(c) $reset(C, t_2, v)$ (d) $up(C)$

Figure 6.4: Illustration of DBM operations

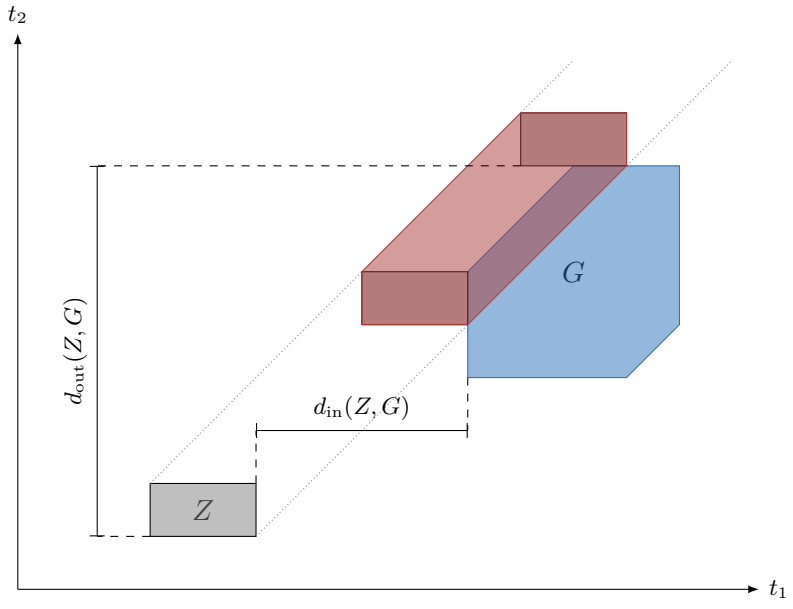


Figure 6.5: Activation and deactivation distances of a guard G from an initial zone Z. The red region is the zone where the guard is enabled.

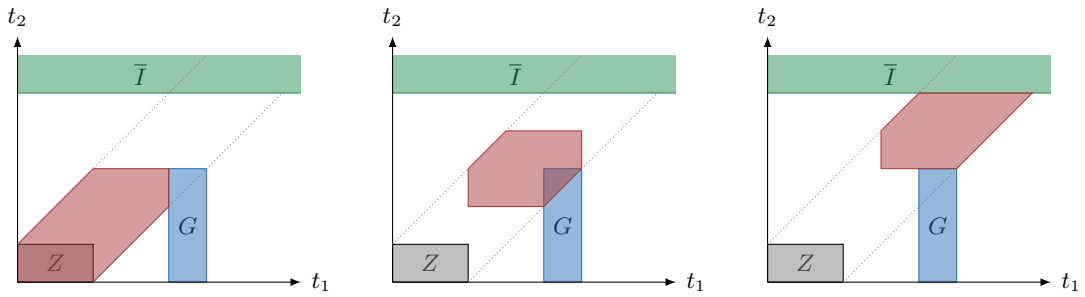


Figure 6.6: The succession of zones from the initial zone Z for a single guard G with an invariant I.

We now have enough elements to compute the succession of zones in a symbolic execution. Starting in an initial zone Z, we compute the two distances associated with all reachable guards, that is, guards that will eventually be enabled if we let time elapse indefinitely. These distances can be totally ordered and we add $(0, \leq)$ for the initial state and $(\infty, \leq)$, the maximal distance for the final state, that is, when no more guards are reachable. Each pair of successive distances defines a zone obtained by *sweeping* the initial zone on the corresponding interval as illustrated in figure 6.6.

Sweep Sweeping an initial zone Z in the interval defined by the two distances $(d_1, \prec_1)$ and $(d_2, \prec_2)$ returns a new zone C defined as follows: we start from the initial zone delayed by $(d_1, \prec_1)$, that is, we add d_1 to all lower bounds of Z and the nature of the limit is given by $\prec_1$ (we start after the change):

$$\forall 1 \leq i < |\mathcal{T}|,\ C_{0i} = (-z_{0i} - d_1, \prec_1) \text{ where } Z_{0i} = (-z_{0i}, \_).$$

The zone stops just before the next change, that is, after a distance $(d_2, \prec_2)$. We thus add d_2 to all upper bounds of Z and the nature of the limit is given by $\overline{\prec_2}$ (we stop before the next change):

$$\forall 1 \leq i < |\mathcal{T}|,\ C_{i0} = (z_{i0} + d_2, \overline{\prec_2}) \text{ where } Z_{i0} = (z_{i0}, \_).$$

Consider again the simple example having only one guard with activation zone $\{3 < t < 5\}$ and the initial zone $\{t = 0\}$. The set of distances is then $\{(0, \leq), (3, <), (5, \leq), (\infty, \leq)\}$ which gives three zones: $\{0 \leq t \leq 3\}$ where the guard is disabled, $\{3 < t < 5\}$ where the guard is enabled, and $\{5 \leq t < \infty\}$ where the guard is disabled again.

Invariants Invariants are conditions that must *always* hold. Zones are thus obtained by intersecting the result of the sweeping mechanism with all the invariants. For instance, if we add the invariant `always {t ≤ 10}` to our simple example, the three zones become: $\{0 \leq t \leq 3\}$, $\{3 < t < 5\}$, and $\{5 \leq t \leq 10\}$. The effect of invariants is illustrated in figure 6.6 where $\overline{I}$ is the negation of the invariant I.

Zone interface

Zones can be manipulating with the following imported functions:

`zall` denotes the complete space: a zone without any constraints.

`zmake(c)` builds a DBM from a constraint `c`.

`is_zempty(z)` tests if the zone `z` is empty.

`zreset(z, t, v)` resets the timer `t` to the value `v` in `z`.

`zinter(z1, z2)` returns the intersection `z1` $\cap$ `z2`.

`zinterfold(l)` returns the intersection of all elements in `l`: $\texttt{l}_0 \cap \texttt{l}_1 \cap ...$, and `zall` if `l` is empty.

`zup(z)` returns the zone obtained by letting time elapse indefinitely from `z`.

`zdist(zi, g)` returns the distances between an initial zone `zi` and a guard activation zone `g`: the distance before activation and the distance before deactivation when the guard is reachable but not enabled, the distance before deactivation if the guard is already enabled, and nothing otherwise.

`zdistmap(zi, gv)` returns the list of distances between an initial zone `zi` and a list of guard activation zones `gv`.

`zsweep(zi, dp, d)` returns the result of sweeping `zi` between distances `dp` and `d`.

`zenabled(zc, gv)` returns a list of booleans characterizing the set of enabled guards. A guard is enabled if its activation zone $\texttt{gv}_i$ intersects the current zone `zc`.

The complete Zélus interface, that is, the type signatures of the imported functions, is:

val zall: zone
val zmake: constraint $\xrightarrow{\text{A}}$ *zone*
val is_zempty: zone $\xrightarrow{\text{A}}$ *bool*
val zreset: zone $\times$ *timer* $\times$ *int* $\xrightarrow{\text{A}}$ *zone*
val zinter: zone $\times$ *zone* $\xrightarrow{\text{A}}$ *zone*
val zinterfold: zone list $\xrightarrow{\text{A}}$ *zone*
val zup: zone $\xrightarrow{\text{A}}$ *zone*
val zdist: zone $\times$ *zone* $\xrightarrow{\text{A}}$ *dist list*
val zdistmap: zone $\times$ *zone list* $\xrightarrow{\text{A}}$ *dist list*
val zsweep: zone $\times$ *dist* $\times$ *dist* $\xrightarrow{\text{A}}$ *zone*
val zenabled: zone $\times$ *zone list* $\xrightarrow{\text{A}}$ *bool list*

6.4 ZSy: an extended subset of Zélus

We now present ZSy, a single assignment kernel of Zélus where the only continuous dynamics are timers extended with nondeterministic constructs: guards and invariants.

The grammar of ZSy is the following.

$$
\begin{array}{rcl}
d & ::= & \texttt{let hybrid } f\texttt{(}p\texttt{) = } e \mid \texttt{let node } f\texttt{(}p\texttt{) = } e \mid \texttt{let } f\texttt{(}p\texttt{) = } e \mid d\ d \\
e & ::= & x \mid v \mid op\texttt{(}e\texttt{)} \mid f\texttt{(}e\texttt{)} \mid \texttt{(}e\texttt{, } e\texttt{)} \mid e \texttt{ fby } e \mid e \texttt{ where rec } E \\
p & ::= & x \mid \texttt{(}p\texttt{, } p\texttt{)} \\
h & ::= & e \rightarrow e \mid \cdots \mid e \rightarrow e \\
E & ::= & x \texttt{ = } e \mid E \texttt{ and } E \mid x \texttt{ = present } h \texttt{ init } e \mid x \texttt{ = present } h \texttt{ else } e \\
 & \mid & \texttt{timer } x \texttt{ init } e \texttt{ reset } h \\
 & \mid & \texttt{always \{ } c \texttt{ \}} \\
 & \mid & \texttt{emit } x \texttt{ when \{ } c \texttt{ \}} \\
c & ::= & \Delta \sim e \mid c \texttt{ \&\& } c \\
\Delta & ::= & x \mid x - x \\
\sim & ::= & < \mid \leq \mid \geq \mid >
\end{array}
$$

Declarations A program is a sequence of declarations (d) of n-ary functions. Functions are declared as continuous (`let hybrid` f`(`p`) =` e), discrete (`let node` f`(`p`) =` e), or combinatorial (`let` f`(`p`) =` e). Functions of the last kind can be used in contexts of the first two kinds.

Expressions The set of expressions comprises variables (x), constants (v), external operator applications (op`(`e`)`), function applications (f`(`e`)`), pairs (`(`e`,` e`)`), initialized unit delays (e `fby` e), and local declarations (e `where rec` E) which return the value of e and where variables used in e can be defined in the set of local equations E.

Patterns A pattern is a variable (x) or a pair of patterns (`(`p`,` p`)`).

Handlers A handler is a list of pairs of conditions and expressions ($c_1 \rightarrow e_1 \mid \cdots \mid c_n \rightarrow e_n$). Conditions must be boolean expressions in discrete contexts and signals in continuous contexts. Conditions $c_1, \ldots, c_n$ are treated sequentially, that is, if two conditions are enabled at the same instant, only the first one has an effect. When condition c_i is enabled, the handler h takes the value e_i of the corresponding expression, and no value otherwise.

Equations A set of equations is either a simple equation (x `=` e), a parallel composition of two sets of equations (E_1 `and` E_2), or the definition of a piecewise constant variable.

A piecewise constant variable can be defined with an initial value (x `=` `present` h `init` e_0). The value changes according to a handler h and the last defined value is maintained when the handler returns nothing. Alternatively, it is possible to provide a default value (x `=` `present` h `else` e). In this case, the variable takes the default value when the handler returns nothing.

The last three constructs are specific to ZSy: (`timer` x `init` e_0 `reset` h) defines a timer initialized with value e_0 and reset according to a handler h, (`always {` c `}`) declares the constraint c as an invariant, and (`emit` s `when {` c `}`) defines a guard which states that signal s may be emitted when constraint c is satisfied.

For simplicity, the only possible action when a guard is fired is to emit a signal. This is not a real restriction since this signal can be used to trigger arbitrary discrete computations when coupled with the `present` constructs.

Since ZSy is a single assignment kernel, no signal can be emitted by two or more distinct guards. After compilation, these signals become inputs of the simulation controlled by the user to handle nondeterministic choices.

Constraints The language of constraints c is limited by the data structure used to represent zones. In this chapter we use DBMs and we only allow constraints of the forms $t_i \sim n$ and $t_i - t_j \sim n$ where t_i and t_j are timers, n is an integer, and $\sim \in \{<, \leq, \geq, >\}$.

6.5 Static typing

As in Zélus, we must statically discriminate between *discrete* and *continuous* computations. In ZSy, the transition between *continuous* and *discrete* contexts is realized via signals emitted by the guards. A variable is typed *discrete* if it is activated on signal emissions, and *continuous* otherwise. We can thus adapt the Zélus type system presented in [BBCP11a, §3.2] to ZSy.

Types and kinds

Each function has a type of the form $t_1 \xrightarrow{k} t_2$ where k is a *kind* with three possible values: C denotes continuous functions that can only be used in continuous contexts, D denotes discrete functions that must be activated on the emission of a signal, A denotes a function that can be used in any context. The subkind relation $\subseteq$ is defined as $\forall k,\ k \subseteq k$ and $\mathrm{A} \subseteq k$. The type language is:

$$\begin{array}{lcl} t & ::= & t \times t \mid \alpha \mid bt \\ k & ::= & \mathrm{D} \mid \mathrm{C} \mid \mathrm{A} \\ bt & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{signal} \mid \texttt{timer} \\ \sigma & ::= & \forall \alpha_1, ..., \alpha_n.\ t \xrightarrow{k} t \end{array}$$

A type (t) can be a pair ($t \times t$), a type variable (α) or a base type (bt). The base types are `int` and `bool` for constants, `signal` for signals emitted by guards, and `timer` for timer variables. Timers have a particular type to prevent their concrete values being used in an expression. Functions are associated to a type scheme σ where type variables are generalized.

A global environment G tracks the type schemes of functions, and another environment H assigns types to variables. We write $x : t$ to state that x is of type t, and if H_1 and H_2 are two environments, $H_1 + H_2$ denotes their union, provided their domains are disjoint.

Generalization and instantiation Type schemes are obtained by generalizing the free variables in function types $t_1 \xrightarrow{k} t_2$:

$$gen(t_1 \xrightarrow{k} t_2) = \forall \alpha_1, \ldots, \alpha_n.\ t_1 \xrightarrow{k} t_2 \text{ where } \{\alpha_1, \ldots, \alpha_n\} = ftv(t_1 \xrightarrow{k} t_2),$$

where $ftv(t)$ denotes the set of free type variables in type t.

A type scheme can be instantiated by substituting type variables with actual types. $Inst(\sigma)$ denotes the set of possible instantiations of a type scheme σ. The kind of a type $t_1 \xrightarrow{k} t_2$ can be instantiated with any kind k' where $k \subseteq k'$:

$$\frac{k \subseteq k'}{(t \xrightarrow{k'} t')[t_1/\alpha_1, ..., t_n/\alpha_n] \in Inst(\forall \alpha_1, ..., \alpha_n.\ t \xrightarrow{k} t')}$$

Typing rules

Typing is defined by four judgments which resemble those of Zélus:

$$\frac{}{G, H \vdash_k e : t}\ (\text{TYP-EXP}) \qquad \frac{}{G, H \vdash_k E : H'}\ (\text{TYP-ENV})$$

$$\frac{}{\vdash_{pat} p : t, H}\ (\text{TYP-PAT}) \qquad \frac{}{G, H \vdash_k h : t}\ (\text{TYP-HANDLER})$$

The judgment (TYP-EXP) states that in environments G and H, expression e has kind k and type t. The judgment (TYP-ENV) states that in environments G and H, a set of equations E has kind k and produces the type environment H'. The judgment (TYP-PAT) states that a pattern p has type t and defines a type environment H. The judgment (TYP-HANDLER) states that in environments G and H, the value defined by a handler h has type t and kind k.

We add a fifth judgment:

$$\frac{}{G, H \vdash_{zone} c}\ (\text{CHECK-ZONE})$$

to check if a constraint defines a valid zone. In particular, (CHECK-ZONE) requires that the definition of zones only involve timer differences and integer bounds.

The initial environment G_0 contains the type of primitive operators, like `fby`, and imported operators, like (+) and (=).

$$\begin{array}{lcl} (+) & : & \texttt{int} \times \texttt{int} \xrightarrow{\text{A}} \texttt{int} \\ (=) & : & \forall\alpha,\ \alpha \times \alpha \xrightarrow{\text{A}} \texttt{bool} \\ \texttt{fby} & : & \forall\alpha,\ \alpha \times \alpha \xrightarrow{\text{D}} \alpha \end{array}$$

Imported operators have kind A since they can be used in any context. The unit delay `fby` has kind D since it is only allowed in discrete contexts.

The typing rules are presented in figure 6.7.

(EQ) An equation x `=` e is well-typed if the types of x and e coincide. The kind of the equation is the kind of e.

(AND) The parallel composition of two sets of equations E_1 `and` E_2 is well-typed if both E_1 and E_2 are well-typed. The kind must be the same for E_1 and E_2.

(PRESENT) The equation x `=` `present` h `init` e_0 activates at instants defined by the handler h. The equation is well-typed if the handler is well-typed and produces a value of type t that coincides with the type of the initialization expression e_0. This equation can be used in continuous and discrete contexts depending on the handler. In any case, the initialization value must be of kind D, even in continuous contexts.

(PRESENT-ELSE) When a default value is provided it must also have the type t returned by the handler h and the same kind. In particular, in continuous contexts the default value is not guarded by a signal and must thus have kind C.

(TIMER) The equation `timer` x `init` e_0 `reset` h defines a variable of type `timer`. This equation is well-typed if the reset handler h is well-typed and returns a value of type `int`, and if the initialization expression is also of type `int`. The reset handler must have kind C and the overall kind is C. Timers can only be defined in continuous contexts.

$$\frac{G, H \vdash_k e : t}{G, H \vdash_k x \texttt{ = } e : [x : t]}\ (\text{EQ})$$

$$\frac{G, H \vdash_k E_1 : H_1 \qquad G, H \vdash_k E_2 : H_2}{G, H \vdash_k E_1 \texttt{ and } E_2 : H_1 + H_2}\ (\text{AND})$$

$$\frac{G, H \vdash_k h : t \qquad G, H \vdash_{\mathrm{D}} e_0 : t}{G, H \vdash_k x \texttt{ = present } h \texttt{ init } e_0 : [x : t]}\ (\text{PRESENT})$$

$$\frac{G, H \vdash_k h : t \qquad G, H \vdash_k e : t}{G, H \vdash_k x \texttt{ = present } h \texttt{ else } e : [x : t]}\ (\text{PRESENT-ELSE})$$

$$\frac{G, H \vdash_{\mathrm{D}} e_0 : \texttt{int} \qquad G, H \vdash_{\mathrm{C}} h : \texttt{int}}{G, H \vdash_{\mathrm{C}} \texttt{timer } x \texttt{ init } e_0 \texttt{ reset } h : [x : \texttt{timer}]}\ (\text{TIMER})$$

$$\frac{G, H \vdash_{zone} c}{G, H \vdash_{\mathrm{C}} \texttt{always \{ } c \texttt{ \}} : []}\ (\text{ALWAYS})$$

$$\frac{G, H \vdash_{zone} c}{G, H \vdash_{\mathrm{C}} \texttt{emit } s \texttt{ when \{ } c \texttt{ \}} : [s : \texttt{signal}]}\ (\text{GUARD})$$

$$G, H \vdash_k 42 : \texttt{int}\ (\text{CONST})$$

$$G, H + [x : t] \vdash_k x : t\ (\text{VAR})$$

$$\frac{G, H \vdash_k e_1 : t_1 \qquad H \vdash_k e_2 : t_2}{G, H \vdash_k (e_1, e_2) : t_1 \times t_2}\ (\text{PAIR})$$

$$\frac{t \xrightarrow{k} t' \in \mathit{Inst}(G(f)) \qquad G, H \vdash_k e : t}{G, H \vdash_k f(e) : t'}\ (\text{APP})$$

$$\frac{G,\ H \vdash_k E : H_e \qquad G,\ H + H_e \vdash_k e : t}{G, H \vdash_k e \texttt{ where rec } E : t}\ (\text{WHERE-REC})$$

$$\frac{\vdash_{pat} p : t_1, H_p \qquad G, H_p \vdash_{\mathrm{C}} e : t_2}{G \vdash \texttt{let hybrid } f\texttt{(}p\texttt{)} \texttt{ = } e : [f : \mathit{gen}(t_1 \xrightarrow{\mathrm{C}} t_2)]}\ (\text{DEF-HYBRID})$$

$$\frac{\vdash_{pat} p : t_1, H_p \qquad G, H_p \vdash_{\mathrm{D}} e : t_2}{G \vdash \texttt{let node } f\texttt{(}p\texttt{)} \texttt{ = } e : [f : \mathit{gen}(t_1 \xrightarrow{\mathrm{D}} t_2)]}\ (\text{DEF-NODE})$$

$$\frac{\vdash_{pat} p : t_1, H_p \qquad G, H_p \vdash_{\mathrm{A}} e : t_2}{G \vdash \texttt{let } f\texttt{(}p\texttt{)} \texttt{ = } e : [f : \mathit{gen}(t_1 \xrightarrow{\mathrm{A}} t_2)]}\ (\text{DEF-ANY})$$

$$\frac{G \vdash d_1 : G_1 \qquad G + G_1 \vdash d_2 : G_2}{G \vdash d_1\ d_2 : G_1 + G_2}\ (\text{DEF-SEQ})$$

$$\vdash_{pat} x : t, [x : t]\ (\text{PAT-VAR})$$

$$\frac{\vdash_{pat} p_1 : t_1, H_1 \qquad \vdash_{pat} p_2 : t_2, H_2}{\vdash_{pat} (p_1,\ p_2) : t_1 \times t_2, H_1 + H_2}\ (\text{PAT-PAIR})$$

$$\frac{\forall i \in \{1, ..., n\} \qquad G, H \vdash_{\mathrm{D}} e_i : t \qquad G, H \vdash_{\mathrm{C}} c_i : \texttt{signal}}{G, H \vdash_{\mathrm{C}} c_1 \ \rightarrow\ e_1 \ \texttt{|}\ \cdots\ \texttt{|}\ c_n \ \rightarrow e_n : t}\ (\text{HANDLER-C})$$

$$\frac{\forall i \in \{1, .., n\} \qquad G, H \vdash_{\mathrm{D}} e_i : t \qquad G, H \vdash_{\mathrm{D}} c_i : \texttt{bool}}{G, H \vdash_{\mathrm{D}} c_1 \ \rightarrow\ e_1 \ \texttt{|}\ \cdots\ \texttt{|}\ c_n \ \rightarrow e_n : t}\ (\text{HANDLER-D})$$

$$\frac{G, H \vdash_{\mathrm{C}} t : \texttt{timer} \qquad G, H \vdash_{\mathrm{C}} e : \texttt{int}}{G, H \vdash_{zone} t \sim e}\ (\text{ZONE-VAR})$$

$$\frac{G, H \vdash_{\mathrm{C}} t_1 : \texttt{timer} \qquad G, H \vdash_{\mathrm{C}} t_2 : \texttt{timer} \qquad G, H \vdash_{\mathrm{C}} e : \texttt{int}}{G, H \vdash_{zone} t_1 - t_2 \sim e}\ (\text{ZONE-DIFF})$$

$$\frac{G, H \vdash_{zone} c_1 \qquad G, H \vdash_{zone} c_2}{G, H \vdash_{zone} c_1 \texttt{ \&\& } c_2}\ (\text{ZONE-AND})$$

Figure 6.7: The typing rules.

(ALWAYS) The equation `always { c }` introduces an invariant and does not define a variable. This equation is well-typed if it has kind C and if the constraint c is a valid zone. Invariants are only allowed in continuous contexts.

(GUARD) The equation `emit` s `when { c }` is well-typed if constraint c defines a valid zone. Variable s is then of type `signal` and the overall kind is C. Guards are only allowed in continuous contexts.

(CONST) The typing of constants is illustrated with the integer constant 42. Constants can be used in any context.

(VAR) A variable of type t can be used in any context.

(PAIR) A pair (e_1, e_2) is of type $t_1 \times t_2$ if e_1 has type t_1 and e_2 has type t_2; e_1 and e_2 must have the same kind.

(APP) An application $f(e)$ is of type t' if e has type t and if $t \xrightarrow{k} t'$ is a valid instantiation of the type scheme of f. The kind of the application $f(e)$ is given by the kind of f.

(WHERE-REC) A local definition e `where rec` E is well-typed if the set of equations E is well-typed and expression e is well-typed in the extended environment.

(DEF-HYBRID) (DEF-NODE) (DEF-ANY) A function definition has type $t_1 \xrightarrow{k} t_2$ if the input pattern p has type t_1 and the defining expression has type t_2. Function types are generalized. The kind is given in the definition: `hybrid` for C, `node` for D, nothing for A.

(DEF-SEQ) Function definitions are typed sequentially.

(PAT-VAR) (PAT-PAIR) Patterns return an environment containing the types of their variables.

(HANDLER-C) (HANDLER-D) A handler $c_1 \rightarrow e_1 \mid \cdots \mid c_n \rightarrow e_n$ is well-typed if all expressions e_i have the same type t and conditions c_i have type `signal` in continuous contexts or `bool` in discrete contexts. The expressions e_i must have kind D since, in any case, they are only activated at discrete instants.

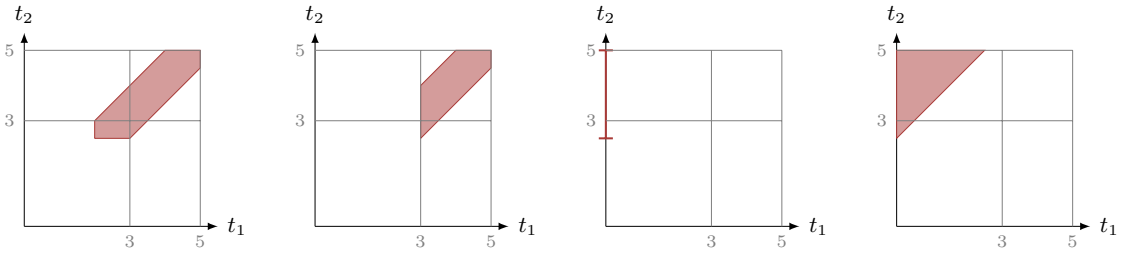
(ZONE-VAR) (ZONE-DIFF) (ZONE-AND) A constraint c defines a valid zone if variables are of type `timer` and bounds are of kind C and type `int`; the values are thus piecewise constant and can only change on signal emissions.

Since expressions of kind A can be executed in any context we also have the following subtyping property:

Property 6.1 (Subtyping)**.**

$$G, H \vdash_{\mathrm{A}} e : t \implies (G, H \vdash_{\mathrm{C}} e : t) \wedge (G, H \vdash_{\mathrm{D}} e : t)$$

Proof. By induction on the typing derivation of $G, H \vdash_{\mathrm{A}} e : t$. □



(a) Current zone zcp (b) Trigger zone zg (c) Initial zone zi (d) New zone zc

Figure 6.8: Computing the new zone when clock `c1` is activated.

6.6 Compilation

In this section, we show how to compile a program mixing continuous and discrete components written in ZSy into a purely discrete program with additional inputs and outputs to handle zones and nondeterministic choices. Recall the simulation scheme described in section 6.1. At each step the user chooses a transition from a set of possible choices: `wait`, if possible, or firing enabled guards. Then there are two possible cases:

1. If the chosen transition is `wait` we compute the new zone by letting time elapse until the next change in the set of enabled guards (if allowed by the active invariants).
2. Otherwise, firing guards triggers discrete computations, possibly resets some timers, and returns a new initial zone. The new zone is obtained by letting time elapse from this initial zone until the next change in the set of enabled guards.

We target a modular compilation scheme where nodes are compiled separately and where function calls are not inlined. Consider again the motivating example of section 6.1:

```
let hybrid archi(t_min, t_max) = c1, c2 where
  rec c1 = metro(t_min, t_max)
  and c2 = metro(t_min, t_max)
```

val archi: int $\times$ *int* $\xrightarrow{\text{C}}$ *signal* $\times$ *signal*

Starting in zone zcp, figure 6.8 shows the succession of computations realized when clock `c1` is activated to obtain the current zone. First we compute the trigger zone zg of the guards fired by the user, that is, the intersection between the current zone and the activation zones of the fired guards (figure 6.8b). Signals emitted when firing a guard can be used to reset timers defined in other functions. The initial zone zi is obtained by applying these resets to zg (figure 6.8c). Finally, the new zone zc is obtained from zi by letting time elapse until the next change in the set of enabled guards which depends on the guards and invariants defined in each function (figure 6.8d). To summarize, when the user fires guards, the computation of the new zone comprises three steps:

1. from the current zone zcp, compute the trigger zone zg,
2. from zg, compute the initial zone zi by applying the resets,
3. compute the new zone zc by letting time elapse from zi.

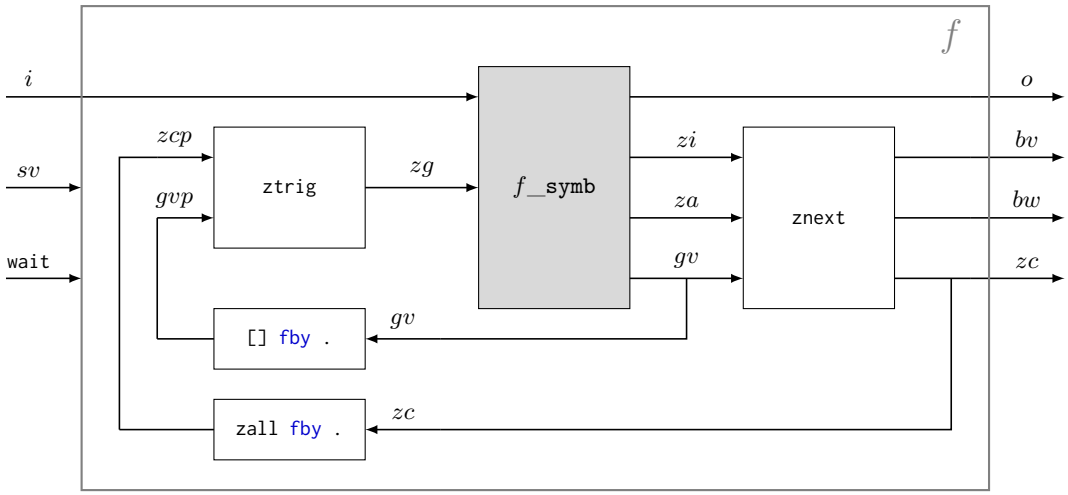


Figure 6.9: Result of the compilation of a continuous function `let hybrid` f`(`i`) =` o into a discrete function `let node` f`(wait,`sv, i`) =` o, bv, bw, zc. Function f`_symb` is obtained by a source-to-source transformation of f.

As the function `archi` shows, the zones zg, zi, and zc all depend on information computed locally in function calls. We implement this logic by generating for each continuous function f a discrete function f`_symb` that returns at each step the information required to compute the new zone, namely the initial zone zi, the conjunction of invariants za and the vector of guard activation zones gv.

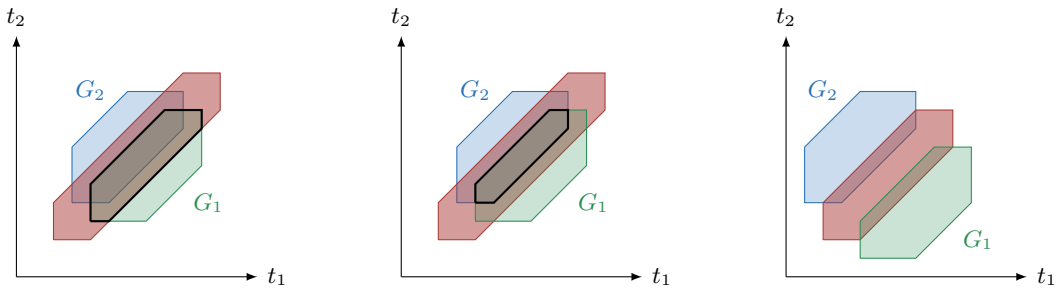
The final result of the compilation is illustrated in figure 6.9. Compared to the initial function `let hybrid` f`(`i`) =` o, there are two additional inputs: sv a boolean vector where elements set to *true* indicate the guards fired by the user, and `wait` a boolean set to *true* when the user chooses the `wait` transition. The resulting function returns three additional outputs: bv a boolean vector that characterizes the set of enabled guards, bw a boolean set to *true* if the `wait` transition is enabled, and zc the current zone.

The `wait` transition and the guard activations are mutually exclusive: if the input `wait` is set to *true*, other inputs are ignored. However, it is possible to simultaneously fire multiple guards. This is another fundamental difference with Uppaal where the user can only choose one transition at a time which precludes the parallel composition of discrete computations triggered by the simultaneous activation of multiple guards. Synchronous languages like Zélus are designed to handle such concurrent computations.

The execution scheme is as follows:

1. Given the current zone zcp and the vector of guard activation zones gvp computed at the previous step, function `ztrig` computes the trigger zone zg.
2. Function f`_symb` triggers the discrete computations and returns the initial zones zi obtained by applying the resets to zg, the conjunction of active invariants za, and the new vector of guard activation zones gv.
3. Function `znext` computes the new zone zc by letting time elapse from zi until the next change in the set of enabled guards.

Compilation produces discrete code that can be compiled with the Zélus compiler and executed.



(a) Firing one guard (b) Firing two guards (c) Exclusive guards

Figure 6.10: Computing the trigger zone.

Causality Timers, guards, and invariants cannot introduce immediate dependencies. The only continuous events are produced by guards and can only be used to reset timers or trigger discrete computations using the `present` constructs. Resets are applied to the trigger zone *zg* obtained from the zones computed at the previous step *gvp* and *zcp* (the `fby` operators in figure 6.9). There is, for example, no instantaneous dependency in the following program:

```
let hybrid causal() = s where
  rec timer t init 0 reset s → 0
  and emit s when {t = 0}
```

val causal: unit $\xrightarrow{C}$ *signal*

If the user triggers the guard, the reset of `t` occurs at the next step. The value of `t` does not instantaneously depend on `s`. The timer may thus be reset indefinitely without time elapsing.

However, we still need to ensure the absence of dependency cycles in discrete equations. The classic causality analysis of synchronous languages suffices. This is done by Zélus when compiling the resulting discrete program.

Computing the trigger zone: `ztrig`

The function `ztrig` computes the trigger zone of the guards fired by the user. If the user fires several guards simultaneously, the result is the intersection of the corresponding trigger zones. Note that guards may be mutually exclusive. Firing two such guards would return an empty zone and block the simulation. It is the user's responsibility to avoid such situations (or to discover them). Figure 6.10 illustrates the possible situations.

```
let node ztrig(sv, zcp, gvp) = zg where
  rec fv = filter(gvp, sv)
  and zg = zinter(zcp, zinterfold(fv))
```

val ztrig: bool list × *zone* × *zone list* $\xrightarrow{D}$ *zone*

Each element set to *true* in the input vector `sv` corresponds to a guard fired by the user. The list of the activation zones of the fired guards `fv` is obtained by filtering `gvp` according to `sv`. The trigger zone `zg` is the intersection of the elements in `fv` with the current zone `zcp`.

Computing the current zone: `znext`

The discrete function `znext` computes the current zone zc and the set of enabled transitions (`wait` or enabled guards) using the information computed by f`_symb`, namely the initial zone `zi`, the conjunction of all invariants `za`, and the vector of guard activation zones `gv`.

```
let node znext(wait, zi, za, gv) = zc, bv, bw where
  rec dp = if wait then (dzero fby d) else dzero
  and dl = zdistmap(zi, gv)
  and d  = mindist(gv, dp)
  and zn = zsweep(zi, dp, d)
  and zc = zinter(zn, za)
  and bv = zenabled(zc, gv)
  and zm = zinter(zup(zn), za)
  and bw = (zc ≠ zm)
```

val znext: bool $\times$ *zone* $\times$ *zone* $\times$ *zone list* $\xrightarrow{\mathrm{D}}$ *zone* $\times$ *bool list*

When the user chooses a `wait` transition, the current zone `zc` is obtained using the sweeping mechanism described in section 6.3. The sweeping mechanism restarts from `dzero` $= (0, \leq)$, each time the user chooses to fire guards instead of `wait` (the outputs of f`_symb` only change when the user fires guards).

From `zi` and `gv` we compute `dl`, the list of distances associated to each guard. The function `mindist(dl, dp)` returns the smallest distance greater than `dp` and is used to enumerate pairs of distances in order (`dp = dzero fby d` is the distance reached at the previous step). Each pair of successive distances `(dp, d)` defines a zone `zn` obtained by sweeping `zi` between `dp` and `d`. The current zone `zc` is the intersection of `zn` and the invariant `za`.

From `zc` and `gv` we compute `bv`, the vector characterizing the set of enabled guards. The maximal zone `zm` is the zone obtained from `zn` by letting time elapse indefinitely with the same invariant `za`. The `wait` transition is enabled as long as `zc` $\neq$ `zm`, the defining equation of `bw`.

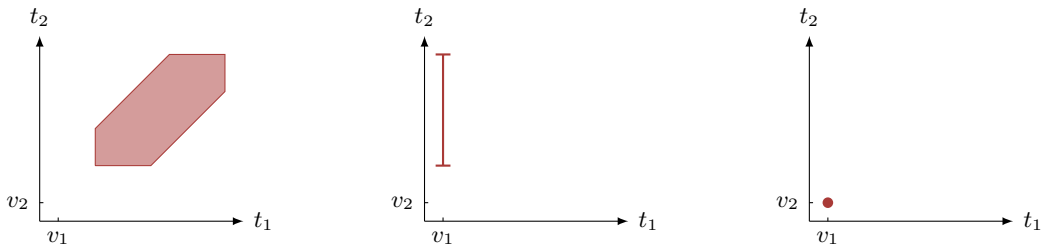
Remark 6.1. If `znext` returned `zm` instead of `zc`, we would obtain a simulator like that of Uppaal where at each step, the user may choose among all reachable guards. The compilation scheme can thus also be adapted for symbolic simulation without `wait` transitions.

Source-to-source generation of f`_symb`

We now show how to adapt the source-to-source compilation of Zélus described in [BBCP11a, §4] to generate the discrete function f`_symb` from a continuous function f. The translation replaces timers, invariants, and guard definitions.

$$\texttt{let hybrid } f(i) = o \quad \longmapsto \quad \texttt{let node } f\texttt{\_symb}(tv, \texttt{wait}, sv, zg, i) = o, zi, za, gv$$

Compared to f, f`_symb` takes four additional inputs: tv a vector of timer identifiers, `wait` a boolean set to *true* when the user chooses the `wait` transition, sv the boolean vector characterizing the guards fired by the user, and zg the trigger zone computed by `ztrig`. It returns three additional outputs: zi the initial zone, za the conjunction of all invariants, and gv the vector of guard activation zones.



(a) Trigger zone zg (b) $zi_1 = reset(zg, t_1, v_1)$ (c) $zi_2 = reset(zi_1, t_2, v_2)$

Figure 6.11: Computing the initial zone.

We compute the initial zone by applying resets to the trigger zone zg and gather the guard activation zones and the invariants. There are three cases:

1. If the transition is `wait` the initial zone is unchanged.
2. If the fired guards do not trigger resets, the initial zone is zg.
3. If the fired guards trigger resets, we apply the resets to zg.

The resets can be applied in any order, the result is the same. Figure 6.11 shows the result of two resets on a trigger zone zg.

The initial zone is computed incrementally. Each timer definition produces a new zone obtained by applying the resets defined in its reset handler to the zone computed so far. The first reset handler applies its resets to zg to produce zi. Then the second reset handler applies its resets to zi to produce zi' and so on. The initial zone is the result produced by the last reset handler.

Additionally, the translation of a timer definition adds a new identifier to the input vector tv. Identifiers defined in function calls are also accumulated in tv. These identifiers allow the discrimination of timers defined in multiple instantiations of the same node (like the timers defined in the two instantiations of `metro` in `archi`). At the end of the compilation process, when combining f`_symb` with `ztrig` and `znext`, timer identifiers are replaced by unique values $1, 2, 3, \ldots$: the associated dimensions in the DBMs.

Similarly, the translation of a guard adds a new element to the input vector sv that represents the signals controlled by the user. Elements corresponding to the guards defined in function calls are also accumulated in sv. At the end of the translation, there is one element in sv for each guard defined in the program.

In parallel, the translation gathers the guard activation zones and the invariants into two vectors, av and gv. At the end of the translation gv becomes an output of f`_symb` and the conjunction of all invariants za is defined as the intersection of the elements in av.

Notations In the following, we write [] for the empty vector and the empty set of equations; $[x_1, \ldots x_n] @ [y_1, \ldots, y_n] = [x_1, \ldots, x_n, y_1, \ldots, y_n]$ for the concatenation of two vectors; and $x_0 :: [x_1, \ldots, x_n] = [x_0, x_1, \ldots, x_n]$ to add an element at the beginning of a vector.

The generation of f_symb is defined by five mutually recursive functions:

TraDef(d) translates declarations. Only continuous declarations introduced by `hybrid` are translated. The translation returns a declaration of a discrete function f_symb with four additional inputs, tv, wait, sv, and zg; and three additional outputs, zi, za, and gv.

TraEq(zi, E) translates equations using the initial zone zi computed so far. It returns a tuple $\langle e', zi', av, gv, sv, tv, E\rangle$, where e' is the translated equation, zi is the new initial zone, vectors av and gv accumulate the invariants and the guard activation zones, vectors sv and tv accumulate the new inputs (signals and timers), and E is the set of equations added by the translation.

Tra(zi, e) translates expressions using the initial zone zi computed so far. It returns a tuple $\langle e', zi', av, gv, sv, tv, E\rangle$ with the same set of vectors to gather information, where e' is the translated expression, zi' is the new initial zone, and E is the set of equations added by the translation.

TraH(zi, h) translates handlers (conditions may contain declarations). It returns a tuple $\langle h', zi', av, gv, sv, tv, E\rangle$ where h' is the translated handler.

TraZ(zi, h) translates constraints (bounds may contain declarations). It returns a tuple $\langle c', zi', av, gv, sv, tv, E\rangle$ where c' is the translated constraint.

The definition of the translation is shown in figure 6.12. The interesting cases are as follows:

(`let hybrid` f`(`p`) = `e) The translation of the function body returns $\langle e', zi', av, gv, sv, tv, E\rangle$ where zi' is the initial zone computed from zg by the translation. The vectors sv and tv are turned into inputs.

`let node` f_symb$(tv,$ wait$, sv, zg, p)$ `= `e', zi, za, gv

The initial zone zi is only updated when the user does not choose the wait transition and is defined via the equation zi `= if` wait `then (`zall `fby` zi`) else` zi'.

The conjunction of all invariants za is defined as the intersection of the elements in av via the equation za `= zinterfold(`av`)`.

(f`(`e`)` when *KindOf*$(f) = \mathrm{C}$) In an application of a continuous function, f is replaced by the function f_symb, with four extra inputs and three extra outputs, in a new equation

(r, zi_f, za, g) `= `f_symb$(t,$ wait$, s, zi', e')$

where e' is the translation of the argument e and zi' is the initial zone obtained by the translation of e.

The input vectors s and t of f_symb are added to sv and tv, and the outputs za and g are added to av and gv. The structure of nested function calls is reflected in the tree structure of sv, tv, and gv.

(`timer` t `init` e_0 `reset` h) The translation of a timer definition adds the timer identifier t to tv and applies the reset defined in its reset handler to the zone obtained by the translation of the handler zi_h. The result is a new zone zi_t whose defining equation is

zi_t `= present (true fby false)` $\rightarrow$ `zreset(`zi_h, t, e_0`)`
`|` c_1' $\rightarrow$ `zreset(`zi_h, t, e_1`)`
`|` $\ldots$
`|` c_n' $\rightarrow$ `zreset(`zi_h, t, e_n`)`
`else` zi_h

$$
\begin{array}{lcl}
Tra(zi, x) & = & \langle x, zi, [], [], [], [], [] \rangle \\
Tra(zi, v) & = & \langle v, zi, [], [], [], [], [] \rangle \\
Tra(zi, op(e)) & = & \mathit{let}\ \langle e', zi', av, gv, sv, tv, E \rangle = Tra(zi, e)\ \mathit{in} \\
 & & \langle op(e'), zi', av, gv, sv, tv, E \rangle \\
Tra(zi, (e_1,\ e_2)) & = & \mathit{let}\ \langle e'_1, zi_1, av_1, gv_1, sv_1, tv_1, E_1 \rangle = Tra(zi, e_1)\ \mathit{in} \\
 & & \mathit{let}\ \langle e'_2, zi_2, av_2, gv_2, sv_2, tv_2, E_2 \rangle = Tra(zi_1, e_2)\ \mathit{in} \\
 & & \langle (e'_1,\ e'_2), zi_2, av_1 \mathbin{@} av_2, gv_1 \mathbin{@} gv_2, sv_1 \mathbin{@} sv_2, tv_1 \mathbin{@} tv_2, E_1 \texttt{ and } E_2 \rangle \\
Tra(zi, f(e)) & = & \mathit{let}\ \langle e', zi', av, gv, sv, tv, E \rangle = Tra(zi, e)\ \mathit{in} \\
\quad \text{if } KindOf(f) = \mathrm{A} & & \langle f(e'), zi', av, gv, sv, tv, E \rangle \\
\bullet\ Tra(zi, f(e)) & = & \mathit{let}\ \langle e', zi', av, gv, sv, tv, E \rangle = Tra(zi, e)\ \mathit{in} \\
\quad \text{if } KindOf(f) = \mathrm{C} & & \langle r, zi_f, za :: av, g :: gv, s :: sv, t :: tv, \\
 & & \ E \texttt{ and } (r, zi_f, za, g) = f\texttt{\_symb}(t, \texttt{wait}, s, zi', e') \rangle \\
 & & \text{where } r,\ zi_f,\ za,\ g,\ s, \text{ and } t \text{ are fresh variables.} \\
Tra(zi, e \texttt{ where rec } E) & = & \mathit{let}\ \langle zi_1, av_1, gv_1, sv_1, tv_1, E_1 \rangle = TraEq(zi, E)\ \mathit{in} \\
 & & \mathit{let}\ \langle e', zi_2, av_2, gv_2, sv_2, tv_2, E_2 \rangle = Tra(zi_1, e)\ \mathit{in} \\
 & & \langle e', zi_2, av_1 \mathbin{@} av_2, gv_1 \mathbin{@} gv_2, sv_1 \mathbin{@} sv_2, tv_1 \mathbin{@} tv_2, E_1 \texttt{ and } E_2 \rangle \\
 & & \text{assuming unique names for variables in } E. \\
TraEq(zi, x \texttt{ = } e) & = & \mathit{let}\ \langle e', zi', av, gv, sv, tv, E \rangle = Tra(zi, e)\ \mathit{in} \\
 & & \langle zi', av, gv, sv, tv, E \texttt{ and } x \texttt{ = } e' \rangle \\
TraEq(zi, E_1 \texttt{ and } E_2) & = & \mathit{let}\ \langle zi_1, av_1, gv_1, sv_1, tv_1, E'_1 \rangle = TraEq(zi, E_1)\ \mathit{in} \\
 & & \mathit{let}\ \langle zi_2, av_2, gv_2, sv_2, tv_2, E'_2 \rangle = TraEq(zi_1, E_2)\ \mathit{in} \\
 & & \langle zi_2, av_1 \mathbin{@} av_2, gv_1 \mathbin{@} gv_2, sv_1 \mathbin{@} sv_2, tv_1 \mathbin{@} tv_2, E'_1 \texttt{ and } E'_2 \rangle \\
TraEq(zi, x \texttt{ = present } h \texttt{ init } e_0) & = & \mathit{let}\ \langle h', zi_h, av_h, gv_h, sv_h, tv_h, E_h \rangle = TraH(zi, h)\ \mathit{in} \\
 & & \langle zi_h, av_h, gv_h, sv_h, tv_h, E_h \texttt{ and } x \texttt{ = present } h' \texttt{ init } e_0 \rangle \\
TraEq(zi, x \texttt{ = present } h \texttt{ else } e) & = & \mathit{let}\ \langle h', zi_h, av_h, gv_h, sv_h, tv_h, E_h \rangle = TraH(zi, h)\ \mathit{in} \\
 & & \mathit{let}\ \langle e', zi', av, gv, sv, tv, E \rangle = Tra(zi_h, e)\ \mathit{in} \\
 & & \langle zi', av_h \mathbin{@} av, gv_h \mathbin{@} gv, sv_h \mathbin{@} sv, tv_h \mathbin{@} tv, \\
 & & \ E_h \texttt{ and } E \texttt{ and } x \texttt{ = present } h' \texttt{ else } e' \rangle \\
\bullet\ TraEq(zi, \texttt{timer } t \texttt{ init } e_0 \texttt{ reset } h) & = & \mathit{let}\ \langle h', zi_h, av_h, gv_h, sv_h, tv_h, E_h \rangle = TraH(zi, h)\ \mathit{in} \\
 & & \mathit{let}\ c'_1 \texttt{ -> } e_1 \texttt{ | } \cdots \texttt{ | } c'_n \texttt{ -> } e_n = h'\ \mathit{in} \\
 & & \langle zi_t, av_h, gv_h, sv_h, t :: tv_h, \\
 & & \ E_h \texttt{ and } zi_t \texttt{ = present (true fby false) -> zreset}(zi_h, t, e_0) \\
 & & \qquad \texttt{| } c'_1 \texttt{ -> zreset}(zi_h, t, e_1) \\
 & & \qquad \texttt{| } \cdots \\
 & & \qquad \texttt{| } c'_n \texttt{ -> zreset}(zi_h, t, e_n) \\
 & & \qquad \texttt{else } zi_h \rangle \\
 & & \text{where } zi_t \text{ is a fresh variable.} \\
\bullet\ TraEq(zi, \texttt{always \{ } c \texttt{ \}}) & = & \mathit{let}\ \langle c', zi_c, av_c, gv_c, sv_c, tv_c, E_c \rangle = TraZ(zi, c)\ \mathit{in} \\
 & & \langle zi_c, za :: av_c, gv_c, sv_c, tv_c, E_c \texttt{ and } za \texttt{ = zmake}(c') \rangle \\
 & & \text{where } za \text{ is a fresh variable.} \\
\bullet\ TraEq(zi, \texttt{emit } s \texttt{ when \{ } c \texttt{ \}}) & = & \mathit{let}\ \langle c', zi_c, av_c, gv_c, sv_c, tv_c, E_c \rangle = TraZ(zi, c)\ \mathit{in} \\
 & & \langle zi_c, av_c, zs :: gv_c, s :: sv_c, tv_c, E_c \texttt{ and } zs \texttt{ = zmake}(c') \rangle \\
 & & \text{where } zs \text{ is a fresh variable.} \\
TraH(zi, c_1 \texttt{ -> } e_1 \texttt{ | } \cdots \texttt{ | } c_n \texttt{ -> } e_n) & = & \mathit{let}\ \langle c'_i, zi_i, av_i, gv_i, sv_i, tv_i, E_i \rangle = Tra(zi_{i-1}, c_i)\ \mathit{in} \\
 & & \langle c'_1 \texttt{ -> } e_1 \texttt{ | } \cdots \texttt{ | } c'_n \texttt{ -> } e_n, zi_n, av_1 \cdots \mathbin{@} av_n, gv_1 \cdots \mathbin{@} gv_n, \\
 & & \ sv_1 \cdots \mathbin{@} sv_n, tv_1 \cdots \mathbin{@} tv_n, E_1 \cdots \texttt{ and } E_n \rangle \\
 & & \text{where } zi_0 = zi. \\
TraZ(zi, \Delta_1 \sim_1 e_1 \cdots \texttt{ \&\& } \Delta_n \sim_n e_n) & = & \mathit{let}\ \langle e'_i, zi_i, av_i, gv_i, sv_i, tv_i, E_i \rangle = Tra(zi_{i-1}, e_i)\ \mathit{in} \\
 & & \langle \Delta_1 \sim_1 e'_1 \cdots \texttt{ \&\& } \Delta_n \sim_n e'_n, zi_n, av_1 \cdots \mathbin{@} av_n, gv_1 \cdots \mathbin{@} gv_n, \\
 & & \ sv_1 \cdots \mathbin{@} sv_n, tv_1 \cdots \mathbin{@} tv_n, E_1 \cdots \texttt{ and } E_n \rangle \\
 & & \text{where } zi_0 = zi. \\
\bullet\ TraDef(\texttt{let hybrid } f(p) \texttt{ = } e) & = & \mathit{let}\ \langle e', zi', av, gv, sv, tv, E \rangle = Tra(zg, e)\ \mathit{in} \\
 & & \texttt{let node } f\texttt{\_symb}(tv, \texttt{wait}, sv, zg, p) \texttt{ = } e', zi, za, gv \texttt{ where} \\
 & & \quad \texttt{rec } E \\
 & & \quad \texttt{and } za \texttt{ = zinterfold}(av) \\
 & & \quad \texttt{and } zi \texttt{ = if wait then (zall fby } zi) \texttt{ else } zi' \\
 & & \text{where } zi \text{ and } za \text{ are fresh variables.}
\end{array}
$$

Figure 6.12: The source-to-source generation of f__symb.

At the first instant `(true fby false)`, the value of t is given by the initialization expression e_0. The first initial zone is thus the conjunction of the initial values of the timers. Otherwise, for each reset condition c_i, t is reset to the value e_i.

If the user fires a guard that does not trigger any reset the initial zone is, as expected, the trigger zone zg passed into f`_symb` by `ztrig`.

(`always { `c` }`) We add the invariant za defined by the following equation to av: za `= zmake(`c'`)` where c' is the translation of c.

(`emit `s` when { `c` }`) The signal s emitted when the guard is fired is added to sv to become an input and we add the activation zone zs of the guard defined by the following equation to gv: zs `= zmake(`c'`)` where c' is the translation of c.

Since ZSy is a single assignment kernel, a signal can only be defined by a unique guard.

In the other cases we simply apply the translation recursively to compute the initial zone and accumulate inputs, guard activation zones, and invariants. We note only the following details:

(E_1 `and` E_2) From a zone zi, the translation of the set of equations E_1 returns zi_1 which is used to start the translation of E_2. We thus chain resets defined in E_1 and E_2. Invariants, guard activation zones, signals, and timers defined in E_1 and E_2 are gathered into the vectors av, gv, sv, and tv.

(e `where rec` E) Local definitions are flattened which is sound provided that variables have unique identifiers. There are no side-effects and equations can be safely reordered.

(x `=` `present` h `init` e_0) The initialization expression e_0 is discrete and thus not translated.

($s_1 \rightarrow e_1$ `|` $\cdots$ `|` $s_n \rightarrow e_n$) Equations e_i are discrete and thus not translated.

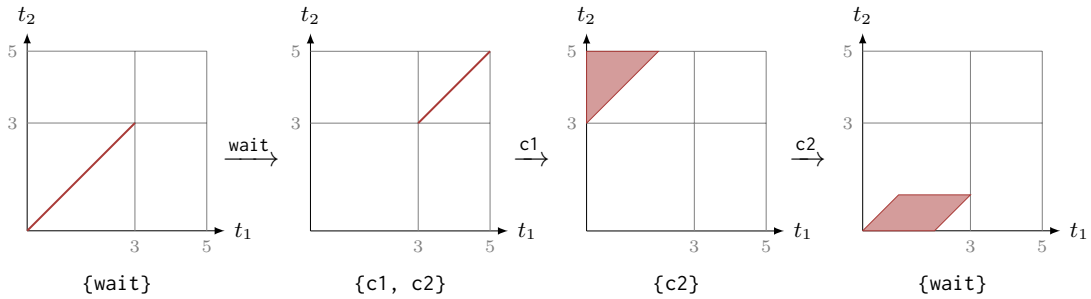
Optimization The constraints defining guard activation zones and invariants can only change on signal emissions, that is, when the user fires guards instead of the `wait` transition. Therefore, the outputs za and gv of f`_symb` only change when the user fires guards. Similarly the equation zi `= if wait then (zall fby` zi`) else` zi' ensures that the initial zone does not change when the user chooses the `wait` transition.

Alternatively, is is possible to only execute `ztrig` and f`_symb` at the initial instant and when the user fires guards using a `present` construct. The outputs of f`_symb` are then stored in memories. We can then remove `wait` from the inputs of f`_symb` and directly return zi', the initial zone obtained by the translation of the function body.

```
init za = zall and init zi = zall and init gv = []
and present (true → not wait) → local zg in
      do zg = ztrig(sv, zcp, gvp)
      and za, zi, gv = f_symb(tv, sv, zg, i) done
```

Illustration: the quasi-periodic architecture

Figure 6.13 illustrates the complete compilation scheme on the example of section 6.1: a two-node quasi-periodic architecture with instantaneous transmission. Figure 6.2 reproduced in figure 6.13, and that we used to motivate our approach, is a valid execution trace of the resulting discrete function `archi`.




```
let hybrid metro(t_min, t_max) = c where
  rec timer t init 0 reset c → 0
  and emit c when {t_min ≤ t}
  and always {t ≤ t_max}


(** Compiling metro **)
let node metro_symb(t, wait, c, zg, (t_min, t_max)) = c, zi, za, [zs] where
  rec zit = present (true fby false) → zreset(zg, t, 0)
            | c → zreset(zg, t, 0)
            else zg
  and zs = zmake({t ≥ t_min})
  and zb = zmake({t ≤ t_max})
  and za = zinterfold([zb])
  and zi = if wait then (zall fby zi) else zit

let node metro(wait, c, (t_min, t_max)) = c', bv, bw, zc where
  rec zg = ztrig([c], zcp, gvp)
  and c', zi, za, gv = metro_symb(1, wait, c, zg, (t_min, t_max))
  and zc, bv, bw = znext(wait, zi, za, gv)
  and zcp = zall fby zc
  and gvp = [] fby gv
```

```
let hybrid archi(t_min, t_max) = c1, c2 where
  rec c1 = metro(t_min, t_max)
  and c2 = metro(t_min, t_max)


(** Compiling archi **)
let node archi_symb((t1, t2), wait, (c1, c2), zg, (t_min, t_max)) = (c1', c2'), zi, za, gv1 @ gv2 where
  rec c1', zi1, za1, gv1 = metro_symb(t1, wait, c1, zg, (t_min, t_max))
  and c2', zi2, za2, gv2 = metro_symb(t2, wait, c2, zi1, (t_min, t_max))
  and za = zinterfold([za1; za2])
  and zi = if wait then (zall fby zi) else zi2

let node archi(wait, (c1, c2), (t_min, t_max)) = (c1', c2'), bv, bw, zc where
  rec zg = ztrig([c1; c2], zcp, gvp)
  and (c1', c2'), zi, za, gv = archi_symb((1, 2), wait, (c1, c2), zg, (t_min, t_max))
  and zc, bv, bw = znext(wait, zi, za, gv)
  and zcp = zall fby zc
  and gvp = [] fby gv
```

Figure 6.13: Compilation of functions `metro` and `archi`.

6.7 Extensions

In this section we discuss two possible extensions of ZSy to improve its expressiveness: valued signals and automata.

Valued signals

In ZSy, signals cannot carry values but it is relatively simple to add this feature to the language by reusing Zélus signals. The type of a signal α *signal* is parametrized by α, the type of its values. A pure signal, without any value, has type *unit signal*. An expression calculating a value to emit on a signal must be of kind D since emissions are discrete computations.

We keep the syntax of Zélus for valued signals:

(*emission*) `emit` s [`=` e]

(*reception*) `present` s`(`v`)` [`on` $P(v)$] $\rightarrow$ e

with the particular case `present` s`()` $\rightarrow$ e for pure signals. The optional condition [`on` $P(v)$] allows to filter the value v received on a signal with a boolean predicate P directly in the branches of the `present` handler.

Automata

A major restriction of ZSy is the absence of state in continuous functions. Conditionals like `if` e_0 `then` e_1 `else` e_2 can be added as an external operator of arity 3, but in that case, the three expressions e_0, e_1, and e_2, and the equations produced by their translations, are computed at every step. It is, however, possible to extend ZSy with hierarchical automata following the compilation technique introduced in [BBCP11b].

Consider the following example:

```
let hybrid auto() = o where
  rec automaton
      | S1 → do o = 1
              and timer t1 init 0 reset c1 → 0
              and emit c1 when {t1 > 3}
              and always {t1 ≤ 5}
          until c1 then S2
      | S2 → do o = 2
              and timer t2 init 0 reset c2 → 0
              and emit c2 when {t2 > 2}
              and always {t2 ≤ 7}
          until c2 then S1
```

val auto: unit $\xrightarrow{\text{C}}$ *int*

An automaton in continuous contexts is translated into a similar discrete automaton where the signals triggering transitions between states are replaced by boolean conditions.

The easiest solution is to duplicate all equations introduced by timers, guards, and invariants during the translation such that, if a variable is defined in one state of the automaton, the same variable returns a dummy value in all other states.

```
let hybrid auto_symb((t1, t2), wait, (c1, c2), zg) = o, zi, za, [zs1; zs2] where
  rec automaton
      | S1 → do o = 1
              and zi1 = present (true fby false) → zreset(zg, t1, 0)
                        | c1 → reset(zg, t1, 0)
                        else zg
              and zs1 = zmake({t1 > 3})
              and za1 = zmake({t1 ≤ 5})
              and zi2 = zall and zs2 = zempty and za2 = zall
          until c1 then S2
      | S2 → do o = 2
              and zi2 = present (true fby false) → zreset(zg, t2, 0)
                        | c2 → reset(zg, t2, 0)
                        else zg
              and zs2 = zmake({t2 > 2})
              and za2 = zmake({t2 ≤ 7})
              and zi1 = zall and zs1 = zempty and za1 = zall
          until c2 then S1
  and za = zinterfold([za1; za2])
  and zi = if wait then (zall fby zi) else zinterfold([zi1; zi2])
```

val auto_symb: (int × int) × bool × (bool × bool) × zone $\xrightarrow{\mathrm{D}}$ *int × zone × zone × zone list*

Each state of the automaton generates a possible initial zone zi. This variable takes the dummy value `zall` in all other states. We gather all these zones into a vector and the global initial zone is the intersection of its elements. The activation zone of a guard defined in one state is empty in all other states (the guard cannot be enabled). An invariant defined in one state becomes `zall` in all other states.

Following [BBCP11b], it is also possible to minimize memory allocations by reusing variables across multiple states. For instance, the pairs of variables (`zi1`, `zi2`) and (`za1`, `za2`) could be merged since they are used in exclusive states. However, we still need to gather all timer identifiers, guard signals, and guard activation zones for interaction with the user.

A complete example: the train gate

We motivated our approach with the example of a quasi-periodic architecture which is both a typical example of a nondeterministic timed system and the main focus of this thesis. But our proposal is more general and ZSy, once extended with valued signals and automata, permits the expression of more complex models. For instance the train gate [BDL06, §4] is a classic example of a system mixing nondeterministic continuous components, the train controllers; with discrete components, the gate controller.

The gate controls access to a bridge for several trains. The bridge can be crossed by only one train at a time and the gate ensures that a train never engages if another is still crossing the bridge. Timing constraints are used to model uncertainty on the speeds of the trains.

Train controller When approaching the bridge, a train waits 10 time units to receive a `stop` signal from the gate controller. If, after this delay, nothing is received, the train starts crossing the bridge. On the other hand, if a signal `stop` is received, the train stops and waits until the gate sends a `go` signal when the bridge is free.

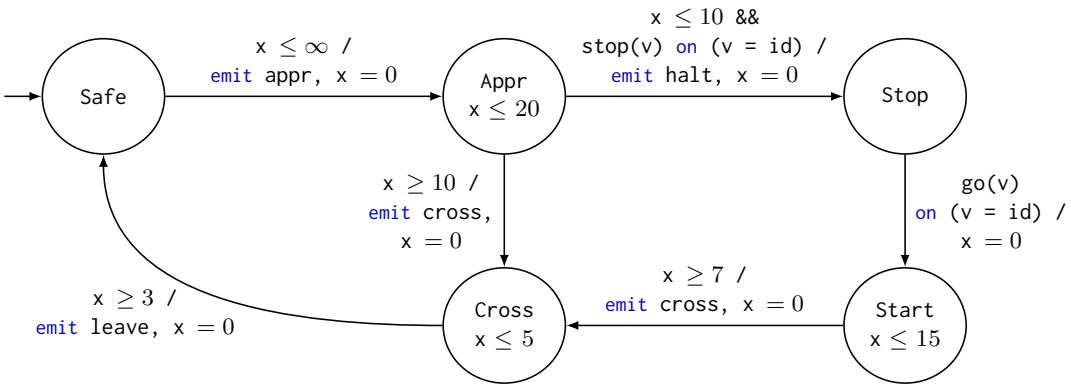


Figure 6.14: [BDL06, Figure 9] Nondeterministic train controller where `x` is a timer reset at each transition and `id` is the train identifier.

The train controller is the five-state automaton illustrated in figure 6.14. Its inputs are the two signals emitted by the gate: `stop` and `go`. The train sends a signal `appr` when approaching the bridge and a signal `leave` when leaving the bridge. A train is characterized by a unique identifier `id`.

```
let hybrid train(id, stop, go) = appr, leave where
  rec timer x init 0
      reset go(v) on (v = id) | halt() | appr() | cross() | leave() → 0
  and automaton
      | Safe  → do emit appr when {x ≤ ∞}
          until appr() then Appr
      | Appr  → do emit halt when {x ≤ 10} && (stop(v) on (v = id))
                and emit cross when {x ≥ 10}
                and always {x ≤ 20}
          until halt() then Stop
           else cross() then Cross
      | Stop  → do
          until go(v) on (v = id) then Start
      | Start → do emit cross when {x ≥ 7}
                and always {x ≤ 15}
          until cross() then Cross
      | Cross → do emit leave when {x ≥ 3}
                and always {x ≤ 5}
          until leave() then Safe
```

val train: ident × ident signal × ident signal $\xrightarrow{C}$ *unit signal × unit signal*

In the initial state `Safe`, a train can approach the bridge, sending a signal `appr` with its `id`, at any time. The constraint for leaving this state is thus `{x ≤ ∞}`.

An approaching train takes at most 20 time units to reach the bridge. This is expressed in the invariant `always {x ≤ 20}` in the `Appr` state. If the train receives a message `stop` that corresponds to its `id` within the first 10 time units (`stop(v) on (v = id) && {x ≤ 10}`) it can be stopped before the bridge. The train then enters the `Stop` state. Otherwise, after 10 time units, the train cannot be stopped and starts crossing, that is, enters the `Cross` state.

At $x = 10$ both transitions are possible and can be activated simultaneously. As in Zélus, transitions are treated sequentially in the `until` handler and in our model `halt` takes priority.

In the `Stop` state the train waits for a signal `go` corresponding to its `id` and then enters the `Start` state to resume its crossing. The train takes between 7 and 15 time units to restart, then it begins crossing the bridge. Finally, the crossing takes between 3 and 5 time units and a signal `leave` is emitted when the train leaves the bridge.

The timer `x` is reset whenever the controller enters a new state.

Gate controller The gate controller is a classic discrete controller. It maintains a queue of approaching trains.

```
let node queue(push, pop) = q where
  rec init q = empty()
  and present
      | push(v) & pop(_) → do q = enqueue(dequeue(last q), v) done
      | push(v) → do q = enqueue(last q, v) done
      | pop(_) → do q = dequeue(last q) done
```

val queue: α *signal* $\times$ β *signal* $\xrightarrow{\mathrm{D}}$ α *queue*

The queue is updated whenever a new train sends a signal `appr` or leaves the bridge and a two-state automaton controls the emission of the `stop` and `go` signals.

```
let node gate(appr, leave) = stop, go where
  rec q = queue(appr, leave)
  and automaton
      | Free → do
          unless appr(v) on (size(q) = 1) then Occ
            else (size(q) > 0) then do emit go = front(q) in Occ
      | Occ  → do
          unless leave() & appr(v) then do emit stop = v in Free
            else leave() then Free
            else appr(v) then do emit stop = v in Occ
```

val gate: ident signal $\times$ *unit signal* $\xrightarrow{\mathrm{D}}$ *ident signal* $\times$ *ident signal*

The controller starts in the `Free` state. If a train approaches while the queue is empty (in which case, `size(q) = 1` since the train is added to the queue) it starts crossing and the controller enters the `Occ` state. On the other hand, if no train is approaching but the waiting queue is not empty (`size(q) > 0`), the controller sends a signal `go` to the first train in the queue and enters the `Occ` state.

In the `Occ` state, the controller waits for a train to leave the bridge. Meanwhile, it sends `stop` signals to all approaching trains. If the two events occur simultaneously we combine the two behaviors. This discrete controller is activated whenever a signal `appr` or `leave` is emitted by one of the trains.

Complete model The components can now all be plugged together, combining the output signals of the train controllers to form two global signals `appr` and `leave`. For instance, the complete code of a gate controlling two trains is:

```
let hybrid train_gate() = () where
  rec appr1, leave1 = train(1, stop, go)
  and appr2, leave2 = train(2, stop, go)
  and present leave1() | leave2() → do emit leave done
  and present
      | appr1() → do emit appr = 1 done
      | appr2() → do emit appr = 2 done
  and present appr(_) | leave() → do stop, go = gate(appr, leave) done
```

val train_gate: unit $\xrightarrow{\mathrm{C}}$ *unit*

Note that in this example, there is a mutual dependency between the nondeterministic controllers of the trains and the discrete gate controller.

6.8 Conclusion

In this chapter, we focused on a particular class of hybrid system: timed discrete-event systems where continuous dynamics is limited to timers but may involve nondeterministic constraints.

We presented an original symbolic simulation scheme where simulation traces are a succession of discrete steps, each step corresponding to a set of timer values and a set of enabled guards. The choice among the set of enabled guards is delegated to the user who can also choose the `wait` transition to let time elapse until the next change in the set of enabled guards. Compared to classic simulators of timed systems like Uppaal, the user maintains a clear view of the succession of possibilities during execution.

Our simulator is based on DBMs, a classic data structure for representing states of timed systems. This choice imposes restrictions on the constraints we are able to express in the language; namely integer bounds on the difference between two timers, but DBMs form a closed set with respect to mode changes, resets, and time elapsing, and allow a symbolic representation of the state of the system without overapproximation. Furthermore, classic operations such as intersection, time elapsing, and reset can be readily computed using DBMs, and we showed how to compute the successive zones of our simulation scheme using elementary operations.

We focused on a kernel of Zélus where the only dynamics are expressed with timers extended with nondeterministic constraints. We showed how to adapt the typing and source-to-source compilation of Zélus to produce discrete code for symbolic simulation. Static typing allowed us to discriminate continuous from discrete computations and to check the restrictions imposed in our subset of Zélus. We used a source-to-source compilation to turn a model mixing discrete and continuous components into a purely discrete function by adding inputs for user choices, and outputs for the resulting zone and the set of enabled transitions. Our kernel language is already sufficient to express simple specifications like those of the quasi-periodic architecture of chapter 3 but is still very limited. We discussed how to extend it with the valued signals and hierarchical automata of Zélus to treat more complex examples.

The main limitation of our proposal is the restrictions on the expressible constraints. The most obvious direction to continue this work is thus to seek less constraining set representations. For instance, octagons [Min06] allow the expression of constraints on both differences and sums of timers.

Another interesting idea is to forsake the idea of capturing all equivalent traces and focus instead on under-approximations. A symbolic trace still captures a set of equivalent traces,

but not all of them. In this case, it is possible to use even more complex representations that are not necessary closed under discrete transitions, like convex polyhedra. This idea of symbolic simulation using under-approximation has been used to improve test coverage in the larger context of hybrid systems [AKRS08, KAI+09].

Finally since we focus on simulation, not verification, decidability of the model is not an issue and an interesting extension would be to add suspensions as in stopwatch automata [CL00]. This would allow the passage of time to be interrupted for a timer defined in one state of an automaton when this state is inactive.

Chapter 7

Conclusion

7.1 Summary

In this thesis we focused on quasi-periodic systems, that is, embedded controllers implemented as a set of unsynchronized periodic processes. To study these systems, we used an approach based on synchronous languages, that is, languages with a precise semantics centered around the notions of time and concurrency. This approach allowed us to clarify and simplify existing work on the treatment of quasi-periodic systems and propose new developments along three themes: verification, implementation, and simulation.

Synchronous modeling

In chapter 3 we formalized the notion of a quasi-periodic architecture. Processes activate quasi-periodically, that is, periodically with bounded jitter, and rely on unsynchronized blackboard communication with bounded transmission delays. This kind of communication introduces multiple sampling artifacts: losses and duplications of data and unintended signal combinations.

Then we presented a synchronous model of quasi-periodic architectures written in the discrete subset of Zélus. We applied classic techniques for modeling asynchronous systems in a synchronous framework, and realized the link with the real-time architecture specifications using input signals; namely for the clocks of processes and the arrival of transmitted messages. Using the continuous part of Zélus, we were then able to propose a complete model where these signals are produced by continuous components. This gave a complete executable model that can be used for testing and simulating quasi-periodic systems.

Although we chose to implement our model in Zélus, we showed that our approach can be adapted to other modeling languages that allow the mixing of discrete-time and continuous-time dynamics, namely Ptolemy and Simulink.

Verification: The quasi-synchronous abstraction

In chapter 4 we focused on the quasi-periodic abstraction, a discrete abstraction proposed by Paul Caspi for model-checking quasi-periodic systems. Caspi's idea was that if processes execute 'almost periodically'—with limited jitter on the activation periods—and if the transmission delay is significantly shorter than the activation period, then the behavior of a quasi-periodic

systems can be captured in a discrete model with two simple conditions: logical step account for transmission delay and a process never activates twice between two activations of another. Starting from the synchronous model of chapter 3 we showed how this abstraction allows a simpler discrete model where possible interleavings of input signals are constrained by a simple predicate and communication channels are replaced by memories.

Despite interest in this abstraction no one has yet posed the question: *Is this abstraction sound with respect to the underlying real-time system?* Based on Lamport's *happened before* relation, we introduced the notion of *unitary discretization* to precisely characterize real-time traces for which transmissions can be safely abstracted as unit delays. We showed that the modeling of transmissions as unit delays links the causalities induced by the communications in real-time traces with the causalities expressible in the discrete model in terms of precedence. By gathering all the constraints on the unitary discretization into a weighted graph, we were able to rephrase the problem of the existence of a unitary discretization in terms of cycles in the corresponding graph. We proved that systems of more than two nodes are in general not unitary discretizable, that is, there are valid real-time traces for which a unitary discretization does not exist. This problem originates from the modeling of transmission delays and can occur in a completely asynchronous model. These results are thus not limited to quasi-periodic systems.

For quasi-periodic systems with known bounds on activation periods and communication delays, we showed that problematic cycles in constraint graphs can be forbidden by constraining both the static communication graph of the application and the real-time characteristics of the architecture. This gives necessary and sufficient conditions to recover soundness. Building on these results, we gave the exact conditions under which the quasi-synchronous abstraction can be correctly applied. We illustrated the quasi-synchronous approach with a classic property of quasi-periodic architectures: bounding the maximum number of successive message losses or duplications.

Finally, we generalized these results to multirate systems where each process is characterized by its own activation period. The quasi-synchronous abstraction then becomes n/m-quasi-synchrony where logical step still account for transmission delay and a process never activates more than n times between m activations of another.

Implementation: Loosely time-triggered architectures

As shown in chapters 3 and 4, quasi-periodic systems are subject to sampling artifacts. These artifacts may be acceptable for robust controllers that can compensate for eventual losses or duplications of data, but they are clearly harmful for discrete logic like state machines. Loosely time-triggered architectures are protocols, proposed as a lightweight alternative to clock synchronization, to ensure the correct execution of an application running on a quasi-synchronous architecture.

We first precisely defined the class of applications that can be implemented on quasi-periodic architectures using LTTA: the *synchronous applications*. By synchronous application, we mean a synchronous program that has been compiled into a composition of communicating Mealy machines. Each machine is executed on a quasi-periodic process and the semantics of the application is given by the sequence of values at each variable. Importantly, we assume that these machines communicate through unit delays, that is, the output of Mealy machines never instantaneously depends on their inputs. This restriction increases node independence and permits simpler protocols.

We showed how the discrete model of chapter 3 can be refined to incorporate the LTTA protocol in the modeling of nodes. Protocol controllers are also synchronous programs: they can be compiled together with application code. Our unified synchronous framework can be instantiated with any LTTA protocol. This gave both a clear framework to reason about the protocols and executable specifications that can be used for testing and simulation using the complete model of chapter 3.

We instantiated our framework with the two historical protocols: *back-pressure* based on acknowledging the receipt of messages and *time-based* which replace acknowledgments with a waiting mechanism. We showed that in our framework these protocols can be expressed as simple two-state automata. Additionally, for each of these protocols, we clarified the required assumptions, gave simplified correctness proofs of the semantics preservation, and gave their theoretical worst-case performance.

We proved that the time-based protocol requires broadcast communication. Based on this observation, we proposed, in the same framework, optimized protocols for systems using broadcast communication where *send* and *execute* phases are merged together. To avoid blocking communication, we extended the controllers with a simple timeout mechanism inspired by fault-detectors of distributed algorithms.

Finally, for comparison purposes, we instantiated our framework again with a basic protocol based on clock synchronization where nodes synchronize on messages sent by a *central master*. Using the executable model of chapter 3 extended with the protocol controllers we were able to simulate all these protocols in the same settings to compare their performance. We discussed and compared the advantages and disadvantages of the various approaches.

Our evaluation suggested that LTTA protocols are at least competitive for jittery architectures where the execution period of the discrete logic is relatively slow compared to the communication latency.

Simulation: Symbolic simulation

Simulations of the LTTA protocols are based on the model of chapter 3 where nondeterministic characteristics of the quasi-periodic architectures—transmission delays and quasi-periodic clocks—are simulated by choosing values at random.

In chapter 6 we focused on such timed systems whose specifications involve tolerances and nondeterminism. Inspired by techniques for model checking timed automata, we proposed a symbolic simulation scheme where a single simulation trace captures a set of possible executions. The dynamics of continuous components is limited to *timers* to measure time elapsing but also involves nondeterministic *guards* on the emission of events, and *invariants*. This is a limited subset of Zélus, but it suffices for the models analyzed in this thesis.

As usual, our synchronous programs advance in discrete time steps but now each step is characterized by a set of enabled guards and corresponds to a set of timer values: a *zone*. Our proposed simulation scheme is the following. At each step the user chooses between firing enabled guards or letting time elapse until the next change in the set of enabled guards (`wait` transitions). Compared to existing simulators for timed systems like Uppaal, our proposal lets the user control time elapsing via `wait` transitions. The user thus has a clear view of the successions of possible transitions.

To represent zones, we use difference-bound matrices (DBMs), a classic data structure for nondeterministic timed systems that can be efficiently represented and manipulated. Using

DBMs, we showed how to compute the succession of zones for our simulation scheme using a simple notion of distance between an initial zone and the activation zone of a guard.

Then we introduced ZSy, a single assignment kernel of Zélus where the only allowed continuous dynamics are timers extended with nondeterministic constructs: guards and invariants. We showed how to exploit the Zélus type system that statically discriminate between discrete and continuous computations. Our adaptations to this system also ensure that the concrete values of timers are never used in an equation and that constraints only involve difference bounds that can be represented with DBMs.

Inspired by the compilation of Zélus, we presented a modular source-to-source compilation scheme that translates the continuous components of ZSy into discrete functions for symbolic simulation. The resulting discrete function has two additional input for the choice of the user at each step (`wait` of firing guards); and three additional outputs for the current zone, the set of enabled guards, and weather or not the `wait` transition is enabled for the next step.

Our kernel language ZSy is quite limited but suffices to simulate quasi-periodic systems and the LTTA protocols studied in this thesis. We discussed possible extensions to improve its expressivity: valued signals and automata. With these extensions we showed that it is already possible to express typical timed models like the train gate controller example.

7.2 Open questions

Real-time requirements

In all our developments we focused on discrete synchronous applications and kept a clear separation between discrete and continuous time. However, for some applications *when* a value is computed is just as important as *what* value is computed. For instance, a controller that must react within $2\,s$ after a particular event, like a user pressing an emergency button. The impact of executing an application with such real-time requirements on a quasi-periodic architecture remains an open question.

The LTTA protocols presented in chapter 5 were designed to ensure the preservation of the synchronous/dataflow semantics of an embedded application. They do so at the cost of additional delays: a node *skips* until it can safely execute or send a value. We studied the slowdown introduced by this skipping mechanism, but by delaying node activations, the real-time behavior of some applications may change. The sequence of values within the program are preserved, but not necessarily their relation to external events and the dynamics of the environment.

Characterizing robust applications

Quasi-periodic architectures are subject to the sampling artifacts presented in section 3.2: message duplication and loss, and unintended signal combinations. These artifacts are in general harmful for discrete logic, but some applications are designed to resist their effects. For instance, a simple master-slave application where the slave must take control if the master crashes. Following the principles of the timeout mechanism introduced in section 5.5, if the slave stops receiving messages from the master for too long it can be sure that the master crashed and can safely take control.

Some applications can thus be executed on quasi-periodic architectures without requiring additional control logic like the LTTA protocols. Characterizing the robustness of such

applications with respect to the targeted architecture and modeling them in a synchronous framework is another promising perspective.

Zélus in a proof assistant

Zélus gives an ideal framework for reasoning about models that mix discrete-time and continuous-time dynamics. Quasi-periodic systems are but one example. Compared to other modeling tools, one of the main advantages is that in Zélus discrete components can be designed in a language that is typically used to program embedded applications. Hence, it is conceivable to produce embedded code directly from the discrete components of a Zélus model thus benefiting from the well-developed compilation techniques for synchronous languages. The gap between the model used for simulation and the actual implementation is significantly reduced.

But, in our work, most of the proofs linking continuous and discrete time—like the soundness of the quasi-synchronous abstraction or the correctness proofs of the LTTA protocols—are still pen-and-paper results. Zélus allowed us to clarify the models and assumptions thus simplifying reasoning, but did not help for the proofs themselves and does not offer the extra confidence on the results that comes with mechanized proof. A long term perspective would be to integrate Zélus into a proof assistant to be able to formally reason and prove properties about hybrid models. This approach would give the advantages of proof assistants while keeping the close proximity between the formal model and the embedded code offered by Zélus.

Model checking

An alternative approach would be to extend the model checking tools dedicated to synchronous languages to handle continuous-time models of the environment. The symbolic simulation scheme presented in chapter 6 might be a first step in this direction for a restricted class of models. One could imagine extending this work to systematically explore all possible choices offered by the symbolic simulation to check safety properties of the model.

Of course, developing a complete model checker would require consequent work on state representation and exploration strategies. But efficient techniques for nondeterministic timed systems have already been developed, for instance in Uppaal, and could be reused in our setting. Compared to Uppaal, this approach would allow us to take advantage of the compilation techniques for synchronous languages to directly generate and execute code from a verified model.

Alternatively, we could take advantage of our source-to-source compilation scheme to verify properties of the resulting discrete model using dedicated verification tools like Kind2. This would require a theory to handle zones of timer values and a translation of discrete programs written in Zélus into the Lustre syntax of Kind2. With this approach, we could keep the benefits of the Zélus type system to analyze program mixing discrete and continuous dynamics and use a state-of-the-art model checker to verify their properties.

7.3 Concluding remark

This thesis focuses on quasi-periodic systems, but we believe that some of our contributions are of broader interest. The notion of unitary discretization and its characterization in terms of cycles in a trace graph holds for any asynchronous distributed system. The symbolic simulation

scheme and our language proposal ZSy can be used to simulate models that mix discrete logic and nondeterministic timed dynamics in more complex ways than does a quasi-periodic architecture.

Overall, our work illustrates the advantages offered by the synchronous approach and its extensions to reason about real-time distributed systems. This approach gives a formal framework to reason about systems involving time and concurrency, well-understood compilation techniques that gives a seamless path from the model to executable code, and dedicated verification and simulation techniques.

The source code presented throughout the thesis can be compiled and executed in version 1.2.3 of Zélus. This code can be downloaded from:

`http://guillaume.baudart.eu/thesis`.

Bibliography

[ACH+95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[ADGFT04] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Symposium on Principles of Distributed Computing (PODC)*, pages 328–337, Canada, July 2004.

[ADLS94] Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41(1):122–152, 1994.

[AKRS08] Rajeev Alur, Aditya Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In *International Conference on Embedded Software (EMSOFT)*, pages 89–98, USA, October 2008.

[Alu11] Rajeev Alur. Formal verification of hybrid systems. In *International Conference on Embedded Software (EMSOFT)*, pages 273–278, Taiwan, October 2011.

[BBB15] Guillaume Baudart, Albert Benveniste, and Timothy Bourke. Loosely Time-Triggered Architectures: Improvements and comparisons. In *International Conference on Embedded Software (EMSOFT)*, pages 85–94, The Netherlands, October 2015. Best paper nominee.

[BBB16] Guillaume Baudart, Albert Benveniste, and Timothy Bourke. Loosely Time-Triggered Architectures: Improvements and comparison. *Transactions on Embedded Computing Systems*, 15(4):71:1–71:26, August 2016.

[BBBC14] Guillaume Baudart, Albert Benveniste, Anne Bouillard, and Paul Caspi. A unifying view of Loosely Time-Triggered Architectures. Technical Report RR-8494, INRIA, March 2014. Corrected version of [BBC10].

[BBC10] Albert Benveniste, Anne Bouillard, and Paul Caspi. A unifying view of Loosely Time-Triggered Architectures. In *International Conference on Embedded Software (EMSOFT)*, pages 189–198, USA, October 2010.

[BBCP11a] Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. Divide and recycle: types and compilation for a hybrid synchronous language. In *Conference on Languages, Compilers, and tools for embedded systems (LCTES)*, pages 61–70, USA, April 2011.

[BBCP11b] Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. A hybrid synchronous language with hierarchical automata: Static typing and translation to synchronous code. In *International Conference on Embedded Software (EMSOFT)*, Taiwan, October 2011.

[BBCP12] Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. Non-standard semantics of hybrid systems modelers. *Journal of Computer and System Sciences*, 78:877–910, May 2012.

[BBP16] Guillaume Baudart, Timothy Bourke, and Marc Pouzet. Soundness of the quasi-synchronous abstraction. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 9–16, USA, October 2016.

[BCDN+07] Albert Benveniste, Paul Caspi, Marco Di Natale, Claudio Pinello, Alberto Sangiovanni-Vincentelli, and Stavros Tripakis. Loosely Time-Triggered Architectures based on Communication-by-Sampling. In *International Conference on Embedded Software (EMSOFT)*, pages 231–239, Austria, September 2007.

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

[BCLG+02] Albert Benveniste, Paul Caspi, Paul Le Guernic, Hervé Marchand, Jean-Pierre Talpin, and Stavros Tripakis. A protocol for Loosely Time-Triggered Architectures. In *International Conference on Embedded Software (EMSOFT)*, pages 252–265, France, October 2002.

[BCP+15] Timothy Bourke, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. A synchronous-based code generator for explicit hybrid systems languages. In *International Conference on Compiler Construction (CC)*, pages 69–88, UK, April 2015.

[BDL06] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A tutorial on Uppaal 4.0*, 2006.

[Ben02] Johan Bengtsson. *Clocks, DBMs and states in timed systems*. PhD thesis, Uppsala University, 2002.

[Ber89] Gérard Berry. Real time programming: special purpose or general purpose languages. Technical Report RR-1065, INRIA, 1989.

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

[BGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the Signal language and its semantics. *Science of Computer Programming*, 16(2):103 – 149, 1991.

[BMY+14] Siddhartha Bhattacharyya, Steven P. Miller, Junxing Yang, Scott Smolka, Baoluo Meng, Christoph Sticksel, and Cesare Tinelli. Verification of quasi-synchronous systems with Uppaal. In *Digital Avionics Systems Conference (DASC)*, pages 8A4–1, USA, October 2014.

[Bos07] Boston Scientific. Pacemaker system specification. `http://www.cas.mcmaster.ca/sqrl/pacemaker_spec.htm`, January 2007.

[BP13] Timothy Bourke and Marc Pouzet. Zélus: A synchronous language with ODEs. In *International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 113–118, USA, April 2013.

[BS01] Gérard Berry and Ellen Sentovich. Multiclock Esterel. In *Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 110–125, UK, September 2001.

[BS09] Timothy Bourke and Arcot Sowmya. Delays in Esterel. In *International Workshop on Synchronous Programming (SYNCHRON)*, Germany, November 2009.

[Car06] Luca P. Carloni. The role of back-pressure in implementing latency-insensitive systems. *Elecronic Notes in Theoretical Computer Science*, 146(2):61–80, 2006.

[Cas00] Paul Caspi. The quasi-synchronous approach to distributed control systems. Technical Report CMA/009931, VERIMAG, Crysis Project, May 2000. *The Cooking Book*.

[Cas01] Paul Caspi. Embedded control: From asynchrony to synchrony and back. In *International Conference on Embedded Software (EMSOFT)*, pages 80–96, USA, October 2001.

[CB08] Paul Caspi and Albert Benveniste. Time-robust discrete control over networked Loosely Time-Triggered Architectures. In *Conference on Decision and Control (CDC)*, pages 3595–3600, Mexico, December 2008.

[CCM+03] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Conference on Languages, Compilers, and tools for embedded systems (LCTES)*, pages 153–162, USA, June 2003.

[CDE+06] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. In *Symposium on Principles of Programming Languages (POPL)*, pages 180–193, USA, January 2006.

[CDE+12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 261–264, USA, October 2012.

[CF99] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[CK07] Jordi Cortadella and Michael Kishinevsky. Synchronous elastic circuits with early evaluation and token counterflow. In *Design Automation Conference (DAC)*, pages 416–419, USA, June 2007.

[CKLS06] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, and Christos P. Sotiriou. Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):1904–1921, 2006.

[CL00] Franck Cassez and Kim G. Larsen. The impressive power of stopwatches. In *International Conference on Concurrency Theory (CONCUR)*, pages 138–152, USA, August 2000.

[CMST16] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The Kind 2 model checker. In *International Conference on Computer Aided Verification (CAV)*, pages 510–517, Canada, July 2016.

[CMSV01] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, 2001.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Symposium on Principles of Programming Languages (POPL)*, pages 178–188, Germany, January 1987.

[Cri96] Flaviu Cristian. Synchronous and asynchronous group communication. *Communications of the ACM*, 1996.

[CSV02] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. Coping with latency in SoC design. *IEEE Micro*, 22(5):24–35, 2002.

[Dil90] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *International Workshop on Automatic verification methods for finite state systems (AVMFSS)*, pages 197–212, France, June 1990.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[DSQ+15] Ankush Desai, Sanjit A. Seshia, Shaz Qadeer, David Broman, and John C. Eidson. Approximate synchrony: An abstraction for distributed almost-synchronous systems. In *International Conference on Computer Aided Verification (CAV)*, pages 429–448, USA, July 2015.

[FLGD+11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable verification of hybrid systems. In Shaz Qadeer Ganesh Gopalakrishnan, editor, *International Conference on Computer Aided Verification (CAV)*, USA, July 2011.

[Flo62] Robert W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[FM82] Michael J. Fischer and Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Symposium on Principles of Database Systems (PODS)*, pages 70–75, USA, March 1982.

[FSS05] Cristof Fetzer, Ulrich Schmid, and Martin Süßkraut. On the possibility of consensus in asynchronous systems with finite average response times. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 271–280, Canada, June 2005.

[Gar02] David Garriou. Symbolic simulation of synchronous programs. *Elecronic Notes in Theoretical Computer Science*, 65(5):11–18, 2002.

[GG03a] Abdoulaye Gamatié and Thierry Gautier. The Signal approach to the design of system architectures. In *International Conference on the Engineering of Computer-Based Systems (ECBS)*, pages 80–88, USA, April 2003.

[GG03b] Abdoulaye Gamatié and Thierry Gautier. Synchronous modeling of avionics applications using the Signal language. In *Real Time Technology and Applications Symposium (RTAS)*, pages 144–151, Canada, May 2003.

[GGBM91] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[Gir05] Alain Girault. A survey of automatic distribution method for synchronous programs. In *International Workshop on Synchronous Languages, Applications and Programs (SLAP)*, UK, April 2005.

[Hal93] Nicolas Halbwachs. Delay analysis in synchronous programs. In *International Conference on Computer Aided Verification (CAV)*, pages 333–346, Greece, June 1993.

[HB02] Nicolas Halbwachs and Siwar Baghdadi. Synchronous modelling of asynchronous systems. In *International Conference on Embedded Software (EMSOFT)*, pages 240–251, France, October 2002.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre. *IEEE Transactions on Software Engineering*, September 1992.

[HLR93] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *International Conference on Algebraic Methodology and Software Technology (AMAST)*, pages 83–96, The Netherlands, June 1993.

[HM06] Nicolas Halbwachs and Louis Mandel. Simulation and verification of aysnchronous systems by means of a synchronous model. In *International Conference on Application of Concurrency to System Design (ACSD)*, pages 3–14, Finland, June 2006.

[HPR94] Nicolas Halbwachs, Yann-Eric Proy, and Pascal Raymond. Verification of linear hybrid systems by means of convex approximations. In *International Static Analysis Symposium (SAS)*, pages 223–237, Belgium, September 1994.

[Huy73] Christiaan Huyghens. *Horologium Oscillatorium: sive de motu pendulorum ad horologia aptato demostrationes geometricae.* 1673. The Pendulum Clock: or geometrical demonstrations concerning the motion of pendula as applied to clocks.

[JHR08] Erwan Jahier, Nicolas Halbwachs, and Pascal Raymond. Synchronous modeling and validation of schedulers dealing with shared resources. Technical Report 2008-10, Verimag, 2008.

[JMO93] Martin Jourdan, Florence Maraninchi, and Alfredo Olivero. Verifying quantitative real-time properties of synchronous programs. In *International Conference on Computer Aided Verification (CAV)*, pages 347–358, Greece, June 1993.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In *World Computer Congress (IFIP)*, pages 471–475, Sweden, August 1974.

[KAI+09] Aditya Kanade, Rajeev Alur, Franjo Ivancic, S. Ramesh, Sriram Sankaranarayanan, and K. C. Shashidhar. Generating and analyzing symbolic traces of Simulink/Stateflow models. In *International Conference on Computer Aided Verification (CAV)*, pages 430–445, France, June 2009.

[KB03] Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[Kop11] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications.* Springer-Verlag, 2011.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[LEWM05] Kang Lee, John C. Eidson, Hans Weibel, and Dirk Mohl. IEEE 1588-standard for a precision clock synchronization protocol for networked measurement and control systems. Technical report, IEEE, 2005.

[LGS15] Wenchao Li, Léonard Gérard, and Natarajan Shankar. Design and verification of multi-rate distributed systems. In *International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 20–29, USA, September 2015.

[LS02] George Logothetis and Klaus Schneider. Extending synchronous languages for generating abstract real-time models. In *Design, Automation, and Test in Europe (DATE)*, page 795, France, March 2002.

[LS14] Robin Larrieu and Natarajan Shankar. A framework for high-assurance quasi-synchronous systems. In *International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 72 – 83, Switzerland, October 2014.

[Lyn96] Nancy A Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[Mar92] Florence Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *International Conference on Concurrency Theory (CONCUR)*, pages 550–564, USA, August 1992.

[MAS06] Gabor Madl, Sherif Abdelwahed, and Douglas C. Schmidt. Verifying distributed real-time properties of embedded systems via graph transformations and model checking. *Real-Time Systems*, 33(1-3):77–100, 2006.

[Mat86] Frederic G. Mather. Primitive clocks. *Popular Science Monthly*, 29:180–192, 1886.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[Mau96] Christophe Mauras. Symbolic simulation of interpreted automata. In *International Workshop on Synchronous Programming (SYNCHRON)*, Germany, December 1996.

[MBT+15] Steven P. Miller, Sidhartha Bhattacharyya, Cesare Tinelli, Scott Smolka, Christoph Sticksel, Baoluo Meng, and Junxing Yang. Formal verification of quasi-synchronous systems. Technical report, DTIC Document, 2015.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.

[Mil06] David L. Mills. *Computer Network Time Synchronization, The Network Time Protocol*. Taylor & Francis Group, 2006.

[Min06] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[MPP11] Louis Mandel, Florence Plateau, and Marc Pouzet. Static scheduling of latency insensitive designs with Lucy-n. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 171–175, USA, October 2011.

[MR01] Florence Maraninchi and Yann Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1–3):61–92, 2001.

[Pla10] Florence Plateau. *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée*. PhD thesis, Université Paris Sud, Paris XI, 2010.

[Pon91] Stephen Ponzio. Consensus in the presence of timing uncertainty: Omission and byzantine failures. In *Symposium on Principles of Distributed Computing (PODC)*, pages 125–138, Canada, August 1991.

[Pou06] Marc Pouzet. *Lucid Synchrone, Tutorial and Reference Manual*, 2006.

[PS92] Stephen Ponzio and Ray Strong. Semisynchrony and real time. In *Workshop on Distributed Algorithms (WDAG)*, Israel, November 1992.

[Pto14] Claudius Ptolemaeus. *System Design, Modeling, and Simulation: Using Ptolemy II*, 2014.

[Ray96] Pascal Raymond. Recognizing regular expressions by means of dataflow networks. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 336–347, Germany, July 1996.

[RJ13] Pascal Raymond and Erwan Jahier. *Lutin Reference manual Version Trilby-1.54*, 2013.

[Roy59] Bernard Roy. Transitivité et connexité. *Comptes Rendus Hebdomadaires des Séances de l'Academie des Sciences*, 249(2):216–218, 1959.

[RRJ08a] Pascal Raymond, Yvan Roux, and Erwan Jahier. Lutin: A language for specifying and executing reactive scenarios. *EURASIP Journal of Embedded Systems*, 2008.

[RRJ08b] Pascal Raymond, Yvan Roux, and Erwan Jahier. Specifying and executing reactive scenarios with Lutin. *Elecronic Notes in Theoretical Computer Science*, 203(4):19–34, 2008.

[RS96] Michel Raynal and Mukesh Singhal. Logical time: Capturing causality in distributed systems. *IEEE Computer*, 29(2):49–56, 1996.

[RS11] Peter Robinson and Ulrich Schmid. The asynchronous bounded-cycle model. *Theoretical Computer Science*, 412(1):5580–5601, 2011.

[SG12] Gideon Smeding and Gregor Goessler. A correlation preserving performance analysis for stream processing systems. In *International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 11–20, USA, July 2012.

[SSC+04] Norman Scaife, Christos Sofronis, Paul Caspi, Stavros Tripakis, and Florence Maraninchi. Defining and translating a safe subset of Simulink/Stateflow into Lustre. In *International Conference on Embedded Software (EMSOFT)*, pages 259–268, Italy, September 2004.

[TPB+08] Stavros Tripakis, Claudio Pinello, Albert Benveniste, Alberto Sangiovanni-Vincentelli, Paul Caspi, and Marco Di Natale. Implementing synchronous models on Loosely Time-Triggered Architectures. *IEEE Transactions on Computers*, 57(10):1300–1314, 2008.

[TSCC05] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time Simulink to Lustre. *Transactions on Embedded Computing Systems*, 4(4):779–818, 2005.

[VC02] Paulo Veríssimo and António Casimiro. The timely computing base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930, 2002.

[Vit84] Paul Vitányi. Distributed elections in an archimedean ring of processors. In *Symposium on the Theory of Computing (STOC)*, pages 542–547, USA, April 1984.

[WS09] Josef Widder and Ulrich Schmid. The theta-model: achieving synchrony without clocks. *Distributed Computing*, 22(1):29–47, 2009.

[Yov96] Sergio Yovine. Model checking timed automata. In *European Educational Forum, School on Embedded Systems*, pages 114–152, Netherlands, November 1996.

[Yov97] Sergio Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1):123–133, 1997.

List of Figures

Index

Résumé

Cette thèse traite de systèmes embarqués contrôlés par un ensemble de processus périodiques non synchronisés. Chaque processus est activé quasi-périodiquement, c'est-à-dire périodiquement avec une gigue bornée. Les délais de communication sont également bornés. De tels systèmes réactifs, appelés *quasi-périodiques*, apparaissent dès que l'on branche ensemble deux processus périodiques. Dans la littérature, ils sont parfois qualifiés de systèmes distribués temps-réels synchrones. Nous nous intéressons aux techniques de conception et d'analyse de ces systèmes qui n'imposent pas de synchronisation globale.

Les langages synchrones ont été introduits pour faciliter la conception des systèmes réactifs. Ils offrent un cadre privilégié pour programmer, analyser, et vérifier des systèmes quasi-périodiques. En s'appuyant sur une approche synchrone, les contributions de cette thèse s'organisent selon trois thématiques: vérification, implémentation, et simulation des systèmes quasi-périodiques.

Vérification: *L'abstraction quasi-synchrone* est une abstraction discrète proposée par Paul Caspi pour vérifier des propriétés de sûreté des systèmes quasi-périodiques. Nous démontrons que cette abstraction est en général incorrecte et nous donnons des conditions nécessaires et suffisantes sur le graphe de communication et les caractéristiques temps-réels de l'architecture pour assurer sa correction. Ces résultats sont ensuite généralisés aux systèmes multi-périodiques.

Implémentation: Les *LTTAs* sont des protocoles conçus pour assurer l'exécution correcte d'une application sur un système quasi-périodique. Nous proposons d'étudier les LTTA dans un cadre synchrone unifié qui englobe l'application et les contrôleurs introduits par les protocoles. Cette approche nous permet de simplifier les protocoles existants, de proposer des versions optimisées, et de donner de nouvelles preuves de correction. Nous présentons également dans le même cadre un protocole fondé sur une synchronisation d'horloge pour comparer les performances des deux approches.

Simulation: Un système quasi-périodique est un exemple de modèle faisant intervenir des caractéristiques temps-réels et des tolérances. Pour ce type de modèle non déterministe, nous proposons une *simulation symbolique*, inspirée des techniques de vérification des automates temporisés. Nous montrons comment compiler un modèle mêlant des composantes temps-réels non déterministes et des contrôleurs discrets en un programme discret qui manipule des ensembles de valeurs. Chaque trace du programme résultant capture un ensemble d'exécutions possibles du programme source.

Mots Clés

Systèmes embarqués, Systèmes distribués temps-réels synchrones, Langages synchrones, Abstraction quasi-synchrone, Architectures LTTA, Simulation symbolique.

Abstract

In this thesis we study embedded controllers implemented as sets of unsynchronized periodic processes. Each process activates quasi-periodically, that is, periodically with bounded jitter, and communicates with bounded transmission delays. Such reactive systems, termed *quasi-periodic*, exist as soon as two periodic processes are connected together. In the distributed systems literature they are also known as synchronous real-time models. We focus on techniques for the design and analysis of such systems without imposing a global clock synchronization.

Synchronous languages were introduced as domain specific languages for the design of reactive systems. They offer an ideal framework to program, analyze, and verify quasi-periodic systems. Based on a synchronous approach, this thesis makes contributions to the treatment of quasi-periodic systems along three themes: verification, implementation, and simulation.

Verification: The *quasi-synchronous abstraction* is a discrete abstraction proposed by Paul Caspi for model checking safety properties of quasi-periodic systems. We show that this abstraction is not sound in general and give necessary and sufficient conditions on both the static communication graph of the application and the real-time characteristics of the architecture to recover soundness. We then generalize these results to multirate systems.

Implementation: *Loosely time-triggered architectures* are protocols designed to ensure the correct execution of an application running on a quasi-periodic system. We propose a unified framework that encompasses both the application and the protocol controllers. This framework allows us to simplify existing protocols, propose optimized versions, and give new correctness proofs. We instantiate our framework with a protocol based on clock synchronization to compare the performance of the two approaches.

Simulation: Quasi-periodic systems are but one example of timed systems involving real-time characteristics and tolerances. For such nondeterministic models, we propose a *symbolic simulation* scheme inspired by model checking techniques for timed automata. We show how to compile a model mixing nondeterministic continuous-time and discrete-time dynamics into a discrete program manipulating sets of possible values. Each trace of the resulting program captures a set of possible executions of the source program.

Keywords

Embedded systems, Synchronous real-time distributed systems, Synchronous languages, Quasi-synchronous abstraction, LTTA architectures, Symbolic simulation.