Programming adaptative real-time systems

Frédéric Fort

École doctorale Mathématiques, sciences du numérique et de leurs interactions (MADIS-631)

Programmation de systèmes temps réels adaptatifs
Programing adaptative real-time systems

Thèse
Informatique

Frédéric Fort
Thèse préparée et soutenue publiquement par Frédéric Fort le 04/10/2022 pour obtenir le grade de Docteur en Informatique

4 Octobre 2022

Direction de Thèse

Directeur
Giuseppe Lipari
Professeur, Université de Lille

Encadrant
Julien Forget
Maître de conférences, Université de Lille

Jury

Rapporteur, Président du Jury
Jean-Pierre Talpin
Directeur de recherche, INRIA

Rapporteur
Robert De Simone
Directeur de recherche, INRIA

Examinateur
Timothy Bourke
Chargé de recherche, INRIA

Examinateur
Claire Pagetti
Directrice d'études, ONERA

Abstract

A *real-time system* is a system whose correctness depends not only on the correctness of the values it produces, but also on the time when it produces those values. The rate at which it must produce values is defined by the environment it operates in.

When programming such a system, it is important that the programming language allows to reason about the constraints introduced by this context. Synchronous languages [14] are well-adapted to the programming of critical real-time systems thanks to their clean formal semantics and to their formally defined compilation process. In this work, we will present extensions to the synchronous language Prelude [67] to tackle two issues: Programming multicore systems predictably and handling system reconfiguration during execution.

Multicore hardware platforms have the potential to increase the performance of real-time systems. However, their architecture, especially the shared central memory, is prone to hard-to-predict delays, outweighing the potential benefits. To address this issue, models such as PREM [71] and AER [32] have been proposed. Our first contribution aims at producing AER-compliant multicore C code from a high-level Prelude program. This shifts the responsibility of low-level implementation concerns related to task communications onto the compiler, saving tedious and error-prone development efforts.

A *multi-mode* real-time system must respect different functional requirements during its execution. A *mode of execution* represents a possible system configurations, for an aircraft control system these may include take-off, cruise and landing. *Mode change protocols* define *transitions* to change safely from one mode to another. Our second contribution proposes *clock views* to decouple the rate of tasks and transitions. The resulting multi-mode support is both formally defined and generic enough to allows programmers to choose the kind of protocol they need for their application. A clock calculus based on refinement typing [39, 83] infers and checks the consistency of rates and views.

Résumé

Un *système temps réel* est un système dont la correction dépend non seulement de la correction des valeurs qu'il produit, mais aussi du temps quand il les produit. Le rythme de production de ces valeurs est défini par l'environnement dans lequel il opère.

La programmation d'un tel système nécessite un langage de programmation permettant de résonner sur les contraintes introduites par ce contexte. Les langages synchrones [14] sont bien adaptés pour la programmation de systèmes critiques temps réel de part leur sémantique et leur processus de compilation formellement définis. Dans ce travail, nous présentons des extensions au langage synchrone Prelude [67] pour aborder deux problèmes: La programmation de systèmes multicœur prédictibles et la reconfiguration du système durant l'exécution.

Les plateformes matérielles multicœurs possèdent un potentiel de performances accrues des systèmes temps réel. Cependant, leur architecture, en particulier la mémoire centrale partagée, est sujette à des délais difficile à prédire, contrebalançant les gains potentiels. Les modèles tel que PREM [71] et AER [32] remédient à cette limitation. Notre première contribution permet de produire du code multicœur en accord avec le modèle AER à partir d'un programme haut-niveau Prelude. Ceci permet de transférer la responsabilité des problèmes bas niveau liés aux communications inter-tâches, évitant des efforts de développement fastidieux et sujets aux erreurs.

Les exigences fonctionnelles d'un système temps-réel *multi-mode* évoluent durant son exécution. Un *mode d'exécution* représente une configuration possible du système, par exemple décollage, croisière ou atterrissage pour un système de contrôle d'avion. Les *protocoles de changement de mode* définissent des *transitions* afin de changer de mode en sécurité. Notre seconde contribution propose les *vues d'horloge* pour découpler le rythme d'exécution des tâches de celui des transitions. Le mécanisme multi-mode résultant est à la fois formellement défini et générique afin de permettre de choisir le type de protocole approprié à application. Un calcul d'horloge basé sur le typage par raffinement [39, 83] infère et vérifie la cohérence des rythmes et vues.

Acknowledgements

No work is ever achieved in isolation, our accomplishments are the result of our interactions with each other. This might be in particular the case for such a long-term work as a PhD thesis. Thus, I would like to thank some people who were there along the way.

First of all, I would like to thank my thesis supervisors, Giuseppe Lipari and Julien Forget. You welcomed me in then Émeraude team after I applied for an out-of-date thesis proposal and knew little of what I was getting into. Your continuous support and helpful feedback was crucial in my thesis work and in training a future researcher.

Next, I would like to thank my thesis jury. Thank you to Jean-Pierre Talpin and Robert De Simone for reviewing my thesis manuscript. Your thorough review greatly helped to shape this manuscrit into the form that is now visible. Thank you to Timothy Bourke and Claire Pagetti for participating in my thesis jury. Your feedback has been highly valuable.

Next, I would like to thank my former colleagues, in particular Fabien, Houssam, Pierre and Sandro with whom I shared an office at different times. I would also like to thank the members of the Association des Doctorant · e · s en Sciences de Lille, in particular Antonin who invested himself so much as a President.

Finally, I would like to thank Mara. You were there for me whenever I needed you to, and words cannot express my feelings for you. Time is the most real with you.

Contents

Part I

Introduction

Chapter 1

Problem Statement

In this Chapter, we will detail the problems this thesis tackles. Together with the next Chapter, you may understand it as an "extended abstract".

1.1 Real-Time Systems

A *real-time system* is a system whose correctness depends not only on the correctness of the values it produces, but also on the time when it produces those values. The rate at which it must produce values is defined by the environment it operates in. A typical example is an aircraft controller which must be able to react to external perturbations, such as a gust of wind, in a timely manner to guarantee the aircraft's safety.

This specific type of constraints results in an approach that is different from more "common" general-purpose computing. In particular, real-time computing should not be confused with high-performance computing. While a certain degree of computing speed is desired, once the system is able to meet its deadlines, further speed gains are superfluous. Indeed, gains in system predictability would be preferred over gains in speed once that point has been reached.

However, designers of real-time systems face the issue that modern computing platforms have an ever increasing complexity. This results in an ever increasing difference between the worst-, average- and best-case scenario. This is an issue for real-time systems as system designers must guarantee that the system respects its deadlines even in the worst case.

The ANR project Corteva [2] aimed at addressing this high variability in next-generation real-time systems by using sound and provably correct programming models. As part of this project, my thesis goal was to extend a programming language to enable the implementation of predictable real-time systems even on modern platforms.

1.2 Synchronous Languages for Real-time Systems

To address the complexity of designing real-time systems, it is important that the used programming language allows to reason about the specific constraints introduced by this context. However, most languages have no model of time, beyond the notion that earlier instructions execute before later ones. Thus, this thesis will focus on the family of synchronous languages which have been proven to be well-adapted to the programming of critical real-time systems.

Central to synchronous languages is the notion of *logical time*. Instead of considering time as something continuous, a synchronous language assumes that time is a sequence of discrete clearly-

separated *instants*. Computations are assumed to happen instantaneously. In essence, logical time assumes that the exact, "physical" time of computations does not matter as long as the frequency of instants is sufficient and computations never execute across instants.

To guarantee the temporal consistency of a program, synchronous languages employ a *clock calculus*. It is in essence a dedicated type system that tries to assign to each expression a *clock*. Clocks characterize sequences of instants and thus allow to predict statically at which instants computations are evaluated.

1.2.1 Prelude

The Prelude language is a synchronous dataflow language and the language we extend in this thesis. A program defines a computational graph operating on infinite streams of values. The particularity of the language is that it allows to express real-time constraints, most notably periodicity. This enables the language to use tools both from synchronous languages and real-time literature. For instance, the system designer can reason about a program using clocks, but then analyze its schedulability using a classical schedulability analysis, or generate multi-task C code targeting a real-time OS.

1.3 Multicore Platforms

The first issue tackled in this thesis is the programming of real-time systems on multicore platforms. Indeed, multicore platforms offer a potential for increasing system performances as cores might process tasks in parallel. However, these platforms are characterized by a central memory that is shared between cores. As cores can and must access this memory and the central memory can only answer a limited amount of requests at a time, a core might suffer a delay whenever it tries to access the main memory.

Accurately predicting these delays is difficult as it depends upon minute details within both hardware and software, as well as their interaction. This forces system designers to assume overly pessimistic execution scenarios, even though these scenarios might be highly improbable or even impossible.

Predictable multi-phase models have been proposed to address these issues. Intuitively, they divide system execution into "execution" and "memory" phases. Execution phases perform the actual computations required by the system tasks, while memory phases manage the unpredictability of memory accesses for the execution phases. For instance, a memory phase might perform cache prefetches such that the following execution phase never has to access the shared memory, all memory accesses being answered by the cache.

However, manually implementing these multi-phase models is non-trivial as the underlying hardware and software offers the illusion of an uniformly accessible memory. Thus, a deeply-nested function call might alter the system state in an unpredictable way, breaking the assumptions of the multi-phase model.

1.4 Multi-mode Systems

The functional requirements of a real-time system may evolve during its execution. For instance, the requirements for an aircraft control system are not identical during take-off, cruise and landing. Meeting all requirements at all times is obviously superfluous and just leads to over-provisioning the system's computing capacities.

A typical pattern in real-time systems is thus to design a system with multiple modes of execution. Each mode handles a specific set of functional requirements and a mode change request triggers a change from one mode to another.

The hardest challenge with this methodology is that while modes might be verified in isolation, the transition from one mode to another requires additional care. Indeed, a system designer has to choose a balance between promptness and system load during a mode change. A mode change with maximal promptness would simply release all new-mode tasks immediately, but this could lead to system overload as old- and new-mode tasks are competing for system resources. On the contrary, delaying the mode change until a "safe" point in time guarantees that the system load remains low, would have poor promptness as the worst-case time until such a point is reached is exponential with the number of tasks within the system.

In addition, the state of the art regarding multi-mode real-time systems is divided between works focusing exclusively on the temporal aspects and works focusing exclusively on functional aspects. However, efficient design of real-time systems requires the ability of both. Ignoring functional aspects would lead to design choices with questionable semantics and ignoring temporal aspects would limit oneself to a restricted set of systems.

Chapter 2

Contribution

The tackled problems will be discussed in individual Parts. Handling multicore is related to the back end of a compiler since it focuses on code-generation. In contrast, multi-mode behavior focuses on the front end of the compiler, especially the clock calculus, a form of static analysis. This allows us to consider improvements on all aspects of the language compiler.

2.1 Compiling Synchronous Languages on Multicore

Our contribution relies on an adaptation of the AER multi-phase model. It leverages the private memories of cores (e.g. caches or scratchpads) as they are not subject to competition. Task execution is divided into three phases: Acquisition, Execution, Restitution. The Acquisition- and Restitution-phases are memory phases. Their role is to copy data respectively from shared to private memory and from private to shared memory. Executing between those phases, the Execution phase thus does not suffer from memory access delays.

In Part III, the PRELUDE compiler is adapted to generate AER-compliant C code from a high-level PRELUDE program. Thus, the responsibility of low-level implementation concerns related to memory accesses are shifted onto the compiler. The automatic and systematic handling of these concerns by the compiler saves tedious and error-prone development efforts. These extensions is implemented as part of the main PRELUDE compiler [1].

2.2 Synchronous Semantics of Multi-mode systems

Our contribution builds upon the previous work of Synchronous State Machines for LUSTRE. State Machines are an established formalism to reason about stateful, reactive systems. A system designer defines states in which the system behavior is defined in isolation. Transitions allow to switch from one state to another. An interesting property of these state machines, is that they are defined using a transpilation. While the programmer writes programs in a surface language with state machines, the compiler transparently translates them into programs in a core language where the state machines have been replaced by lower-level language constructs.

A limitation of Synchronous State Machines is that have been defined for languages without multiple real-time constraints. Thus, a direct transposition of this feature to PRELUDE would prevent

[1] As part of the `ffort_aer` branch, either available at the ONERA forge or my personal mirror.

programmers from using a core aspect of the language. Indeed, this limitation arises from the lower-level constructs used in the transpilation which PRELUDE directly inherits from LUSTRE.

Our contribution thus focuses on extending these language constructs. In essence, these extensions allow us to decouple the rate of tasks from the rate of mode changes. This in turn required us to change the clock calculus of the PRELUDE language. As the required modifications are quite profound, these extensions resulted in a rewrite of the PRELUDE compiler [2].

2.3 Thesis Overview

This Section will detail what to expect from and how to read this document.

Part II will present the "Related Works" of this thesis. It presents established models and languages used in the context of real-time systems which will serve as a basis for our contributions. Part III will present the first contribution detailed in Section 2.1. Part IV will present the second contribution detailed in Section 2.2. Finally, Part V will conclude this thesis and discusses perspectives for future works.

While this document is readable from "beginning to end", the contributions are mostly independent. Thus, certain readers might prefer reading first towards one contribution and then towards the second.

In all cases, Chapter 3, Chapter 6 and Chapter 7 are must-reads to get a proper understanding of the contribution. Chapter 3 presents a base model for real-time systems which originated from pioneer works in real-time scheduling [55]. However, this model is only suited for analyzing *a posteriori* an existing system. Thus, Chapter 6 presents languages to actually program real-time systems. It will in particular focus on synchronous languages [14]. Finally, Chapter 7 will present the synchronous language PRELUDE [67] upon which these contributions build.

For the contribution in Part III, one should read Chapter 4 to better understand the issues introduced by multicore platforms in the context of real-time systems. If it isn't already done, reading Section 6.7 will also detail how these issues have been addressed from the point of view of programming languages in the state of the art.

For the contribution in Part IV, one should read Chapter 5 and Section 6.6 to understand how this issue has been tackled in the state of the art from the point of view of real-time scheduling and programming languages.

[2] Available here

Part II

Background and Definitions

Chapter 3

Simple Model for Real-Time Tasks

In this Chapter, we present an overview of real-time systems and a simple model for real-time tasks. In the following, Chapters 4 to 5, this model will be expanded upon.

In essence, real-time systems are cyber-physical systems which simultaneously perceive and influence an environment. Their reactions must be provided at a rate that is adapted to the evolution rate of the environment. A failure to do so would have critical consequences, e.g. deaths, high financial or environmental damages. Readers looking for a more in-depth review may consult [21, 22].

A real-time system is modeled as a directed acyclic graph $(\mathcal{T}, \mathcal{D})$ where $\mathcal{T}$ is a set of tasks $\tau_i \in \mathcal{T}$ and $\mathcal{D}$ is a set of data-dependencies between pairs of jobs $(\tau_i^n, \tau_j^m) \in \mathcal{D}$.

3.1 Tasks

Each task $\tau_i = (O_i, T_i, D_i, C_i) \in \mathcal{T}$ releases a sequence of non-overlapping jobs where τ_i^n denotes the n-th job of τ_i. The set of jobs is $\mathcal{J}$. A task can be interpreted as an OS thread that executes an infinite loop, while a job represents one execution of the loop. Figure 3.1 illustrates this simplified view.

As alluded above, a task is defined as a tuple of *real-time attributes*. We consider the following attributes.

Definition 1 (Offset)**.** The *offset* O_i denotes the date when the first job τ_i^0 of τ_i is released.

```
void task_i (void)
{
  // Start of task τ_i
  for(int n = 0; true; ++n)
  {
    // Wait for activation of job τ_i^n
    wait_activation();
    // Perform computation associated to job
    do_computation();
  }
}
```

Figure 3.1: Simplified view of tasks and jobs

Definition 2 (Period)**.** The *period* T_i denotes the time interval between successive releases of jobs of τ_i, i.e. τ_i^n is released at date $r_i^n = O_i + n * T_i$.

Definition 3 (Deadline)**.** The *relative deadline* D_i denotes the maximum amount of time a job of τ_i has after its release to finish its execution. A relative deadline D_i is *implicit*, if $D_i = T_i$. In most models, non-implicit deadlines are still constrained such that $D_i \leqslant T_i$. In certain models, different jobs of the same task may have different relative deadlines.

The *absolute deadline* is the point in time at which τ_i^n must have finished its execution. It is $d_i^n = r_i^n + D_i$.

Definition 4 (Execution time)**.** The *execution time* c_i denotes the amount of time a job requires to compute. An important metric is the *Worst-Case Execution Time* (WCET) C_i because of its use in Scheduling Analysis (Section 3.4).

Definition 5 (Utilization)**.** The system utilization is the total load of the system. It is defined as $U = \sum_{\tau_i \in \mathcal{T}} = \frac{C_i}{T_i}$.

Definition 6 (Hyperperiod)**.** The *hyperperiod* of a task(sub)set is the least common multiple of the task periods. We denote H the hyperperiod of the entire system and $H_{i,j}$ the hyperperiod of a pair of tasks τ_i and τ_j.

Definition 7 (Job begin)**.** We denote $begin(\tau_i^n)$ the date at which τ_i^n starts executing (which may be after r_i^n) for a specific schedule.

Definition 8 (Job end)**.** We denote $end(\tau_i^n)$ the date at which τ_i^n stops executing (which may be before d_i^n) for a specific schedule.

Definition 9 (Schedule)**.** A *schedule* is a possible execution trace of a real-time system. Scheduling decisions, e.g. which job should execute at which point in time, are the responsibility of the scheduling policy (Section 3.3). A schedule is *valid*, iff $\forall \tau_i^n . r_i^n \leqslant begin(\tau_i^n) \wedge end(\tau_i^n) \leqslant d_i^n$, i.e. all jobs terminated before their deadline and started executing after their release.

3.2 Data-dependencies

Each data-dependency, also called communication, $\tau_i^n \rightarrow \tau_j^m = (\tau_i^n, \tau_j^m) \in \mathcal{D}$ indicates that job τ_i^n produces data that is used by job τ_j^m. We assume that data-dependencies are *causal*.

Definition 10 (Causal data-dependencies)**.** A *causal data-dependency* $\tau_i^n \rightarrow \tau_j^m$ requires:

$$end(\tau_i^n) < begin(\tau_j^m)$$

For each data-dependency $\tau_i^n \rightarrow \tau_j^m$, τ_i^n produces its outputs at $end(\tau_i^n)$ and τ_j^m acquires its inputs at $begin(\tau_j^m)$.

Moreover, to efficiently implement a real-time system, we only consider data-dependencies between tasks that follow a predefined sequence. The data-dependency between tasks τ_i and τ_j is a tuple $\mathcal{D}_{i,j} = (\mathcal{D}_{i,j}^{pref}, I_{i,j}^{pref}, \mathcal{D}_{i,j}^{pat}, I_{i,j}^{pat})$ where $I_{i,j}^X$ is an interval length in time units, $\mathcal{D}_{i,j}^X$ is a data-dependency template, $pref$ indicates that it belongs to the *prefix* and pat indicates that it belongs to the *pattern*. The prefix defines the data-dependencies between the two tasks at the start of the system between

dates 0 and $I_{i,j}^{pref}$. Thus, $\mathcal{D}_{i,j}^{pref}$ contains data-dependencies between jobs active between date 0 and $I_{i,j}^{pref}$. The pattern is a template that repeats after date $I_{i,j}^{pref}$ each $I_{i,j}^{pat}$ time units.

Because of the cyclical nature of data-dependencies, we can define $\mathcal{D}$ by using $\mathcal{D}_{i,j}^{pref}$ for the first $I_{i,j}^{pref}$ time units and then unrolling $\mathcal{D}_{i,j}^{pat}$ each $I_{i,j}^{pat}$ time units to obtain data-dependencies between concrete job instances. For instance, the first iteration of $\mathcal{D}_{i,j}^{pat}$ defines data-dependencies between dates $I_{i,j}^{pref}$ and $I_{i,j}^{pref} + I_{i,j}^{pat}$. Data-dependency $(\tau_i^0, \tau_j^0) \in \mathcal{D}_{i,j}^{pat}$ then indicates a data-dependency between the first jobs of τ_i andτ_j released within that interval (typically at date $I_{i,j}^{pref} + I_{i,j}^{pat}$).

Definition 11 (Data-dependencies by unrolling)**.** Assuming an assignment of data-dependencies prefixes and patterns ($\mathcal{D}_{i,j}^{pref}$,$\mathcal{D}_{i,j}^{pat}$) and their length ($I_{i,j}^{pref}$, $I_{i,j}^{pat}$), the set of data-dependencies $\mathcal{D}$ is defined as:

$$\begin{aligned}
\mathcal{D} &= \mathcal{D}^{pref} \cup \mathcal{D}^{pat} \\
\mathcal{D}^{pref} &= \left(\bigcup_{\tau_i, \tau_j \in \mathcal{T}} \mathcal{D}_{i,j}^{pref} \right) \\
\mathcal{D}^{pat} &= \{(\tau_i^p, \tau_j^q) \mid k \in \mathbb{N}, \tau_i, \tau_j \in \mathcal{T}, (\tau_i^n, \tau_j^m) \in \mathcal{D}_{i,j}^{pat}, \\
&\qquad p = n + \frac{I_{i,j}^{pref}}{T_i} + k\frac{I_{i,j}^{pat}}{T_i}, \\
&\qquad q = n + \frac{I_{i,j}^{pref}}{T_j} + k\frac{I_{i,j}^{pat}}{T_j}\}
\end{aligned}$$

3.3 Scheduling Policies

The pioneer works of the policies Rate-Monotonic (RM), Deadline-Monotonic (DM), and Earliest-Deadline First (EDF) spawned a large field of research with dedicated conferences and journals. Scheduling policies (and analysis) being out-of-scope for this work, we will focus on this short presentation. Curious readers might consider the earlier cited references for more in-depth discussions [21,22].

Because jobs compete for accessing shared resources (most notably the CPU), a *scheduling policy* is required to determine which job is to execute at each point in time, using real-time attributes to guide its decisions. A real-time system is *feasible* under a policy, iff the decisions of the scheduling policy guarantee that each job finishes executing before its absolute deadline. A policy is *optimal* for a specific class of systems, iff it is able to produce a feasible schedule for a feasible systems within that class.

DM belongs to the class of task-level static priority policies. All jobs of a task share the same priority. Jobs with lower relative deadline have a higher priority. EDF belongs to the class of job-level static priority policies. A job's priority is determined by its absolute deadline. Jobs with an earlier deadline have a higher priority. EDF is the more general policy: it is optimal on mono-processor systems. DM is only optimal on mono-processor systems within the class of task-level static priority policies.

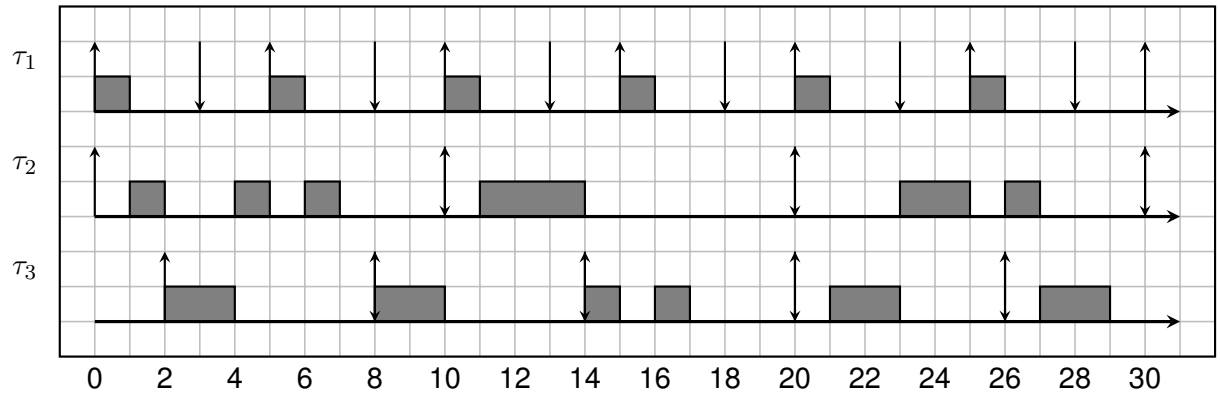
3.4 Schedulability Analysis

A *schedulability analysis* is a static analysis that verifies the feasibility of a real-time system. The complexity of the schedulability analysis depends on the scheduling policy and the complexity of the system. For a real-time system scheduled by EDF without data-dependencies and implicit deadlines, the schedulability analysis is $\sum_{\tau_i \in \mathcal{T}} \frac{C_i}{T_i} \leqslant 1$. Adding data-dependencies or non-implicit deadlines requires to verify that the *demand-bound function* [12] holds, i.e. $\forall t \in \mathbb{N}.\ t \geqslant \sum_{\tau_i \in \mathcal{T}} C_i * (\lfloor \frac{t-D_i}{T_i} \rfloor + 1)$.

Example 1 (Real-time systems with and without data-dependencies)**.** Let us consider the following real-time system without data-dependencies.

τ_i	O_i	T_i	D_i	C_i
τ_1	0	5	3	1
τ_2	0	10	10	3
τ_3	2	6	6	2

Executing this system according to the *Earliest Deadline First* policy on a monocore platform yields the following schedule shown as a *timing diagram*. The diagram indicates when a given task executes. Each task has its own timeline. A grey box indicates that a task executes during that time interval. Since tasks compete for access to the processor, only one may be active at a time. Arrows pointing upwards indicate a job release, while arrows pointing downwards indicate that corresponding job's absolute deadline.



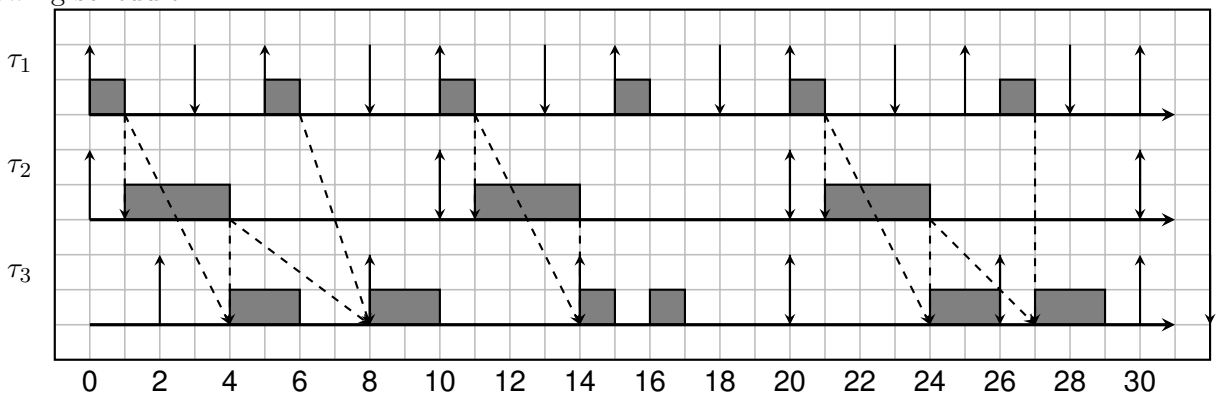
A selection of attributes:

$$H = 30 \qquad H_{1,2} = 10$$
$$r_1^1 = 5 \qquad d_1^1 = 8$$
$$begin(\tau_2^0) = 1 \qquad end(\tau_2^0) = 7$$

Now, let us consider the same system with data-dependencies. The data-dependency patterns are shown below.

$$\mathcal{D}_{1,2}^{pat} = \{(\tau_1^0, \tau_2^0)\} \qquad I_{1,2}^{pat} = 10$$
$$\mathcal{D}_{1,3}^{pat} = \{(\tau_1^0, \tau_3^0), (\tau_1^1, \tau_3^1), (\tau_1^2, \tau_3^2), (\tau_1^4, \tau_3^3), (\tau_1^5, \tau_3^4)\} \qquad I_{1,2}^{pat} = 30$$
$$\mathcal{D}_{2,3}^{pat} = \{(\tau_2^0, \tau_3^0), (\tau_2^0, \tau_3^1), (\tau_2^1, \tau_3^1), (\tau_2^2, \tau_3^4), (\tau_2^2, \tau_3^4)\} \qquad I_{2,3}^{pat} = 30$$

Executing this system according to the EDF policy, while respecting data-dependencies, yields the following schedule.



Chapter 4

Multicore Real-Time Systems

Multicore systems pose new issues for real-time systems. On the one hand, they offer a potential for increasing system performances by allowing to execute one job per core simultaneously (i.e. offering parallelism in addition to the inherent concurrency of real-time systems). On the other hand, multicore systems are characterized by a global memory that is shared between cores via a bus. Accesses to this memory may be delayed if multiple cores try to access it simultaneously. These delays, called contentions, have a high variability between their worst-case and average values because they depend upon minute details within task codes, task interferences and the contention resolution mechanisms [72]. This in turn leads to an overly pessimistic WCET even though the conditions for the actual worst-case are highly unlikely or even impossible.

4.1 Scheduling

Scheduling policies similar to those used in monocore exist in multicore. However, these policies exist either as a *partitioned* or as a *global* variant. In a partitioned policy, tasks are assigned at system design to a core and may only execute on that core. Thus, scheduling decisions need only to take into account core-local properties. In a global policy, tasks may execute on any processor and a single scheduler makes decisions for the whole system. These two variants are *incomparable*, meaning that there exist real-time systems which are only feasible under partitioned scheduling and others which are are only feasible under global scheduling.

4.2 Hardware model

To elaborate our hardware model, let us look at how a multicore system is composed. Such a system is composed of processor cores which are connected via a bus to the RAM which is a *shared central memory*. Thus, access to the RAM is subject to contentions. Each core also has access to *private memories* which can only be accessed by that core and are not subject to contentions. The most common are cache memories which can replicate content of the RAM for faster access. When accessing a memory address, if the location is replicated in the cache, the processor can access it faster and contention-free from there.

Certain embedded platforms also have *scratchpad memories* (SPM). These private memories have the same timing properties as caches, but instead of implicitly replicating the shared memory, scratchpad memories possess their own memory space and must thus be manipulated explicitly.

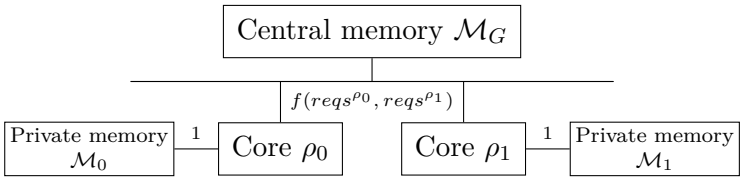


Figure 4.1: An example hardware component graph

Using those observations, we can see that the switch to multicore platforms requires to explicit a *hardware component graph*. It also exists in monocore platforms, but represents trivial instance of such a graph, i.e. accessing the central memory is trivial when there is only one processor/core accessing it. The edges of this graph are either CPU cores $\rho_i \in \Pi$ or memories $\mathcal{M}_i \in \mathfrak{M}$. The vertices of this graph connect CPU cores to memory components and are annotated with *access costs*.

Figure 4.1 displays an example graph. Each core ρ_i has access to its private memory $\mathcal{M}_i$, but cannot access to another private memory. The access cost is constant in this example, 1 instruction cycle, which is typical for scratchpads or caches. Each core can however access the central memory $\mathcal{M}_G$. The access cost in this case is however a function depending on the memory requests emitted by all CPU cores.

4.3 PREM

To address the pessimism induced by contentions, one solution is to reduce the number of instructions they can impact. The PRedictable Execution Model (PREM) [71] achieves this by decoupling contention-inducing (communication) phases from computation phases.

The idea is that most jobs in a real-time system will load values from a shared buffer or sensor, perform computations and then write results back into a buffer or actuator. When writing code without PREM-compliance in mind, one could mix those buffer accesses and computations, e.g. first load a value and perform some computations with it before loading a second value. Worse, those buffer accesses could be located deeply-nested within functions called by the task code.

To write with PREM-compliance in mind, contending accesses have to be isolated such that once all such operations are completed, computations can execute without risking to produce contentions. To do so, all required data and code should be located within memories that are guaranteed contention-free, e.g. registers, private caches or scratchpad-memory.

Figure 4.2 shows a simple task code with and without PREM-compliance. Without PREM-compliance in Figure 4.2a, the task is implemented using opaque subroutines where certain buffers are accessed mid-execution. Figure 4.2b shows an implementation with PREM-compliance in mind. Three distinct phases are visible. First, all necessary buffers are read in a contention-inducing communication phase. Second, subroutines are called in a contention-free computation phase. Note that deeply-nested buffer accesses are replaced by explicit parameters. Lastly, a new communication phase stores the result in an output buffer.

4.4 AER

The AER [32] model is similar to the PREM model but further restricts the structure of tasks. Each job τ_i^n of task τ_i must be divided into a sequence of 3 phases: Acquisition (A_i^n), Execution (E_i^n), Restitution (R_i^n). In the Acquisition phase, data is copied into private memory from global memory.

```
void g(int x)
{
  x = g1(x);
  int y = read2();
  x = g2(x, y);
  write3(x);
}

void do_computation()
{
  //Reads buffer 1
  int x = f()
  //Reads buffer 2
  //Writes buffer 3
  g(x)
}
```

(a) Without PREM-compliance

```
int g(int x, int y)
{
  x = g1(x);
  x = g2(x, y);
  return x;
}

void do_computation()
{
  int x = load_buf1()
  int y = load_buf2()
  x = f(x)
  y = g(x, y)
  store_buf3(y)
}
```

(b) With PREM-compliance

Figure 4.2: Job body with and without PREM-compliance

In the Execution phase, computations are executed contention-free using the private memory. Finally, in the Restitution phase, results of the computation are copied from the private memory into the global memory.

This declination is advantageous because it is close to the the model of synchrony (Section 7.4) where computations (Execution phase) are assumed to execute without side-effects except input acquisition (Acquisition phase) and output production (Restitution phase). Because of this, we will rely on the AER model in this work.

Looking at Figure 4.2b, this example matches with the AER model. The calls to `load_bufX()` are the Acquisition phase. The calls to `f` and `g` are the Execution phase and `store_buf3(y)` is the Restitution phase.

4.5 Related Works

The difficulty of predicting the timing properties of multicore real-time systems because of the minute interactions between task codes, task interferences, and the contention resolution mechanisms, has been identified in [72, 73, 85].

The PREM [71] and AER [32, 58] models have been proposed to address this issue. They both divide task execution in *memory phases* which interact with the global memory and *execution phases* which perform computations contention-free, i.e. with accessing the global memory. The main difference between those two is that the AER model is more explicit about the role of the different phases. Unless stated otherwise, we will use PREM and AER interchangeably.

Both models have gained significant attention from the real-time scheduling community such that numerous scheduling algorithms and schedulability analyses have been proposed [4–6, 13, 18, 57, 63, 84, 106, 108, 109].

Concrete applications of these models have been considered at different levels of the software-stack.

4.5.1 OS-level support

At the lowest level, support for PREM can be introduced at the OS-level. This has been studied in [90, 93–95, 105]. The system developers writes their task code using the provided OS primitives. The compilation process will produce an OS image that will use the scratchpad memory as appropriate.

Similarly, others have studied the introduction of PREM-support using a hypervisor [51]. In [40], such a technique is introduced to support mixed-criticality systems.

4.5.2 Source code refactoring

Many real-time systems are composed of legacy code, in general non-PREM-compliant C code. Refactoring this code to be PREM-compliant is non-trivial and requires a thorough understanding of the code to be converted. Using memory profiling tools, Light-PREM [59] refactors legacy source code in such manner that it is PREM-compliant.

4.5.3 Binary code generation

Another possibility is to modify the compiler, in this case the LLVM compiler, such that it produces PREM-compliant code. In [71], a new `predictable`-block is introduced. Code within this block is generated such that it is PREM-compliant. In [62], no new language construct is introduced, but the compiler plugin is required to perform more advanced analyses and ILP-based scheduling. In [91], a specialized LLVM plugin produces PREM-compliant code using SPM.

4.5.4 Predictable GPU-code

The problems of multicore-platforms that led to the development of the PREM-model, also exist on platforms featuring integrated GPUs. Unlike dedicated GPUs which possess their own memory and communicate with the main memory via a bus such as PCI-Express, integrated GPUs share their memory with the CPU. To address this issue, solutions such as HePREM [37, 38] and SiGamma [24] have been proposed.

Chapter 5

Multi-Mode Real-Time Systems

One restriction of the model in Chapter 3 is that the task set $\mathcal{T}$ is statically defined at system design. It is not possible to specify changing functional requirements throughout system execution. However, certain real-time systems have a *multi-moded* behavior. A typical example would be an aircraft control system with modes such as take-off, cruise and landing.

In this Chapter, we will discuss two ways to model real-time systems with the ability to perform system reconfiguration during execution. Section 5.1 presents *mode change protocols*, a generic mechanism to switch between different modes. Section 5.2 presents *mixed-criticality systems*, a special case of mode change protocols where the current mode sets the minimum *criticality* of tasks allowed to run.

5.1 Mode Change Protocols

When using *Mode Change Protocols* (MCP) [82], the system is defined as a set of distinct task sets. Each task set represents a possible and valid system configuration with potentially different timing constraints. The MCP is then responsible for switching from one task set (the old mode) to another (the new mode) when a *Mode Change Request* (MCR) is emitted. During this *transition phase*, the MCP must guarantee that jobs respect their deadlines.

Note that this approach asks different levels of detail for different aspects of the system. The system designer (or the used tool) must define precisely each individual mode in its "pure" form, i.e. how the mode executes ignoring the arrival of MCRs. However, beyond the choice of a specific mode change protocol, there is little control over how the system behaves between the modes.

In this context, different protocols offer different advantages. We compare MCPs according to two criteria: *promptness* and *overload*. The promptness of a MCP is its worst-case transition time. The overload induced by a MCP is due to the transition phase. For certain MCPs, a dedicated schedulability analysis is required.

A protocol with maximal promptness would start releasing all new-mode jobs immediately upon receiving the MCR. This leads however to poor schedulability because the system would need to be able to schedule both modes simultaneously during the transition phase.

A protocol with maximal schedulability would wait until no job of the old-mode is waiting to be executed. In the worst case, this requires to wait until a date that is a multiple of the hyperperiod. This protocol leads to no temporary overload and is easy to analyze. However the worst-case transition time is exponential with the number of tasks.

There is no best protocol and as a consequence, a system designer must choose a MCP that suits their system. The real-time literature [82] classifies MCPs according to three criteria:

- *Overlapping*[1]: When do the *new-mode tasks* start executing?
- *Periodicity*: Are *unchanged tasks* impacted by mode changes?
- *Retirement*: How long can *old-mode tasks* continue executing?

Non-overlapping protocols release the new-mode tasks only at the end of the transition phase. Tasks of both modes thus never co-exist. *Overlapping* protocol allow new-mode and old-mode tasks to be both executed during the transition phase. The potential overlap between modes allows for a more prompt transition at the cost of a scheduling analysis specific to the transition phase.

Periodic protocols do not interfere with the execution of *unchanged tasks*, i.e. tasks which are present both in the old- and the new-mode. *Aperiodic* protocols suspend unchanged tasks for the duration of the transition phase.

Late-retirement protocols gracefully decommission old-mode tasks by continuing their execution for a given amount of time, e.g. until they complete their current activation. *Early-retirement* protocols abort old-mode tasks as soon as the MCR is triggered.

One should note that this classification of multi-mode real-time systems [82] does not take into account the semantics of the system. For instance, how do early-retirement or aperiodicity interact with data-dependencies ? This issue will be addressed in our work.

Example 2 (Multi-mode real-time systems)**.** Let us consider the following real-time system with the modes A and B. For simplicity, we assume that all tasks have $O_i = 0$.

τ_i	T_i	D_i	C_i
τ_1	5	3	1
τ_2	20	20	4
τ_3	10	10	3
τ_4	10	8	2

$$\mathcal{T}_A = \{\tau_1, \tau_2, \tau_4\}$$
$$\mathcal{T}_B = \{\tau_1, \tau_3, \tau_4\}$$

Data-dependencies within mode A are:

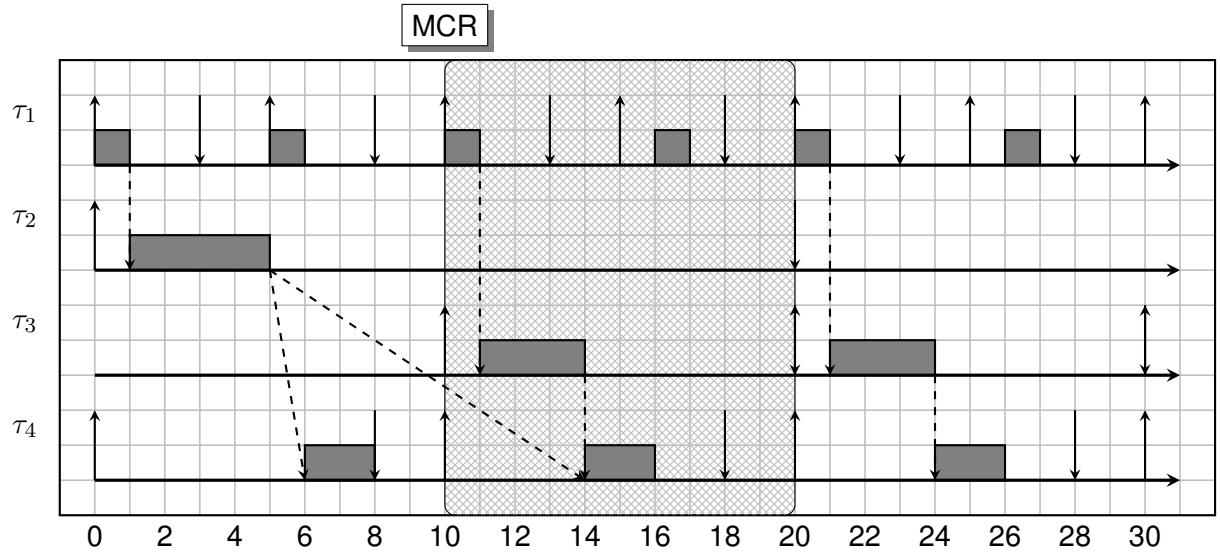
$$\mathcal{D}^{pat}_{1,2} = \{(\tau^0_1, \tau^0_2)\} \qquad I^{pat}_{1,2} = 20$$
$$\mathcal{D}^{pat}_{2,4} = \{(\tau^0_2, \tau^0_4), (\tau^0_2, \tau^1_4)\} \qquad I^{pat}_{1,3} = 20$$

Data-dependencies within mode B are:

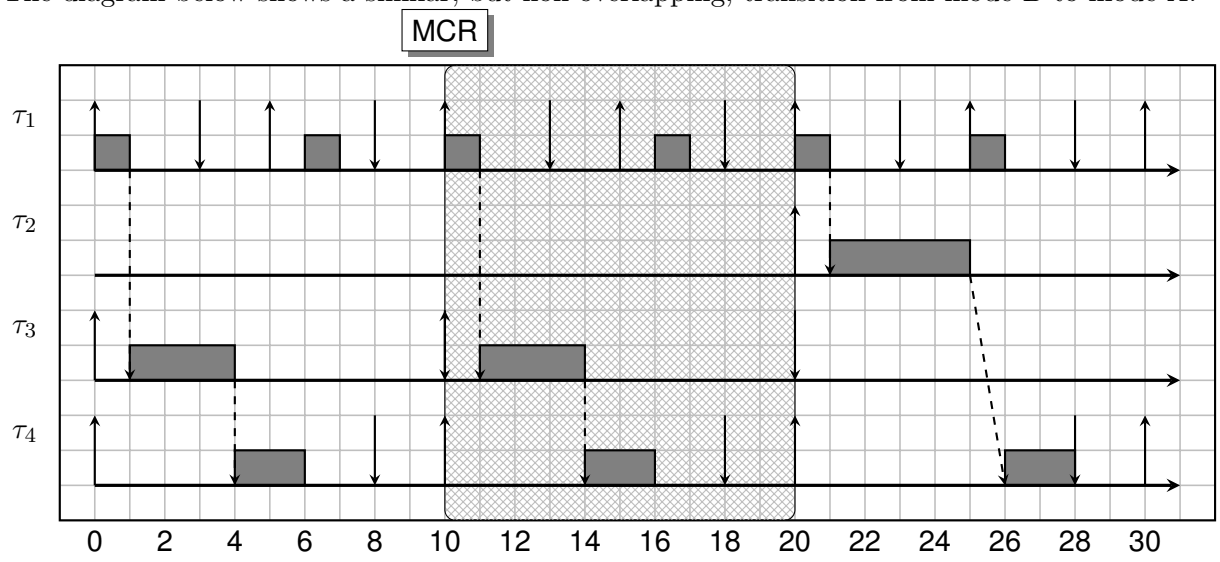
$$\mathcal{D}^{pat}_{1,3} = \{(\tau^0_1, \tau^0_3)\} \qquad I^{pat}_{1,2} = 10$$
$$\mathcal{D}^{pat}_{3,4} = \{(\tau^0_3, \tau^0_4)\} \qquad I^{pat}_{3,4} = 12$$

Below we will show a transition from mode A to mode B using an overlapping, periodic protocol with late-retirement. A MCR is emitted at date 10, which starts a transition phase (grey box). As can be seen, during the transition period, tasks τ_2 and τ_3 co-exist during this period.

[1] Also called synchronicity [82], we renamed it to avoid confusion with synchronous languages.



The diagram below shows a similar, but non-overlapping, transition from mode B to mode A.



5.1.1 Related Works

A survey [82] presents a good overview of real-time scheduling of multi-mode systems.

- The *idle time protocol* [98] waits for an *idle time* (i.e. an instant where no job is waiting to be executed) to transition. The worst-case waiting time is the hyperperiod of old- and new-mode tasks.
- The *maximum-period offset protocol* [9] waits for a time equal to the maximum of the old- and new-mode task periods to transition.
- The *minimum single offset protocol* [81] interrupts the release of old-mode jobs upon reception of an MCR and waits for the termination of currently running old-mode jobs;
- The *utilization-based protocol* [86] keeps a ledger of the current system utilization. New-mode tasks are added gradually as old-mode tasks finish executing, reclaiming the liberated utilization.

- Tindell's *asynchronous protocol with periodicity* [99] is late-retiring. Old-mode jobs finish their current execution, but no new old-mode jobs are released. Unchanged tasks may have different attributes between modes.
- Pedro's *asynchronous protocol without periodicity* [70] behaves similarly. Breaking periodicity allows however to schedule a greater number of task sets.
- The paper's own protocol is a mixture of Tindell's and Pedro's protocol. It is an extension of Tindell's protocol that takes inspiration from Pedro's protocol to improve promptness. By default, it is a periodic protocol, but it can introduce aperiodicity to guarantee schedulability.

A limitation of this work is that it does not consider data-dependency. Instead, it considers *consistency* of shared resources, i.e. mutexes, which is too simplified in our opinion.

Later work on *real-time calculus* [75–77, 92], improves this by considering that tasks consume data from a buffer. However, it is not considered how these buffers are filled.

Beyond this work, some work expanded upon the study of multi-mode systems for special cases such as multicore systems [65, 66], high-performance multimedia systems [100] or systems with arbitrary deadlines [61]. However, these papers do not solve this limitation.

5.2 Mixed-criticality systems

A special case of multi-mode systems are mixed-criticality systems [19, 20, 102]. In the base model of real-time systems, all tasks are considered equally critical: any deadline-miss is considered catastrophic. This is not the case in a mixed-criticality system. In its simplest form, each task has either a high- (HI) or low-criticality (LO). LO tasks may execute in a degraded state, if this may allow HI tasks to respect their deadlines.

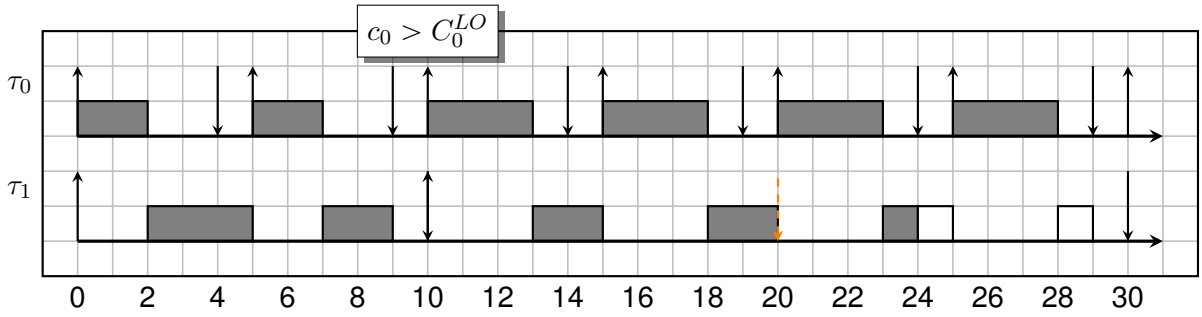
The main motivation for mixed-criticality originates from software standards such as the avionics standard DO-178C [30] where software has different levels of criticity. DO-178C defines Software Levels from A, "Catastrophic", to E, "No Effect". Failure in Software Level A could cause deaths, while failure in Software Level E could not negatively impact safety, aircraft operation, or crew workload (but could cause passenger discomfort). Mixed-criticality systems offer a way to execute software of different levels on the same platform while guaranteeing that lower levels have no negative impact on software of higher levels [20].

Example 3 (Mixed-criticality real-time systems)**.** Let us consider the following, simplified, mixed-criticality real-time system. It has two levels of criticality, LO and HI. In each mode, two tasks execute, τ_0 and τ_1. Task τ_0 is a high criticality task that must be able to execute at any cost, while task τ_1 is of lower criticality and may experience a decrease in quality of service.

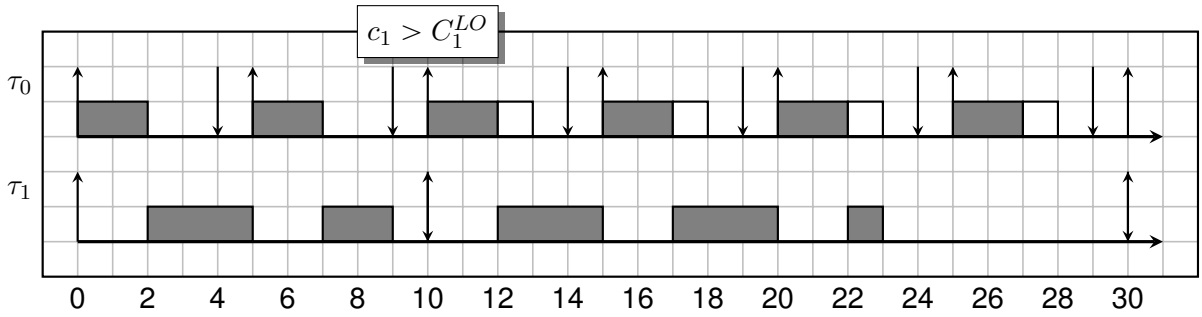
Their real-time attributes are shown below. As can be seen, task period, deadline and WCET vary between the LO- and HI-mode. Most notably, the WCET in the LO-mode is lower for both tasks. This WCET-value is the "optimistic" WCET which upper-bounds the majority of execution traces. On the other hand, the values for the HI-mode are the actual WCET values which upper-bound all execution traces. The examples below will assume a semi-clairvoyant scheduler (see below) that is able to increase task periods and deadlines during execution, but not abort jobs.

		LO			HI		
Task	O_i	T^{LO}	D^{LO}	C^{LO}	T^{HI}	D^{HI}	C^{HI}
τ_0	0	5	4	2	5	4	3
τ_1	0	10	10	5	20	20	7

The timing diagram below illustrates a potential criticality change due to τ_0 requiring more execution time than C_0^{LO}. This means that the absolute deadline of τ_1^0 is delayed by 10 time units to reflect the increase of T_1 and D_1 (orange doted line at date 10). In white, we indicate the time intervals τ_1^0 would use in case it too required more execution time.



This second diagram illustrates a criticality change triggered by τ_1 this time. Again, white boxes indicate time slices that would have been consumed in case jobs of τ_0 required their high-criticality execution time.



5.2.1 Related Works

A survey [20] presents a good overview of real-time mixed-criticality systems. It is regularly updated which makes it a valuable resource.

The seminal work of mixed-criticality [102] presents the ground works of the mixed-criticality model. While notions such as HI and LO tasks has been reused in further work, this paper presents a simplistic model. Tasks have a set of real-time attributes for each criticality level. Thus, the WCET value represents an execution time budget. The scheduler tracks the execution time of tasks. When a tasks exceeds that budget, the scheduler changes the criticality level. Certain tasks would have no period in the new mode, meaning they would be aborted immediately without taking into consideration data-dependencies. Moreover, the paper defines no mechanism to return to a lower criticality level.

In [19], it has been identified that mixed-criticality systems are a special case of multi-mode systems. Reconsidering them under this optic allows us to reapply previous results. For instance, returning to a lower criticality can be solved by re-using an existing mode change protocol.

Work on *semi-clairvoyant scheduling* for mixed-criticality systems [3, 11], addresses the issue of the nebulous semantics in the original model. A semi-clairvoyant scheduler is able to know at job-release (thus before any instruction is executed) certain properties about the job-execution. A semi-clairvoyant scheduler can use a parametric WCET [10, 23] to compute a more exact WCET value for the current job execution. Thus it may potentially trigger a change in criticality mode.

5.2.2 Link with Mode Change Protocols

As has been illustrated, mixed-criticality systems literature often lacks considerations for semantics. Reconsidering them as multi-mode system could solve this issue. Our contribution (Part IV), offers thus a way to program mixed-criticality systems with a formally sound basis.

Chapter 6

Synchronous Languages for Real-Time Systems

In Chapters 3 to 5, the presentation of real-time systems was fairly abstract, it focused mostly on the temporal properties of these systems. In this Chapter, we present synchronous languages that have been designed for programming real-time systems. Moreover, we will also evaluate how these languages tackle the problems of programming multicore and multi-mode systems.

6.1 Synchronous Languages

Synchronous languages are a family of programming languages that have been designed to tackle the challenges of programming safety-critical real-time embedded systems [14]. They are characterized by combining three properties:

1. Synchrony to abstract time as a sequence of discrete instants;
2. Deterministic concurrency via explicit language constructs;
3. Simple formal underlying models to make verification tractable.

Within that design space, different solutions emerged. In Sections 6.2 to 6.5, we will review these solutions and later contributions. We will both consider the original designs and later contributions that addressed the design of multicore and multi-mode systems.

6.2 Lustre

The Lustre programming language [42] is based on the formalism of control engineers who define a system via equations or dataflow networks. Under this formalism, a system is a sequence of equations $x = e$ where x is a variable that appears exactly once on the left side of an equation and e is an expression describing the value of x at each point in time.

A Lustre program is thus structured around *nodes* rather than functions. The main difference is that in a node variables represent dataflows, i.e. infinite sequences of values, instead of individual scalar values. Instead of being composed of expressions or statements, the node-body is composed of *equations*. Node equations can be understood like mathematical equations: They define one dataflow by composition of other dataflows.

The synchronous model of time in the LUSTRE language is structured around *clocks*. Each dataflow has a clock and when a clock is *present*, i.e. "it produces a tick", its associated dataflow is present. A *clock calculus* [29] verifies the consistency of clocks within the program. In the case of LUSTRE, it can be solved using techniques similar to Hindley-Milner typing.

A *base clock* `base` specifies the fastest possible rate, i.e. `base` is present at each instant. Externally, the system designer can specify a period for the base clock, e.g. $1ms$, but internally the language operates purely on ticks of the base clock. The clock operator *ck* **on** `C(c)` produces a clock that is present at each instant where *ck* is present and the dataflow `c` (whose clock is *ck*) produces a value `C`.

An expression `e when C(c)` conditionally sub-samples the dataflow `e` by keeping only the values of `e` at instants where dataflow `c` simultaneously produces the value `C`. The expression has clock *ck* **on** `C(c)` while both `e` and `c` have clock *ck*.

The expression `merge(c, C0->e0, C1->e1)` merges the *complementary* dataflows with respective clocks *ck* **on** `C0(c)` and *ck* **on** `C1(c)`, i.e. only one dataflow is present each instant and the union of their instants is the instants of clock *ck*. The resulting dataflow has clock *ck*. For each instant, the `merge` operator simply selects the currently present dataflow and produces its value for the current instant. Another operator to manipulate dataflows with **on** operators is `current`. It accepts a dataflow with clock *ck* **on** `C(c)` and returns a dataflow with clock *ck*. For each instant where *ck* **on** `C(c)` is not present, the operator "fills the hole" by repeating the input dataflow's last value. A limitation of this operator is that, unlike the `merge` operator, it's behavior is undefined is *ck* **on** `C(c)` is not present the first time *ck* is present.

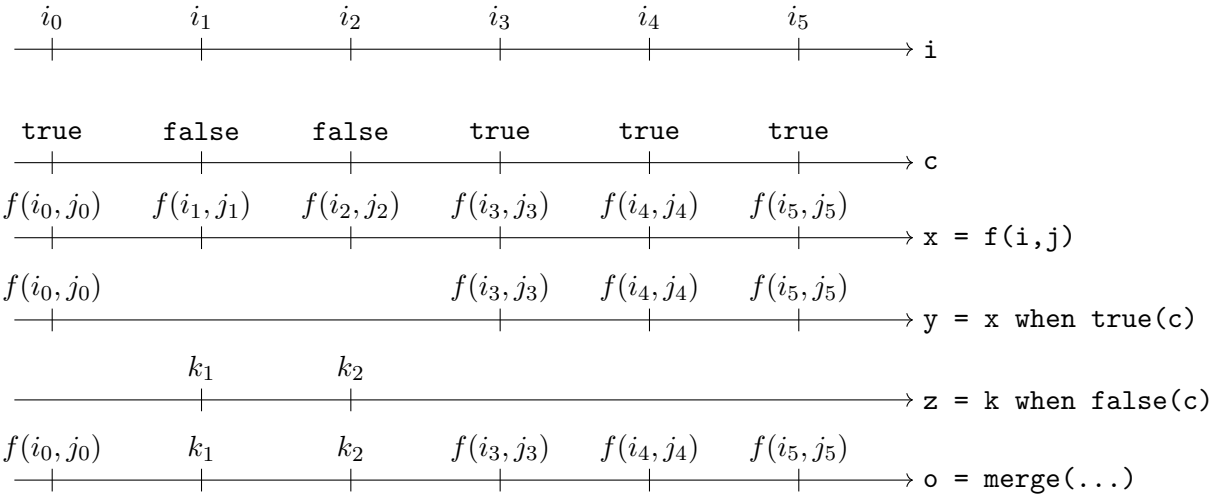
To relate logical time to physical time, a typical pattern is to define `base` as the fastest rate in the system, e.g. a period of $10ms$, and then define sub-samplings of this clock that define slower rates. For instance, if dataflow `each2` is true only once each two successive instants, clock `base` **on** `true(each2)` would represent a clock with a period of $20ms$. However, this scheme has the limitation that it does not compose well. For instance, the type system has no knowledge of how `base` **on** `true(each2)`, `base` **on** `true(each3)` and `base` **on** `true(each6)` relate to each other.

Example 4 (Lustre)**.** The program below is a minimal LUSTRE program featuring a single node `main`. This node has four inputs, `i`, `j`, `k`, and `c`, and one output, `o`. The first equation defines the local dataflow `x` as the result of `f(i, j)` where `f` is a function over scalars that has been lifted to operate on dataflows via point-wise application. For consistency reasons, such a lifted function requires its arguments to be *synchronous*, i.e. that they possess the same clock. The next two dataflows are conditional sub-samplings using the `when` operator. The resulting dataflows have the respective clocks `base` **on** `true(c)` and `base` **on** `false(c)`. Finally, the output `o` is the result of merging the complementary dataflows together to obtain a new dataflow that has clock `base`.

```
node main(i,j,k,c) returns (o)
var x,y,z;
let
  x = f(i, j);
  y = x when true(c);
  z = k when false(c);
  o = merge(c, true->y, false->z);
tel
```

The timing diagram below illustrates a possible trace of this program. It illustrates the values carried by the dataflows over time. Note that it views the values carried by the dataflow from the point of view of logical time, i.e. computations are supposed to execute instantaneously.

Inputs `i` and `c` have clock `base` and are thus present at each instant. Dataflow `x` is the result of applying the scalar function f pairwise on the values of `i` and `j`. Since the input rate defines the output rate, it also has clock `base`. Dataflows `y` and `z` are not present at each instant because they are on clocks `base` **on** `true(c)` and `base` **on** `false(c)` respectively.



6.3 Signal

The SIGNAL language [15, 52, 54] is a synchronous dataflow language similar to LUSTRE. However, clocks are first-class objects in SIGNAL. Thus, it is possible to express *polychronous* systems, i.e. systems where clocks do not form a single hierarchy. In contrast, LUSTRE is said to be *endochronous*.

Programmers can manipulate clocks the following ways:

1. Any boolean dataflow, called *signal* in the language's lingo, can be lifted to a clock. In LUSTRE, the right-hand side of a `when` expression is always a data constructor and a variable. The LUSTRE expression `e when C(c)` is written in SIGNAL `e when (C = c)`. More complicated expressions such as `e when (n >= 1)` are also possible. The expression `event e` returns the clock of `e`, it is a short-hand for `true when (e=e)`.

2. In addition to dataflow equations `v := e` where v is a variable and e an expression, SIGNAL allows for *clock equations*. An equation `v ^= e` specifies that `v` and `e` have the same clock. Equation `x ^= y ^+ z` specifies that the clock of `x` is the union of the clocks of `y` and `z`. Dataflow equations themselves also introduce clock equations. For instance, equation `v := e when c` introduces the equation `v ^< e`, meaning `v` is at most as frequent as `e`.

To verify the clock consistency of a polychronous program, the clock calculus solves the set of all clock equations. While decidable, solving such equations is NP-hard. To mitigate this, efficient strategies have been proposed that trade-off completeness for efficiency [7, 88]. These strategies do not accept all correct programs, but, for typical systems, the clock calculus efficiently verifies consistency. In [97] an approach for basing the clock calculus of SIGNAL on refinement types has been proposed.

For real-time systems, *affine clock transformations* [88] are of particular interest. An affine clock transformation is a tuple (n, φ, d). Two clocks H and K are related by such a transformation, if there is a clock P such that: 1. `H ^< P` and `K ^< P`; 2. The $i-th$ instant of H is the $n*i-th$ instant of P; 3. The $i-th$ instant of K is the $(d*i+\varphi)-th$ of P. This form of clocks allows to express efficiently

the relations between periodic tasks. However, this formalism does not directly relate instants to physical time. Moreover, compilation of such a model to multi-task code has not been studied to our knowledge.

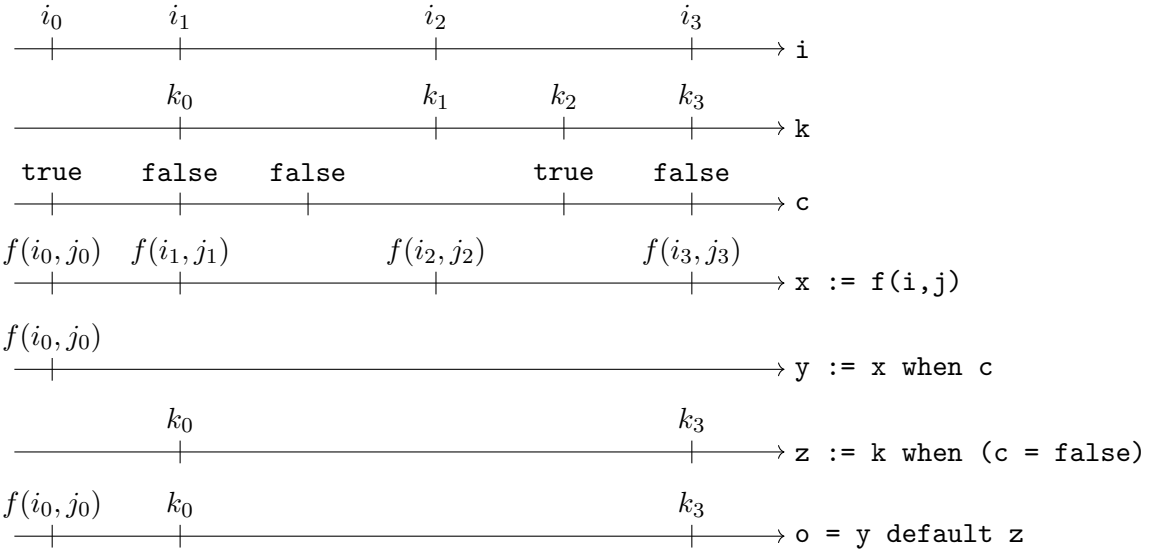
Example 5 (Signal)**.** Let us discuss the implementation of Example 4 in SIGNAL. It could be implemented as shown below. Instead of defining a node, a SIGNAL program is a *process* defined via the parallel composition of equations.

We already discussed the `when` construct. However, signal does not feature a `merge` expression. Instead, the `default` operator performs deterministic merge of arbitrary expressions. Expression `y default z` is the union of the values of `y` and `z`. In case both dataflows are present simultaneously, `y` takes priority.

The clock equation added at the end of the example is not strictly required. We added it to guarantee that both examples exhibit the same behavior. Without it, the example becomes polychronous. Variables `i` and `j` must have the same clock because of they are arguments of `f`, but beyond that, variables may have unrelated clocks.

```
process main =
  { ? int i, j, k; bool c
    ! int o }
(| x := f(i, j)
 | y := x when c
 | z := k when (c = false)
 | o := y default z
 | o ^= i ^= j ^= k ^= c
 |)
```

The timing diagram below illustrates a possible execution trace of the program without the clock equation. As can be seen, without added restriction, the output is not guaranteed to be as frequent as the inputs.



6.4 Prelude

As shown in Section 6.2 and Section 6.3, the previously presented dataflow languages possess some limitations in their ability to directly express real-time constraints.

The PRELUDE language [36, 67] solves this issue by introducing *strictly periodic clocks*. Using the tagged-signal model [53], each instant has a tag corresponding to its date, i.e. the first instant has tag/date 0, the second 1, etc. A strictly periodic clock (n, p) produces its i-th tag at date $p + i * n$. Note that a clock (n, p) has its i-th tag when a task with $T = n$ and $O = p$ releases its i-th job. These similarities allow for efficient implementation as a set of dependent real-time tasks [35].

To bridge the gap between dataflows of different clocks, the language defines *rate-transition operators*. The expression `e*^k` produces a dataflow that is k times as fast as `e` by repeating each value of `e` k times. If `e` has clock (n, p), `e*^k` has clock $(n/k, p)$. In contrary, the expression `e/^k` produces a dataflow that is k times slower than `e` by only preserving the first of k successive values. It has clock $(n * k, p)$. Note that rate-transition operators introduce new constraints. For instance, `e*^k` is well-formed, iff n is a multiple of k. It is the role of the clock calculus to verify the consistency the resulting dataflows.

A current limitation of the language that will be addressed in this thesis is that the operators `when` and `merge` directly inherit their definition from LUSTRE. This implies two restrictions. First, the arguments of a `when` or `merge` must all have the same clock. Second, it is not possible to apply a rate-transition operator after applying a `when`.

Chapter 7 will provide an in-depth presentation of a base kernel of the PRELUDE language used throughout this thesis.

Example 6 (Prelude)**.** Let us discuss the implementation of Example 4 in PRELUDE. The language reuses a syntax similar to LUSTRE for most syntactic constructs. To better illustrate the language, we introduced multi-periodicity in our example. Dataflows `i`, `c`, `o` all have clock $(10, 0)$ (for `c` and `o` it is inferred by the clock calculus). Since `j` has clock $(20, 0)$, it cannot be applied directly as an argument to `f` with `i`. A rate-transition is applied first.

```
-- Clocks inferred by the compiler:
-- c: rate (10,0)
-- o: rate (10,0)
-- x: rate (10,0)
-- y: rate (10,0) on true(c)
-- z: rate (10,0) on false(c)
node main(i: rate (10,0); j: rate(20,0); k: rate ( 5,0); c)
returns (o)
var x, y, z;
let
  x = f(i,j*^2);
  y = x when true(c);
  z = (k/^2) when false(c);
  o = merge(c, true->y, false->z);
tel
```

6.5 Imperative synchronous languages

Instead of relying on a dataflow semantics, the ESTEREL language [17] relies on *control-flow semantics*. A program is composed of parallel imperative threads that communicate via deterministic signals. This paradigm is especially suited for systems that do not continuously sample their environment but instead react to individual events.

Example 7 (Esterel)**.** The program below illustrates the typical ABRO example. It implements an ESTEREL module with three inputs, `A`, `B`, and `C`, and one output `O`. The module implements an infinite loop. The loops restarts when signal `R` is present, even if the loop body didn't finish executing. In the first instruction of the loop body, the program spawns two threads that respectively wait for `A` and `B` to be present. The parallel construct terminates when all threads terminated executing. Both signals need not be present within the same reaction of the system. When both signals have been present, the module emits its output `O`.

```
module ABRO:
input A, B, R;
output O;
loop
  [ await A || await B ];
  emit O
each R
end module
```

The FOREC language [46, 110] is a C-like language that introduces deterministic concurrency via ESTEREL-constructs.

Recently, the model of *Sequentially Constructive Concurrency* [43, 89] (SCC) has gained traction as a conservative extension of control-flow semantics that is easier to reason about than the model of ESTEREL, increasing the number of accepted programs. The language achieves this by allowing concurrent read and write operations on the same variable. In non-synchronous languages, this would lead to non-determinism and race conditions. In classical ESTEREL, the compiler would reject such a program. In SCC, the language offers mechanisms to describe how these concurrent operations should be resolved. In particular, SCCHARTS is of relevance to our work because of its ability to describe multi-mode systems.

6.6 Related Works on Multi-Mode Systems

The need for multi-mode real-time systems has motivated language designers to conceive language constructs for such systems. In this Section, we will discuss solutions proposed by these languages, compare them to our contribution and describe their ability using the terminology introduced in Chapter 5.

6.6.1 Lustre

In [28] a conservative extension of the LUSTRE and LUCID SYNCHRONE [25] languages introduces *Synchronous State Machines* as a way to implement multi-mode systems. The extension relies on

a *transpilation*[1] process that accepts programs in a surface language with automata and produces programs in a core language without. In [27], a semantics for these automata has been provided that does not rely on the translation process.

Our method relies on a fairly similar translation process. However, our contribution allows to express systems with multiple rates of execution.

The mode change protocol offered by Synchronous State Machines is a non-overlapping, periodic, late-retirement protocol. It is similar to a special case of the minimum single offset protocol where all tasks share the same period.

Example 8 (State machines in Lustre)**.** The automaton below implements a crossbar switch. It has two inputs, `i` and `j`, and two outputs, `o` and `p`. The state of the automaton dictates to which output the inputs are redirected. A switch from one state to another is triggered when `c` evaluates to true. In state `S1`, this transition is *strong*, i.e. if `c` is true the switch is immediate. In state `S2`, this transition is *weak*, i.e. the switch to the new mode is taken only in the next instant.

```
node switch(i, j, c) returns (o,p)
let
  automaton
  | S1 ->
    unless c then S2;
    o = i;
    p = j;
  | S2 ->
    o = j;
    p = i;
    until c then S1;
  end
tel
```

Below we have the result of the transpilation process. It introduces three variables `s`, `ns` and `pns`. Variable `s` holds the current automaton *state* which dictates the equations to evaluate. Variable `ns` holds the automaton's *next state*, i.e. the state the automaton will enter into at the beginning of the next instant. If a transition is triggered, it holds the destination state of that variable, otherwise it holds the current state. The variable `pns` holds the *previous next state*, i.e. the value of `ns` at the previous instant. This is thus the state the automaton should enter, if no strong transition is triggered.

The transpilation process guarantees that only one set of equations is evaluated by introducing *projected* equations and then merging those partial dataflows. For instance, `o` is defined in state `S1` as `o = i;` and in state `S2` as `o = j;`. The transpilation process thus introduces two equations `o_S1 = i when S1(s);` and `o_S2 = j when S2(s);`. In each of those equations, a `when` is applied to the inputs, `i` and `j`. Thus, equation `o_S1` and `o_S2` are only evaluated when the automaton is in state `S1` and `S2` respectively. The dataflow `o` is then created by merging these equations `o = merge(s, S1->o_S1, S2->o_S2);`.

```
node switch(i, j) returns (o,p)
let
```

[1] From *trans-* and *compilation*, a compilation process where both the input and output are "source" programming languages.

```
  pns = S1 fby ns;
  s = merge(pns, S1->s_S1, S2->s_S2);
  ns = merge(s, S1->ns_S1, S2->s_S2);

  o = merge(s, S1->o_S1, S2->o_S2);
  p = merge(s, S1->p_S1, S2->p_S2);

  s_S1 = if (c when S1(pns)) then S2 else S1;
  s_S2 = S2;

  ns_S1 = S1;
  ns_S2 = if (c when S2(s)) then S1 else S2;

  o_S1 = i when S1(s);
  o_S2 = j when S2(s);

  p_S1 = j when S1(s);
  p_S2 = i when S2(s);
tel
```

6.6.2 Signal

In [96], an extension of SIGNAL with State Machines similar to [28] is provided. The mechanism is polychronous in the sense that dataflows within the system may operate on unrelated clocks. However, the dataflow defining the automaton state is required to be present at each instant where the automaton inputs are present.

Our contribution also allows for dataflows within states with different rates of execution. However, our contribution requires that partial definitions of one dataflow to share the same clock across modes.

The mode change protocol implemented by Polychronous State Machines is also a non-overlapping, periodic, late-retirement protocol similar to the minimum single offset protocol. This is due to the fact that even tough SIGNAL is polychronous, it follows a strict definition of logical time similar to LUSTRE: any computation started within an instant terminates before the next one. Strong transitions are resolved before any computations are started within that instant such that the first non-transition computation belongs to the new state. Weak transitions are resolved after all computations within that instant terminated. Hence, the system is in the new state at the beginning of the next instant.

Example 9 (State machines in Signal)**.** Let us illustrate this by implementing the `switch` example in SIGNAL.

```
process switch =
  { ? i, j, c
    ! o, p }
(| init S1 : (| o = i | p = j |)
 |      S2 : (| o = j | p = i |)
 | c => S1 ->> S2
 | c => S2 ->  S1
 |)
```

The first two lines implement the states `S1` and `S2`. The lower two lines implement transitions. An arrow `->>` denotes a strong transition, while `->` denotes a weak transition.

No clock constraint exists between `i`, `j` and `c`. Thus, `o` can produce values at a different rate between modes `S1` and `S2`. This is not possible in our contribution. The clock calculus would require `i` and `j` to have the same clock.

6.6.2.1 Prelude

Being similar to the Lustre programming language, the technique presented in [28] can be applied to construct synchronous state machines in Prelude. However, the semantics of the **`when`** and **`merge`** operators, borrowed from Lustre, do not allow to mix multi-periodicity with state machines. All arguments of a **`when`** and **`merge`** must share the same clock. Furthermore, rate-transition operators may not be applied on arguments whose clock feature an **on**.

Until this work, the issue of how multi-periodicity and mode changes interact in a synchronous language had been identified, but not tackled [34]. Part IV of this work will tackle this problem in detail.

6.6.2.2 Esterel

Historically, control-flow languages have been considered better suited to implement systems relying on state machines until contributions such as [28]. State machines can be written fairly easily without relying on explicit constructs, legacy code may in fact hide "implicit state machines" [56]. Specialized constructs for state machines have been proposed for control-flow synchronous languages such as SyncCharts [8] and SCCharts [104].

6.6.2.3 Non-synchronous languages

Statecharts This graphical language [44] describes multi-moded reactive systems. The language does not directly target real-time systems, but certain features have been developed with such systems in mind. A statecharts program is a hierarchical and/or orthogonal composition of *states*. A Statecharts observes *events* which trigger the transition from one state into another. Computations performed by a statecharts program are only loosely defined via *actions*. An action can be performed upon *entry* of, *exit* of, or *throughout* a state. Which data such actions consume or produce is not specified. In general, the semantics was poorly defined. Leading to numerous contradictory implementations of Statecharts variants [103]. For instance, it is unclear how to handle *instantaneous states*, i.e. how to handle when upon entering a state, another transition is possible. The original semantics leave this unclear. In particular, the Argos language [60] introduces concepts of synchronous languages to offer a well-formed semantics, e.g. forbidding instantaneous states and restricting transitions to a finite number. Certain semantics however [49] impose no such constraint at all.

Giotto Multi-mode systems are at the core of Giotto [45]. The language relies on what is called *Logical Execution Time* (LET). The LET model is similar to the model of synchronous languages. However, it requires that jobs acquire their inputs when they are released and not when they start executing. This means that all tasks need to communicate via buffers which are pre-filled with a default value at the start of the system. This leads to poor data freshness in communication chains: the n-th task in such a chain will have to wait its $(n+1)$-th activation before it can compute a value that depends upon the first sensor reading.

The mode change protocol offered by Giotto is a non-overlapping, periodic, late-retirement protocol. Mode changes may be performed at multiples of the hyperperiod of the tasks affected by the mode change.

Giotto also being a multi-periodic language, it is possible to draw some similarities with our contribution. In the case where our clock calculus opted for the non-overlapping protocol, both protocols are similar. However, the absence of forced communication delays and ability to perform mode overlap are improvements over Giotto's protocol.

AADL The "Architecture Analysis & Design Language" [16] is an architecture language that allows to model the hardware and software components of an embedded real-time system. While it allows for tasks of different periods within a single mode, its mode change protocol features poor promptness. To guarantee soundness of communications, when a MCR is received, it first waits for a synchronization point between the old-mode tasks, which takes in the worst-case a full hyperperiod. Then, the mode transition begins which may take up to one hyperperiod to complete. This protocol is thus a non-overlapping, periodic, late-retirement protocol.

The most notable property of this language is the poor promptness of the mode change protocol. Requiring in the worst-case two hyperperiods to complete a mode change is not required in any of the languages reviewed in this Section.

6.7 Related Works on Multicore Systems

6.7.1 Prelude

In [68] a compilation scheme for predictable execution of Prelude on a multicore platform is presented. However, their solution features design choices, such as offline computed non-preemptive schedule and bare metal execution, that are specific to their targeted avionics platform. These design choices are necessary to pass strict standards.

In comparison, our solution is not compatible with those standards, but allows to implement real-time systems for less strict applications.

6.7.2 ForeC

The ForeC language [110] is C-like language extended with a synchronous semantics similar to Esterel. In [46], a compilation scheme for ForeC is proposed. It relies on a *PREcision Timed* (PRET) architecture, i.e. an architecture designed to experience as little execution time variability as possible. Access to the global memory is arbitraged via a Time-Division Multiple Access (TDMA) scheme.

As for the previous contribution, this contribution restricts itself to a very specific application (PRET architecture with TDMA) which our contribution doesn't.

6.7.3 OS-level solutions

With the OS-level approaches of [90, 93–95, 105], we observe similarities between the RTOS API exposed by a PREM-compliant OS and our proposed high-level language. However, we believe that leveraging a high-level language brings advantages such as better static analysis tools via specialized type systems.

6.8 Summary

In this Chapter, we reviewed languages dedicated to real-time systems. In particular, we studied synchronous languages which our contribution builds upon. Recall their key properties:

1. Synchrony to abstract time as a sequence of discrete instants;
2. Deterministic concurrency via explicit language constructs;
3. Simple formal underlying models to make verification tractable.

Chapter 7

A base synchronous language: Definitions and reminders

In this Chapter, we will present the semantics of the synchronous language Prelude [67]. This language should be understood as a *core language* that is manipulated by the compiler. The programmer uses a *surface language* that is easier to manipulate and whose conversion into the core language is fairly trivial (e.g. expressions converted in Normal Form).

We will use the following notations:

Definition 12 (Divisibility constraint)**.** The relation x **div** $y \Leftrightarrow y \bmod x = 0$, reads "$x$ divides y".

Definition 13 (Sequences)**.** A sequence $s = hd.tl$ is composed of a *head* hd and a *tail* tl. The head is a value, while the tail is another sequence. The $\prod$ operator extends the . operator over ranges. For instance, $4.7.10.13 = \left(\prod_{i=0}^{2}(4 + 3 * i)\right).13$.

If a total ordering of sequence elements is possible, e.g. for a sequence of strictly increasing instants, we use a set-theoretic notation. For instance, we have $s = 4.7.10.13 = \{4 + 3 * i \mid i \in \mathbb{N}, 0 \leqslant i \leqslant 3\}$ and $10 \in s$.

7.1 Clocks and Dataflows

In this Section, we will formally define *clocks* and *dataflows* which have been illustrated in Chapter 6.

Time is represented as a sequence of equidistant *instants*. Following the tagged-signal model [53], we associate to each instant a tag corresponding to its date, i.e. the first instant has tag/date 0, the second 1, etc. A *clock* is a sequence of instants. A clock ck is *present* at an instant t, iff $t \in ck$.

Timing constraints are specified using a specific class of clocks called *strictly periodic clocks*.

Definition 14 (Strictly Periodic Clock)**.** A strictly periodic clock is denoted as a pair (n, p), with $n \in \mathbb{N}^+, p \in \mathbb{N}$, and:

- The infinite sequence of tags generated by (n, p), denoted $(n, p)^{\#}$, is defined as follows: $(n, p)^{\#} = \{i * n + p \mid i \in \mathbb{N}\}$.
- $\pi((n, p)) = n$ is the *period* and $\varphi((n, p)) = p$ is the *offset* of (n, p).

Clock operators allow to construct a strictly periodic clock from another. The acceleration operator $(ck *. k)$ produces a clock that is present k times for each instant where ck is present. The deceleration operator $(ck /. k)$ produces a clock that is present once for each k instants where ck is present. The delay operator $(ck \rightarrow. k)$ produces a clock that is present k instants after ck. Note that clock operations are only well-defined for certain clocks. For instance, $ck *. k$ is defined, iff k **div** $\pi(ck)$.

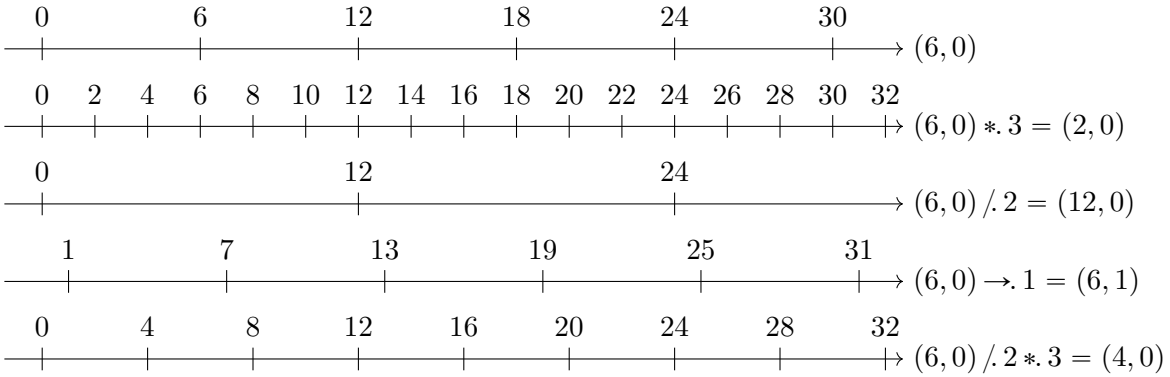
Definition 15 (Periodic Clock Operators)**.**

$$
\begin{aligned}
(ck *. k)^{\#} &= \{t \mid t' \in ck^{\#}, i \in [\![0..k[\![, t = t' + i\frac{\pi(ck)}{k}\} && \text{, if } k \textbf{ div } \pi(ck)\\
(ck /. k)^{\#} &= \{t \mid t \in ck^{\#}, i \in [\![0..k[\![, t = \pi(ck) * k * i + \varphi(ck)\}\\
(ck \rightarrow. k)^{\#} &= \{t \mid t' \in ck^{\#}, t = t' + k\} && \text{, if } \varphi(ck) + k \geqslant 0
\end{aligned}
$$

$$
(n, p) *. k = (n/k, p) \qquad (n, p) /. k = (n * k, p) \qquad (n, p) \rightarrow. k = (n, n + k)
$$

$$
\begin{aligned}
\pi(ck *. k) &= \pi(ck)/k & \varphi(ck *. k) &= \varphi(ck)\\
\pi(ck /. k) &= \pi(ck) * k & \varphi(ck /. k) &= \varphi(ck)\\
\pi(ck \rightarrow. k) &= \pi(ck) & \varphi(ck \rightarrow. k) &= \varphi(ck) + k
\end{aligned}
$$

Example 10 (Periodic clocks)**.** Below, we illustrate the resulting clocks of applying different periodic clock operators to the clock $(6, 0)$. The timing diagram displays the instants where each clock is present, each "clock tick" being annotated with the respective instant.



Clocks allow us to specify *when* things happen, but do not carry any meaning about *what* is happening. *Dataflows* produce infinite sequences of tagged values. Meaning that they carry the information at which date certain values are produced. We define them as follows.

Definition 16 (Dataflows)**.** The dataflow s produces an infinite sequence of tagged values denoted $s^{\#} = (v, t).s'^{\#}$. Its head (v, t) is composed of the value v produced at tag t and tail is $s'^{\#}$.

Because tags provide a total ordering over value-tag pairs, we can use a set-theoretic notation for flows $s = \{(v, t) \mid (v, t) \in s^{\#}\}$. We denote (v_n, t_n) the n-th value (according to the tag ordering relation) v_n of a flow produced at its n-th tag t_n.

The clock of a dataflow s is its sequence of tags and denoted $\hat{s} = \{t \mid (v, t) \in s^{\#}\}$.

Intuitively, when a dataflow s produces a pair (v, t), the value v is the value carried by the flow until date $t + \pi(\hat{s})$. In the case where there is no conditional sub-sampling (Section 7.1.1), this is until the next next value-tag pair is produced.

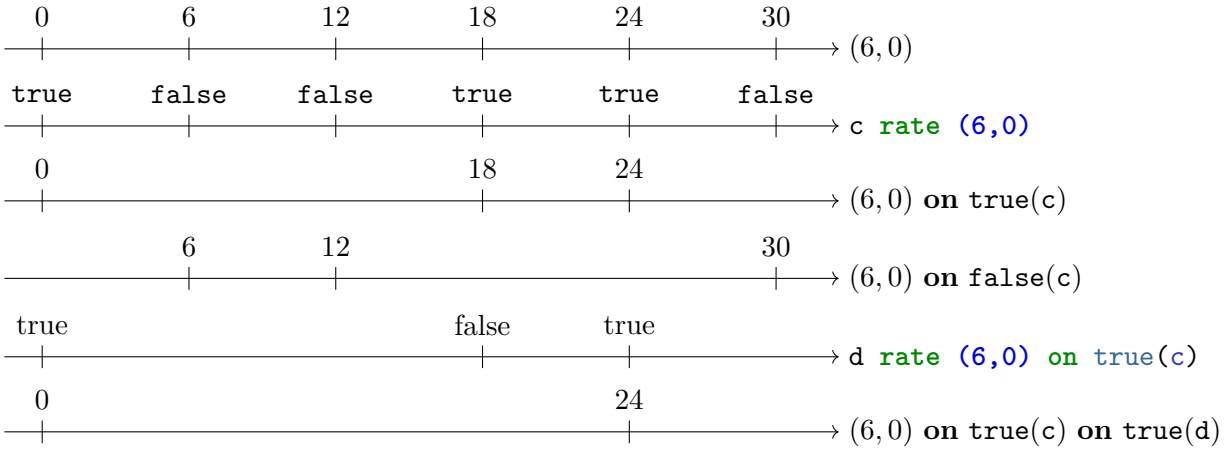
7.1.1 Mono-Periodic Conditional Sub-Sampling

Conditional sub-sampling allows to sub-sample a clock such that it is only present when another dataflow produces a specified value. We present here the mono-periodic definition of conditional sub-sampling as used in existing synchronous languages, and also in PRELUDE prior to our work. The multi-periodic definition will be presented in Part IV.

Definition 17 (Mono-periodic conditional sub-sampling)**.** A clock ck **on** $C(c)$ denotes a clock ck that is sub-sampled on condition dataflow c. It produces a tag t, iff ck produces that tag t and dataflow c produces at t the value C.

$$(ck \textbf{ on } C(c))^{\#} = \{t \mid t \in ck^{\#} \wedge (C, t) \in c^{\#}\}$$

Example 11 (Mono-periodic conditional sub-sampling)**.** Below, we illustrate some sub-samplings of clock $(6, 0)$ according to different dataflows



7.2 Language Syntax

Figure 7.1 details the syntax of the PRELUDE language. A program is a collection of *declarations*, either a type declaration, node import or user-defined node declaration.

A *type declaration* allows to define an enumeration type which is used by the `when` and `merge` expressions (and will be used by automatons). A node is a synchronous dataflow language's equivalent of a function. Instead of operating on scalar values, it operates on streams of values. An *imported node* lifts a function over scalar values from the target language (e.g. C) to flows by performing a point-wise application of the function. A *user-specified node* defines via equations how local and output variables are computed from input variables.

An *equation* of the form `x = e;` defines variable `x` as equal to expression `e`. Unlike statements in imperative languages, equations are unordered.

An *expression* can be a variable (`x`), a constant (`42`), a node application (`f(a,b)`), a clock annotation (`e rate (10,0)`) or the result of the application of one of the following built-in operators (assuming `a` has clock ck):

- `a/^k` keeps the first out of k successive values of `a` and has clock $ck \,/.\, k$;
- `a*^k` repeats each value of `a` k times and has clock $ck \,*.\, k$

- `a~>k` delays each value of `a` by k and has clock $ck \rightarrow .\, k$
- `cst fby a` produces the value `cst` followed by the values of `a` and has clock ck, effectively delaying values of `a` by $\pi(ck)$
- `cst::a` produces the values of `a` prepended by the value `cst` and has clock $ck \rightarrow .\, -\pi(ck)$
- `tail a` skips the first value of `a` and has clock $ck \rightarrow .\, \pi(ck)$
- `a when C(c)` sub-samples `a` such that it produces values only if `c` produces `C`. It has clock ck **on** $C(c)$. Note that without our contribution (Part IV), `c` must have clock `c`.
- `merge(c, C0->a_c0, C1->a_c1)` combines the complementary flows `a_c0` and `a_c1` with respective clocks ck **on** $C0(c)$ and ck **on** $C1(c)$ and has clock ck.

$$
\begin{aligned}
\langle prog \rangle &::= \langle decl \rangle* \\
\langle decl \rangle &::= \langle nd \rangle \mid \langle ind \rangle \mid \langle tydecl \rangle \\
\langle nd \rangle &::= \text{`}\texttt{node}\text{'}\ \langle id \rangle\ \text{`}\texttt{(}\text{'}\ \langle vars \rangle\ \text{`}\texttt{)}\text{'} \\
&\quad\ \text{`}\texttt{returns}\text{'}\ \text{`}\texttt{(}\text{'}\ \langle vars \rangle\ \text{`}\texttt{)}\text{'} \\
&\quad\ (\text{`}\texttt{var}\text{'}\ \langle vars \rangle\ \text{`}\texttt{;}\text{'})? \\
&\quad\ \text{`}\texttt{let}\text{'}\ \langle eq \rangle+\ \text{`}\texttt{tel}\text{'} \\
\langle ind \rangle &::= \text{`}\texttt{imported}\text{'}\ \text{`}\texttt{node}\text{'}\ \langle id \rangle\ \text{`}\texttt{(}\text{'}\ \langle vars \rangle\ \text{`}\texttt{)}\text{'} \\
&\quad\ \text{`}\texttt{returns}\text{'}\ \text{`}\texttt{(}\text{'}\ \langle vars \rangle\ \text{`}\texttt{)}\text{'} \\
&\quad\ \langle indprop \rangle\ \text{`}\texttt{;}\text{'} \\
\langle tydecl \rangle &::= \text{`}\texttt{type}\text{'}\ \langle id \rangle\ \text{`}\texttt{=}\text{'}\ (\text{`}\texttt{|}\text{'}\ \langle id \rangle)+ \\
\langle eq \rangle &::= \langle id \rangle(\text{`}\texttt{,}\text{'}\ \langle id \rangle)*\ \text{`}\texttt{=}\text{'}\ \langle expr \rangle \\
\langle expr \rangle &::= \langle atom \rangle \mid \langle id \rangle\ \text{`}\texttt{(}\text{'}\ \langle atom \rangle(\text{`}\texttt{,}\text{'}\ \langle atom \rangle)*\ \text{`}\texttt{)}\text{'} \mid \langle atom \rangle\ \text{`}\texttt{rate}\text{'}\ \langle ck \rangle \\
&\quad \mid \langle atom \rangle\ \text{`}\texttt{*\^{}}\text{'}\ \langle int \rangle \mid \langle atom \rangle\ \text{`}\texttt{/\^{}}\text{'}\ \langle int \rangle \mid \langle atom \rangle\ \text{`}\texttt{\~{}>}\text{'}\ \langle int \rangle \\
&\quad \mid \langle const \rangle\ \text{`}\texttt{fby}\text{'}\ \langle atom \rangle \mid \langle const \rangle\ \text{`}\texttt{::}\text{'}\ \langle atom \rangle \mid \text{`}\texttt{tail}\text{'}\ \langle atom \rangle \\
&\quad \mid \langle atom \rangle\ \text{`}\texttt{when}\text{'}\ \langle id \rangle\ \text{`}\texttt{(}\text{'}\ \langle id \rangle\ \text{`}\texttt{)}\text{'} \\
&\quad \mid \text{`}\texttt{merge}\text{'}\ \text{`}\texttt{(}\text{'}\ \langle id \rangle\ (\text{`}\texttt{,}\text{'}\ \langle id \rangle\ \text{`}\texttt{->}\text{'}\ \langle atom \rangle)+\ \text{`}\texttt{)}\text{'} \\
\langle atom \rangle &::= \langle id \rangle \mid \langle const \rangle \\
\langle vars \rangle &::= \langle var \rangle \mid \langle var \rangle\ \text{`}\texttt{;}\text{'}\ \langle vars \rangle \\
\langle var \rangle &::= \langle id \rangle \mid \langle id \rangle\ \text{`}\texttt{:}\text{'}\ \langle typ \rangle?\ (\text{`}\texttt{rate}\text{'}\langle ck \rangle)? \\
\langle typ \rangle &::= \text{`}\texttt{int}\text{'} \mid \text{`}\texttt{bool}\text{'} \mid \langle typ \rangle\ \text{`}\texttt{[}\text{'}\ \langle int \rangle\ \text{`}\texttt{]}\text{'} \mid \ldots \\
\langle ck \rangle &::= \text{`}\texttt{(}\text{'}\ \langle int \rangle\ \text{`}\texttt{,}\text{'}\ \langle int \rangle\ \text{`}\texttt{)}\text{'} \mid \langle ck \rangle\ \text{`}\texttt{on}\text{'}\ \langle id \rangle\ \text{`}\texttt{(}\text{'}\ \langle id \rangle\ \text{`}\texttt{)}\text{'} \\
\langle indprop \rangle &::= \langle wcet \rangle? \\
\langle wcet \rangle &::= \text{`}\texttt{wcet}\text{'}\ \langle int \rangle
\end{aligned}
$$

Figure 7.1: Syntax of the PRELUDE language

7.3 Running Example

In this Section, we will illustrate the running example used in this Chapter and in Part III, Part IV will have dedicated examples.

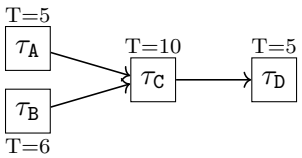
The PRELUDE code below defines a system with two sensors, `A` and `B`, and one actuator, `D`. An imported node `C` takes as input the rate-transitioned dataflows of these sensors, producing the dataflow `tmp`. The dataflow provided to `D` is the result of up-sampling dataflow `tmp`.

```
imported node C(i,j: int) returns (o: int) wcet 2;
sensor A wcet 1;
sensor B wcet 1;
actuator D wcet 1;

node main(A: int rate (5,0); B: rate (6,0)) returns (D: int)
var tmp;
let
  tmp = C(A/^2, B*^3/^5);
  D = tmp *^2;
tel
```

The PRELUDE compiler then produces the following real-time system from this program. Section 7.6 will go into more detail how it is produced. For now note how only the following constructs produce actual tasks:

- Node inputs produce sensor tasks;
- Node outputs produce actuator tasks;
- Imported node calls produce intermediate tasks.



τ_i	O_i	T_i	D_i	C_i
τ_A	0	5	5	1
τ_B	0	6	6	1
τ_C	0	10	10	2
τ_D	0	5	5	1

$$\mathcal{D}^{pat}_{A,C} = \{(\tau^0_A, \tau^0_C)\} \qquad I^{pat}_{A,C} = 10$$
$$\mathcal{D}^{pat}_{B,C} = \{(\tau^0_B, \tau^0_C), (\tau^1_B, \tau^1_C), (\tau^3_B, \tau^2_C)\} \qquad I^{pat}_{B,C} = 30$$
$$\mathcal{D}^{pat}_{C,D} = \{(\tau^0_C, \tau^0_D), (\tau^0_C, \tau^1_D)\} \qquad I^{pat}_{C,D} = 10$$

7.4 Synchronous Kahn Networks

In this Section, we present the semantics of the core language based on synchronous Kahn networks [26, 48]. The term $\diamond^{\#}(s_0, \ldots, s_n)$ denotes the flow resulting from the application of the operator $\diamond$ on flows $s_0, \ldots, s_n$. Operators fall into three categories: imported operators, rate-transition operators and

conditional operators. The semantics for the first two categories remain as previously defined in [36] and are recalled below.

An imported operator $op^{\#}$, is an operator op_f over scalars from the target language (e.g. C) that has been lifted to operate on dataflows. Thus, it is the direct equivalent to the application of an imported node in the source language.

Definition 18 (Kahn semantics of imported operators)**.**

$$op^{\#}(s_0, \dots, s_n) = \{(op_f(v_0, \dots, v_n), t) \mid (v_0, t) \in s_0^{\#}, \dots, (v_n, t) \in s_n^{\#}\}$$

Rate-transition Kahn operators are directly equivalent to the rate-transition operators in the source language.

Definition 19 (Kahn semantics of rate-transition operators)**.**

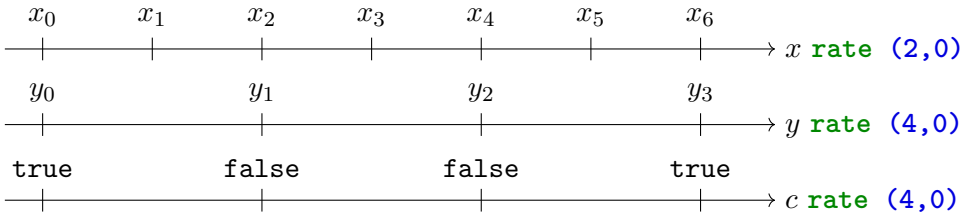
$$\begin{aligned}
*^{\wedge\#}(s,k) &= \{(v, t + i * \pi(\widehat{s})/k) \mid (v,t) \in s^{\#}, i \in [\![0..k[\![\ \} \\
/^{\wedge\#}(s,k) &= \{(v,t) \mid (v,t) \in s^{\#} \wedge t \in (\widehat{s} /. k)^{\#}\} \\
\sim>^{\#}(s,k) &= \{(v, t+k) \mid (v,t) \in s^{\#}\} \\
\texttt{fby}^{\#}(v,s) &= \{(v,t_0)\} \cup \{(v_i, t_{i+1}) \mid (v_i,t_i), (v_{i+1}, t_{i+1}) \in s^{\#}\} \\
::^{\#}(v,s) &= \{(v,t)\} \cup s \qquad \text{, where } t = t_0 \in \widehat{s} \rightarrow . \ -\pi(\widehat{s}) \\
\texttt{tail}^{\#}(s) &= \{(v_i, t_i) \mid (v_i,t_i) \in s^{\#}, i \neq 0\}
\end{aligned}$$

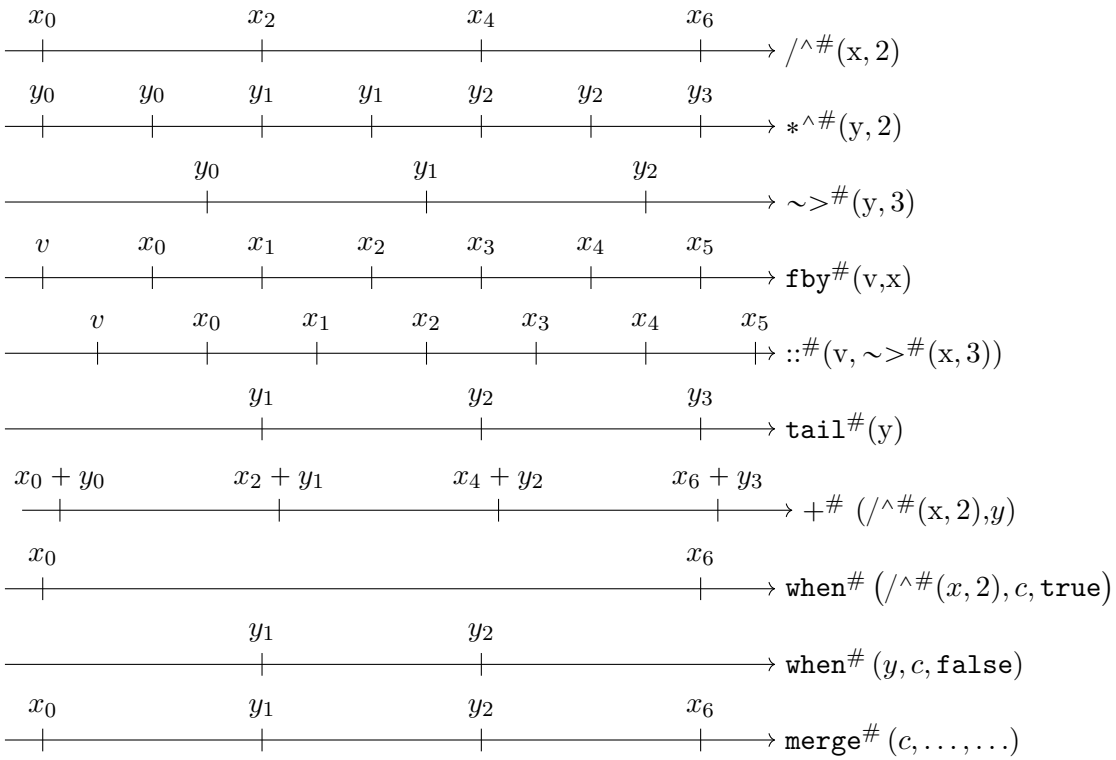
The $\texttt{when}^{\#}$ operator is a direct transposition of the **on** operator on flows. The $\texttt{merge}^{\#}$ combines flows that have complementary clocks.

Definition 20 (Kahn semantics of conditional operators)**.**

$$\begin{aligned}
\texttt{when}^{\#}\,(s, c, C) &= \{(v,t) \mid (v,t) \in s, t \in (\widehat{s} \ \textbf{on} \ C(c))^{\#}\} \\
\texttt{merge}^{\#}\,(c, s_0, \dots, s_n) &= \bigcup_{i=0}^{n} s_i^{\#}
\end{aligned}$$

Example 12 (Synchronous Kahn Semantics)**.**





7.5 Clock Calculus

Note that $\mathtt{merge}^{\#}$ is deterministic iff the merged flows are *complementary*, i.e. only one is present at any given instant. Similarly, the imported node application requires arguments that are *synchronous*, i.e. that have the same clock. Also, $*\hat{}^{\#}(s, k)$ is defined iff k **div** $\pi(\widehat{s})$. Finally, the semantics for $\mathtt{when}^{\#}$ and $\mathtt{merge}^{\#}$ require clock views, which are not specified by the program. Instead, they are inferred by the compiler.

A *clock calculus* is required to infer and check these properties. Previously, PRELUDE used a clock calculus based on Hindley-Milner typing extended with subtyping [78]. Readers may refer to [34, 36] for a presentation of this type system. Part IV will present a clock calculus based on refinement typing [39, 83] that supports our extensions.

7.6 Compiling Prelude Equations into Real-Time Tasks

The compilation target of a PRELUDE program is a set of real-time tasks [34, 67] implemented in a host language (e.g. C). The compiler operates in three steps: First it generates a *task graph*, second it transforms it into a *task set*, finally it produces host language code to be integrated in the OS.

The task graph generation starts with a direct translation of equations. The vertices of the graph are node inputs (future sensors), node outputs (future actuators) and operators and the edges are data-dependencies. A *graph reduction procedure* then gradually removes rate-transition and conditional operators from the vertices and replaces them by annotations on edges. The resulting task graph then only features vertices for sensors, actuators and imported operators.

The goal of the task set construction is to obtain a system that can be scheduled according to a classical scheduling policy such as EDF or DM. To obtain this, edges are replaced with *non-blocking*

communication protocols which dictate for each task which instance reads/writes into their buffers. Then, real-time attributes are adjusted such that the choices of the scheduling policy guarantee that the executing system respects the semantics of the communication protocols.

Finally, this task set can be fairly easily translated into host language code. For each task, a *step function* is generated. One execution of this function represents one job execution. Each step function follows the same structure: 1. If dictated by the read protocol, read from the appropriate buffers; 2. Call the user-provided function implementing the task functionality; 3. If dictated by the write protocol, read from the appropriate buffers.

7.7 Summary

In this Chapter, we presented the synchronous language Prelude. To this effect we defined:

- A synchronous model of time using clocks and dataflows producing data according to a model;
- A syntax for the core language of Prelude;
- A formal semantics based on synchronous Kahn networks.

Upon this basis, we will be able to present our contributions in Part III and Part IV.

Part III

Compiling Synchronous Languages on Multicore

Chapter 8

Introduction

In this Part, we will address the issue of implementing the presented language on a multicore platform.

As a reminder, a multicore platform is a computing platform composed of multiple cores which compete for access to a shared memory, the RAM. Because it is shared, multiple cores can attempt to access it simultaneously. However, the memory can only answer a limited amount of requests at a time. When there are more requests than the memory can serve, this is called a contention. Contentions block the affected cores until they may access the memory.

These delays are hard to quantify because they depend upon minute details within task codes, task interferences and the contention resolution mechanisms [72].

Multicore platforms also feature private memories. A private memory is accessible by only one processor. Thus, it is never subject to contentions. However, it is limited in size. The most common form of private memory is cache memory. It implicitly replicates the content of the global memory when the processor performs memory accesses, bypassing the need for accessing the global memory if the data is replicated in the cache. Certain embedded platforms also feature scratchpad memory (SPM). In contrast to caches, this form of memory is explicit. It can be accessed using memory addresses just like the RAM and must be managed by manual copy operations.

Tasks model such as PREM [71] and AER [32] advocate to decouple memory operations from computation operations into distinct phases. During memory phases, data is copied between the local and global memory. This allows the execution phase to execute using only the local memory, preventing any contention from happening, greatly improving their execution time predictability.

We will focus on the AER model where each task τ_i follows the sequence of phases Acquisition-Execution-Restitution. During the Acquisition phase A_i data is copied from local to global memory. This allows to perform the Execution phase E_i using only local memory. Finally, in the Restitution phase R_i results are copied back to global memory.

However, implementing these task models manually is tedious and error-prone. The goal of our contribution is to propose a compilation scheme that accepts a high-level synchronous program and produces multi-task AER-compliant task code. The generated code leverages scratchpad memories with little additional programming effort.

8.1 Platform Description

In this Section, we will present the makeup of our targeted platform, both software and hardware. We will also present the running example used throughout this Part.

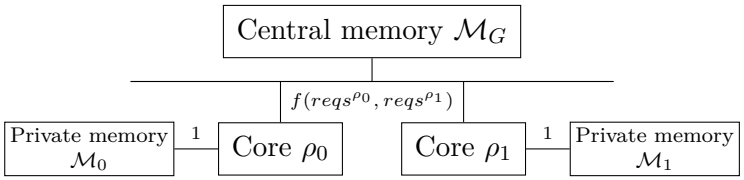


Figure 8.1: An example hardware component graph

8.1.1 Hardware

Similar to Section 4.2, we assume a platform with multiple cores, $\rho_i \in \Pi$. Cores compete for access to a shared memory $\mathcal{M}_G$. Each core ρ_i has access to a private scratchpad memory $\mathcal{M}_i$. Figure 8.1 displays an example hardware component graph of our kind of target platform. We assume the access cost to private scratchpad memories is constant, 1. Accurately bounding the access cost is out of scope of this work. Thus, we model access costs as a function $f(reqs^{\rho_0}, reqs^{\rho_1}) > 1$.

8.1.2 Software

Our platform features a real-time operating system (RTOS). The RTOS scheduler implements a partitioned preemptive policy such as Deadline-Monotonic or Earliest-Deadline First. Moreover, the RTOS offers mutexes and semaphores as primitives. The scheduler must handle them appropriately, e.g. using priority inheritance [22, 87] to prevent deadline misses. Mutexes and semaphores will be used to handle data-dependencies in Chapter 9.

8.2 Running Example

Throughout this Part, we will reuse the running example from Chapter 7, shown below. Recall that it defines a system with two sensors, `A` and `B`, and one actuator, `D`. An imported node `C` takes as input the rate-transitioned dataflows of these sensors, producing the dataflow `tmp`. The dataflow provided to `D` is the result of up-sampling dataflow `tmp`.

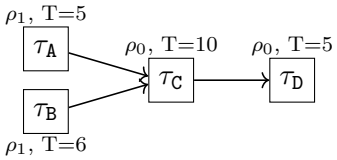
```
imported node C(i,j: int) returns (o: int) wcet 2;
sensor A wcet 1;
sensor B wcet 1;
actuator D wcet 1;

node main(A: int rate (5,0); B: rate (6,0)) returns (D: int)
var tmp;
let
  tmp = C(A/^2, B*^3/^5);
  D = tmp *^2;
tel
```

8.3 Contribution Goal

The goal of our contribution is to produce C code from a Prelude program in accordance with the AER model. For our running example, this would result in a real-time system shown below. Mapping

tasks to cores being out-of-scope of this work, we assume from here on that it is provided by the developer, potentially from an external tool. For how such a mapping could be produced, readers may refer to works such as [74, 80].



τ_i	O_i	T_i	D_i	C_i	π_i
$\tau_{\mathtt{A}}$	0	5	5	1	ρ_1
$\tau_{\mathtt{B}}$	0	6	6	1	ρ_1
$\tau_{\mathtt{C}}$	0	10	10	2	ρ_0
$\tau_{\mathtt{D}}$	0	5	5	1	ρ_0

$$\mathcal{D}^{pat}_{\mathtt{A},\mathtt{C}} = \{(\tau^0_{\mathtt{A}}, \tau^0_{\mathtt{C}})\} \qquad I^{pat}_{\mathtt{A},\mathtt{C}} = 10$$
$$\mathcal{D}^{pat}_{\mathtt{B},\mathtt{C}} = \{(\tau^0_{\mathtt{B}}, \tau^0_{\mathtt{C}}), (\tau^1_{\mathtt{B}}, \tau^1_{\mathtt{C}}), (\tau^3_{\mathtt{B}}, \tau^2_{\mathtt{C}})\} \qquad I^{pat}_{\mathtt{B},\mathtt{C}} = 30$$
$$\mathcal{D}^{pat}_{\mathtt{C},\mathtt{D}} = \{(\tau^0_{\mathtt{C}}, \tau^0_{\mathtt{D}}), (\tau^0_{\mathtt{C}}, \tau^1_{\mathtt{D}})\} \qquad I^{pat}_{\mathtt{C},\mathtt{D}} = 10$$

We will present our contribution in Chapter 9. Section 9.1 will extend the PRELUDE model with multi-phase communications. After discussing the monocore compilation process in Section 9.2, Sections 9.3 to 9.4 will present the compilation process from PRELUDE source code into AER-compliant C code.

Chapter 9

Code Generation

In this Chapter, we will present how code is generated for our multicore platform. We reuse notions from previous work [67] and will thus focus on aspects specific to multicore.

9.1 Multi-phase Communications

In this Section, we will present how we adapt the model of data-dependencies to multi-phase communications in accordance to the AER model [32]. As stated above, data-dependencies are causal, i.e. for all $\tau_i^n \rightarrow \tau_j^m$:

- $end(\tau_i^n) < begin(\tau_j^m)$;
- τ_i^n produces its outputs at $end(\tau_i^n)$;
- τ_j^m acquires its inputs at $begin(\tau_j^m)$.

Recall from Section 7.6 that when a PRELUDE program is compiled, the compiler produces one task per imported node call, main node input and output. However, to remain in accordance with the AER model, the compiler will now have to generate *phases* instead of tasks.

9.1.1 Precedence Constraints

Data-dependencies describe *communications*, i.e. $\tau_i^n \rightarrow \tau_j^m$ indicates that τ_i^n produces data consumed by τ_j^m. A *precedence constraint* $X_i^n \rightarrow Y_j^m$ describes that the Y phase of τ_j^m cannot execute before the X-phase of τ_i^n terminates. Both notions are distinct, but related. A data-dependency describes the semantics of a program, while a precedence constraint is a scheduling restriction to be complied with in order to respect the program semantics.

Section 9.1.2 will provide a base definition of how to construct phases and precedence constraints from a PRELUDE program. While this procedure is correct, we observed that it may be improved upon. Thus, Section 9.1.3 will expand upon it to provide a semantics-preserving optimization of this procedure. Our optimization will reduce the amount of shared memory accesses performed, at the cost of increased use of the private memories.

9.1.2 Base Definition

Before defining the precedence constraints between phases, we must first define the construction of our phases. In our base definition, we simply provide a A-, E- and R-phase for each job.

Definition 21 (Phase set)**.**

$$\mathcal{P} = \{A_i^n \mid \tau_i^n \in \mathcal{J}\} \cup \{E_i^n \mid \tau_i^n \in \mathcal{J}\} \cup \{R_i^n \mid \tau_i^n \in \mathcal{J}\}$$

With our phases defined, we can now define the precedence constraints that exist between those. We distinguish between intra-job precedence constraints and inter-job precedence constraints. Intra-job precedence constraints relate phases within the same job, while inter-job precedence constraints relate phases of different jobs.

Definition 22 (Precedence constraints)**.** The set of precedence constraints $\mathcal{P}_{prec}$ is the union of intra-job precedence constraints and inter-job precedence constraints.

$$\mathcal{P}_{prec} = \mathcal{P}_{intra} \cup \mathcal{P}_{inter}$$

For our implementation of the AER-model, the definition of $\mathcal{P}_{intra}$ is straight-forward. The A-phase must execute before the E-phase and the E-phase must execute before the R-phase. Since each job has all 3 phases, its definition is straight-forward.

Definition 23 (Intra-job precedence constraints)**.**

$$\begin{aligned}\mathcal{P}_{intra} = &\{A_i^n \rightarrow E_i^n \mid A_i^n, E_i^n \in \mathcal{P}\} \cup \\ &\{E_i^n \rightarrow R_i^n \mid E_i^n, R_i^n \in \mathcal{P}\}\end{aligned}$$

Between jobs of different tasks, certain data-dependencies impose redundant precedence constraints. In our running example, data-dependency $\tau_{\mathtt{C}}^0 \rightarrow \tau_{\mathtt{D}}^1$ is redundant, because there is already $\tau_{\mathtt{C}}^0 \rightarrow \tau_{\mathtt{D}}^0$.

Thus, we introduce the notion of *relevant* data-dependencies. Non-relevant data-dependencies can be safely ignored for the construction of $\mathcal{P}_{inter}$.

Definition 24 (Data-dependency relevance)**.**

$$\begin{aligned}relevant(\tau_i^n \rightarrow \tau_j^m) \Leftrightarrow &\nexists \tau_i^n \rightarrow \tau_j^{m'}.\ m' < m \ \wedge \\ &\nexists \tau_i^{n'} \rightarrow \tau_j^m.\ n < n'\end{aligned}$$

Using this notion of relevance, we then define inter-job precedence constraints.

Definition 25 (Inter-job precedence constraints)**.**

$$\mathcal{P}_{inter} = \{R_i^n \rightarrow A_j^m \mid R_i^n, A_j^m \in \mathcal{P}, \tau_i^n \rightarrow \tau_j^m \in \mathcal{D}, relevant(\tau_i^n \rightarrow \tau_j^m)\}$$

Example 13 (Precedence constraints)**.** In our running example, data-dependency $\tau_{\mathtt{C}}^0 \rightarrow \tau_{\mathtt{D}}^1$ is not relevant because of $\tau_{\mathtt{C}}^0 \rightarrow \tau_{\mathtt{D}}^1$. It is the only non-relevant data-dependency in this set. Thus, $\mathcal{P}_{inter}$ is defined from the following elements:

$$R_{\mathtt{A}}^0 \rightarrow A_{\mathtt{C}}^0 \qquad R_{\mathtt{B}}^0 \rightarrow A_{\mathtt{C}}^0 \qquad R_{\mathtt{B}}^1 \rightarrow A_{\mathtt{C}}^1 \qquad R_{\mathtt{B}}^3 \rightarrow A_{\mathtt{C}}^2 \qquad R_{\mathtt{C}}^0 \rightarrow A_{\mathtt{D}}^0$$

9.1.3 Phase Optimizations

The presented precedence constraints are sufficient to define a sound real-time system. However, communicating via the global memory when tasks are on the same core is unnecessary. A data-dependency $\tau_i^n \to \tau_j^m$ is *local*, iff $\pi_i = \pi_j$, otherwise it is *distant*. We define thus improved inter-job precedence constraints which use the global memory only if necessary, local communications are handled inside the E-phase.

Thus, we review our definition of the phase set. In particular, there is no need to emit a A- or R-phase, if a task has only local data-dependencies.

Definition 26 (Improved phase set)**.**

$$\begin{aligned}\mathcal{P} = &\{A_i^n \mid \tau_i^n \in \mathcal{J}, \tau_j^m \to \tau_i^{n'} \in \mathcal{D}, \pi_i \neq \pi_j\} \cup \\ &\{E_i^n \mid \tau_i^n \in \mathcal{J}\} \cup \\ &\{R_i^n \mid \tau_i^n \in \mathcal{J}, \tau_i^{n'} \to \tau_j^m \in \mathcal{D}, \pi_i \neq \pi_j\}\end{aligned}$$

There is no need to change the definition of intra-job precedence constraints since by construction these require both an A- and E-phase inside $\mathcal{P}$. However, for inter-job precedence constraints, we must distinguish local and distant communications.

Definition 27 (Improved inter-job precedence constraints)**.**

$$\begin{aligned}\mathcal{P}_{inter} = &\{R_i^n \to A_j^m \mid R_i^n, A_j^m \in \mathcal{P}, \tau_i^n \to \tau_j^m \in \mathcal{D}, relevant(\tau_i^n \to \tau_j^m), \pi_i \neq \pi_j\} \cup \\ &\{E_i^n \to E_j^m \mid E_i^n, E_j^m \in \mathcal{P}, \tau_i^n \to \tau_j^m \in \mathcal{D}, relevant(\tau_i^n \to \tau_j^m), \pi_i = \pi_j\}\end{aligned}$$

Example 14 (Improved precedence constraints)**.** With these improvements, precedence constraint $R_{\mathtt{C}}^0 \to A_{\mathtt{D}}^0$ in $\mathcal{P}_{inter}$ is replaced by $E_{\mathtt{C}}^0 \to E_{\mathtt{D}}^0$:

$$R_{\mathtt{A}}^0 \to A_{\mathtt{C}}^0 \qquad R_{\mathtt{B}}^0 \to A_{\mathtt{C}}^0 \qquad R_{\mathtt{B}}^1 \to A_{\mathtt{C}}^1 \qquad R_{\mathtt{B}}^3 \to A_{\mathtt{C}}^2 \qquad E_{\mathtt{C}}^0 \to E_{\mathtt{D}}^0$$

9.2 Monocore Compilation

9.2.1 Compilation Process

Let us briefly discuss the compilation of Prelude as it is defined for monocore platforms. A Prelude program is compiled into a single C file that contains one function per phase as well as all buffers. Some files are not generated by the compiler. First, the code of imported nodes has to be provided by the user. Second, the user must provide code that integrates the generated files into the final application. These files can either be part of the Prelude distribution or written by the programmer. Finally, and using those files, the compilation process produces a single binary.

9.2.2 Code Structure

In this Section, we will illustrate the generated code for monocore platforms. Figure 9.1 displays the generated task code for tasks $\tau_{\mathtt{A}}$ $\tau_{\mathtt{C}}$.

If required, the compiler generates an *instance counter* that counts how often the tasks execute (Line 7 and 18). It will be used below.

```
1  static int A_C_buff[A_C_size];
2  static int B_C_buff[B_C_size];
3  static int C_D_buff;
4
5  void A_task()
6  {
7    static int instance = 0;
8    static int A_C_idx = 0;
9
10   const int a_loc = A();
11
12   if (must_write_A_C(instance))
13     A_C_buff[A_C_idx] = a_loc;
14
15   if (must_post_A_C(instance))
16     post_sem(sem_A_C);
17
18   ++instance;
19 }
```

```
1  void C_task()
2  {
3    static int instance = 0;
4    static int A_C_idx = 0;
5    static int B_C_idx = 0;
6
7    wait_sem(sem_A_C);
8    if (must_wait_B_C(instance))
9      wait_sem(sem_B_C);
10
11   const int a_loc = A_C_buff[A_C_idx];
12   const int b_loc = B_C_buff[B_C_idx];
13
14   A_C_idx = (A_C_idx + 1) % A_C_size;
15   if (must_change_B_C(instance))
16     B_C_idx = (B_C_idx + 1) % B_C_size;
17
18   const int c_loc = C(a_loc, b_loc);
19
20   C_D_buff = c_loc;
21
22   post_sem(sem_C_D);
23
24   ++instance;
25 }
```

Figure 9.1: Monocore task code of $\tau_{\mathtt{A}}$ and $\tau_{\mathtt{C}}$

For each data-dependency $\mathcal{D}_{i,j}$, a *communication buffer* is allocated (Lines 1-3). The sizes of buffers are automatically computed and since `C_D_buff` is of size 1, a single value is allocated instead of an array. A *buffer index variable* (Line 4) holds the buffer cell to be accessed for operations. The compiler also generates code to compute the next value of that variable (Line 14). If only some executions of the task must change the value of the index, the compiler generates a *protocol function* (Line 15) to control when it is modified.

Working variables (Line 10 and 12) hold the current processed values of dataflows. They are denoted `_loc` and are declared local and `const` since they are only used by that task.

Finally, the generated code also handles semaphores (Line 7 and 22). Similarly to communication buffers, the compiler may generate protocol functions (Line 8 and 15).

9.3 Multicore Compilation Process

Figure 9.2 provides an overview for the code generation process. A Prelude program is compiled and produces one C file per processor as well as one C file for the global data. The generated C code contains one function per phase allocated to their respective core. Buffers for local communications are defined inside the core-specific file. Buffers for distant communications are defined as global data.

As for monocore compilation, some files are not generated by our compiler. For imported nodes, the compilation process expects one file per core that implements the relevant functions. Using those

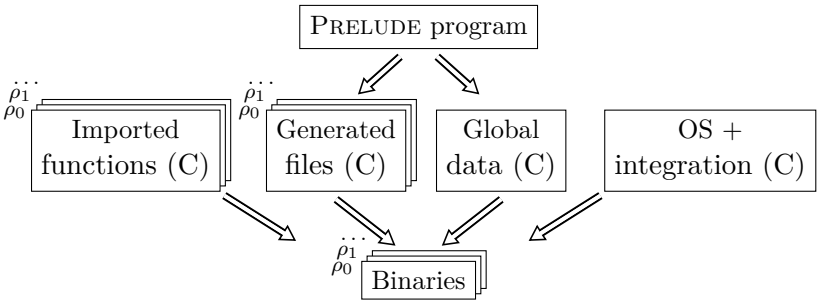


Figure 9.2: Overview of the multicore compilation chain

files, the compilation process then produces one binary per core.

9.4 Multicore Code Structure

In this Section, we will illustrate the structure of our generated code. using Figure 9.3. It displays the generated code for tasks $\tau_{\texttt{C}}$ and $\tau_{\texttt{A}}$. The compiler generates one function per phase.

Working variable are allocated in $\mathcal{M}_i$. While certain variables (Line 28) are used only within a single phase and are thus generated unchanged, others are used in multiple phases (Lines 2, 16, 28). Thus, they are declared `static`.

Communication buffers are allocated differently whether the communicating tasks are co-located or not. If they are, the buffer is allocated in private memory (Line 4) as a variable. Buffers residing in global memory are accessed using opaque handles which are then passed to functions `read_val` and `write_val`.

The code generation for protocol functions and semaphores is unchanged.

```
// CPU 0
static int a_loc;
static int b_loc;
static int C_D_buff;

void C_A()
{
  static int instance = 0;
  static int A_C_idx = 0;
  static int B_C_idx = 0;

  wait_sem(sem_A_C);
  if (must_wait_B_C(instance))
    wait_sem(sem_B_C);

  a_loc = read_val(A_C_buff, A_C_idx);
  b_loc = read_val(B_C_buff, B_C_idx);

  A_C_idx = (A_C_idx + 1) % A_C_size
  if (must_change_B_C(instance))
    B_C_idx = (B_C_idx + 1) % B_C_size;

  ++instance;
}

void C_E()
{
  const int c_out = C(a_loc, b_loc);
  C_D_buff = c_out;
  post_sem(sem_C_D);
}
```

```
// CPU 1
static int a_loc;

void A_E()
{
  a_loc = A();
}

void A_R() {
  static int instance = 0;
  static int A_C_idx = 0;

  if (must_write_A_C(instance))
    write_val(A_C_buff, A_C_idx, a_loc);

  if (must_post_A_C(instance))
    post_sem(sem_A_C);

  ++instance;
}
```

Figure 9.3: Generated task code of τ_A and τ_C

Chapter 10

Evaluation

To evaluate our approach, we use the ROSACE [69] case study. It implements a longitudinal flight controller that measures the airspeed, vertical speed and altitude of the aircraft and controls the aircraft accordingly. We simplified parts dedicated to environment simulation (which are not meant to be embedded), so that corresponding tasks return dummy values. We implemented the case study on a FPGA platform with two softcores running the ERIKA [33] OSEK-compliant RTOS. This allowed us not only to change between AER compliant and AER non-compliant code generation, but also between a scratchpad-based and cache-based memory architecture. Relying on an FPGA platform allowed us to guarantee that the architectures differ only in the private memories and, not the least important, that they used the same "silicon" budget.

10.1 Hardware Platform

In order to allow the comparison between different hardware architectures, we rely on an FPGA development board, a Cyclone III by Altera with two NIOS-II softcores, depicted in Figure 10.1. The data and instruction *master ports* connect the processor to the *Avalon Interconnect Fabric*, a partial crossbar with a master/slave behavior, which serves as a hub to access shared resources of the board. In our case, only NIOS-II processors are masters, other devices connected to the crossbar are slaves. Each master is only connected to a subset of slaves. Accesses from two masters to two different slaves can execute simultaneously. Simultaneous accesses to the same slave are arbitrated with a round-robin policy. *Tightly-coupled* data and instruction ports offer private contention-free access to processor-dedicated on-chip memories. Each processor has access to a tightly-coupled memory for data and to another for instructions. These memories serve as scratchpad memory ($\mathcal{M}_i$). Processors share access to an on-chip shared RAM ($\mathcal{M}_G$). On a real embedded board, the shared memory would be an external component (e.g. SDRAM, SRAM), with typically longer access time. Therefore, our shared RAM is controlled by an artificially slower clock which mimics these slower accesses. Finally, processors are also connected to an on-chip mutex, on-board IOs and timers, through the master ports. We use the mutex to implement synchronizations, because the processors do not have dedicated built-in primitives.

In addition to the scratchpad architecture we just detailed, we implement an alternative cache-based architecture. It features a cache on each master port, with access performances similar to the scratchpads. The FPGA has tight space limitations (ERIKA is not available on more recent FPGA boards), memory sizes are reported in Table 10.1. Space reserved for SPM in the scratchpad-based architecture is instead reserved for the main memory in the cache-based architecture. As can be

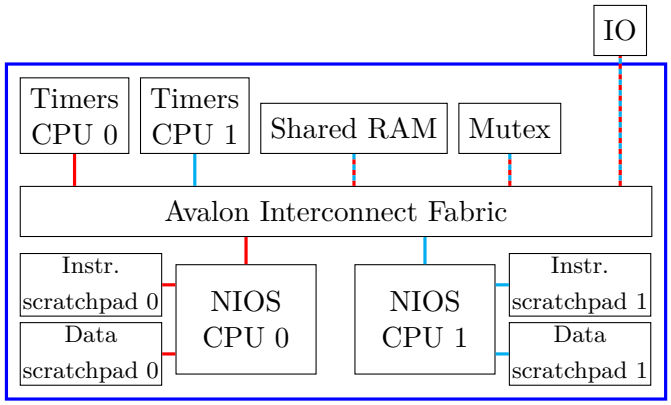


Figure 10.1: The hardware design

Memory (SPM architecture)	Size
Data SPM	ρ_0: 5kB, ρ_1: 4kB
Instruction SPM	ρ_0: 12kB, ρ_1: 8kB
Main	2kB
Memory (cache architecture)	**Size**
Data cache	2kB
Instruction cache	4kB
Main	29kB

Table 10.1: Size of memories for the experiments

seen, restricting both platforms to the same architecture reveals the "hidden cost" of cache platforms: Mechanisms such as cache coherency required FPGA space which limited the amount of available private memory.

10.2 OSEK-compliant code

ERIKA is an OSEK-compliant RTOS, so tasks must all be declared statically in an *OIL* configuration file, which is generated by the PRELUDE compiler in our case. The OIL file is divided into several sections, in particular:

- `CPU_DATA` sections, which describe the hardware processors (identifier, source files, Hardware Abstraction Layer, stack address space, ...);
- `TASK` sections define the task set (CPU allocation, events to handle synchronizations, stack size, ...). In our case, each phase is declared as a separate `TASK`;
- `EVENT` sections which enable us to implement binary semaphores.

10.3 Measurements

To evaluate our modified PRELUDE compiler, we measured the response time of each task for different hardware/software configurations as shown in Figure 10.2. We measured the speedup of a

AER-compliant scratchpad-based architecture with respect to a AER non-compliant cache-based architecture with different global memory speeds. Under the different scenarios, the global memory was:

1. as fast as private memory (red);
2. 4 times slower than the private memory (green);
3. 8 times slower than the private memory (blue).

The final value corresponds to observed latencies on external SRAM on similar boards.

We chose this metric because it best illustrates the impact of AER-compliance when the cost of communications, i.e. the cost of A-/R-phases, changes. Scenario 3 represents the "typical" execution context, while scenario 1 represents an exceptional case where shared memory is essentially as fast as private memory. Varying other factors seemed of little interest. For instance, varying the size of available SPM would simple be a "pass-fail" test, either the compiled code fits inside, or it doesn't.

We provide mean results for 20 executions for each configuration (variance is very low). The observed speedup is proportional to the RAM clock. When the shared RAM is the slowest, the average speedup is 6.29 with a standard deviation of 2.19. When the shared RAM has the same clock as the global clock, the SPM implementation barely outperforms the cache one. The average speedup is 1.09 with a standard deviation of 0.31. This is likely due to the OS overheads of the AER-compliant implementation, since each phase is implemented as a separate task.

10.4 Summary

This evaluation illustrates the advantages of our our approach. We could test and compare different, competing hardware architectures and the necessary development efforts were limited to the integration of the generated C code with the RTOS. For the ROSACE case study, we could thus measure that the AER-compliant variant relying on scratchpad memory outperforms the cache-based variant.

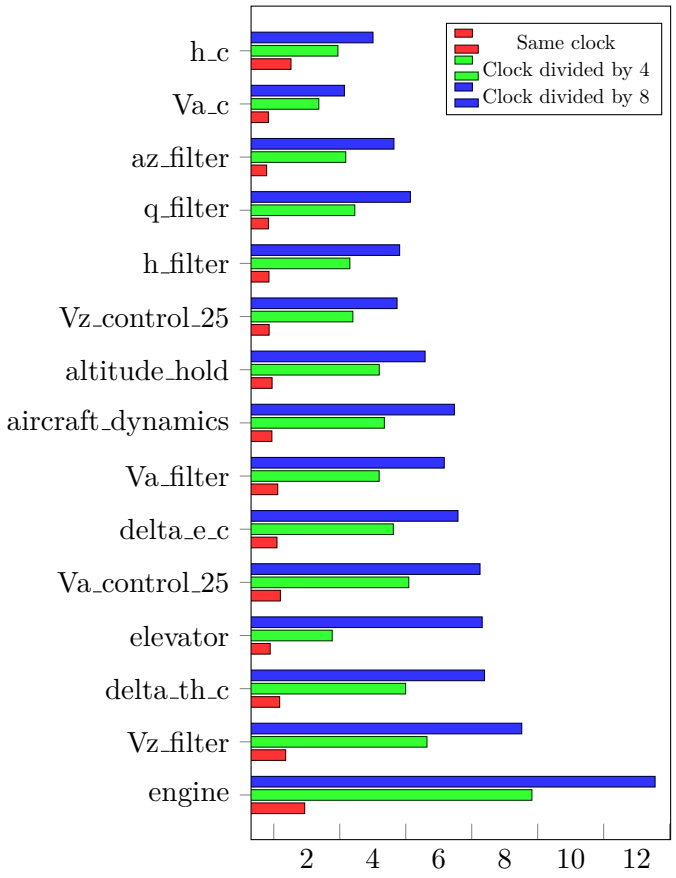


Figure 10.2: Observed speedup (higher is better)

Part IV

Synchronous semantics of multi-mode multi-periodic systems

Chapter 11

Introduction

In this Part, we tackle the problem of designing and programming a real-time system with multiple modes of execution as well as multiple real-time constraints. A *multi-mode* real-time system, is characterized by a set of modes. Each mode is defined by a different task set and the current mode defines the task set to execute. A Mode Change Request (MCR) triggers the transition from one mode to another. The transition is executed in accordance to a Mode Change Protocol (MCP).

As detailed in Section 6.6, Synchronous State Machines have been proposed in [28] for Lustre [42] and Lucid Synchrone [25] to tackle the problem of programming of such system using a formally defined language. In these languages, an automaton state corresponds to a mode of execution. However, these languages (and the automata by extension) only support mono-periodic systems.

This limitation arises from the mono-periodic semantics of these languages. The semantics of Synchronous State Machines is defined via a translation semantics that relies on lower-level constructs (`when` and `merge`) which are only defined for mono-periodic systems.

When looking at the Prelude language presented in Chapter 7, it is in essence mono-mode. Imported and rate-transition operators do not allow to change the behavior of the system. Only conditional sub-sampling operators offer a construct to perform a choice between dataflows at runtime. However, their definition limits them to operands with the same period and offset. In addition, once a conditional sub-sampling operator has been applied, it is not possible to apply further rate-transition operators.

The main problem to tackle is the definition of a semantics for a system where the period of Mode Change Requests (MCR) and the period of tasks within a mode are not all the same. Thus, the semantics must define how to handle tasks which perceive MCRs differently. A language for multi-mode systems must be able to ensure the soundness and determinism of the system under these conditions.

This Part will present an extension of the language of Chapter 7 to support such systems. The extension consists in transposing the Synchronous State Machines of [28] to Prelude.

11.1 Current Limitations

To understand why one cannot simply combine Synchronous State Machines with Prelude, one has to look at how the Synchronous Kahn Semantics (Section 7.4) and Clocks (Section 7.1) interact. The semantics define how a program behaves, but if we are unable to ascribe a clock to dataflows in our semantics, we cannot guarantee the program has a well-defined semantics.

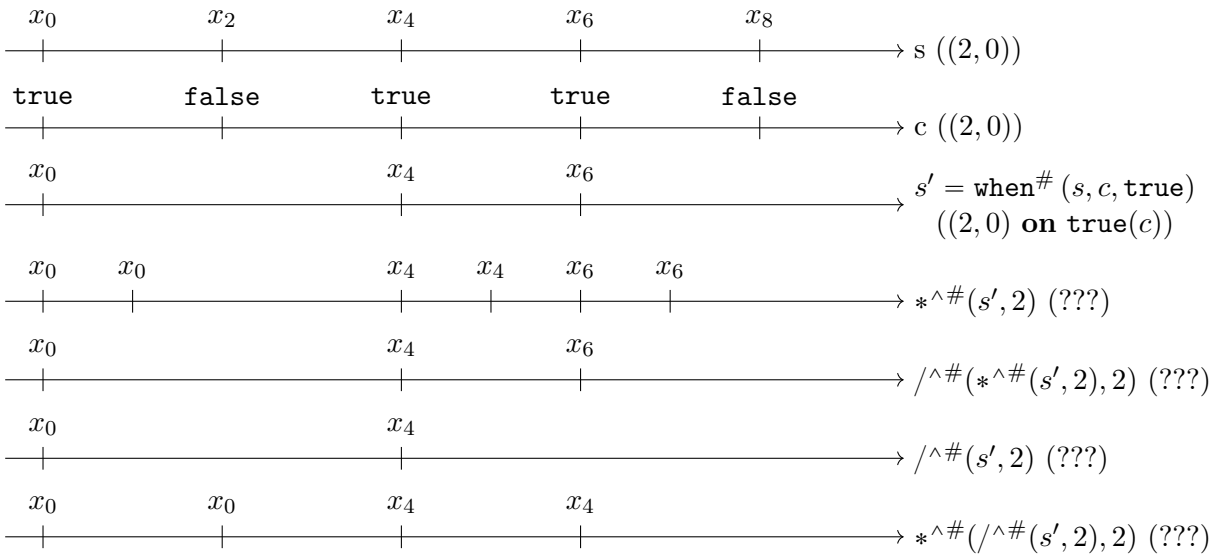


Figure 11.1: Applying a periodic clock operators after conditional sub-sampling

The first problem we encounter is that the semantics of the $\texttt{when}^{\#}(s, c, C)$ operator do not define the values of the program when s and c have different clocks. For simple cases, such as harmonic periods, one can easily imagine a sensible behavior. However, what should the program behavior be if $\widehat{s} = (7, 0)$ and $\widehat{c} = (11, 0)$?

The second problem we encounter is shown in Figure 11.1. Dataflows s and c have the same clock. While their semantics are not well-defined for dataflows whose clock feature a **on** operator, we can follow the "intent" of the semantics. In our case, the $*^{\wedge\#}$ operator repeats each value k times, while the $/^{\wedge\#}$ operator skips values whose tags do not coincide with those of a clock slower by factor k. Thus, after applying the $\texttt{when}^{\#}$ operator, and we applied those periodic operators. We can see that the order of operators yields incompatible dataflows whose clocks we can't accurately describe. In our case, dataflow $*^{\wedge\#}(/^{\wedge\#}(s', 2), 2)$ is present at instant 2, while dataflow $/^{\wedge\#}(*^{\wedge\#}(s', 2), 2)$ isn't. In [34], it was identified that such dataflows would have clocks similar to s **on** $true(/^{\wedge\#}(*^{\wedge\#}(c, 2), 2))$ and s **on** $true(*^{\wedge\#}(/^{\wedge\#}(c, 2), 2))$. Such an extension was however not pursued as it requires a complex dependent type system, because clock conditions may be not just names, but arbitrary expressions.

Our contribution thus aims at defining a formal semantics for a multi-periodic $\texttt{when}^{\#}$ operator and clock system that allows to reason efficiently about such dataflows.

11.2 Overview

Chapter 12 presents *clock views*, our solution to decouple the rate of MCRs and tasks. Chapter 13 presents the extended language semantics, in particular of the `when` and `merge` operators. Chapter 14 illustrates the extended syntax to support Synchronous State Machines. Chapter 15 defines a clock calculus able to reason about our extended clocks and operators.

Chapter 12

Clock views

In this Chapter, we will present our solution to the issues presented in Section 11.1. Recall that one of the issues is that we cannot give a sound type to expressions such as `x when true(c) *^2 /^2` and `x when true(c) /^2 *^2`. Informally, the problem is that those expressions perceive the condition `c` differently, but there is no way to express this difference using existing clocks.

We introduce *clock views* as an extension of conditional clocks to express this difference. Their definition is given below.

Definition 28 (Multi-periodic conditional sub-sampling with clock views)**.** A multi-periodic conditional clock is denoted ck **on** $C(c, w)$ where ck is a clock, C is a constant, c is a condition dataflow, and w is a strictly periodic clock called *view*.

The infinite sequence of tags generated by ck **on** $C(c, w)$, denoted $(ck \textbf{ on } C(c, w))^{\#}$, is defined as follows:

$$(ck \textbf{ on } C(c, (n, p)))^{\#} = \{t \mid t \in ck^{\#}, t' \in (n, p)^{\#}, (v, t'') \in c^{\#}, \\ t' \leqslant t < t' + n \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C\}$$

A view is valid, iff

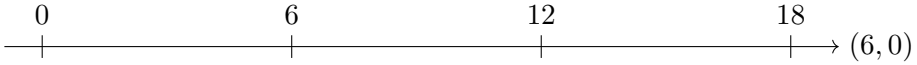
- $\pi(ck)$ **div** $\pi(w) \wedge \pi(\widehat{c})$ **div** $\pi(w)$;
- $\varphi(\widehat{c}) \leqslant p$.

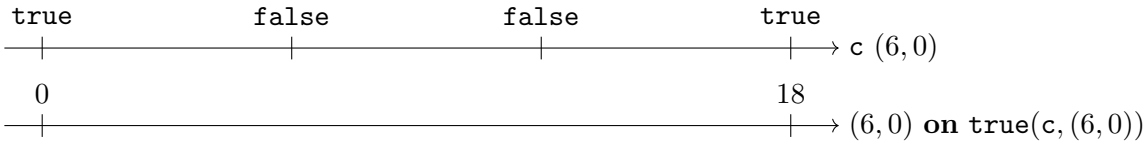
Moreover, we extend the π and φ operators:

$$\pi(ck \textbf{ on } C(c, w)) = \pi(ck) \qquad\qquad \varphi(ck \textbf{ on } C(c, w)) = \varphi(ck)$$

Intuitively, the view w delimits intervals where only a single value produced by c is perceived for all tags produced by ck within that interval. Each interval starts whenever a tag of w is present and ends when the next interval begins. If the observed value equals C, then ck **on** $C(c, w)$ produces all tags within that interval. Otherwise, it produces no tag within that interval.

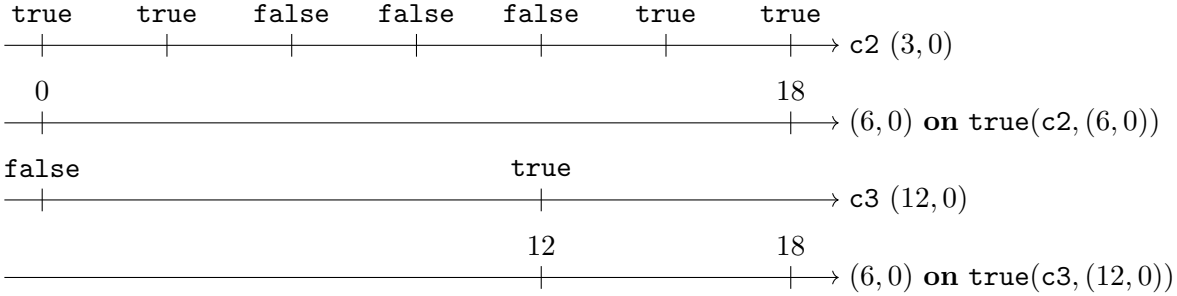
Example 15 (Simple clock views)**.** Let us first illustrate this extension on a mono-periodic program. Thus, let us assume the following clocks and dataflows. In this case, the view could have the same clock,$(6, 0)$. This clock then produces the same values as $(6, 0)$ **on** `true(c)`.



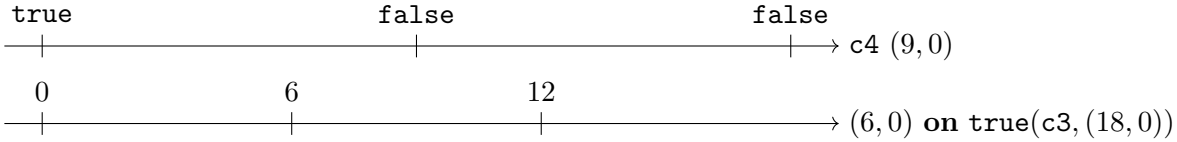


However, it is also possible to use a larger view. Chapter 13 and Chapter 15 will illustrate in more detail why this is useful. It this example, the clock produces the tag 6, but not the tag 18. This is because the view produces the tags 0, 12 and 24. Thus, all tags t such that $0 \leqslant t < 12$ observe the value of `c` produced at instant $0 + 0 - 0 = 0$ and all tags t such that $12 \leqslant t < 24$ observe the value of `c` produced at instant $12 + 0 - 0$. Thus, both 0 and 6 observe the value `true`, while both 12 and 18 observe the value `false`.

Stepping up the complexity a little bit, we have the following possible dataflows. As can be seen, if one clock has a period that is a multiple of another, we can use this clock for the view. If the condition dataflow has the faster clock, we consider only the first of each value. If the condition dataflow has the slower clock, it's value is reused for consecutive tags.



If clocks have unrelated periods, we must provide a view with a period divisible by both the clock and condition (Definition 28). Thus, the least-common multiple of periods is a period satisfying this condition.



Clock views allows us to give a clock (and thus a well-defined semantics) to dataflows such as $*^{\wedge\#}(\texttt{when}^{\#}(\texttt{s}, \texttt{c}, \texttt{true}, (12, 0)), 2)$. However, assuming $\widehat{\texttt{s}} = \widehat{\texttt{c}} = (6, 0)$, the most precise clock we can construct with the current Definition is $((6, 0)$ **on** $\texttt{true}(\texttt{c}, (12, 0))) *. 2$. With increasing program sizes, the size of clocks would increase too. Thus, we would like to simplify clocks such that we can safely display a more readable clock expression. In our case, this would be $(3, 0)$ **on** $\texttt{true}(\texttt{c}, (12, 0))$.

Property 1 below shows how we can apply periodic clock operators on clocks with clock views. Note that some care has to be given because we have to take into account the impact of the operator on clock views. For the $*.$ and $/.$ operator, the impact is straightforward. The $*.$ operator inserts tags, but can't create information about the condition *ex nihilo*. Thus, the view remains unchanged, but the operator is applied recursively. For the $/.$ operator, we also apply the operator recursively. However, applying this operator may or may not change the view as illustrated in Example 16. This is reflected in the new view value.

For the $\rightarrow.$ operator, we must distinguish two cases to prepare the definition of our Synchronous Kahn Semantics (Chapter 13). The $\sim>^{\#}$ operator delays all values of a dataflow, while the $\texttt{tail}^{\#}$

operator drops the first value. Without conditional sub-sampling, dropping the first value and delaying each value by one period yields dataflows with the same clock, even though the produced values differ. This is not the case with conditional sub-sampling. Thus, we differentiate between the $\Rightarrow.$ operator that delays *both* the conditioned clock and the view and the $\rightharpoonup.$ operator that delays *only* the conditioned clock. Both are equivalent to $\rightarrow.$ in the absence of conditional sub-sampling.

Property 1 (Periodic Clock Operators and Clock Views)**.**

$$\begin{aligned}
((ck \textbf{ on } C(c,w)) *. k)^{\#} &= ((ck *. k) \textbf{ on } C(c,w))^{\#} \\
&\quad \text{, if } k \textbf{ div } \pi(ck \textbf{ on } C(c,w)) \\
((ck \textbf{ on } C(c,(n,p))) /. k)^{\#} &= ((ck /. k) \textbf{ on } C(c,(\pi(ck) * lcm(n/\pi(ck),k),p)))^{\#} \\
&\quad \text{, if } (\pi(ck) * k) \textbf{ div } n \vee n \textbf{ div } (\pi(ck) * k) \\
((ck \textbf{ on } C(c,w)) \Rightarrow. k)^{\#} &= ((ck \Rightarrow. k) \textbf{ on } C(c, w \rightarrow. k))^{\#} \\
(ck \textbf{ on } C(c,w) \rightharpoonup. k)^{\#} &= ((ck \rightharpoonup. k) \textbf{ on } C(c,w))^{\#} \\
((n,p) \Rightarrow. k)^{\#} &= ((n,p) \rightarrow. k)^{\#} \\
((n,p) \rightharpoonup. k)^{\#} &= ((n,p) \rightarrow. k)^{\#}
\end{aligned}$$

While the equivalences for $\Rightarrow.$ and $\rightharpoonup.$ hold by construction, those for $*.$ and $/.$ must be proven.

Equivalence of Periodic Acceleration. Recall the definitions of $*.$ and **on**.

$$\begin{aligned}
(ck *. k)^{\#} &= \{t \mid t' \in ck^{\#}, i \in [\![0..k[\![, t = t' + i\frac{\pi(ck)}{k}\} \\
(ck \textbf{ on } C(c,(n,p)))^{\#} &= \{t \mid t \in ck^{\#}, t' \in (n,p)^{\#}, (v,t'') \in c^{\#}, \\
&\qquad t' \leqslant t < t' + n \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C\}
\end{aligned}$$

Thus, we have

$$\begin{aligned}
&(ck \textbf{ on } C(c,(n,p)) *. k)^{\#} \\
&= \{t \mid t_0 \in (ck \textbf{ on } C(c,(n,p)))^{\#}, i \in [\![0,k[\![, \\
&\qquad t = t_0 + i * \pi(ck \textbf{ on } C(c,(n,p)))/k\} \\
&= \{t \mid t_0 \in ck^{\#}, t' \in (n,p)^{\#}, (v,t'') \in c^{\#}, i \in [\![0..k[\![, \\
&\qquad t' \leqslant t_0 < t' + n \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C \\
&\qquad t = t_0 + i * \pi(ck \textbf{ on } C(c,(n,p)))/k\} \\
&= \{t \mid t_0 \in ck^{\#}, t' \in (n,p)^{\#}, (v,t'') \in c^{\#}, i \in [\![0..k[\![, \\
&\qquad t' \leqslant t_0 < t' + n \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C \\
&\qquad t = t_0 + i * \pi(ck)/k\}
\end{aligned}$$

And

$$\begin{aligned}
&(ck *. k) \textbf{ on } C(c,(n,p))^{\#}\\
=\ &\{t \mid t \in (ck *. k)^{\#}, t' \in (n,p)^{\#}, (v,t'') \in c^{\#}, i \in [\![0,k[\![,\\
&\quad t' \leqslant t < t' + n \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C\}\\
=\ &\{t \mid t_0 \in ck^{\#}, t' \in (n,p)^{\#}, (v,t'') \in c^{\#}, i \in [\![0,k[\![,\\
&\quad t = t_0 + i * \pi(ck)/k,\\
&\quad t' \leqslant t < t' + n \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C\}
\end{aligned}$$

These equations differ only in one point:

$$\begin{array}{ccc}
(ck \textbf{ on } C(c,(n,p)) *. k)^{\#} & & (ck *. k) \textbf{ on } C(c,(n,p))^{\#}\\
t' \leqslant t_0 < t' + n & \Leftrightarrow & t' \leqslant t < t' + n
\end{array}$$

Since $\pi(ck)$ **div** n, we have $i * \pi(ck)/k < n$. Thus, whatever the difference is between t and t_0, it is less than n, the minimal difference between two tags within (n,p). Thus, for a given value of t_0 its derived ts, there exists only one t' that satisfies the constraints. □

Equivalence of Periodic Deceleration. Recall the definitions of /. and **on**.

$$\begin{aligned}
(ck /. k)^{\#} &= \{t \mid t \in ck^{\#}, i \in [\![0..k[\![, t = \pi(ck) * k * i + \varphi(ck)\}\\
(ck \textbf{ on } C(c,(n,p)))^{\#} &= \{t \mid t \in ck^{\#}, t' \in (n,p)^{\#}, (v,t'') \in c^{\#},\\
&\qquad t' \leqslant t < t' + n \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C\}
\end{aligned}$$

Thus, we have

$$\begin{aligned}
&((ck \textbf{ on } C(c,(n,p))) /. k)^{\#}\\
=\ &\{t \mid t \in (ck \textbf{ on } C(c,(n,p)))^{\#}, i \in [\![0..k[\![,\\
&\quad t = \pi(ck \textbf{ on } C(c,(n,p))) * k * i + \varphi(ck)\}\\
=\ &\{t \mid t \in (ck \textbf{ on } C(c,(n,p)))^{\#}, i \in [\![0..k[\![,\\
&\quad t = \pi(ck) * k * i + \varphi(ck)\}\\
=\ &\{t \mid t \in ck^{\#}, i \in [\![0..k[\![,\\
&\quad t = \pi(ck) * k * i + \varphi(ck)\\
&\quad t' \leqslant t < t' + n \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C\}
\end{aligned}$$

And

$$\begin{aligned}
&((ck /. k) \textbf{ on } C(c,(\pi(ck) * lcm(n/\pi(ck),k),p)))^{\#}\\
=\ &\{t \mid t \in (ck /. k)^{\#}, t' \in (\pi(n) * lcm(n/\pi(ck),k),p)^{\#}, (v,t'') \in c^{\#}, i \in \mathbb{N},\\
&\quad t' \leqslant t < t' + (\pi(n) * lcm(n/\pi(ck),k)) \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C\}\\
=\ &\{t \mid t \in ck^{\#}, t' \in (\pi(n) * lcm(n/\pi(ck),k),p)^{\#}, (v,t'') \in c^{\#}, i \in \mathbb{N},\\
&\quad t' \leqslant t < t' + (\pi(n) * lcm(n/\pi(ck),k)) \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C\\
&\quad t = \pi(ck) * k * i + \varphi(ck)\}
\end{aligned}$$

We now distinguish 2 cases.

Case 1. $\pi(ck) * k$ **div** n

Because of our assumption, we have $n = \pi(ck) * k * a$. Thus, we have

$$\begin{aligned}
&\pi(ck) * lcm(n/\pi(n), k) \\
&= \pi(ck) * lcm(\pi(ck) * k * a/\pi(ck), k) \\
&= \pi(ck) * lcm(k * a, k) \\
&= \pi(ck) * k * a = n
\end{aligned}$$

Returning to our set definitions, we have

$$\begin{aligned}
&\{t \mid t \in ck^{\#}, t' \in (n,p)^{\#}, (v,t'') \in c^{\#}, i \in \mathbb{N}, \\
&\quad t = \pi(ck) * k * i + \varphi(ck), \\
&\quad t' \leqslant t < t' + n \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C\} \\
&= \{t \mid t \in ck^{\#}, t' \in (\pi(ck) * k * a, p)^{\#}, (v,t'') \in c^{\#}, i \in \mathbb{N}, \\
&\quad t = \pi(ck) * k * i + \varphi(ck), \\
&\quad t' \leqslant t < t' + \pi(ck) * k * a \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C\}
\end{aligned}$$

And

$$\begin{aligned}
&\{t \mid t \in ck^{\#}, t' \in (\pi(ck) * lcm(n/\pi(ck), k), p)^{\#}, (v,t'') \in c^{\#}, i \in \mathbb{N}, \\
&\quad t' \leqslant t < t + (\pi(ck) * lcm(n/\pi(ck), k)) \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C, \\
&\quad t = \pi(ck) * k * i + \varphi(ck)\} \\
&= \{t \mid t \in ck^{\#}, t' \in (\pi(ck) * k * a, p)^{\#}, (v,t'') \in c^{\#}, i \in \mathbb{N}, \\
&\quad t' \leqslant t < t' + \pi(ck) * k * a \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C, \\
&\quad t = \pi(ck) * k * i + \varphi(ck)\}
\end{aligned}$$

Case 2. n **div** $\pi(ck) * k$

Because of our assumption, we have

$$\begin{aligned}
n &= \pi(ck) * a \\
\pi(ck) * k &= n * b \\
k &= a * b
\end{aligned}$$

Thus, we have

$$\begin{aligned}
&\pi(ck) * lcm(n/\pi(ck), k) \\
&= \pi(ck) * lcm(\pi(ck) * a/\pi(ck), a * b) \\
&= \pi(ck) * lcm(a, a * b) \\
&= \pi(ck) * a * b = \pi(ck) * k
\end{aligned}$$

Returning to our set definitions, we have

$$\begin{aligned}
&\{t \mid t \in ck^{\#}, t' \in (n,p)^{\#}, (v,t'') \in c^{\#}, i \in \mathbb{N}, \\
&\quad t = \pi(ck) * k * i + \varphi(ck), \\
&\quad t' \leqslant t < t' + n \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C\} \\
&= \{t \mid t \in ck^{\#}, t' \in (\pi(ck) * a, p)^{\#}, (v,t'') \in c^{\#}, i \in \mathbb{N}, \\
&\quad t = \pi(ck) * a * b * i + \varphi(ck), \\
&\quad t' \leqslant t < t' + \pi(ck) * a \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C\}
\end{aligned}$$

And

$$\begin{aligned}
&\{t \mid t \in ck^{\#}, t' \in (\pi(ck) * lcm(n/\pi(ck), k), p)^{\#}, (v,t'') \in c^{\#}, i \in \mathbb{N}, \\
&\quad t' \leqslant t < t + (\pi(ck) * lcm(n/\pi(ck), k)) \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C, \\
&\quad t = \pi(ck) * k * i + \varphi(ck)\} \\
&= \{t \mid t \in ck^{\#}, t' \in (\pi(ck) * a * b, p)^{\#}, (v,t'') \in c^{\#}, i \in \mathbb{N}, \\
&\quad t' \leqslant t < t' + \pi(ck) * a * b \wedge t'' = t' + \varphi(\widehat{c}) - p \wedge v = C, \\
&\quad t = \pi(ck) * a * b * i + \varphi(ck)\}
\end{aligned}$$

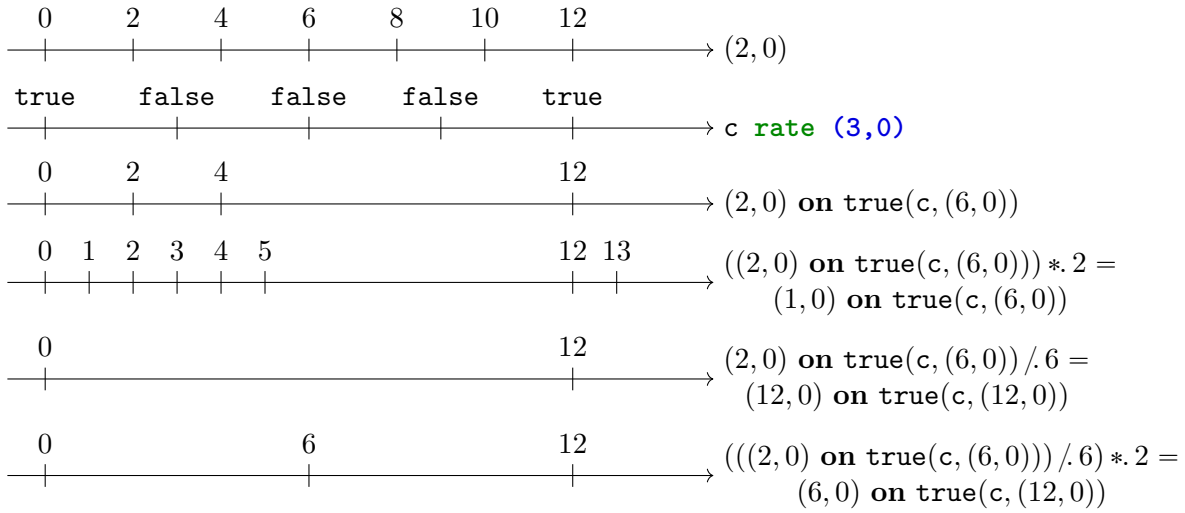
Those set definitions are only different, iff

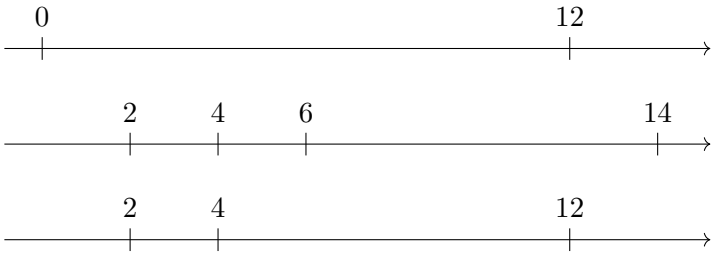
$$\begin{aligned}
&\exists t.t = \pi(ck) * a * b * i + \varphi(ck) \wedge t'_{before} = \pi(ck) * a * r + p \wedge t'_{after} = \pi(ck) * a * b * s + p \wedge \\
&\quad t'_{before} \leqslant t < t'_{before} + \pi(ck) * a \wedge t'_{after} \leqslant t < t'_{after} + \pi(ck) * a * b \wedge t'_{before} \neq t'_{after}
\end{aligned}$$

This is proven as UNSAT by the program in Appendix A.

□

Example 16 (Periodic Clock Operators and Clock Views)**.**





$$(((2,0)\ \textbf{on}\ \texttt{true}(\texttt{c},(6,0)))\mathbin{*.}2)\mathbin{/.}6 = (6,0)\ \textbf{on}\ \texttt{true}(\texttt{c},(6,0))$$

$$(2,0)\ \textbf{on}\ \texttt{true}(\texttt{c},(6,0))\mathbin{\Rightarrow.}2 = (2,2)\ \textbf{on}\ \texttt{true}(\texttt{c},(6,2))$$

$$(2,0)\ \textbf{on}\ \texttt{true}(\texttt{c},(6,0))\mathbin{\rightharpoonup.}2 = (2,2)\ \textbf{on}\ \texttt{true}(\texttt{c},(6,0))$$

Chapter 13

Extended Synchronous Kahn Semantics

In this Chapter, we will present the extended Synchronous Kahn Semantics to support our clock views. Many definitions are equivalent to those provided in Section 7.4. We will highlight the changed definitions to improve clarity.

For the imported operators, there is no need to change the semantics, as we keep the requirement that the arguments are synchronous.

Definition 29 (Extended Kahn semantics of imported operators)**.**

$$op^{\#}(s_0, \dots, s_n) = \{(op_f(v_0, \dots, v_n), t) \mid (v_0, t) \in s_0^{\#}, \dots, (v_n, t) \in s_n^{\#}\}$$

For the rate-transition operators, we only need to change the definition of the $::^{\#}$ operator. Since its definition relies on the $\rightarrow.$ operator, we must change its definition. In our case, this is the $\rightharpoonup.$ operator, since we prepend a value to the dataflow without changing the rest of the dataflow.

Definition 30 (Extended Kahn semantics of rate-transition operators)**.**

$$
\begin{aligned}
*^{\wedge\#}(s,k) &= \{(v, t + i * \pi(\widehat{s})/k) \mid (v,t) \in s^{\#}, i \in [\![0..k[\![\} \\
/^{\wedge\#}(s,k) &= \{(v,t) \mid (v,t) \in s^{\#} \wedge t \in (\widehat{s} /. k)^{\#}\} \\
\sim>^{\#}(s,k) &= \{(v, t+k) \mid (v,t) \in s^{\#}\} \\
\texttt{fby}^{\#}(v,s) &= \{(v,t_0)\} \cup \{(v_i, t_{i+1}) \mid (v_i,t_i), (v_{i+1}, t_{i+1}) \in s^{\#}\} \\
::^{\#}(v,s) &= \{(v,t)\} \cup s \qquad \text{, where } t = t_0 \in \widehat{s} \rightharpoonup. - \pi(\widehat{s}) \\
\texttt{tail}^{\#}(s) &= \{(v_i, t_i) \mid (v_i,t_i) \in s^{\#}, i \neq 0\}
\end{aligned}
$$

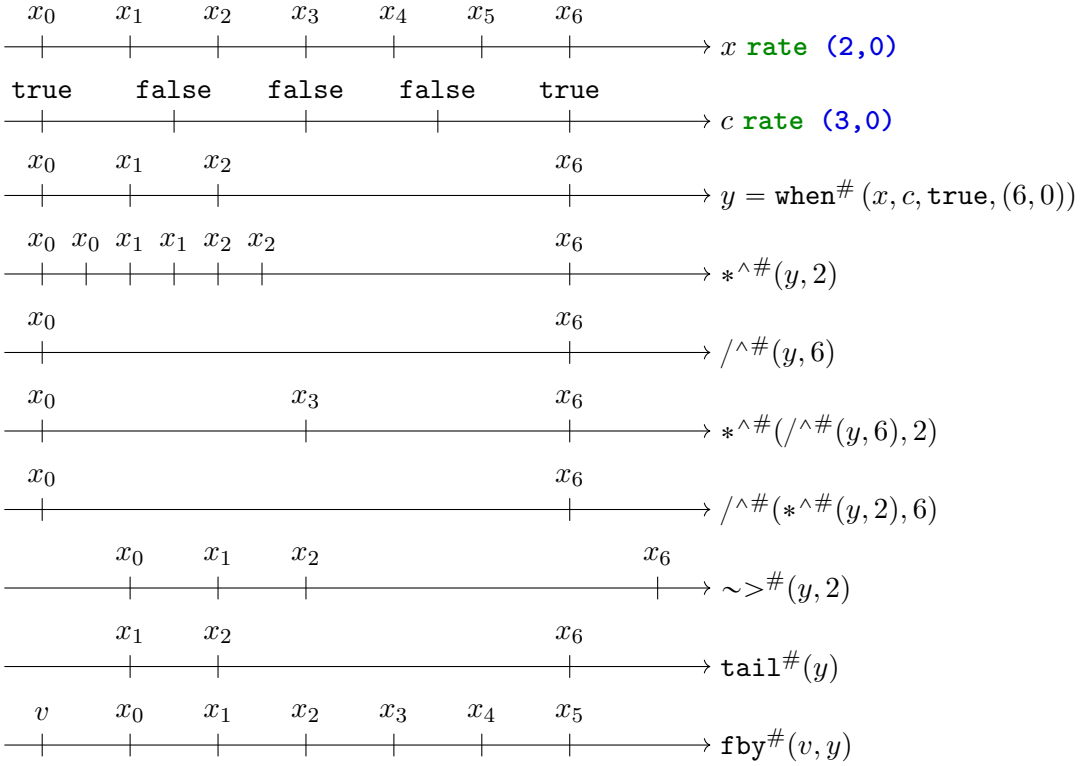
For the conditional operators, we must only change the definition of $\texttt{when}^{\#}$, to use the extended **on** operator. The $\texttt{merge}^{\#}$ operator remains unchanged, because we require the arguments $s_0, \dots, s_n$ to have the same view. This guarantees that the merged dataflows are *complementary*, i.e. there is never more than one value-tag pair present at any instant and their union results in a dataflow whose clock has the last-level **on** dropped.

Definition 31 (Extended Kahn semantics of conditional operators)**.**

$$\texttt{when}^{\#}\,(s, c, C, w) = \{(v, t) \mid (v, t) \in s,\ t \in (\widehat{s}\ \textbf{on}\ C(c, w))^{\#}\}$$

$$\texttt{merge}^{\#}\,(c, s_0, \ldots, s_n) = \bigcup_{i=0}^{n} s_i^{\#}$$

Example 17 (Extended Synchronous Kahn Semantics)**.** See Example 12 for examples without the $\texttt{when}^{\#}$ operator.



Chapter 14

Extended Language Syntax

In this, Chapter, we will present the syntax of our extended language supporting our Synchronous State Machines. We reuse the same transpilation technique as [28]. Thus, we will first define a *surface language* featuring state machines and then a core language without those. To improve clarity, we will highlight changes in definitions which differ only slightly with those provided in Section 7.2.

14.1 Surface Language

Figure 14.1 illustrates the surface language the programmer uses. It is mostly similar to the language defined in Figure 7.1. The notable differences are:

First, some minor cosmetic changes help with writing programs: Expressions need not be in Administrative Normal Form, i.e. arguments of function and operator applications may be arbitrary expressions, and one can define constants. The change from one form to another is trivial, however.

Second, a node body isn't a sequence of equations, but of *definitions*. A definition is either an equation or an automaton. Equations are unchanged from their previous definition.

The syntax of automata follows closely the one defined in [28]. *Automata* are composed of *states*. Each state is composed of an *unique identifier*, optional *strong transitions* and mandatory *sub-definitions* (i.e. automata may be hierarchical). The sub-definitions define how the dataflows are computed within that state. A transition is composed of a *boolean expression* and a *destination state*. When it evaluates to true, the automaton transitions to the destination state.

Note that we do not support *weak transitions*. The difference between a strong and a weak transition lies in the instant when the automaton state changes. Transitions and the automaton state are dataflows which all share the same rate. When a strong transition fires, the automaton state is immediately updated. On the contrary, when a weak transition fires, the automaton state is only updated the next time it is present. While strong transitions remain easy to grasp when combined with clock views, the same cannot be said about weak transitions [1]. See Example 18 below for an illustration of this issue.

As an additional restriction, we require that user-defined nodes have their rates manually annotated by the programmer. This guarantees that we can infer the correct clocks in the Clock Calculus (Chapter 15).

[1] As of time of writing, I may be the only person who is not confused and/or repulsed by this language feature, but merely disappointed.

Example 18 (Counter-intuitivity of weak transitions with clock views)**.** Suppose an automaton state with clock $(10, 0)$. State `S1` has a strong transition to state `S2` it has clock $(10, 0)$ **on** `S1`$(\texttt{state}, (10, 0))$. State `S2` has a weak transition to state `S1` it has clock $(10, 0)$ **on** `S2`$(\texttt{state}, (10, 0))$. Thus, each transition is only evaluated in its corresponding state and sees state changes immediately.

Suppose now that the automaton output `o` has clock $(10, 0)$ and view $(30, 0)$, i.e. `S1_o` and `S2_o` have respective clocks $(10, 0)$ **on** `S1`$(\texttt{state}, (30, 0))$ and $(10, 0)$ **on** `S2`$(\texttt{state}, (30, 0))$.

Assume the automaton is in state `S1` at some date $t \equiv 0 \bmod 30$. If the strong transition fires at date t, the state will change to `S2` at date t and `o` will observe the change immediately. If the transition fires at dates $t + 10$ or $t + 20$, the dataflow `state` will change immediately too, but `o` will only observe the change at date $t + 30$. This can be easily derived from the views and repeats every 30 time units.

Assume now that the automaton is in state `S2` at some date $t \equiv 0 \bmod 30$. If the weak transition fires at date t, the state will change to `S1` at date $t + 10$ and `o` will thus only observe the change at date $t + 30$. The same applies for date $t + 10$. However, if the transition fires at date $t + 20$, the automaton state changes at date $t + 30$ and thus `o` will observe the change without additional delay at date $t + 30$.

In general, a weak transition is without additional delay if it is fired at date $\pi(w) - \pi(s) \bmod \pi(w)$, where w is the view clock and s is the state clock.

Example 19 (Example program in the surface language)**.** The program below implements a simple crossbar-switch in the surface language. It consists of a node with three inputs, `i`, `j`, and `c`, and two outputs, `o` and `p`. When the automaton is in state `S1`, input `i` is directed to output `o` and input `j` is directed to output `p`. When the automaton is in state `S2`, it is the opposite. However, we first apply rate-transition operators such that rates match.

```
node main(i: int rate(10,0); j: int rate (20,0);
          c: bool rate (15,0))
returns (o,p)
let
  automaton
  | S1 ->
    unless c then S2;
    o = i;
    p = j;
  | S2 ->
    unless c then S1;
    o = j *^2;
    p = i /^2;
  end
tel
```

14.2 Core Language

The core language illustrated in Figure 14.2 is syntactically identical to the one presented in Section 7.2.

Recall that for an atom `a` with clock ck:

- `a/^k` sub-samples `a` by factor k and has clock $ck \,/.\, k$;

- `a*^k` up-samples `a` by repeating values k times and has clock $ck *. k$;
- `a~>k` delays each value of `a` by k and has clock $ck \Rightarrow. k$;
- `cst fby a` produces the value `cst` followed by the values of `a` and has clock ck, effectively delaying values of `a` by $\pi(ck)$;
- `cst :: a` produces the values of `a` prepended by the value `cst` and has clock $ck \rightharpoonup. - \pi(ck)$;
- `tail a` skips the first value of `a` and has clock $ck \rightharpoonup. \pi(ck)$;
- `a when C(c)` sub-samples `a` such that it produces values only if `c` as viewed by w produces `C`. It has clock ck **on** $C(c, w)$. Note that the view is not specified by the program, but instead inferred by the compiler;
- `merge(c, C0->a_c0, C1->a_c1)` combines the complementary flows `a_c0` and `a_c1` with respective clocks ck **on** $C0(c, w)$ and ck **on** $C1(c, w)$ and has clock ck.

14.3 Surface-to-Core Transpilation

Transpiling the surface to the core language is performed in three steps:

1. Replacing all constants by the appropriate literal
2. Transpiling automata into equations
3. Rewriting equations into ANF

Constant inlining and ANF transpilation are fairly straightforward. For the first, we simply substitute names representing constants by the appropriate value. For the second, we must modify function and operator applications. For each application argument, if it is not an atom (a constant or a variable), we introduce a fresh variable and sustitute this argument by it. Then, we introduce an equation whose left-hand side is the fresh variable and right-hand side the sustituted expression. If this expression is itself not in ANF, we apply this procedure recursively.

14.3.1 Automata Transpilation

In [28], a transpilation process for a synchronous dataflow language is provided. To remain self-contained, we will provide a similar definition in this Section. We chose the automaton transpilation technique for our work because it is a well-established technique that illustrates well the expressivity of clock views. Future work could study other language constructs or even enabling metaprogramming to construct a library of mode changing facilities.

To *flatten* automata definitions into equations, we define a family of functions, $flat$, in Figure 14.3. In addition, we define the function $proj$ which "projects" an expression into an automaton state.

The first function $flat_{def}$ takes a definition and returns its flattened representation. Thus, for an equation, it simply returns the equation. For an automaton, it returns the union of equations produced by $flat^{vars}_{auto}$, $flat^{stvs}_{auto}$ and $flat^{defs}_{auto}$. Note that the fresh integer i uniquely identifies this automaton.

Function $flat^{vars}_{auto}$ introduces the equations computing the automaton state. The dataflow `'i_s` holds the current automaton state which will be used by the actual equations of the automaton body. Dataflow `'i_ps` holds the previous value of `'i_s`, i.e. the state it should enter if no transition fires.

Let us illustrate how these dataflows interact. When the system starts, `'i_ps` is initialised at `S0` because of the `fby`. That means that we assume that the automaton is supposed to enter state `S0`. However, we must first verify whether or not transitions may be fired as these may change the state immediately such that we never enter state `S0`. The `merge` defining `'i_s` evaluates the strong transitions of the state we are supposed to enter, `S0`, and updates `'i_s` as appropriate. Having determined which state we are supposed to enter, this information can then be propagated to the dataflows of the state such that they may be evaluated (more on this later). Now, the next time we want to evaluate the automaton state, `'i_ps` holds the value computed for `'i_s` the previous time. However, we must first check if this state has some transitions which might fire. And thus the cycle repeats.

Function $flat^{stvs}_{auto}$ returns the equations which actually compute the transitions. The expressions defining those dataflows are themself defined by $flat^{trans}_{auto}$. The programmer may provide multiple transitions with different destination states. The notation [] denotes the empty list and $x :: [y]$ denotes a list whose head is x and tail is a list of ys. The semantics of our automata state that the equation defined earlier in syntactic order have higher priority. Thus, we evaluate transitions as nested `if` expressions. The expression evaluating a non-empty list of transitions is an `if` expression whose condition is the *projected* transition guard of the head, the `then`-branch is the destination state and the `else`-brach is the expression evaluating the tail of the list. The expression evaluating an empty list of transitions is simply the current state.

Function $flat^{defs}_{auto}$ returns the flattened definitions of the automaton body. It is itself the result of mapping $flat^{def}_{auto}$ onto those definitions. Applying $flat^{def}_{auto}$ on an equation results into applyinng $flat^{eq}_{auto}$ which produces a new equation with the left-hand side mangled such that there is no conflict between equations with the same left-hand side accross modes and right-hand side projected into the automaton state. Applying $flat^{def}_{auto}$ on an automaton results in first applying $flat_{def}$ onto that automaton and then mapping the same mangling/projection function onto the resulting equations.

The projection function $proj(\mathtt{e}, S_j, \mathtt{c})$ in Figure 14.4 itself turns an expression e into an expression which is only evaluated, iff the automaton, whose state dataflow is `c`, is within state S_j. The most important operation is that it applies a `when Sj(s)` on identifiers such that their dataflows are filtered as appropriate. For the other cases, it simply applies itself recursively.

Example 20 (Automaton transpilation)**.** The program below is the result of the automaton transpilation of the example above (with some cosmetic improvements).

```
node main(i: int rate(10,0); j: int rate (20,0);
          c: bool rate (15,0))
returns (o,p)
var s, ps;
let
  s = merge(ps, S1->S1_s, S2->S2_s);
  ps = S1 fby ns;
  S0_s = if c when S1(ps) then S2 else S1;
  S1_s = if c when S2(ps) then S1 else S2;

  o = merge(s, S1->S1_o, S2->S2_o);
  p = merge(s, S1->S1_p, S2->S2_p);

  S1_o = i when S1(s);
```

```
  S2_o = j when S2(s);

  S1_p = (j *^2) when S1(s);
  S2_p = (i /^2) when S2(s);
tel
```

$$
\begin{aligned}
\langle prog\rangle &::= \langle decl\rangle* \\
\langle decl\rangle &::= \langle nd\rangle \mid \langle ind\rangle \mid \langle tydecl\rangle \mid \langle constdef\rangle \\
\langle nd\rangle &::= \text{`}\texttt{node}\text{'}\ \langle id\rangle\ \text{`}\texttt{(}\text{'}\ \langle vars\rangle\ \text{`}\texttt{)}\text{'} \\
&\quad\ \text{`}\texttt{returns}\text{'}\ \text{`}\texttt{(}\text{'}\ \langle vars\rangle\ \text{`}\texttt{)}\text{'} \\
&\quad\ (\text{`}\texttt{var}\text{'}\ \langle vars\rangle\ \text{`}\texttt{;}\text{'})? \\
&\quad\ \text{`}\texttt{let}\text{'}\ \langle def\rangle+\ \text{`}\texttt{tel}\text{'} \\
\langle ind\rangle &::= \text{`}\texttt{imported}\text{'}\ \text{`}\texttt{node}\text{'}\ \langle id\rangle\ \text{`}\texttt{(}\text{'}\ \langle vars\rangle\ \text{`}\texttt{)}\text{'} \\
&\quad\ \text{`}\texttt{returns}\text{'}\ \text{`}\texttt{(}\text{'}\ \langle vars\rangle\ \text{`}\texttt{)}\text{'} \\
&\quad\ \langle indprop\rangle\ \text{`}\texttt{;}\text{'} \\
\langle tydecl\rangle &::= \text{`}\texttt{type}\text{'}\ \langle id\rangle\ \text{`}\texttt{=}\text{'}\ (\text{`}\texttt{|}\text{'}\ \langle id\rangle)+ \\
\langle constdef\rangle &::= \text{`}\texttt{const}\text{'}\ \text{`}\texttt{=}\text{'}\ \langle const\rangle \\
\langle def\rangle &::= \langle auto\rangle \mid \langle eq\rangle \\
\langle auto\rangle &::= \text{`}\texttt{automaton}\text{'}\ \langle state\rangle+\ \text{`}\texttt{end}\text{'} \\
\langle state\rangle &::= \text{`}\texttt{|}\text{'}\ \langle id\rangle\ \text{`}\texttt{->}\text{'}\ \langle strans\rangle*\ \langle def\rangle+\ \langle wtrans\rangle* \\
\langle strans\rangle &::= \text{`}\texttt{unless}\text{'}\ \langle expr\rangle\ \text{`}\texttt{then}\text{'}\ \langle id\rangle\ \text{`}\texttt{;}\text{'} \\
\langle eq\rangle &::= \langle id\rangle(\text{`}\texttt{,}\text{'}\ \langle id\rangle)*\ \text{`}\texttt{=}\text{'}\ \langle expr\rangle\text{`}\texttt{;}\text{'} \\
\langle expr\rangle &::= \langle atom\rangle \mid \langle id\rangle\ \text{`}\texttt{(}\text{'}\ \langle expr\rangle\ (\text{`}\texttt{,}\text{'}\ \langle expr\rangle)*\ \text{`}\texttt{)}\text{'} \mid \langle expr\rangle\ \text{`}\texttt{rate}\text{'}\ \langle ck\rangle \\
&\quad \mid \langle expr\rangle\ \text{`}\texttt{*\^{}}\text{'}\ \langle int\rangle \mid \langle expr\rangle\ \text{`}\texttt{/\^{}}\text{'}\ \langle int\rangle \mid \langle expr\rangle\ \text{`}\texttt{\~{}>}\text{'}\ \langle int\rangle \\
&\quad \mid \langle atom\rangle\ \text{`}\texttt{fby}\text{'}\ \langle expr\rangle \mid \langle atom\rangle\ \text{`}\texttt{::}\text{'}\ \langle expr\rangle \mid \text{`}\texttt{tail}\text{'}\ \langle expr\rangle \\
&\quad \mid \langle expr\rangle\ \text{`}\texttt{when}\text{'}\ \langle id\rangle\ \text{`}\texttt{(}\text{'}\ \langle id\rangle\ \text{`}\texttt{)}\text{'} \\
&\quad \mid \text{`}\texttt{merge}\text{'}\ \text{`}\texttt{(}\text{'}\ \langle id\rangle\ (\text{`}\texttt{,}\text{'}\ \langle id\rangle\ \text{`}\texttt{->}\text{'}\ \langle expr\rangle)+\ \text{`}\texttt{)}\text{'} \\
\langle atom\rangle &::= \langle id\rangle \mid \langle const\rangle \\
\langle vars\rangle &::= \langle var\rangle \mid \langle var\rangle\ \text{`}\texttt{;}\text{'}\ \langle vars\rangle \\
\langle var\rangle &::= \langle id\rangle \mid \langle id\rangle\ \text{`}\texttt{:}\text{'}\ \langle typ\rangle?\ (\text{`}\texttt{rate}\text{'}\ \langle ck\rangle)? \\
\langle typ\rangle &::= \text{`}\texttt{int}\text{'} \mid \text{`}\texttt{bool}\text{'} \mid \langle typ\rangle\ \text{`}\texttt{[}\text{'}\ \langle int\rangle\ \text{`}\texttt{]}\text{'} \mid \ldots \\
\langle ck\rangle &::= \text{`}\texttt{(}\text{'}\ \langle int\rangle\ \text{`}\texttt{,}\text{'}\ \langle int\rangle\ \text{`}\texttt{)}\text{'} \mid \langle ck\rangle\ \text{`}\texttt{on}\text{'}\ \langle id\rangle\ \text{`}\texttt{(}\text{'}\ \langle id\rangle\ \text{`}\texttt{)}\text{'} \\
\langle indprop\rangle &::= \langle wcet\rangle? \\
\langle wcet\rangle &::= \text{`}\texttt{wcet}\text{'}\ \langle int\rangle
\end{aligned}
$$

Figure 14.1: Syntax of the PRELUDE language

$$
\begin{aligned}
\langle prog\rangle &::= \langle decl\rangle* \\
\langle decl\rangle &::= \langle nd\rangle \mid \langle ind\rangle \mid \langle tydecl\rangle \\
\langle nd\rangle &::= \text{`}\texttt{node}\text{'}\ \langle id\rangle\ \text{`}\texttt{(}\text{'}\ \langle vars\rangle\ \text{`}\texttt{)}\text{'} \\
&\quad\ \text{`}\texttt{returns}\text{'}\ \text{`}\texttt{(}\text{'}\ \langle vars\rangle\ \text{`}\texttt{)}\text{'} \\
&\quad\ (\text{`}\texttt{var}\text{'}\ \langle vars\rangle\ \text{`}\texttt{;}\text{'})? \\
&\quad\ \text{`}\texttt{let}\text{'}\ \langle eq\rangle+\ \text{`}\texttt{tel}\text{'} \\
\langle ind\rangle &::= \text{`}\texttt{imported}\text{'}\ \text{`}\texttt{node}\text{'}\ \langle id\rangle\ \text{`}\texttt{(}\text{'}\ \langle vars\rangle\ \text{`}\texttt{)}\text{'} \\
&\quad\ \text{`}\texttt{returns}\text{'}\ \text{`}\texttt{(}\text{'}\ \langle vars\rangle\ \text{`}\texttt{)}\text{'} \\
&\quad\ \langle indprop\rangle\ \text{`}\texttt{;}\text{'} \\
\langle tydecl\rangle &::= \text{`}\texttt{type}\text{'}\ \langle id\rangle\ \text{`}\texttt{=}\text{'}\ (\text{`}\texttt{|}\text{'}\ \langle id\rangle)+ \\
\langle eq\rangle &::= \langle id\rangle(\text{`}\texttt{,}\text{'}\ \langle id\rangle)*\ \text{`}\texttt{=}\text{'}\ \langle expr\rangle \\
\langle expr\rangle &::= \langle atom\rangle \mid \langle id\rangle\ \text{`}\texttt{(}\text{'}\ \langle atom\rangle(\text{`}\texttt{,}\text{'}\ \langle atom\rangle)*\ \text{`}\texttt{)}\text{'} \mid \langle atom\rangle\ \text{`}\texttt{rate}\text{'}\ \langle ck\rangle \\
&\quad \mid \langle atom\rangle\ \text{`}\texttt{*\^{}}\text{'}\ \langle int\rangle \mid \langle atom\rangle\ \text{`}\texttt{/\^{}}\text{'}\ \langle int\rangle \mid \langle atom\rangle\ \text{`}\texttt{\~{}>}\text{'}\ \langle int\rangle \\
&\quad \mid \langle const\rangle\ \text{`}\texttt{fby}\text{'}\ \langle atom\rangle \mid \langle const\rangle\ \text{`}\texttt{::}\text{'}\ \langle atom\rangle \mid \text{`}\texttt{tail}\text{'}\ \langle atom\rangle \\
&\quad \mid \langle atom\rangle\ \text{`}\texttt{when}\text{'}\ \langle id\rangle\ \text{`}\texttt{(}\text{'}\ \langle id\rangle\ \text{`}\texttt{)}\text{'} \\
&\quad \mid \text{`}\texttt{merge}\text{'}\ \text{`}\texttt{(}\text{'}\ \langle id\rangle\ (\text{`}\texttt{,}\text{'}\ \langle id\rangle\ \text{`}\texttt{->}\text{'}\ \langle atom\rangle)+\ \text{`}\texttt{)}\text{'} \\
\langle atom\rangle &::= \langle id\rangle \mid \langle const\rangle \\
\langle vars\rangle &::= \langle var\rangle \mid \langle var\rangle\ \text{`}\texttt{;}\text{'}\ \langle vars\rangle \\
\langle var\rangle &::= \langle id\rangle \mid \langle id\rangle\ \text{`}\texttt{:}\text{'}\ \langle typ\rangle?\ (\text{`}\texttt{rate}\text{'}\langle ck\rangle)? \\
\langle typ\rangle &::= \text{`}\texttt{int}\text{'} \mid \text{`}\texttt{bool}\text{'} \mid \langle typ\rangle\ \text{`}\texttt{[}\text{'}\ \langle int\rangle\ \text{`}\texttt{]}\text{'} \mid \ldots \\
\langle ck\rangle &::= \text{`}\texttt{(}\text{'}\ \langle int\rangle\ \text{`}\texttt{,}\text{'}\ \langle int\rangle\ \text{`}\texttt{)}\text{'} \mid \langle ck\rangle\ \text{`}\texttt{on}\text{'}\ \langle id\rangle\ \text{`}\texttt{(}\text{'}\ \langle id\rangle\ \text{`}\texttt{)}\text{'} \\
\langle indprop\rangle &::= \langle wcet\rangle? \\
\langle wcet\rangle &::= \text{`}\texttt{wcet}\text{'}\ \langle int\rangle
\end{aligned}
$$

Figure 14.2: Syntax of the core language

$$
\begin{aligned}
flat_{def}(\texttt{x = e;}) &= \texttt{x = e;} \\
flat_{def}(\texttt{automaton } S_0\texttt{->}sb_0 \ldots S_n\texttt{->}sb_n) &= flat^{vars}_{auto}(S_0 \ldots S_n, i) \cup (\bigcup_{j=0}^{n} flat^{stvs}_{auto}(S_j, s_b, i)) \\
&\quad \cup (\bigcup_{j=0}^{n} flat^{defs}_{auto}(S_j, s_b, i)) \\
&\quad \text{where } i = fresh(\texttt{int}) \\
flat^{vars}_{auto}(S_0 \ldots S_n, i) &= \\
&\quad \texttt{'i\_s = merge('i\_pns, S0->'i\_S0\_s, ... , Sn->'i\_Sn\_s)} \\
&\quad \texttt{'i\_ns = merge('i\_s, S0->'i\_S0\_ns, ... , Sn->'i\_Sn\_ns)} \\
&\quad \texttt{'i\_pns = S0 fby 'i\_ns} \\
flat^{stvs}_{auto}(S_j, (ts, \_, tw), i) &= \\
&\quad \texttt{'i\_Sj\_s =} flat^{trans}_{auto}(S_j, ts, \texttt{'i\_pns}) \\
&\quad \texttt{'i\_Sj\_ns =} flat^{trans}_{auto}(S_j, tw, \texttt{'i\_s}) \\
flat^{trans}_{auto}(S_j, (e, S_k) :: [ts], c) &= \texttt{if } proj(\texttt{e}, S_j, c) \texttt{ then Sk else } flat^{ts}_{auto}(S_j, ts, c) \\
flat^{trans}_{auto}(S_j, [], c) &= \texttt{Sj} \\
flat^{defs}_{auto}(S_j, (\_, defs, \_), i) &= map(\lambda def.\ flat^{def}_{auto}(S_j, def, i), defs) \\
flat^{def}_{auto}(S_j, \texttt{x = e}, i) &= flat^{eq}_{auto}(S_j, \texttt{x = e}, \texttt{'i\_Sj\_s}) \\
flat^{def}_{auto}(S_j, \texttt{automaton } auto, i) &= map(\lambda eq.\ flat^{eq}_{auto}(S_j, eq, \texttt{'i\_Sj\_s}), eqs) \\
&\quad \text{where } eqs = flat_{def}(\texttt{automaton } auto) \\
flat^{eq}_{auto}(S_j, \texttt{x = e}, i) &= \\
&\quad \texttt{'i\_Sj\_x = } proj(\texttt{e}, S_j, c)
\end{aligned}
$$

Figure 14.3: Automata flattening procedure

$$
\begin{aligned}
proj(\texttt{id}, S_j, c) &= \texttt{id when Sj(c)} \\
proj(\texttt{const}, S_j, c) &= \texttt{const} \\
proj(\texttt{f(i)}, S_j, c) &= \texttt{f}(proj(\texttt{i}, S_j, c)) \\
proj(\texttt{e rate (n,p)}, S_j, c) &= proj(\texttt{e}, S_j, c)\ \texttt{rate (n,p) on Sj(c)} \\
proj(\texttt{e *\^{}k}, S_j, c) &= proj(\texttt{e}, S_j, c)\ \texttt{*\^{}k} \\
proj(\texttt{e /\^{}k}, S_j, c) &= proj(\texttt{e}, S_j, c)\ \texttt{/\^{}k} \\
proj(\texttt{e \~{}>k}, S_j, c) &= proj(\texttt{e}, S_j, c)\ \texttt{\~{}>k} \\
proj(\texttt{const fby e}, S_j, c) &= \texttt{const fby}\ proj(\texttt{e}, S_j, c) \\
proj(\texttt{const :: e}, S_j, c) &= \texttt{const ::}\ proj(\texttt{e}, S_j, c) \\
proj(\texttt{tail e}, S_j, c) &= \texttt{tail}\ proj(\texttt{e}, S_j, c) \\
proj(\texttt{e when C(d)}, S_j, c) &= proj(\texttt{e}, S_j, c)\ \texttt{when C(d)} \\
proj(\texttt{merge(d, C0->e0, C1->e1)}, S_j, c) &= \texttt{merge(d, C0->}proj(\texttt{e0}, S_j, c)\texttt{, C1->}proj(\texttt{e1}, S_j, c)\texttt{)}
\end{aligned}
$$

Figure 14.4: Automata projection procedure

Chapter 15

Clock calculus

In this Chapter, we will present a clock calculus, that is able to ascribe a clock to programs in our extended language. Most notably, it is able to reason about views. While the previous clock calculus of the PRELUDE language relied on Hindley-Milner type inference extended with subtyping [78], this clock calculus will rely on *bidirectional refinement typing*.

A note on terminology before talking in more detail about the clock calculus: Clocks (Chapter 12) are elements of the synchronous Kahn semantics (Chapter 13), the operational semantics of the language. Thus, they define the behavior of an abstract machine. The clock types of this Chapter are their reconstruction from a program as done by the the clock calculus. When it is without ambiguity, we use the terms clock and clock type interchangeably.

15.1 Bidirectional Typing

Our work relies on bidirectional typing [31] rather than Hindley-Milner (HM) type inference. Their difference is best illustrated by the structure of their typing judgments. In HM type inference, the judgment $E \vdash e : t$ associates to expression e the type t under environment E. However, how this information is obtained is unspecified. On the other hand in bidirectional typing, such a statement could have any one of two forms, $E \vdash e \Leftarrow t$ or $E \vdash e \Rightarrow t$. The first states a type check, i.e. we verify that t is a valid type for e even though e might be associated to a different type u. The second states a type synthesis, i.e. we construct a type t for e under the environment E.

While this distinction is of little use in the simply typed lambda calculus, HM type inference is often undecidable for more complex type systems. This is because it may perform type checking or type synthesis at any point. Type checking is decidable since the environment, expression and type are all inputs and one must simply verify whether the relation holds. In type synthesis however, the type is an output, making it potentially undecidable. Making the distinction explicit allows to delimit which parts of the type system are decidable and thus where the type checker may infer types automatically, and which are undecidable and thus require programmer input.

15.2 Refinement Typing

Refinement typing [39, 83, 101] is a typing discipline where types may be *refined* by logical predicates. A refinement type is written $\{\nu{:}b \mid r\}$. It reads: "a *base type* b (e.g. `int` or `bool list`) is *refined* by a boolean predicate r (e.g. $\nu > 0$ or $length(\nu) < x$) such that r is true for all values inhabiting $\{\nu : b \mid r\}$ and where the special variable ν represent the value of the typed expression".

Let us illustrate refinement types. In more classical programming languages, the expression `4` would have type `int`. In a language with refinement types, it would have type $\{\nu{:}\texttt{int} \mid \nu = 4\}$, meaning "an `int` whose value is equal to 4". Functions may also have refined types. The function `(/)` would have type $a{:}\texttt{int} \to b{:}\{\nu{:}\texttt{int} \mid \nu \neq 0\} \to \{\nu{:}\texttt{int} \mid \nu = a/b\}$. This type describes a function taking an argument a of type `int` and an argument b of type `int` whose value is not equal to 0, and returning an `int` whose value is equal to a/b.

When typing an expression such as `(/) a b`, a refinement typer must perform two tasks. First, it must verify the consistency of base types. This can be solved with classical techniques such as HM typing [78], in our case bidirectional typing [31]. Second, it must verify the consistency of logical predicates. Typically, a dedicated solver is used for this. Our work will rely on the Z3 SMT solver [64] to check the satisfiability of predicates.

The kind of constraints in a refinement type system is important too. Ideally, they should belong to a decidable class of problems. A problem is decidable, iff there exists an *effective method* to solve it [1]. A method is effective for solving a problem, if it consists of a finite number of instructions to execute and these instructions can be followed rigorously without any form of ingenuity. For instance, addition of integers is a decidable problem, because of the existence of effective methods as taught in primary school.

For our clock calculus, we will use constraints belonging to the decidable logic of Quantifier-Free Linear Integer Arithmetic (QFLIA), also called Presburger arithmetic [79]. "Linear" here means that the only operation allowed between integer values are addition and subtraction, adding multiplication or division would lead to undecidability. However, our formalization in Chapter 12 and Chapter 13 features multiplication and division. Luckily, a limited subset of multiplication and division is still expressible in this logic. When submitting constraints to our SMT solver, we can use the following properties which gives us sufficient expressivity to reason about our clocks. Assuming a and b are variables and k is an explicit constant:

1. **Multiplication**: An expression $a * k$ may be substituted by $a + a + \ldots + a$ where a is added to itself k times;

2. **Division**: An expression a/k may be substituted by the fresh variable b and a new constraint $b * k = a$ where a is the dividend, k the divisor and b the quotient;

3. **Divisibility**: A constraint k **div** a is similar to division except that we only care that the relation holds, the result is irrelevant, and thus the constraint may be substituted similarly by the constraint $b * k = a$ where b is a fresh variable.

For a more in-depth review of refinement types, readers may either refer to the cited literature or this excellent tutorial [47].

15.3 Overview

Our clock calculus is divided into three passes. This division is typical in type systems based on refinement types [47] as it allows us to divide the complex typing problem into simpler individual problems:

1. Structural Clock Calculus

2. Refinement Clock Calculus

```
1  node main(i : int rate (10,0); j: int rate (45,0); c : bool rate (15,0))
2  returns (o : int rate (30,0))
3  var a,b,x,y,z;
4  let
5    a = i when true(c);
6    b = a /^3;
7    x = j /^2;
8    y = x when false(c);
9    z = y *^3;
10   o = merge(c, true->b, false->z);
11 tel
```

Figure 15.1: The running example

$\texttt{i} : \langle \texttt{pck} \mid 10, 0 \rangle$ $\quad$ $\texttt{j} : \langle \texttt{pck} \mid 45, 0 \rangle$

$\texttt{c} : \langle \texttt{pck} \mid 15, 0 \rangle$ $\quad$ $\texttt{o} : \langle \texttt{pck} \mid 30, 0 \rangle$

$\texttt{a} : \langle \texttt{pck on true} (\texttt{c}, \langle \texttt{pck} \mid 90, 0 \rangle) \mid 10, 0 \rangle$ $\quad$ $\texttt{b} : \langle \texttt{pck on true} (\texttt{c}, \langle \texttt{pck} \mid 90, 0 \rangle) \mid 30, 0 \rangle$

$\texttt{x} : \langle \texttt{pck} \mid 90, 0 \rangle$

$\texttt{y} : \langle \texttt{pck on true} (\texttt{c}, \langle \texttt{pck} \mid 90, 0 \rangle) \mid 90, 0 \rangle$ $\quad$ $\texttt{z} : \langle \texttt{pck on true} (\texttt{c}, \langle \texttt{pck} \mid 90, 0 \rangle) \mid 30, 0 \rangle$

Figure 15.2: The result of the clock calculus on the running example

3. View Closing

In the *Structural Clock Calculus*, only the structure of clocks is inferred. This pass is very similar to a typical typing pass. Refinements are ignored and replaced by *refinement holes*, i.e. refinement placeholders. Refinement holes are opaque and type checking them always succeeds in this pass. After this pass, expressions and nodes are fully typed except for their refinements. In the *Refinement Clock Calculus*, the actual refinements of clocks are verified. Finally, we delay view computation until the last point, the *View Closing* pass. Before that, views only collect constraints without checking them.

15.4 Running Example

Throughout this Chapter, we will illustrate our clock calculus via the running example in Figure 15.1. It features three sensors, `i`, `j`, and `c`, and one actuator `o`. We use `c` to conditionally sub-sample `i` and `j`. We apply rate-transition operators before and after this sub-sampling and then **`merge`** the results back together to produce the output `o`.

Figure 15.2 shows the clocks of different variables as determined by our clock calculus. Examples will illustrate how this is determined.

15.5 Structural Clock Calculus

The goal of this pass is to infer the structure of clocks, that is to say clock types where refinements are left unknown and represented by refinement holes. This means in particular that clock conditions

are inferred during this pass, while periods and offsets are inferred during the following pass, the refinement clock calculus.

The judgments of the structural clock calculus are the following. The S indicates that these are judgments of the structural clock calculus.

- Synthesis $H \vdash x \overset{S}{\Rightarrow} \sigma$: Under environment H, a type σ could be constructed for x;
- Checking $H \vdash x \overset{S}{\Leftarrow} \sigma$: Under environment H, type σ is valid for x;
- Subtyping $H \vdash \sigma \overset{S}{<:} \sigma'$: Under environment H, type σ is a subtype of σ';
- Well-formedness $H \vdash x\checkmark^S$: Under environment H, x is well-formed;
- Instantiation $H \vdash \sigma \overset{S}{\Rrightarrow} \sigma'$: Under environment H, type σ can be instantiated into type σ';
- Membership $x \overset{S}{\in} H$: Environment H contains x.

15.5.1 Clock Language

Figure 15.3 illustrates the clock system of the structural clock calculus. A clock type (σ) is either a polymorphic clock ($\forall\alpha.\sigma$) or a clock expression. A clock expression (ck_e) is either a dependent clock function ($x{:}ck_r \to ck_e$), a tuple ($ck_e \times ck_e$) or a refined clock. A refined clock (ck_r) is a base clock refined by a refinement hole ($\{\nu{:}ck_b \mid \star\}$). A base clock ($ck_b$) is either a periodic clock (`pck`), a clock variable (α), a conditioned base clock (ck_b **on** $C\,(c, ck_r)$), or the structural equivalent of a base clock ($\zeta(ck_b, \star)$).

An environment (H) is either empty or an environment extended by a variable binding ($H; x{:}\sigma$), a type ($H; \alpha$), or a refinement hole ($H; \star$).

Function definitions are always curried even though our semantics do not support partial application. This allows us however to simplify the type checker: without curried functions, we would have to define dependent tuples in addition to dependent functions.

Let us discuss the notion of structural equivalence. We focus on this type constructor in particular as it is something unique to our work. As shown in Chapter 12, applying a periodic clock operator on a clock changes not only the period and offset of the dataflow, but also the period and offset of the views. Thus, the type system must also be able to reflect this. This is the role of the structural equivalence type constructor ζ. If we take the `~>k` operator, its semantics states that it accepts a dataflow whose clock features an arbitrary amount of **on** in it and returns a dataflow whose clock has the same "structure", i.e. the same amount of **on** applied to it, but whose offset and view offsets are delayed by k.

In Section 15.6.3, its clock is shown to be $\forall\alpha.e{:}\{\nu{:}\alpha \mid \star_0\} \to \{\nu{:}\zeta(\alpha, \star_2) \mid \star_1\}$. Dissecting this clock type, we see that it is a polymorphic function and thus indeed accepts any clock as input. For instance, if we apply it to an argument `i` with clock type $\{\nu{:}$`pck` **on** `C` $(\mathtt{c}, \{\nu{:}\mathtt{pck} \mid \star_3\}) \mid \star_4\}$, we can instantiate α following the instantiation rules (Section 15.5.5). However, we cannot reuse the same instantiated clock for the output as obviously the offset of the output has changed and we must thus represent this in the refinements. This is where the ζ operator becomes relevant. When instantiating such a type, one has to follow the DeStInstSEq-1 and DeStInstSEq-2 rules (Figure 15.7). If α has been instantiated to a `pck`, it is unchanged. However, if α has been instantiated to a **on**, the refinement (hole) of the **on** has to be substituted by a new refinement (hole). This will allow us to accurately describe the change a rate-transition operator applies to the views of its output. The formal definition

$$
\begin{array}{lcl}
\sigma & ::= & \forall\alpha.\sigma \mid ck_e \\
ck_e & ::= & x{:}ck_r \rightarrow ck_e \mid ck_e \times ck_e \mid ck_r \\
ck_r & ::= & \{\nu{:}ck_b \mid \star\} \\
ck_b & ::= & \texttt{pck} \mid \alpha \mid ck_b \textbf{ on } C\,(c, ck_r) \mid \zeta(ck_b, \star) \\
H & ::= & \varnothing \mid H; x{:}\sigma \mid H; \alpha \mid H; \star \\
\multicolumn{3}{c}{x : \text{Variable}}
\end{array}
$$

Figure 15.3: Declarative Structural Clock System

of ζ relies on the instantiation rules in Section 15.5.5 (structural clock calculus) and Section 15.6.6 (refinement clock calculus).

15.5.2 Initial Environment

The initial environment of the structural clock calculus is shown in Figure 15.4. It contains the built-in operators. As this is the structural clock calculus, most details are hidden away, but one can already see the use of the ζ operator.

15.5.3 Well-formedness Rules

The first kind of rule we will discuss are the well-formedness rules in Figure 15.5. Their role is to verify that the clock types we manipulate are sound. In OCaml for instance, they distinguish sound types such as `int` or `(int -> bool) list`, from unsound types such as `list` or `int int`. The rules are:

- DeStAbsWF: A polymorphic clock is well-formed, if the polymorphic type is sound within the environment extended by the polymorphic variable;
- DeStFunWF: A clock function is well-formed, if the input clock is well-formed and the output clock is well-formed in the environment extended by the input clock, i.e. functional clocks are dependent types;
- DeStTupWF: A clock tuple is well-formed, if the individual clocks are well-formed;
- DeStRefWF: A refined clock is well-formed, if both the base clock and refinement hole are well-formed;
- DeStReHWF: A refinement hole is well-formed, if it is part of the environment;
- DeStSEqWF: A structurally equivalent clock is well-formed, if its constituents are well-formed;
- DeStPckWF: A `pck` is always well-formed;
- DeStVarWF: A variable is well-formed, if it is part of the environment;
- DeStOnWF: A **on** is well-formed, if it's constituents are well-formed.

`e when C(c)` :

$$\forall \alpha.e{:}\{\nu{:}\alpha \mid \star_0\} \to c{:}\{\nu{:}\zeta(\alpha, \star_1) \mid \star_2\} \to \{\nu{:}\zeta(\alpha, \star_3)\ \mathbf{on}\ C\,(c, \{\nu{:}\texttt{pck} \mid \star_4\}) \mid \star_5\}$$

`merge(c, C0->e0, C1->e1)` :

$$\forall \alpha.c{:}\{\nu{:}\alpha \mid \star_0\} \to e0{:}\{\nu{:}\zeta(\alpha, \star_1)\ \mathbf{on}\ C0\,(c, \{\nu{:}\texttt{pck} \mid \star_2\}) \mid \star_3\} \to e1{:}\{\nu{:}\zeta(\alpha\ \mathbf{on}\ C1\,(c, \{\nu{:}\texttt{pck} \mid \star_4\})\,, \star_5) \mid \star_6\} \to \{\nu{:}\zeta(\alpha, \star_7) \mid \star_8\}$$

`e*^k` :

$$\forall \alpha.e{:}\{\nu{:}\alpha \mid \star_0\} \to \{\nu{:}\zeta(\alpha, \star_1) \mid \star_2\}$$

`e/^k` :

$$\forall \alpha.e{:}\{\nu{:}\zeta(\alpha, \star_0) \mid \star_1\} \to \{\nu{:}\zeta(\alpha, \star_2) \mid \star_3\}$$

`C fby e`

$$\forall \alpha.e{:}\{\nu{:}\alpha \mid \star_0\} \to \{\nu{:}\zeta(\alpha, \star_1) \mid \star_2\}$$

`e~>k` :

$$\forall \alpha.e{:}\{\nu{:}\alpha \mid \star_0\} \to \{\nu{:}\zeta(\alpha, \star_1) \mid \star_2\}$$

`C::e` :

$$\forall \alpha.e{:}\{\nu{:}\zeta(\alpha, \star_0) \mid \star_1\} \to \{\nu{:}\zeta(\alpha, \star_2) \mid \star_3\}$$

`tail e` :

$$\forall \alpha.e{:}\{\nu{:}\alpha \mid \star_0\} \to \{\nu{:}\zeta(\alpha, \star_1) \mid \star_2\}$$

`e rate (n,p)` :

$$e{:}\{\nu{:}\texttt{pck} \mid \star_0\} \to \{\nu{:}\texttt{pck} \mid \star_1\}$$

Figure 15.4: Initial Declarative Structural Clock Environment

$$\textsc{DeStAbsWF}\ \frac{H, \alpha \vdash \sigma\checkmark^S}{H \vdash \forall\alpha.\sigma\checkmark^S} \qquad \textsc{DeStFunWF}\ \frac{H \vdash ck_r\checkmark^S \qquad H, x : ck_r \vdash ck_e\checkmark^S}{H \vdash x{:}ck_r \to ck_e\checkmark^S} \qquad \textsc{DeStTupWF}\ \frac{H \vdash ck_e\checkmark^S \qquad H \vdash ck'_e\checkmark^S}{H \vdash ck_e \times ck'_e\checkmark^S}$$

$$\textsc{DeStRefWF}\ \frac{H \vdash ck_b\checkmark^S \qquad H \vdash \star\checkmark^S}{H \vdash \{\nu{:}ck_b \mid \star\}\checkmark^S} \qquad \textsc{DeStReHWF}\ \frac{\star \overset{S}{\in} H}{H \vdash \star\checkmark^S} \qquad \textsc{DeStSEqWF}\ \frac{H \vdash ck_b\checkmark^S \qquad H \vdash \star\checkmark^S}{H \vdash \zeta(ck_b, \star)\checkmark^S}$$

$$\textsc{DeStPckWF}\ \frac{}{H \vdash \texttt{pck}\checkmark^S} \qquad \textsc{DeStVarWF}\ \frac{\alpha \overset{S}{\in} H}{H \vdash \alpha\checkmark^S} \qquad \textsc{DeStOnWF}\ \frac{H \vdash ck_b\checkmark^S \qquad H \vdash \star\checkmark^S}{H \vdash ck_b\ \mathbf{on}\ C\,(c, \{\nu{:}\texttt{pck} \mid \star\})\checkmark^S}$$

Figure 15.5: Declarative Structural Clock Well-Formedness

15.5.4 Subtyping Rules

A subtype relation $H \vdash t \overset{S}{<:} u$ states that under environment H, t is a subtype of u, meaning that all values inhabiting t also inhabit u. We didn't define subtyping rules for clock functions and polymorphic clocks as our semantics do not support this kind of dataflow. The rules are shown in Figure 15.6:

- DeStSubPck: A `pck` is a subtype of itself;
- DeStSubVar: A clock variable is a subtype of itself;
- DeStSubOn: A conditional clock is a subtype of another, if their constructor and condition are identical, and the first conditioned clocks is a subtype of the second, and the first view is a subtype of the second;
- DeStSubRef: A refined clock is a subtype of another, if the first base clock is a subtype of the second. Note that we do not check refinements;
- DeStSubTup: A tuple clock is a subtype of another, if their respective constituents are subtypes of another.

15.5.5 Instantiation Rules

Instantiation rules describe how to construct (instantiate) an "abstract" clock into a more "concrete" one. We use them for two cases. First, to construct a clock from the syntactic annotations of the programmer (or the lack thereof). Second, when using expressions featuring polymorphic clocks. As an expression with a polymorphic clock, such as a rate-transition operator or an imported node, must obviously first be instantiated before being applied. The rules are shown in Figure 15.7:

- DeStDeclPck: A strictly periodic rate annotation instantiates into a (well-formed) refined `pck`;

$$\textsc{DeStSubPck}\quad H \vdash \texttt{pck} \overset{S}{<:} \texttt{pck}$$

$$\textsc{DeStSubVar}\quad H \vdash \alpha \overset{S}{<:} \alpha$$

$$\textsc{DeStSubOn}\quad \frac{H \vdash ck_b \overset{S}{<:} ck_b' \qquad H \vdash w \overset{S}{<:} w'}{H \vdash ck_b\ \mathbf{on}\ C\,(c, w) \overset{S}{<:} ck_b'\ \mathbf{on}\ C\,(c, w')}$$

$$\textsc{DeStSubRef}\quad \frac{H \vdash ck_b \overset{S}{<:} ck_b'}{H \vdash \{\nu{:}ck_b \mid \star\} \overset{S}{<:} \{\nu{:}ck_b' \mid \star'\}}$$

$$\textsc{DeStSubTup}\quad \frac{H \vdash ck_r^0 \overset{S}{<:} ck_r^A \qquad H \vdash ck_r^1 \overset{S}{<:} ck_r^B}{H \vdash ck_r^0 \times ck_r^1 \overset{S}{<:} ck_r^A \times ck_r^B}$$

Figure 15.6: Declarative Structural Subtyping Rules

- DeStDeclOn: To instantiate a conditional rate annotation, first instantiate the clock declaration without the conditional and then apply an **on** operator;
- DeStDeclEmpty: An empty rate annotation can be instantiated into any well-formed refined clock;
- DeStInstPoly: To instantiate a polymorphic clock, substitute each occurrence of the polymorphic clock by a well-formed base clock and then instantiate the resulting clock type;
- DeStInstFun: To instantiate a function, instantiate its components;
- DeStInstTup: To instantiate a tuple, instantiate its components;
- DeStInstRef: To instantiate a refined clock, instantiate its base clock;
- DeStInstSEq-1: An instantiated structural equivalent of a `pck`, is a `pck`;
- DeStInstSEq-2: To instantiate the structural equivalent of a conditional clock, first instantiate the structural equivalent of the conditioned clock and then reapply the **on** with a new view;
- DeStInstPck: A `pck` instantiates to itself;
- DeStInstOn: To instantiate an **on**, instantiate the conditioned clock and the view.

Example 21 (Structural Instantiation Rules)**.** The structural instantiation rules are applied whenever a built-in operator is used. For instance, when instantiating the `/^2` operator at Line 7 in Figure 15.1, the following judgments are produced by the clock calculus. First (Equation (15.4)), the rule DeStInstPoly states that to instantiate a polymorphic type, we substitute the type variable α by a well-formed base clock. We must then recursively instantiate this clock type after performing the substitution. Here, the constraints of the program require us to substitute α by `pck`. After this, the recursive call requires us to apply rule DeStInstFun (Equation (15.3)). This rule simply requires us to instantiate the input and output clock types. Continuing with the input type, this requires us to apply rule DeStInstRef (Equation (15.2)) which requires us to instantiate the base clock of the refined clock. To instantiate this clock, we must follow rule DeStInstSEq-1 (Equation (15.1)) which tells us how to instantiate a ζ applied to a `pck`. Since there are no view refinements in a `pck`, the instantiation is `pck` itself. After this, similar judgments are performed for the output type and the clock calculus can conclude the instantiation of `/^2`.

$$\frac{}{H \vdash \zeta(\texttt{pck}, \star_0) \overset{S}{\Rightarrow} \texttt{pck}} \tag{15.1}$$

$$\frac{H \vdash \zeta(\texttt{pck}, \star_0) \overset{S}{\Rightarrow} \texttt{pck}}{H \vdash \{\nu{:}\zeta(\texttt{pck}, \star_0) \mid \star_1\} \overset{S}{\Rightarrow} \{\nu{:}\texttt{pck} \mid \star_1\}} \tag{15.2}$$

$$\frac{H \vdash \{\nu{:}\zeta(\texttt{pck}, \star_0) \mid \star_1\} \overset{S}{\Rightarrow} \{\nu{:}\texttt{pck} \mid \star_1\} \qquad H \vdash \{\nu{:}\zeta(\texttt{pck}, \star_2) \mid \star_3\} \overset{S}{\Rightarrow} \{\nu{:}\texttt{pck} \mid \star_3\}}{H \vdash e{:}\{\nu{:}\zeta(\texttt{pck}, \star_0) \mid \star_1\} \to \{\nu{:}\zeta(\texttt{pck}, \star_2) \mid \star_3\} \overset{S}{\Rightarrow} e{:}\{\nu{:}\texttt{pck} \mid \star_1\} \to \{\nu{:}\texttt{pck} \mid \star_3\}} \tag{15.3}$$

$$\frac{\begin{array}{c} H \vdash \texttt{pck}\checkmark^S \\ H \vdash e{:}\{\nu{:}\zeta(\alpha, \star_0) \mid \star_1\} \to \{\nu{:}\zeta(\alpha, \star_2) \mid \star_3\}[\alpha := \texttt{pck}] \overset{S}{\Rightarrow} e{:}\{\nu{:}\texttt{pck} \mid \star_1\} \to \{\nu{:}\texttt{pck} \mid \star_3\} \end{array}}{H \vdash \forall\alpha.e{:}\{\nu{:}\zeta(\alpha, \star_0) \mid \star_1\} \to \{\nu{:}\zeta(\alpha, \star_2) \mid \star_3\} \overset{S}{\Rightarrow} e{:}\{\nu{:}\texttt{pck} \mid \star_1\} \to \{\nu{:}\texttt{pck} \mid \star_3\}} \tag{15.4}$$

Now, let's look at an example featuring an **on** operator. At Line 9 in the running example, operator `*^3` is instantiated. Because it is applied to an argument featuring an **on**, we must substitute α by $\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\})$ in rule DeStInstPoly (Equation (15.8)). We then apply rule DeStInstFun (Equation (15.7)) similarly. This time however, the instantiation of the input clock type is fairly straightforward and we don't show it. However, when instantiating the output clock type, we must apply rule DeStInstSEq-2 (Equation (15.6)) which replaces the refinement hole of the view by a new one, indicating the change it produces wrt. the view of the input clock type. Finally, the instantiation of the ζ operator recurses and terminates with rule DeStInstSEq-1 (Equation (15.5)).

$$\frac{}{H \vdash \zeta(\texttt{pck}, \star_4) \overset{S}{\Rightarrow} \texttt{pck}} \tag{15.5}$$

$$\frac{H \vdash \zeta(\texttt{pck}, \star_4) \overset{S}{\Rightarrow} \texttt{pck} \qquad H \vdash \star_5\checkmark^S}{H \vdash \{\nu{:}\zeta(\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\})\,, \star_1) \mid \star_2\} \overset{S}{\Rightarrow} \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_5\}) \mid \star_3\}} \tag{15.6}$$

$$\frac{\begin{array}{c} H \vdash \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\}) \mid \star_0\} \overset{S}{\Rightarrow} \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\}) \mid \star_0\} \\ H \vdash \{\nu{:}\zeta(\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\})\,, \star_1) \mid \star_2\} \overset{S}{\Rightarrow} \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_5\}) \mid \star_3\} \end{array}}{\begin{array}{c} H \vdash e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\}) \mid \star_0\} \to \{\nu{:}\zeta(\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\})\,, \star_1) \mid \star_2\} \overset{S}{\Rightarrow} \\ e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\}) \mid \star_1\} \to \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_5\}) \mid \star_3\} \end{array}} \tag{15.7}$$

$$\frac{\begin{array}{c} H \vdash \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\})\,\checkmark^S \\ H \vdash e{:}\{\nu{:}\alpha \mid \star_0\} \to \{\nu{:}\zeta(\alpha, \star_1) \mid \star_2\}[\alpha := \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\})] \overset{S}{\Rightarrow} \\ e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\}) \mid \star_1\} \to \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_5\}) \mid \star_3\} \end{array}}{\begin{array}{c} H \vdash \forall\alpha.e{:}\{\nu{:}\zeta(\alpha, \star_0) \mid \star_1\} \to \{\nu{:}\zeta(\alpha, \star_2) \mid \star_3\} \overset{S}{\Rightarrow} \\ e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\}) \mid \star_1\} \to \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_5\}) \mid \star_3\} \end{array}} \tag{15.8}$$

$$\textsc{DeStDeclPck}\ \frac{H \vdash \{\nu{:}\texttt{pck} \mid \star\}\checkmark^S}{H \vdash \texttt{rate (n,p)} \overset{S}{\Rrightarrow} \{\nu{:}\texttt{pck} \mid \star\}}$$

$$\textsc{DeStDeclOn}\ \frac{H \vdash \texttt{rate}\ \langle ck \rangle \overset{S}{\Rrightarrow} \{\nu{:}ck_b \mid \star\} \qquad H \vdash \left\{\nu{:}\texttt{pck} \mid \star'\right\}\checkmark^S}{H \vdash \texttt{rate}\ \langle ck \rangle\ \texttt{on}\ \texttt{C(c)} \overset{S}{\Rrightarrow} \left\{\nu{:}ck_b\ \textbf{on}\ C\left(c, \left\{\nu{:}\texttt{pck} \mid \star'\right\}\right) \mid \star\right\}}$$

$$\textsc{DeStDeclEmpty}\ \frac{H \vdash \{\nu{:}ck_b \mid \star\}\checkmark^S}{H \vdash \texttt{" "} \overset{S}{\Rrightarrow} \{\nu{:}ck_b \mid \star\}}$$

$$\textsc{DeStInstPoly}\ \frac{H \vdash ck_b\checkmark^S \qquad H \vdash \sigma[\alpha := ck_b] \overset{S}{\Rrightarrow} ck_e}{H \vdash \forall\alpha.\sigma \overset{S}{\Rrightarrow} ck_e}$$

$$\textsc{DeStInstFun}\ \frac{H \vdash ck'_r \overset{S}{\Rrightarrow} ck_r \qquad H \vdash ck'_e \overset{S}{\Rrightarrow} ck_e}{H \vdash x{:}ck'_r \to ck'_e \overset{S}{\Rrightarrow} x{:}ck_r \to ck_e}$$

$$\textsc{DeStInstTup}\ \frac{H \vdash ck_e^A \overset{S}{\Rrightarrow} ck_e^0 \qquad H \vdash ck_e^B \overset{S}{\Rrightarrow} ck_e^1}{H \vdash ck_e^A \times ck_e^B \overset{S}{\Rrightarrow} ck_e^0 \times ck_e^1}$$

$$\textsc{DeStInstRef}\ \frac{H \vdash ck'_b \overset{S}{\Rrightarrow} ck_b}{H \vdash \left\{\nu{:}ck'_b \mid \star\right\} \overset{S}{\Rrightarrow} \{\nu{:}ck_b \mid \star\}}$$

$$\textsc{DeStInstSEq-1}\ \frac{}{H \vdash \zeta(\texttt{pck}, \star) \overset{S}{\Rrightarrow} \texttt{pck}}$$

$$\textsc{DeStInstSEq-2}\ \frac{H \vdash \zeta(ck'_b, \star) \overset{S}{\Rrightarrow} ck_b \qquad H \vdash \star''\checkmark^S}{H \vdash \zeta(ck'_b\ \textbf{on}\ C\left(c, \left\{\nu{:}\texttt{pck} \mid \star'\right\}\right), \star) \overset{S}{\Rrightarrow} ck_b\ \textbf{on}\ C\left(c, \left\{\nu{:}\texttt{pck} \mid \star''\right\}\right)}$$

$$\textsc{DeStInstPck}\ \frac{}{H \vdash \texttt{pck} \overset{S}{\Rrightarrow} \texttt{pck}}$$

$$\textsc{DeStInstOn}\ \frac{H \vdash ck_b \overset{S}{\Rrightarrow} ck'_b \qquad H \vdash w \overset{S}{\Rrightarrow} w'}{H \vdash ck_b\ \textbf{on}\ C\left(c, w\right) \overset{S}{\Rrightarrow} ck'_b\ \textbf{on}\ C\left(c, w'\right)}$$

Figure 15.7: Declarative Structural Clock Instantiation

15.5.6 Node Rules

Node rules describe how to verify the consistency of clocks inside the node body as well as describe which type should be added to the environment afterwards. As stated above, in order to contain the complexity of the clock calculus, we consider nodes to have curried function types, even though our semantics do not support partial applications. The function $curry(l_{in}, l_{out})$ takes a list of input types and a list of output types and returns a curried function type accepting the input types in order and returning a tuple of the output types. Recall our notation for lists: [] is the empty list and $x :: [y]$ a list whose head is the element x and tail is a list of ys. The rules are:

- DeStImpNd: To construct the clock of an imported node, we first construct a clock for the first argument v_{in}. Then, we construct a second type that is the structural equivalent of the first type. Then, we construct a curried function type from the input and output types.
- DeStNd: To construct the clock of a user-defined node, we first instantiate the input, local and output clocks, and then we verify the well-formedness of clocks within an environment extended with the input, local and output clocks. The clock type of the node is a curried function constructed of the input and output types.

DeStImpNd

$$\frac{H;\alpha \vdash \{\nu{:}\alpha \mid \star\}\checkmark^S \qquad \{\nu{:}\alpha \mid \star\} = ck_r \qquad H;\alpha \vdash \left\{\nu{:}\zeta(\alpha,\star') \mid \star''\right\}\checkmark^S \qquad \left\{\nu{:}\zeta(\alpha,\star') \mid \star''\right\} = ck_r' \qquad currify((v_{in},ck_r)::[v_{ins},ck_r'],[v_{out},ck_r']) = ck_e}{H \vdash \texttt{imported node}\ N((v_{in}:t_{in})::[v_{ins}:t_{ins}])\ \texttt{returns}\ ([v_{out}:t_{out}])\texttt{;} \overset{S}{\Rightarrow} \forall\alpha.ck_e}$$

DeStNd

$$\frac{H \vdash [ckdecl_{in}] \overset{S}{\Rightarrow} [ck_{in}] \qquad H \vdash [ckdecl_{loc}] \overset{S}{\Rightarrow} [ck_{loc}] \qquad H \vdash [ckdecl_{out}] \overset{S}{\Rightarrow} [ck_{out}] \qquad H;[v_{in}:ck_{in}];[v_{loc}:ck_{loc}];[v_{out}:ck_{out}] \vdash eqs\checkmark^S \qquad currify\,([v_{in},ck_{in}],[v_{out},ck_{out}]) = ck_e}{H \vdash \texttt{node}\ N([v_{in}:t_{in}\ ckdecl_{in}])\ \texttt{returns}\ ([v_{out}:t_{out}\ ckdecl_{out}])\ \texttt{var}\ [v_{loc}:t_{loc}\ ckdecl_{loc}]\ \texttt{;}\ \texttt{let}\ eqs\ \texttt{tel} \overset{S}{\Rightarrow} ck_e}$$

Figure 15.8: Declarative Structural Node Rules

DeStEq

$$\frac{H \vdash x \overset{S}{\Rightarrow} \sigma \qquad H \vdash e \overset{S}{\Rightarrow} \sigma}{H \vdash x = e\checkmark^S}$$

DeStEqs

$$\frac{H \vdash eq_1\checkmark^S \qquad H \vdash eq_2\checkmark^S}{H \vdash eq_1;eq_2\checkmark^S}$$

Figure 15.9: Declarative Structural Equation Rules

15.5.7 Equation Rules

Equation rules are well-formedness rules which verify that equations are valid within the environment of a node. The rules are:

- DeStEq: A single equation is well-formed, if the left-hand side and the right-hand side both synthesize the same type.
- DeStEqs: Multiple equations are well-formed, if the individual equations are well-formed.

15.5.8 Expression Rules

Expression rules are fairly straightforward rules as can be found in other refinement typing languages. The rules are:

- DeStChk: To check an expression e against a clock σ, we must first synthesize a clock σ' for e and then verify that σ' is a subtype of σ;
- DeStVar: To synthesize a clock ck_e for a variable x, we first fetch the clock σ bound to this variable inside the environment and then instantiate it into clock ck_e;
- DeStCst: A constant can be lifted to a dataflow of any clock;
- DeStAppl: When applying an expression to a node or built-in operator (a "function-like"), we first synthesize a clock for our function-like and then check its argument against the input clock and return a subtype of the the output type, i.e. the application has the same type upto refinements.

$$\textsc{DeStChk}\ \frac{H \vdash e \overset{S}{\Rightarrow} \sigma' \qquad H \vdash \sigma' \overset{S}{<:} \sigma}{H \vdash e \overset{S}{\Leftarrow} \sigma} \qquad \textsc{DeStVar}\ \frac{(x : \sigma) \overset{S}{\in} H \qquad H \vdash \sigma \overset{S}{\Rightarrow\!\!\!\!\!\!\Rightarrow} ck_e}{H \vdash x \overset{S}{\Rightarrow} ck_e} \qquad \textsc{DeStCst}\ \frac{}{H \vdash c \overset{S}{\Rightarrow} \sigma}$$

$$\textsc{DeStAppl}\ \frac{H \vdash N \overset{S}{\Rightarrow} x{:}ck_r \to ck_e \qquad H \vdash a \overset{S}{\Leftarrow} ck_r \qquad H \vdash ck'_e \overset{S}{<:} ck_e}{H \vdash N(a) \overset{S}{\Rightarrow} ck'_e}$$

Figure 15.10: Declarative Structural Expression Rules

Example 22 (Structural Expression Rules)**.** The inference below shows the judgments performed by the structural clock calculus when synthesizing a clock for `j /^2` (Line 7). First (Equation (15.9), rule DeStAppl states that we must instantiate a clock for this instance of `/^2` (see Example 21). Then, we must check `j` against the input clock, $\{\nu{:}\texttt{pck} \mid \star_{di}\}$. Finally, the application has the output clock of the function type with new refinements. To check `j` against the input type, we must apply rule DeStChk (Equation (15.10)). For this, we simply fetch the clock type of `j` from the environment and verify that it is a subtype of the input clock. To verify the subtype check (Equation (15.11)), we apply rule DeStSubRef which states that the subtype base clock must be a subtype of the supertype base clock. Recall, that refinement holes are ignored in this step of the clock calculus. As `pck` is a subtype of itself, the subtyping check succeeds and we can synthesize clock $\{\nu{:}\texttt{pck} \mid \star_x\}$ for our expression.

$$\frac{H \vdash \texttt{/\^2} \overset{S}{\Rightarrow} e{:}\{\nu{:}\texttt{pck} \mid \star_{di}\} \to \{\nu{:}\texttt{pck} \mid \star_{do}\} \qquad H \vdash \texttt{j} \overset{S}{\Leftarrow} \{\nu{:}\texttt{pck} \mid \star_{di}\} \qquad H \vdash \{\nu{:}\texttt{pck} \mid \star_x\} \overset{S}{<:} \{\nu{:}\texttt{pck} \mid \star_{do}\}}{H \vdash \texttt{j /\^2} \overset{S}{\Rightarrow} \{\nu{:}\texttt{pck} \mid \star_x\}} \tag{15.9}$$

$$\frac{H \vdash \texttt{j} \overset{S}{\Rightarrow} \{\nu{:}\texttt{pck} \mid \star_j\} \qquad H \vdash \{\nu{:}\texttt{pck} \mid \star_j\} \overset{S}{<:} \{\nu{:}\texttt{pck} \mid \star_{di}\}}{H \vdash \texttt{j} \overset{S}{\Leftarrow} \{\nu{:}\texttt{pck} \mid \star_{di}\}} \tag{15.10}$$

$$\frac{H \vdash \texttt{pck} \overset{S}{<:} \texttt{pck}}{H \vdash \{\nu{:}\texttt{pck} \mid \star_j\} \overset{S}{<:} \{\nu{:}\texttt{pck} \mid \star_{di}\}} \tag{15.11}$$

Taking something slightly more complicated, let's look at the judgments produced by the clock calculus for `a /^3`. The clock calculus starts as in the above example with rule DeStAppl (Equation (15.12)). This time, the substitute for α features a **on** operator. Next, we apply rule DeStChk (Equation (15.13)) to check `a` against the input type. This means that we must synthesize a clock type for `a` and then verify that it is a subtype of the input type. To verify this subtype check, we apply rule DeStSubRef (Equation (15.14)) which tells us to verify the subtype relation between the base types. As the base types feature **on** operators, we apply rule DeStSubOn (Equation (15.15)). It requires to verify the subtype relation of base types (which succeeds as both are `pck`) and the subtype relation of views (which succeeds similarly as in Equation (15.11)). With this, the subtype check succeeds and we can synthesize clock type $\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{w0}\}) \mid \star_b\}$ for our expression.

$$\frac{H \vdash \texttt{/\^{}3} \overset{S}{\Rightarrow} e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{fwi}\}) \mid \star_{fi}\} \to \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{fwo}\}) \mid \star_{fo}\} \quad H \vdash \texttt{a}{\Leftarrow}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{fwi}\}) \mid \star_{fi}\} \quad H \vdash \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{w0}\}) \mid \star_{b}\} \overset{S}{<:} \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{fwo}\}) \mid \star_{fo}\}}{H \vdash \texttt{a}\ \texttt{/\^{}3} \overset{S}{\Rightarrow} \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{w0}\}) \mid \star_{b}\}} \tag{15.12}$$

$$\frac{H \vdash \texttt{a} \overset{S}{\Rightarrow} \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{wa}\}) \mid \star_{a}\} \quad H \vdash \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{wa}\}) \mid \star_{a}\} \overset{S}{<:} \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{fwi}\}) \mid \star_{fi}\}}{H \vdash \texttt{a}{\Leftarrow}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{fwi}\}) \mid \star_{fi}\}} \tag{15.13}$$

$$\frac{H \vdash \texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{wa}\}) \overset{S}{<:} \texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{fwi}\})}{H \vdash \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{wa}\}) \mid \star_{a}\} \overset{S}{<:} \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{fwi}\}) \mid \star_{fi}\}} \tag{15.14}$$

$$\frac{H \vdash \texttt{pck} \overset{S}{<:} \texttt{pck} \qquad H \vdash \{\nu{:}\texttt{pck} \mid \star_{wa}\} \overset{S}{<:} \{\nu{:}\texttt{pck} \mid \star_{fwi}\}}{H \vdash \texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{wa}\}) \overset{S}{<:} \texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_{fwi}\})} \tag{15.15}$$

15.6 Refinement Clock Calculus

The goal of this pass is to infer the refinements of clock types in place of the refinement holes. This means in particular that other parts of clock types remain fixed.

The judgments of the refinement clock calculus are the following.

- Synthesis $H \vdash x \Rightarrow \sigma$: Under environment H, a type σ could be constructed for x;
- Checking $H \vdash x{\Leftarrow}\sigma$: Under environment H, type σ is valid for x
- Subtyping $H \vdash \sigma <: \sigma'$: Under environment H, type σ is a subtype of σ';
- Well-formedness $H \vdash x\checkmark^S$: Under environment H, x is well-formed;
- Instantiation $H \vdash \sigma^S{\twoheadrightarrow}\,\sigma{\twoheadrightarrow}\,\sigma'$: Under environment H and using the structural type σ^S as a guide, type σ can be instantiated into type σ';
- Membership $\sigma \in H$: Type σ is part of environment H.

We assume that the results of the previous pass can be accessed freely, as if part of the environment. In practice, they are stored as field of the datatypes produced by the structural clock calculus.

15.6.1 Clock Language

Figure 15.11 illustrates the clock system of our clock calculus. A clock type (σ) is either an polymorphic clock ($\forall\alpha.\sigma$) or a clock expression. A clock expression (ck_e) is either a dependent clock function ($x{:}ck_r \to ck_e$), a tuple ($ck_e \times ck_e$) or a refined clock. A refined clock (ck_r) is a base clock refined by a boolean refinement ($\{\nu{:}ck_b \mid r\}$). A base clock (ck_b) is either a periodic clock (`pck`), a clock variable (α), a conditioned base clock (ck_b **on** $C\,(c, ck_r)$), or the structural equivalent of a base clock where the refinements of views in ck_b are substituted by refinement r ($\zeta(ck_b, r)$).

A refinement (r) is either the conjunction or disjunction of two refinements ($r \wedge r$, $r \vee r$), the equality or inequality between two arithmetic expressions ($d = d$, $d \geqslant d$, $d \leqslant d$), a divisibility constraint (d **div** d), a divisibility constraint within a constant ($d \overset{k}{\textbf{div}}\, d$) or the boolean *true*.

An arithmetic expression (d) is either a constant (k), a clock property, the sum, difference, product, or quotient of arithmetic expressions ($d + d$, $d - d$, $d * d$, d/k), or the least-common multiple of two arithmetic expressions.

A clock property (p) references either the period (π) or the offset (φ) of a clock. The forms $\pi(\nu)$ and $\varphi(\nu)$ reference the clock of the current refinement, i.e. the one whose ν most closely bound. For instance, in a clock $\{\nu{:}\texttt{pck}$ **on** $C\,(c, \{\nu{:}\texttt{pck} \mid \pi(\nu) = 20 \wedge \pi(\nu) = 10\}) \mid \pi(\nu) = 10 \wedge \varphi(\nu) = 0\}$ is the clock type of the clock $(10, 0)$ **on** $C(c, (20, 10))$. The forms with subscript reference a clock at a certain *depth*, either bound in the environment ($\pi_n(x)$, $\varphi_n(x)$), or within the hierarchy of a refined clock ($\pi_n(\nu)$, $\varphi_n(\nu)$). If the depth-indicator is a constant k, it references a clock as defined in Section 15.6.2. The special depth-indicator ξ is used in conjunction with the constructor ζ (see Section 15.6.6). When instantiating a $\zeta(ck_b, r)$ clock type, occurrences of ξ inside r are substituted by the current depth of ck_b.

An environment (H) is either empty or an environment extended by a variable binding ($H; x{:}\sigma$), or a type ($H; \alpha$).

Definition 32 (Divisible within k)**.** The relation $\overset{k}{\textbf{div}}$ called "divisible within k" is a special case of the division relation **div**. While the relation a **div** b is a constraint stating $\exists x \in \mathbb{N}.a * x = b$, the relation $a \overset{k}{\textbf{div}}\, b$ adds a constraint x **div** $k \wedge x > 1$, i.e. x must be a product of prime factors of k. This constraint has the advantage that it is linear, and thus decidable in more cases. The relation a **div** b is linear iff a is an explicit constant. On the other hand, the relation $a \overset{k}{\textbf{div}}\, b$ is linear iff k is an explicit constant, a and b can be both variables. The translation of $\overset{k}{\textbf{div}}$ into a linear constraint is shown in Section 15.6.2.

The table below shows under which conditions the different forms of **div** and **div**k are linear. In these examples, a, b and c are all variables.

$$
\begin{array}{lcl}
\sigma & ::= & \forall\alpha.\sigma \mid ck_e \\
ck_e & ::= & x{:}ck_r \to ck_e \mid ck_e \times ck_e \mid ck_r \\
ck_r & ::= & \{\nu{:}ck_b \mid r\} \\
ck_b & ::= & \mathtt{pck} \mid \alpha \mid ck_b \ \mathbf{on}\ C\,(c, ck_r) \mid \zeta(ck_b, r) \\
r & ::= & r \wedge r \mid r \vee r \mid d = d \mid r \geqslant d \mid r \leqslant d \mid d\ \mathbf{div}\ d \mid d \overset{k}{\mathbf{div}}\ d \mid true \\
d & ::= & k \mid p \mid d + d \mid d - d \mid d * k \mid d/k \mid lcm(d, d) \\
p & ::= & \pi(\nu) \mid \varphi(\nu) \mid \pi_n(\nu) \mid \varphi_n(\nu) \mid \pi_n(x) \mid \varphi_n(x) \\
n & ::= & k \mid \xi \\
H & ::= & \varnothing \mid H; x{:}\sigma \mid H; \alpha
\end{array}
$$

x : Variable $\quad k$: Constant

Figure 15.11: Declarative Refinement Clock System

Constraint	Linear	Translated form
$2\ \mathbf{div}\ 4$	Yes	$\exists x \in \mathbb{N}.2 * x = 4$
$2\ \mathbf{div}\ b$	Yes	$\exists x \in \mathbb{N}.2 * x = b$
$a\ \mathbf{div}\ 4$	No	$\exists x \in \mathbb{N}.a * x = 4$
$a\ \mathbf{div}\ b$	No	$\exists x \in \mathbb{N}.a * x = b$
$2 \overset{4}{\mathbf{div}} 8$	Yes	$\exists x \in \mathbb{N}.2 * x = 8 \wedge x\ \mathbf{div}\ 4 \wedge x > 1$
$2 \overset{4}{\mathbf{div}} c$	Yes	$\exists x \in \mathbb{N}.2 * x = c \wedge x\ \mathbf{div}\ 4 \wedge x > 1$
$2 \overset{b}{\mathbf{div}} 8$	No	$\exists x \in \mathbb{N}.2 * x = 8 \wedge x\ \mathbf{div}\ b \wedge x > 1$
$2 \overset{b}{\mathbf{div}} c$	No	$\exists x \in \mathbb{N}.2 * x = c \wedge x\ \mathbf{div}\ b \wedge x > 1$
$a \overset{4}{\mathbf{div}} 8$	Yes	$\exists x \in \mathbb{N}.a * x = 8 \wedge x\ \mathbf{div}\ 4 \wedge x > 1$
$a \overset{4}{\mathbf{div}} c$	Yes	$\exists x \in \mathbb{N}.a * x = c \wedge x\ \mathbf{div}\ 4 \wedge x > 1$
$a \overset{b}{\mathbf{div}} c$	No	$\exists x \in \mathbb{N}.a * x = c \wedge x\ \mathbf{div}\ b \wedge x > 1$

To retain brevity, we define a shorter notation for common types of refined clocks.

Definition 33 (Shorthands for Common Clocks)**.**

$$
\begin{aligned}
\langle ck_b \mid d, d' \rangle &= \{\nu{:}ck_b \mid \pi(\nu) = d \wedge \varphi(\nu) = d'\} \\
\langle ck_b \mid \_, d' \rangle &= \{\nu{:}ck_b \mid \varphi(\nu) = d'\} \\
\langle ck_b \mid d, d' \mid r \rangle &= \{\nu{:}ck_b \mid \pi(\nu) = d \wedge \varphi(\nu) = d' \wedge r\} \\
\zeta\,\langle ck_b \mid d, d' \rangle &= \zeta(ck_b, \pi(\nu) = d \wedge \varphi(\nu) = d')
\end{aligned}
$$

15.6.2 Functions on Refinement Clocks

We define two functions on refinement clocks, the *depth* function and the lowering function $\lfloor\ \rfloor$.

The *depth* function determines the number of **on** operators applied to a ck_b and this allows to determine which clock is referenced by clock properties of the form $\pi_k(e)$.

The lowering function $\lfloor\ \rfloor$ lowers elements of the clock language into the language of the SMT solver. Lowering an environment is the conjunction of the lowered variable bindings. Lowering a variable is the conjunction of the lowered refinements of the ck_r and the lowered ck_b. When lowering the refinement of the ck_r bound to x, substitute occurrences of ν by x and occurrences of $\pi(\nu)$ and $\varphi(\nu)$ by $\pi_0(x)$ and $\varphi_0(x)$ respectively. When lowering the refinements of a view inside a ck_b, substitute occurrences of ν by x and occurrences of $\pi(\nu)$ and $\varphi(\nu)$ by $\pi_k(x)$ and $\varphi_k(x)$ respectively, where k is the depth of the ck_b. The lowering of `pck` is simply $true$.

The lowering of refinements is as follows:

- $r \wedge r'$: The lowering of the conjunction of refinements is the conjunction of the lowered refinements;
- $r \vee r'$: The lowering of the disjunction of refinements is the disjunction of the lowered refinements;
- $d = d'$, $d \geqslant d'$, $d \leqslant d'$: The lowering of an (in)equality between arithmetic expressions is the (in)equality between the lowered arithmetic expressions;
- $true$: The lowering of $true$ is itself;
- d **div** d': The lowering of a divisibility constraint, is the constraint that the first lowered expression modulo the second lowered expression equals 0;
- $d \overset{k}{\textbf{div}} d'$: The lowering of a divisibility constraint is the disjunction of constraints stating that the first lowered expression times a possible multiplier is equal to the second lowered constraint;
- $\pi_k(x)$, $\varphi_k(x)$: The lowering of a clock property is simply the variable introduced to the SMT solver to represent it;
- $d + d$, $d - d$, $d * d$, d/d: The lowering of a binary operation, is the same binary operation applied to the lowered expressions;
- $lcm(d, k)$: The lowering of the least-common multiple between an expression and a constant is a fresh variable to which the constraint lcm_{SMT} forces it to be the least-common multiple of the lowered expression and the constant;
- k: The lowering of a constant is the constant.

Note that when lowering a $\overset{k}{\textbf{div}}$ the disjunction $\bigvee_{k' \in \{x | x \textbf{ div } k, x>1\}}$ can be enumerated explicitly. For instance, $\lfloor a \overset{6}{\textbf{div}} b \rfloor$ is lowered into $(a * 2 = b) \vee (a * 3 = b) \vee (a * 6 = b)$ which is linear and decidable, even if a and b are both variables.

Note also that when lowering a lcm expression, we can do the same. The operator `ite` is the "if-then-else" built-in of the SMT solver and has type $\forall\alpha.bool \to \alpha \to \alpha \to \alpha$. Thus, $\lfloor lcm(a, 6) \rfloor$ is lowered into

$$
\begin{array}{l}
\texttt{ite}\ (6\ \textbf{div}\ a)(\texttt{x} = a) \\
\quad (\texttt{ite}\ (6\ \textbf{div}\ (a * 2))(\texttt{x} = a * 2) \\
\quad\quad (\texttt{ite}\ (6\ \textbf{div}\ (a * 3))(\texttt{x} = a * 3)(\texttt{x} = a * 6)))
\end{array}
$$

$$
\begin{aligned}
depth(\texttt{pck}) &= 0 \\
depth(ck_b \textbf{ on } C\,(c, w)) &= depth(ck_b) + 1
\end{aligned}
$$

$$
\begin{aligned}
\lfloor \varnothing \rfloor &= true \\
H; x{:}\sigma &= \lfloor H \rfloor \wedge \lfloor \sigma \rfloor^x \\
H; \alpha &= \lfloor H \rfloor
\end{aligned}
$$

$$
\begin{aligned}
\lfloor \{\nu{:}ck_b \mid r\} \rfloor^x &= \lfloor ck_b \rfloor^x \wedge \lfloor r[\nu := x, \pi(\nu) := \pi_0(x), \varphi(\nu) := \varphi_0(x)] \rfloor \\
\lfloor ck_b \textbf{ on } C\,(c, \{\nu{:}\texttt{pck} \mid r\}) \rfloor^x &= \lfloor ck_b \rfloor^x \wedge \lfloor r[\nu := x, \pi(\nu) := \pi_k(x), \varphi(\nu) := \varphi_k(x)] \rfloor \\
&\qquad \text{where } k = depth(ck_b) + 1 \\
\lfloor \texttt{pck} \rfloor^x &= true
\end{aligned}
$$

$$
\begin{aligned}
\lfloor r \wedge r' \rfloor &= \lfloor r \rfloor \wedge \lfloor r' \rfloor & \qquad \lfloor r \vee r' \rfloor &= \lfloor r \rfloor \vee \lfloor r' \rfloor \\
\lfloor d = d' \rfloor &= (\lfloor d \rfloor = \lfloor d' \rfloor) & \qquad \lfloor true \rfloor &= true \\
\lfloor d \geqslant d' \rfloor &= \lfloor d \rfloor \geqslant \lfloor d' \rfloor & \qquad \lfloor d \leqslant d' \rfloor &= \lfloor d \rfloor \leqslant \lfloor d' \rfloor \\
\lfloor d \textbf{ div } d' \rfloor &= (\lfloor d' \rfloor \bmod \lfloor d \rfloor = 0) & \qquad \lfloor d \overset{k}{\textbf{ div }} d' \rfloor &= \bigvee_{k' \in \{x \mid x \textbf{ div } k, x > 1\}} (\lfloor d \rfloor * k' = \lfloor d' \rfloor) \\
\lfloor \pi_k(x) \rfloor &= \texttt{x-k-period} & \qquad \lfloor \varphi_k(x) \rfloor &= \texttt{x-k-offset} \\
\lfloor d + d' \rfloor &= \lfloor d \rfloor + \lfloor d' \rfloor & \qquad \lfloor d - d' \rfloor &= \lfloor d \rfloor - \lfloor d' \rfloor \\
\lfloor d * d' \rfloor &= \lfloor d \rfloor * \lfloor d' \rfloor & \qquad \lfloor d / d' \rfloor &= \lfloor d \rfloor / \lfloor d' \rfloor \\
\lfloor lcm(d, k) \rfloor &= \texttt{\$lcm-var} & \qquad &, lcm_{SMT}(\texttt{\$lcm-var}, \lfloor d \rfloor, k) \\
\lfloor k \rfloor &= k
\end{aligned}
$$

$$
\begin{aligned}
&lcm_{SMT}(\texttt{x}, v, k) = \\
&(\texttt{ite } (k \textbf{ div } v)(\texttt{x} = v) \\
&\qquad (\texttt{ite } (k \textbf{ div } (v * k_0))(\texttt{x} = (v * k_0)) \\
&\qquad \ldots \\
&\qquad (\texttt{ite } (k \textbf{ div } (v * k_n))(\texttt{x} = (v * k_n))(\texttt{x} = (v * k)))) \\
&k_i \in \{x \mid x \textbf{ div } k, x > 1\}
\end{aligned}
$$

Figure 15.12: Functions on Refinement Clocks

15.6.3 Initial Environment

The initial environment of the refinement clock calculus is shown in Figure 15.13. It contains the built-in operators. Note that the clock types of the built-in operators rely heavily on Property 1. Recall that this property allows us to simplify clocks featuring both periodic operators and conditional operators. The Property pushes the periodic operator deeper into the clock hierarchy, potentially changing the views of the **on** operators it encounters, until it reaches a periodic clock where it can be applied according to Definition 15. Without this Property, we would have to layer refinement clock types on top of each other to accurately describe the clocks of the program. An expression `x /^2 ~>3` would have to be of the form $\{\nu{:}\{\nu{:}type\ of\ \texttt{x} \mid refinements\ of\ \texttt{/\^2}\} \mid \text{refinements of } \texttt{\~{}>3}\}$ which not only increases the complexity of the type system (refinement types may be arbitrarily nested), but also greatly decreases the readability of clock types.

Let us go into more detail over the different operators. They all have polymorphic clocks:

- `e when C(c)`: The function takes an argument e without restrictions and an argument c whose offset must be equal to the offset of e and whose views must have the same period and offset than the views of e. The output has the same period and offset than e and an additional **on** applied to its base clock. The view on this last-level **on** has an offset equal to the offset of e and a period that is divisible by the periods of e and c.

- `merge(c, C0->e0, C1->e1)`: The function takes 3 arguments. The first argument c has no restrictions. For the second argument $e0$, the refinement also state that there are no restrictions. However, the base clock of this argument features two changes. First, the ζ operator states that whatever has been instantiated for α, we replace view refinements by $true$, i.e. there are no requirements on the views of $e0$. Second, we apply an **on** operator afterwards where the constructor is `C0` and condition `c`. This, simply enforces that the arguments we merge together are indeed conditionally sub-sampled on the first argument c. The second argument $e1$ must have the same period and offset than $e1$, i.e. we can only merge dataflows with the same rate. Its base clock features a ζ operator. Here, it enforces that the views of $e1$ have the same period and offset than the same-level views of $e0$, i.e. we can only merge dataflows which perceive the condition `c` in the same way. The output has the last-level **on** removed, but apart from that the same clock as $e0$.

- `e*^k`: The function takes an argument e whose period is divisible by k. The output has the period of e divided by k, but the offset remains unchanged. The use of the ζ operator states that views remain unchanged, as specified by Property 1.

- `e/^k`: The function takes an argument e without restrictions on its period or offset. However, the ζ operator introduces requirements on the views of e. These requirements are the preconditions defined in Property 1 encoded in a refinement. Thus, we require that each view must have a period that either is divisible by the output period or that divides (within k) the output period. The output has the period of e multiplied by k and an unchanged offset. The ζ operator in the output clock similarly, encodes in a refinement the right-hand side of Property 1. Offsets remain unchanged.

- `C fby e`: The function takes an argument e without restrictions. Note that even though the operator is syntactically binary, as one operand is a constant, we do not consider it during the clock calculus. The output has the same period and offsets and views are also unchanged, since the `fby` operator only delays values, but not the tags when the dataflow is present.

- `e~>k`: The function similarly takes an argument e without restrictions. The output has an unchanged period, but an offset increased by k. The ζ operator states that the views of the output have the same period than the same-level views of e, but an offset that is delayed by k compared to the input e. This is due to the fact that `~>` delays the input by $\Rightarrow.k$ and thus Property 1 requires us to reflect this delay both in the offset and the views.
- `C::e`: The function takes an argument e whose offset must be greater or equal than it's period, i.e. we can't insert a value before the initial date 0. In addition, the ζ operator requires that each view of e must be lower by at least a period compared to the offset of the dataflow, i.e. we can't insert a value before the first time the condition is present. The output has an unchanged period, but an offset lower by one period. The ζ of the output type states that views also remain unchanged. This is because we inserted a value inside the dataflow, but did not change how the condition is observed.
- `tail e`: The function takes an argument e without restrictions. The output has an unchanged period, but an offset delayed by one period. The ζ operator inside the output type states that the periods and offsets of views are unchanged wrt. the input e. Note the difference between this operator and `~>`. Because this operator simply drops a value but does not change how the remaining values of the dataflow perceive the condition, views are unchanged. However, because `~>` shifts all values of the dataflow, this changes how the condition is perceived.
- `e rate (n,p)`: The function takes an argument e which matches exactly the clock (n, p) and returns a value with the same clock.

15.6.4 Well-formedness Rules

The well-formedness rules of the refinement clock calculus in Figure 15.14 are fairly similar to those of the structural clock calculus:

- DeReAbsWF: A polymorphic clock is well-formed, if the polymorphic clock is well-formed within the environment extended by the polymorphic variable;
- DeReFunWF: A clock function is well-formed, if the input clock is well-formed and the output clock is well-formed in the environment extended by the input clock, i.e. functional clocks are dependent types;
- DeReTupWF: A clock tuple is well-formed, if the individual clocks are well-formed;
- DeReRefWF: A refined clock is well-formed, if both the base clock and refinement are well-formed;
- DeReRfPWF: A refinement is well-formed, if it has type `bool` within the environment (and respects the grammar defined in Section 15.6.1);
- DeReSEqWF: A structuraly equivalent clock is well-formed, if its constituents are well-formed;
- DeRePckWF: A `pck` is always well-formed;
- DeReVarWF: A variable is well-formed, if it is part of the environment;
- DeReOnWF: A **on** is well-formed, if it's constituents are well-formed.

`e when C(c)` :

$$\forall\alpha.e{:}\{\nu{:}\alpha \mid true\} \to c{:}\,\langle\zeta\,\langle\alpha \mid \pi_\xi(e),\varphi_\xi(e)\rangle \mid \_,\varphi_0(e)\rangle \to$$
$$\langle\zeta\,\langle\alpha \mid \pi_\xi(e),\varphi_\xi(e)\rangle \ \mathbf{on}\ C\,(c,\langle \texttt{pck} \mid \_,\varphi_0(e) \mid \pi_0(e)\ \mathbf{div}\ \pi(\nu) \wedge \pi_0(c)\ \mathbf{div}\ \pi(\nu)\rangle) \mid \pi_0(e),\varphi_0(e)\rangle$$

`merge(c, C0->e0, C1->e1)` :

$$\forall\alpha.c{:}\{\nu{:}\alpha \mid true\} \to e0{:}\{\nu{:}\zeta(\alpha,true)\ \mathbf{on}\ C0\,(c,\{\nu{:}\texttt{pck} \mid true\}) \mid true\} \to$$
$$e1{:}\,\langle\zeta\,\langle\alpha\ \mathbf{on}\ C1\,(c,\{\nu{:}\texttt{pck} \mid true\}) \mid \pi_\xi(e0),\varphi_\xi(e0)\rangle \mid \pi_0(e0),\varphi_0(e0)\rangle \to$$
$$\langle\zeta\,\langle\alpha \mid \pi_\xi(e0),\varphi_\xi(e0)\rangle \mid \pi_0(e0),\varphi_0(e0)\rangle$$

`e*^k` :

$$\forall\alpha.e{:}\{\nu{:}\alpha \mid k\ \mathbf{div}\ \pi(\nu)\} \to \langle\zeta\,\langle\alpha \mid \pi_\xi(e),\varphi_\xi(e)\rangle \mid \pi_0(e)/k,\varphi_0(e)\rangle$$

`e/^k` :

$$\forall\alpha.e{:}\left\{\nu{:}\zeta(\alpha,(\pi_0(\nu)*k)\ \mathbf{div}\ \pi(\nu) \vee \pi(\nu) \overset{k}{\mathbf{div}}\ \pi_0(\nu)*k) \mid true\right\} \to$$
$$\left\langle\zeta\left\langle\alpha \mid \pi_0(e)*lcm\left(\frac{\pi_\xi(e)}{\pi_0(e)},k\right),\varphi_\xi(e)\right\rangle \mid \pi_0(e)*k,\varphi_0(e)\right\rangle$$

`C fby e`

$$\forall\alpha.e{:}\{\nu{:}\alpha \mid true\} \to \langle\zeta\,\langle\alpha \mid \pi_\xi(e),\varphi_\xi(e)\rangle \mid \pi_0(e),\varphi_0(e)\rangle$$

`e~>k` :

$$\forall\alpha.e{:}\{\nu{:}\alpha \mid true\} \to \langle\zeta\,\langle\alpha \mid \pi_\xi(e),\varphi_\xi(e)+k\rangle \mid \pi_0(e),\varphi_0(e)+k\rangle$$

`C::e` :

$$\forall\alpha.e{:}\{\nu{:}\zeta(\alpha,\varphi(\nu) \leqslant \varphi_0(\nu)-\pi_0(\nu)) \mid \varphi(\nu) \geqslant \pi(\nu)\} \to \langle\zeta\,\langle\alpha \mid \pi_\xi(e),\varphi_\xi(e)\rangle \mid \pi_0(e),\varphi_0(e)-\pi(\nu)\rangle$$

`tail e` :

$$\forall\alpha.e{:}\{\nu{:}\alpha \mid true\} \to \langle\zeta\,\langle\alpha \mid \pi_\xi(e),\varphi_\xi(e)\rangle \mid \pi_0(e),\varphi_0(\nu)+\pi(\nu)\rangle$$

`e rate (n,p)` :

$$e{:}\,\langle\texttt{pck} \mid n,p\rangle \to \langle\texttt{pck} \mid n,p\rangle$$

Figure 15.13: Initial Declarative Refinement Clock Environment

$$\textsc{DeReAbsWF}\ \frac{H, \alpha \vdash \sigma\checkmark}{H \vdash \forall\alpha.\sigma\checkmark} \qquad \textsc{DeReFunWF}\ \frac{H \vdash ck_r\checkmark \qquad H, x : ck_r \vdash ck_e\checkmark}{H \vdash x{:}ck_r \to ck_e\checkmark} \qquad \textsc{DeReTupWF}\ \frac{H \vdash ck_e\checkmark \qquad H \vdash ck'_e\checkmark}{H \vdash ck_e \times ck'_e\checkmark}$$

$$\textsc{DeReRefWF}\ \frac{H \vdash ck_b\checkmark \qquad H \vdash r\checkmark}{H \vdash \{\nu{:}ck_b \mid r\}\checkmark} \qquad \textsc{DeReRfPWF}\ \frac{H \vdash r : \texttt{bool}}{H \vdash r\checkmark} \qquad \textsc{DeReSEqWF}\ \frac{H \vdash ck_b\checkmark \qquad H \vdash r\checkmark}{H \vdash \zeta(ck_b, r)\checkmark}$$

$$\textsc{DeRePckWF}\ \frac{}{H \vdash \texttt{pck}\checkmark} \qquad \textsc{DeReVarWF}\ \frac{\alpha \in H}{H \vdash \alpha\checkmark} \qquad \textsc{DeReOnWF}\ \frac{H \vdash ck_b\checkmark \qquad H \vdash r\checkmark}{H \vdash ck_b\ \textbf{on}\ C\,(c, \{\nu{:}\texttt{pck} \mid r\})\,\checkmark}$$

Figure 15.14: Declarative Refinement Clock Well-Formedness

15.6.5 Subtyping Rules

The subtyping rules in Figure 15.15 introduce in particular the DeReVerif rule which handles the satisfiability of the refinements:

- DeReSubPck: A `pck` is a subtype of itself;
- DeReSubVar: A clock variable is a subtype of itself;
- DeReSubOn: A conditional clock is a subtype of another, if their constructor and condition are identical, and the first conditioned clocks is a subtype of the second, and the first view is a subtype of the second;
- DeReSubRef: A refined clock is a subtype of another, if the first base clock is a subtype of the second and the first refinement implies the second;
- DeReVerif: A refinement implication is verified, if the SMT solver can prove that the lowered implication of the refinements;
- DeReSubTup: A tuple clock is a subtype of another, if the constituents of the subtype are subtypes of the respective constituent of the supertype.

Note that for decidability reasons, our implementation does not submit $\lfloor H \rfloor \wedge \forall\nu.\lfloor r \rfloor \to \lfloor r' \rfloor$ to the SMT solver, but instead $\lfloor H \rfloor \wedge \lfloor r \rfloor \wedge \neg\lfloor r' \rfloor$. This form allows us to verify the same property, if we change how results have to be interpreted. Instead of verifying that "Within environment H, for all values of ν, if it respects r, it also respects r'", we verify "Within environment H, is there a value of ν, such that it respects r, but not r' ?". Thus, the subtyping relation holds, iff the query returns UNSAT. When it returns SAT, the assignment to variables shows us a possible assignment of periods and offsets where the subtyping relation does not hold.

15.6.6 Instantiation Rules

A specificity of the instantiation rules in Figure 15.16 is that the instantiated type must preserve the same structure as the result of the previous pass. Thus, the structural clock type becomes an input

$$\textsc{DeReSubPck}\quad H \vdash \texttt{pck} <: \texttt{pck}$$

$$\textsc{DeReSubVar}\quad H \vdash \alpha <: \alpha$$

$$\textsc{DeReSubOn}\quad \frac{H \vdash ck_b <: ck'_b \qquad H \vdash w <: w'}{H \vdash ck_b \textbf{ on } C\,(c, w) <: ck'_b \textbf{ on } C\,\big(c, w'\big)}$$

$$\textsc{DeReSubRef}\quad \frac{H \vdash ck_b <: ck'_b \qquad H \vdash r \to r'}{H \vdash \{\nu{:}ck_b \mid r\} <: \big\{\nu{:}ck'_b \mid r'\big\}}$$

$$\textsc{DeReVerif}\quad \frac{SAT(\lfloor H \rfloor \wedge \forall \nu \in .\lfloor r \rfloor \to \lfloor r' \rfloor)}{H \vdash r \to r'}$$

$$\textsc{DeReSubTup}\quad \frac{H \vdash ck^0_r <: ck^A_r \qquad H \vdash ck^1_r <: ck^B_r}{H \vdash ck^0_r \times ck^1_r <: ck^A_r \times ck^B_r}$$

Figure 15.15: Declarative Refinement Subtyping Rules

of the instantiation statement. Note that rule DeReInstSEq-2 is the rule responsible for replacing the ξ in π_ξ and $\varphi_\xi(b)$ y a concrete k. The rules are:

- DeReDeclPck: A strictly periodic annotation can be instantiated by simply replacing the refinement hole by the appropriate refinement;
- DeReDeclOn: A conditional clock annotation can be instantiated by instantiating the clock without the highest-level **on** and instantiating the clock view from an empty annotation, and then, re-applying the **on** operator;
- DeReDeclEmptyPck: An empty annotation of a refined `pck` can be instantiating by replacing the the refinement hole by a well-formed refinement;
- DeReDeclEmptyOn: An empty annotation of a refined conditional clock, can be instantiated by performing the instantiation recursively as in DeReDeclOn;
- DeReInstPoly: To instantiate a polymorphic clock, substitute the polymorphic variable by a base clock and instantiate this clock recursively;
- DeReInstFun: To instantiate a function, instantiate its components;
- DeReInstTup: To instantiate a tuple, instantiate its components;
- DeReInstRef: To instantiate a refined clock, instantiate the base clock and replace the refinement hole by the provided refinement;
- DeReInstSEq-1: An instantiated structural equivalent of a `pck` is a `pck`
- DeReInstSEq-2: To instantiate the structural equivalent of a conditional clock, first instantiate the structural equivalent of the conditioned clock and then reapply the **on** where the view refinement has been replaced by the provided refinement where ξ has been replaced by the depth of the clock;
- DeReInstOn: To instantiate an **on**, instantiate the conditioned clock and replace the view refinement hole by the provided refinement;

- DeReInstPck: A pck instantiates to itself.

Example 23 (Refinement Instantiation Rules)**.** We will look at the instantiation of the same operators as in Example 21.

To instantiate the /^2 operator at Line 7, the following judgments are produced by the clock calculus. First (Equation (15.19)), the rule DeReInstPoly states that to instantiate a polymorphic type, using the type produced during the structural clock calculus as a guide, we must provide a well-formed base clock, here pck, to replace the type variable α and then, we must recursively instantiate this clock type after substituting α by the provided base clock.

After this, the recursive call requires us to apply rule DeReInstFun (Equation (15.18)). This rule simply requires us to instantiate the input and output clock types. Note that when instantiating the output clock type in the refinement clock calculus, the environment is extended by the input binder e bound to the instantiated input clock type. When instantiating the input type, we must apply rule DeReInstRef (Equation (15.17)) which requires us to instantiate the base clock of the refined clock and replace the refinement hole by the refinement of the clock to instantiate. Instantiating the base clock follows rule DeReInstSEq-1, simply returns pck. After this, similar judgments are performed for the output type and the clock calculus can conclude the instantiation of /^2.

$$\frac{}{H \vdash \mathtt{pck} \twoheadrightarrow \zeta(\mathtt{pck}, (\pi_0(\nu) * 2)\ \mathbf{div}\ \pi(\nu) \vee \pi(\nu)\ \overset{2}{\mathbf{div}}\ \pi_0(\nu) * 2) \twoheadrightarrow \mathtt{pck}} \tag{15.16}$$

$$\frac{H \vdash \mathtt{pck} \twoheadrightarrow \zeta(\mathtt{pck}, (\pi_0(\nu) * 2)\ \mathbf{div}\ \pi(\nu) \vee \pi(\nu)\ \overset{2}{\mathbf{div}}\ \pi_0(\nu) * 2) \twoheadrightarrow \mathtt{pck}}{H \vdash \{\nu{:}\mathtt{pck} \mid \star_1\} \twoheadrightarrow \left\{\nu{:}\zeta(\mathtt{pck}, (\pi_0(\nu) * 2)\ \mathbf{div}\ \pi(\nu) \vee \pi(\nu)\ \overset{2}{\mathbf{div}}\ \pi_0(\nu) * 2) \mid true\right\} \twoheadrightarrow \{\nu{:}\mathtt{pck} \mid true\}} \tag{15.17}$$

$$\frac{\begin{array}{c} H \vdash \{\nu{:}\mathtt{pck} \mid \star_1\} \twoheadrightarrow \left\{\nu{:}\zeta(\mathtt{pck}, (\pi_0(\nu) * 2)\ \mathbf{div}\ \pi(\nu) \vee \pi(\nu)\ \overset{2}{\mathbf{div}}\ \pi_0(\nu) * 2) \mid true\right\} \twoheadrightarrow \{\nu{:}\mathtt{pck} \mid true\} \\ H; e{:}\{\nu{:}\mathtt{pck} \mid true\} \vdash \{\nu{:}\mathtt{pck} \mid \star_3\} \twoheadrightarrow \left\langle \zeta \left\langle \mathtt{pck} \mid \pi_0(e) * lcm\left(\frac{\pi_\xi(e)}{\pi_0(e)}, 2\right), \varphi_\xi(e) \right\rangle \mid \pi_0(e) * 2, \varphi_0(e) \right\rangle \twoheadrightarrow \\ \langle \mathtt{pck} \mid \pi_0(e) * 2, \varphi_0(e) \rangle \end{array}}{\begin{array}{c} H \vdash e{:}\{\nu{:}\mathtt{pck} \mid \star_1\} \to \{\nu{:}\mathtt{pck} \mid \star_3\} \\ \twoheadrightarrow e{:}\left\{\nu{:}\zeta(\mathtt{pck}, (\pi_0(\nu) * 2)\ \mathbf{div}\ \pi(\nu) \vee \pi(\nu)\ \overset{2}{\mathbf{div}}\ \pi_0(\nu) * 2) \mid true\right\} \to \\ \left\langle \zeta \left\langle \mathtt{pck} \mid \pi_0(e) * lcm\left(\frac{\pi_\xi(e)}{\pi_0(e)}, 2\right), \varphi_\xi(e) \right\rangle \mid \pi_0(e) * 2, \varphi_0(e) \right\rangle \\ \twoheadrightarrow e{:}\{\nu{:}\mathtt{pck} \mid true\} \to \langle \mathtt{pck} \mid \pi_0(e) * 2, \varphi_0(e) \rangle \end{array}} \tag{15.18}$$

$$\frac{\begin{array}{c} H \vdash \texttt{pck}\checkmark \qquad H \vdash e{:}\{\nu{:}\texttt{pck} \mid \star_1\} \to \{\nu{:}\texttt{pck} \mid \star_3\} \\ \twoheadrightarrow e{:}\left\{\nu{:}\zeta(\alpha, (\pi_0(\nu) * 2) \mathbin{\textbf{div}} \pi(\nu) \vee \pi(\nu) \overset{2}{\mathbin{\textbf{div}}} \pi_0(\nu) * 2) \mid true\right\} \to \\ \left\langle \zeta \left\langle \alpha \mid \pi_0(e) * lcm\left(\frac{\pi_\xi(e)}{\pi_0(e)}, 2\right), \varphi_\xi(e)\right\rangle \mid \pi_0(e) * 2, \varphi_0(e)\right\rangle [\alpha := \texttt{pck}] \\ \twoheadrightarrow e{:}\{\nu{:}\texttt{pck} \mid true\} \to \langle \texttt{pck} \mid \pi_0(e) * 2, \varphi_0(e)\rangle \end{array}}{\begin{array}{c} H \vdash e{:}\{\nu{:}\texttt{pck} \mid \star_1\} \to \{\nu{:}\texttt{pck} \mid \star_3\} \\ \twoheadrightarrow \forall\alpha.e{:}\left\{\nu{:}\zeta(\alpha, (\pi_0(\nu) * 2) \mathbin{\textbf{div}} \pi(\nu) \vee \pi(\nu) \overset{2}{\mathbin{\textbf{div}}} \pi_0(\nu) * 2) \mid true\right\} \to \\ \left\langle \zeta \left\langle \alpha \mid \pi_0(e) * lcm\left(\frac{\pi_\xi(e)}{\pi_0(e)}, 2\right), \varphi_\xi(e)\right\rangle \mid \pi_0(e) * 2, \varphi_0(e)\right\rangle \\ \twoheadrightarrow e{:}\{\nu{:}\texttt{pck} \mid true\} \to \langle \texttt{pck} \mid \pi_0(e) * 2, \varphi_0(e)\rangle \end{array}} \tag{15.19}$$

Now, let's look at an example featuring an **on** operator. At Line 9, operator `*^3` is instantiated. Because it is applied to an argument featuring an **on**, we must substitute clock variable α by the clock $\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid true\})$ in rule DeReInstPoly (Equation (15.24)). We then apply rule DeReInstFun (Equation (15.23)) similarly. This time however, we will focus of the instantiation of the output clock type. For this, we apply rule DeReInstRef (Equation (15.22)).

This requires us to apply rule DeReInstSEq-2 (Equation (15.21)), demonstrating the impact of the ζ operator. The refinement of the view is replaced by the refinement passed to the ζ operator after replacing the special variable ξ by the current depth of the clock, 1 in our case. This allows us to express the relation between the views of the input and output clock type. In our case, this allows us to express the fact that the `*^3` operator does not change the view of the output clock. Finally, the instantiation of the ζ operator recurses and terminates with rule DeStInstSEq-1 (Equation (15.20)).

$$\frac{}{H; e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid true\}) \mid k \mathbin{\textbf{div}} \pi(\nu)\} \vdash \texttt{pck} \twoheadrightarrow \zeta\,\langle \texttt{pck} \mid \pi_\xi(e), \varphi_\xi(e)\rangle \twoheadrightarrow \texttt{pck}} \tag{15.20}$$

$$\frac{H; e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid true\}) \mid k \mathbin{\textbf{div}} \pi(\nu)\} \vdash \texttt{pck} \twoheadrightarrow \zeta\,\langle \texttt{pck} \mid \pi_\xi(e), \varphi_\xi(e)\rangle \twoheadrightarrow \texttt{pck}}{\begin{array}{c} H; e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid true\}) \mid k \mathbin{\textbf{div}} \pi(\nu)\} \vdash \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_5\}) \\ \twoheadrightarrow \zeta\,\langle \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid true\}) \mid \pi_\xi(e), \varphi_\xi(e)\rangle \twoheadrightarrow \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \langle \texttt{pck} \mid \pi_1(e), \varphi_1(e)\rangle) \end{array}} \tag{15.21}$$

$$\frac{\begin{array}{c} H; e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid true\}) \mid k \mathbin{\textbf{div}} \pi(\nu)\} \vdash \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_5\}) \\ \twoheadrightarrow \zeta\,\langle \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid true\}) \mid \pi_\xi(e), \varphi_\xi(e)\rangle \twoheadrightarrow \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \langle \texttt{pck} \mid \pi_1(e), \varphi_1(e)\rangle) \end{array}}{\begin{array}{c} H; e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid true\}) \mid k \mathbin{\textbf{div}} \pi(\nu)\} \vdash \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_5\}) \mid \star_3\} \\ \twoheadrightarrow \langle \zeta\,\langle \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid true\}) \mid \pi_\xi(e), \varphi_\xi(e)\rangle \mid \pi_0(e)/k, \varphi_0(e)\rangle \\ \twoheadrightarrow \langle \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \langle \texttt{pck} \mid \pi_1(e), \varphi_1(e)\rangle) \mid \pi_0(e)/k, \varphi_0(e)\rangle \end{array}} \tag{15.22}$$

$$\frac{\begin{array}{c}H \vdash \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\}) \mid \star_1\} \twoheadrightarrow \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \mathit{true}\}) \mid k\ \textbf{div}\ \pi(\nu)\} \\ \twoheadrightarrow \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \mathit{true}\}) \mid k\ \textbf{div}\ \pi(\nu)\} \\ H; e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \mathit{true}\}) \mid k\ \textbf{div}\ \pi(\nu)\} \vdash \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_5\}) \mid \star_3\} \\ \twoheadrightarrow \langle \zeta\ \langle \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \mathit{true}\}) \mid \pi_\xi(e), \varphi_\xi(e)\rangle \mid \pi_0(e)/k, \varphi_0(e)\rangle \\ \twoheadrightarrow \langle \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \langle \texttt{pck} \mid \pi_1(e), \varphi_1(e)\rangle) \mid \pi_0(e)/k, \varphi_0(e)\rangle \end{array}}{\begin{array}{c} H \vdash e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\}) \mid \star_1\} \to \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_5\}) \mid \star_3\} \\ \twoheadrightarrow e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \mathit{true}\}) \mid k\ \textbf{div}\ \pi(\nu)\} \to \\ \langle \zeta\ \langle \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \mathit{true}\}) \mid \pi_\xi(e), \varphi_\xi(e)\rangle \mid \pi_0(e)/k, \varphi_0(e)\rangle \\ \twoheadrightarrow e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \mathit{true}\}) \mid k\ \textbf{div}\ \pi(\nu)\} \to \\ \langle \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \langle \texttt{pck} \mid \pi_1(e), \varphi_1(e)\rangle) \mid \pi_0(e)/k, \varphi_0(e)\rangle \end{array}} \tag{15.23}$$

$$\frac{\begin{array}{c} H \vdash \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \mathit{true}\})\ \checkmark \\ H \vdash e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\}) \mid \star_1\} \to \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_5\}) \mid \star_3\} \\ \twoheadrightarrow e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \mathit{true}\}) \mid k\ \textbf{div}\ \pi(\nu)\} \to \\ \langle \zeta\ \langle \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \mathit{true}\}) \mid \pi_\xi(e), \varphi_\xi(e)\rangle \mid \pi_0(e)/k, \varphi_0(e)\rangle \\ \twoheadrightarrow e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \mathit{true}\}) \mid k\ \textbf{div}\ \pi(\nu)\} \to \\ \langle \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \langle \texttt{pck} \mid \pi_1(e), \varphi_1(e)\rangle) \mid \pi_0(e)/k, \varphi_0(e)\rangle \end{array}}{\begin{array}{c} H \vdash e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_4\}) \mid \star_1\} \to \{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \star_5\}) \mid \star_3\} \\ \twoheadrightarrow \forall\alpha.e{:}\{\nu{:}\alpha \mid k\ \textbf{div}\ \pi(\nu)\} \to \langle \zeta\ \langle \alpha \mid \pi_\xi(e), \varphi_\xi(e)\rangle \mid \pi_0(e)/k, \varphi_0(e)\rangle \\ \twoheadrightarrow e{:}\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \{\nu{:}\texttt{pck} \mid \mathit{true}\}) \mid k\ \textbf{div}\ \pi(\nu)\} \to \\ \langle \texttt{pck}\ \textbf{on}\ \texttt{false}\,(\texttt{c}, \langle \texttt{pck} \mid \pi_1(e), \varphi_1(e)\rangle) \mid \pi_0(e)/k, \varphi_0(e)\rangle \end{array}} \tag{15.24}$$

15.6.7 Node Rules

The node rules in Figure 15.17 resemble those of Section 15.6.7, but we introduce refinements. The rules are:

- DeReImpNd: We produce a clock type similar as in DeStImpNd, but in addition the refinements states that all periods and offsets are equal among arguments and outputs;
- DeReNd: We instantiate clock types for the input, local and output variables using the results of DeStNd as a guide and then verify the well-formedness of equations within the extended body.

15.6.8 Equation Rules

Except for the missing S, the rules in Figure 15.18 are identical to the ones in Section 15.6.7. They are:

- DeReEq: An equation is well-formed, if the left-hand side and the right-hand side both synthesize the same type;
- DeReEqs: Multiple equations are well-formed, if the individual equations are well-formed.

DeReDeclPck

$$\frac{}{H \vdash \{\nu{:}\texttt{pck} \mid \star\} \twoheadrightarrow \texttt{rate (n,p)} \twoheadrightarrow \langle \texttt{pck} \mid n, p\rangle}$$

DeReDeclOn

$$\frac{H \vdash \left\{\nu{:}ck_b^S \mid \star\right\} \twoheadrightarrow \texttt{rate}\langle ck\rangle \twoheadrightarrow \{\nu{:}ck_b \mid r\} \qquad H \vdash w_s \twoheadrightarrow \texttt{" "} \twoheadrightarrow w}{H \vdash \left\{\nu{:}ck_b^S \textbf{ on } C\,(c, w_s) \mid \star\right\} \twoheadrightarrow \texttt{rate}\langle ck\rangle\texttt{on C(c)} \twoheadrightarrow \{\nu{:}ck_b \textbf{ on } C\,(c, w) \mid r\}}$$

DeReDeclEmptyPck

$$\frac{H \vdash r\checkmark}{H \vdash \{\nu{:}\texttt{pck} \mid \star\} \twoheadrightarrow \texttt{" "} \twoheadrightarrow \{\nu{:}\texttt{pck} \mid r\}}$$

DeReDeclEmptyOn

$$\frac{H \vdash \left\{\nu{:}ck_b^S \mid \star\right\} \twoheadrightarrow \texttt{" "} \twoheadrightarrow \{\nu{:}ck_b \mid r\} \qquad H \vdash w_s \twoheadrightarrow \texttt{" "} \twoheadrightarrow w}{H \vdash \left\{\nu{:}ck_b^S \textbf{ on } C\left(c, w^S\right) \mid \star\right\} \twoheadrightarrow \texttt{" "} \twoheadrightarrow \{\nu{:}ck_b \textbf{ on } C\,(c, w) \mid r\}}$$

DeReInstPoly

$$\frac{H \vdash ck_b\checkmark \qquad H \vdash ck_e^S \twoheadrightarrow \sigma[\alpha := ck_b] \twoheadrightarrow ck_e}{H \vdash ck_e^S \twoheadrightarrow \forall\alpha.\sigma \twoheadrightarrow ck_e}$$

DeReInstFun

$$\frac{H \vdash ck_r^S \twoheadrightarrow ck_r' \twoheadrightarrow ck_r \qquad H; x{:}ck_r \vdash ck_e^S \twoheadrightarrow ck_e' \twoheadrightarrow ck_e}{H \vdash x{:}ck_r^S \to ck_e^S \twoheadrightarrow x{:}ck_r' \to ck_e' \twoheadrightarrow x{:}ck_r \to ck_e}$$

DeReInstTup

$$\frac{H \vdash ck_e^{S0} \twoheadrightarrow ck_e^A \twoheadrightarrow ck_e^0 \qquad H \vdash ck_e^{S1} \twoheadrightarrow ck_e^B \twoheadrightarrow ck_e^1}{H \vdash ck_e^{S0} \times ck_e^{S1} \twoheadrightarrow ck_e^A \times ck_e^B \twoheadrightarrow ck_e^0 \times ck_e^1}$$

DeReInstRef

$$\frac{H \vdash ck_b^S \twoheadrightarrow ck_b' \twoheadrightarrow ck_b}{H \vdash \left\{\nu{:}ck_b^S \mid \star\right\} \twoheadrightarrow \left\{\nu{:}ck_b' \mid r\right\} \twoheadrightarrow \{\nu{:}ck_b \mid r\}}$$

DeReInstSEq-1

$$\frac{}{H \vdash \texttt{pck} \twoheadrightarrow \zeta(\texttt{pck}, r) \twoheadrightarrow \texttt{pck}}$$

DeReInstSEq-2

$$\frac{H \vdash ck_b^S \twoheadrightarrow \zeta(ck_b', r) \twoheadrightarrow ck_b}{\begin{array}{l} H \vdash ck_b^S \textbf{ on } C\,(c, \{\nu{:}\texttt{pck} \mid \star\}) \twoheadrightarrow \zeta(ck_b' \textbf{ on } C\left(c, \left\{\nu{:}\texttt{pck} \mid r'\right\}\right), r) \twoheadrightarrow \\ \quad ck_b \textbf{ on } C\left(c, \left\{\nu{:}\texttt{pck} \mid r[\pi_\xi := \pi_{depth(ck_b)+1}, \varphi_\xi := \varphi_{depth(ck_b)+1}]\right\}\right) \end{array}}$$

DeReInstOn

$$\frac{H \vdash ck_b^S \twoheadrightarrow ck_b' \twoheadrightarrow ck_b}{H \vdash ck_b^S \textbf{ on } C\,(c, \{\nu{:}\texttt{pck} \mid \star\}) \twoheadrightarrow ck_b' \textbf{ on } C\,(c, \{\nu{:}\texttt{pck} \mid r\}) \twoheadrightarrow ck_b \textbf{ on } C\,(c, \{\nu{:}\texttt{pck} \mid r\})}$$

DeReInstPck

$$\frac{}{H \vdash \texttt{pck} \twoheadrightarrow \texttt{pck} \twoheadrightarrow \texttt{pck}}$$

Figure 15.16: Declarative Refinement Clock Instantiation

DeReImpNd

$$\frac{H;\alpha \vdash \{\nu{:}\alpha \mid true\}\checkmark \qquad H;\alpha;v_{in}{:}\{\nu{:}\alpha \mid true\} \vdash \left\langle \zeta \left\langle \alpha \mid \pi(v_{in})^{\xi}, \varphi(v_{in})^{\xi} \right\rangle \mid \pi(v_{in}^{0}), \varphi(v_{in}^{0}) \right\rangle \checkmark \quad b_{in} = (v_{in}, \{\nu{:}\alpha \mid true\}) :: [v_{ins}, \left\langle \zeta \left\langle \alpha \mid \pi(v_{in})^{\xi}, \varphi(v_{in})^{\xi} \right\rangle \mid \pi(v_{in}^{0}), \varphi(v_{in}^{0}) \right\rangle] \quad b_{out} = [a_{out}, \left\langle \zeta \left\langle \alpha \mid \pi(v_{in})^{\xi}, \varphi(v_{in})^{\xi} \right\rangle \mid \pi(v_{in}^{0}), \varphi(v_{in}^{0}) \right\rangle] \qquad currify\,(b_{in}, b_{out}) = ck_e}{H \vdash \texttt{imported node}\ N((v_{in} : t_{in}) :: [v_{ins} : t_{ins}])\ \texttt{returns}\ ([v_{out} : t_{out}]); \overset{S}{\Rightarrow} \forall\alpha.ck_e}$$

DeReNd

$$\frac{\begin{array}{c} H \vdash [\texttt{rate}\langle ck\rangle_{in}] \overset{S}{\Rightarrow} [ck_{in}^{S}] \qquad H \vdash [ck_{in}^{S}] \twoheadrightarrow [\texttt{rate}\langle ck\rangle_{in}] \twoheadrightarrow [ck_{in}] \\ H \vdash [\texttt{rate}\langle ck\rangle_{loc}] \overset{S}{\Rightarrow} [ck_{loc}] \qquad H;[v_{in}]{:}[ck_{in}] \vdash [ck_{loc}^{S}] \twoheadrightarrow [\texttt{rate}\langle ck\rangle_{loc}] \twoheadrightarrow [ck_{loc}] \\ H \vdash [\texttt{rate}\langle ck\rangle_{out}] \overset{S}{\Rightarrow} [ck_{out}] \qquad H;[v_{in}]{:}[ck_{in}] \vdash [ck_{out}^{S}] \twoheadrightarrow [\texttt{rate}\langle ck\rangle_{out}] \twoheadrightarrow [ck_{out}] \\ H;[v_{in} : ck_{in}];[v_{loc} : ck_{loc}];[v_{out} : ck_{out}] \vdash eqs\checkmark \qquad currify\,([v_{in}, ck_{in}], [v_{out}, ck_{out}]) = ck_e \end{array}}{\begin{array}{l} H \vdash \texttt{node}\ N([v_{in} : t_{in}\ \texttt{rate}\langle ck\rangle_{in}])\ \texttt{returns}\ ([v_{out} : t_{out}\ \texttt{rate}\langle ck\rangle_{out}])\ \texttt{var}\ [v_{loc} : t_{loc}\ \texttt{rate}\langle ck\rangle_{loc}]\ \texttt{;} \\ \qquad \texttt{let}\ eqs\ \texttt{tel} \Rightarrow ck_e \end{array}}$$

Figure 15.17: Declarative Refinement Node Rules

DeReEq

$$\frac{H \vdash x \Rightarrow \sigma \qquad H \vdash e \Rightarrow \sigma}{H \vdash x = e\checkmark}$$

DeReEqs

$$\frac{H \vdash eq_1\checkmark \qquad H \vdash eq_2\checkmark}{H \vdash eq_1; eq_2\checkmark}$$

Figure 15.18: Declarative Refinement Equation Rules

15.6.9 Expression Rules

Expression rules of the refinement clock calculus in Figure 15.19 introduce subtleties which are linked to the presence of refinements. The rules are:

- DeReChk: To check an expression e against a clock σ, we must first synthesize a clock σ' for e and then verify that σ' is a subtype of σ;
- DeReVar: To synthesize a clock for ck_e for a variable x, we first fetch the clock σ bound to this variable inside the environment and then, using clock σ_S synthesized during the structural clock calculus, instantiate it into clock ck_e;
- DeReCst: A constant can be lifted to a dataflow of any clock;
- DeReAppl: When applying an expression e to a node or built-in operator N (a "function-like"), we first synthesize a clock $x{:}ck_r \rightarrow ck_e$ for our function-like and then check the argument e against ck_r. The expression $N(e)$ has clock ck_e after substituting the input binder x for the actual e.

Example 24 (Refinement Expression Rules)**.** Returning to our previous example, the judgments preformed by the refinement clock calculus when synthesizing a clock for `j /^2` at Line 7 are shown below. First (Equation (15.25)), rule DeReAppl states that we must instantiate a clock for this

instance of `/^2` (see Example 23). Then, we check `j` against the input clock, $\{\nu{:}\texttt{pck} \mid true\}$. Finally, the application has the output clock of the function type where occurrences of the input binder e have been substituted with the actual `j`. To check `j` against the input type, we must apply rule DeReChk (Equation (15.26)). It states that we must synthesize a clock for `j` and then verify that it is a subtype of the input clock. To synthesize the clock, we apply rule DeReVar (Equation (15.27)). It states that we must fetch the clock bound in the environment and then instantiate it using the type generated during the structural clock calculus. With this type, we can then apply rule DeReSubRef (Equation (15.28)). It states that the base clock must be a subtype of the supertype base clock (which holds trivially as both are `pck`) and then we submit a request to the SMT solver to verify that the subtype refinement implies the supertype refinement. This check is verified trivially. Thus, we can synthesize clock $\langle \texttt{pck} \mid 45, 0\rangle$ for our expression.

$$\frac{H \vdash \texttt{/\^2} \Rightarrow e{:}\{\nu{:}\texttt{pck} \mid true\} \to \langle \texttt{pck} \mid \pi(e) * 2, \varphi(e)\rangle \qquad H \vdash \texttt{j} \Leftarrow \{\nu{:}\texttt{pck} \mid true\}}{H \vdash \texttt{j}\ \texttt{/\^2} \Rightarrow \langle \texttt{pck} \mid \pi_0(\texttt{j}) * 2, \varphi_0(\texttt{j})\rangle} \tag{15.25}$$

$$\frac{H \vdash \texttt{j} \Rightarrow \langle \texttt{pck} \mid 45, 0\rangle \qquad H \vdash \langle \texttt{pck} \mid 45, 0\rangle <: \{\nu{:}\texttt{pck} \mid true\}}{H \vdash \texttt{j} \Leftarrow \{\nu{:}\texttt{pck} \mid true\}} \tag{15.26}$$

$$\frac{(\texttt{j} : \langle \texttt{pck} \mid 45, 0\rangle) \in H \qquad H \vdash \left\{\nu{:}\texttt{pck} \mid \star_{\texttt{j}}\right\} \twoheadrightarrow \langle \texttt{pck} \mid 45, 0\rangle \twoheadrightarrow \langle \texttt{pck} \mid 45, 0\rangle}{H \vdash \texttt{j} \Rightarrow \langle \texttt{pck} \mid 45, 0\rangle} \tag{15.27}$$

$$\frac{H \vdash \texttt{pck} <: \texttt{pck} \qquad \dfrac{SAT(\forall \texttt{period} \in \mathbb{N}, \texttt{offset} \in \mathbb{N}.(\texttt{period} = 45 \wedge \texttt{offset} = 0) \to true)}{H \vdash (\pi(\nu) = 45 \wedge \varphi(\nu) = 0) \to true}}{H \vdash \langle \texttt{pck} \mid 45, 0\rangle <: \{\nu{:}\texttt{pck} \mid true\}} \tag{15.28}$$

Looking at our second example, `a /^3`, the clock calculus performs the following judgments. As in the example above, we start with rule DeReAppl (Equation (15.29). We instantiate a clock for this instance of `/^3` and then check `a` against the input clock. The check `a` against the input type, we apply rule DeReChk (Equation (15.30)). It states that we must synthesize a clock for `a` and then verify that it is a subtype of the input clock. To synthesize the clock, we apply rule DeReVar similarly as above. To perform the subtyping check, we use rule DeReSubRef as above and submit a request to our SMT solver. However, this time the base clock features an **on**. Thus, we apply rule DeReSubOn to verify that the view of `a` is indeed a subtype of the view of the input clock. Note that we only pass refinements relative to the offset of clocks here. Period refinements of views are handled later (see Section 15.7). With the subtyping check succeeding, we can synthesize for `a/^3` the clock type $\left\langle \texttt{pck}\ \textbf{on}\ \texttt{true}\left(\texttt{c}, \left\langle \texttt{pck} \mid \pi_0(\texttt{a}) * lcm\left(\frac{\pi_1(\texttt{a})}{\pi_0(\texttt{a})}, 3\right), \varphi_1(\texttt{a})\right\rangle\right) \mid \pi_0(\texttt{a}) * 3, \varphi_0(\texttt{a})\right\rangle$.

$$\frac{\begin{array}{c} H \vdash \texttt{/\^3} \Rightarrow e{:}\left\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{true}\left(\texttt{c}, \left\{\nu{:}\texttt{pck} \mid (\pi_0(\nu) * 3\ \textbf{div}\ \pi(\nu)) \vee \pi(\nu) \overset{3}{\textbf{div}} (\pi_0(\nu) * 3)\right\}\right) \mid true\right\} \to \\ \left\langle \texttt{pck}\ \textbf{on}\ \texttt{true}\left(\texttt{c}, \left\langle \texttt{pck} \mid \pi_0(e) * lcm\left(\frac{\pi_1(e)}{\pi_0(e)}, k\right), \varphi_1(e)\right\rangle\right) \mid \pi_0(e) * 3, \varphi_0(e)\right\rangle \\ H \vdash \texttt{a} \Leftarrow \left\{\nu{:}\texttt{pck}\ \textbf{on}\ \texttt{true}\left(\texttt{c}, \left\{\nu{:}\texttt{pck} \mid (\pi_0(\nu) * 3\ \textbf{div}\ \pi(\nu)) \vee \pi(\nu) \overset{3}{\textbf{div}} (\pi_0(\nu) * 3)\right\}\right) \mid true\right\} \end{array}}{H \vdash \texttt{a}\ \texttt{/\^3} \Rightarrow \left\langle \texttt{pck}\ \textbf{on}\ \texttt{true}\left(\texttt{c}, \left\langle \texttt{pck} \mid \pi_0(\texttt{a}) * lcm\left(\frac{\pi_1(\texttt{a})}{\pi_0(\texttt{a})}, 3\right), \varphi_1(\texttt{a})\right\rangle\right) \mid \pi_0(\texttt{a}) * 3, \varphi_0(\texttt{a})\right\rangle} \tag{15.29}$$

$$\textsc{DeReChk}\ \frac{H \vdash e \Rightarrow \sigma' \qquad H \vdash \sigma' <: \sigma}{H \vdash e \Leftarrow \sigma} \qquad \textsc{DeReVar}\ \frac{(x : \sigma) \in H \qquad H \vdash x \Rightarrow \sigma^S \qquad H \vdash \sigma^S \twoheadrightarrow \sigma \twoheadrightarrow ck_e}{H \vdash x \Rightarrow ck_e}$$

$$\textsc{DeReCst}\ \frac{}{H \vdash c \Rightarrow \sigma} \qquad \textsc{DeReAppl}\ \frac{H \vdash N \Rightarrow x{:}ck_r \to ck_e \qquad H \vdash e \Leftarrow ck_e}{H \vdash N(e) \Rightarrow ck_e[x := e]}$$

Figure 15.19: Declarative Refinement Expression Rules

$$\frac{H \vdash \mathtt{a} \Rightarrow \langle \mathtt{pck}\ \mathbf{on}\ \mathtt{true}\,(\mathtt{c}, \langle \mathtt{pck} \mid \_, \varphi_0(\mathtt{i}) \mid \pi_0(\mathtt{i})\ \mathbf{div}\ \pi(\nu) \wedge \pi_0(\mathtt{c})\ \mathbf{div}\ \pi(\nu)\rangle) \mid \pi_0(\mathtt{i}), \varphi_0(\mathtt{i})\rangle = ck_{syn} \qquad H \vdash ck_{syn} <: ck_{chk}}{H \vdash \mathtt{a} \Leftarrow \left\{ \nu{:}\mathtt{pck}\ \mathbf{on}\ \mathtt{true}\left(\mathtt{c}, \left\{ \nu{:}\mathtt{pck} \mid (\pi_0(\nu) * 3\ \mathbf{div}\ \pi(\nu)) \vee \pi(\nu) \overset{3}{\mathbf{div}} (\pi_0(\nu) * 3) \right\}\right) \mid true \right\} = ck_{chk}} \tag{15.30}$$

$$\frac{\dfrac{H \vdash \mathtt{pck} <: \mathtt{pck} \qquad H \vdash \varphi_0(\mathtt{c}) \to true}{H \vdash \mathtt{pck}\ \mathbf{on}\ \mathtt{true}\,(\mathtt{c}, w_{syn}) <: \mathtt{pck}\ \mathbf{on}\ \mathtt{true}\,(\mathtt{c}, w_{chk})} \qquad H \vdash (\pi(\nu) = \pi_0(\mathtt{i}) \wedge \varphi(\nu) = \varphi_0(\mathtt{i})) \to true}{H \vdash ck_{syn} <: ck_{chk}} \tag{15.31}$$

15.7 View closing

The goal of this pass is to "close" the views inferred during the refinement clock calculus. The constraints used in our clock calculus are linear except for the `when` operator as it introduces a new view because constraints of the form $\pi(i)$ **div** $\pi(\nu)$ are nonlinear. However, since we require programmers to annotate the rates of node inputs, we can postpone view resolution until after the refinement clock calculus of a node succeeds but before inserting it inside the environment. At this point, periods and offsets are "solved", i.e. they can be substituted by constants. Thus, before this point, we simply collect the constraints on views and perform the view closing then.

The view closing is performed in two steps. First, we perform constant propagation. All clocks are thus of the form $\langle ck_b \mid n, p\rangle$ where n and p are constants. Then, we can submit the constraints on views to the SMT solver. The resulting system has few variables which can be rather easily solved by the solver.

Example 25 (View closing)**.** When closing node `main`, constant propagation yields an environment containing the following bindings. Note that `z` has two period constraints. The first originates from

the `*^3` application on Line 9. The second originates from the `merge` on Line 10.

$$
\begin{aligned}
&\texttt{i} : \langle \texttt{pck} \mid 10, 0 \rangle \\
&\texttt{j} : \langle \texttt{pck} \mid 45, 0 \rangle \\
&\texttt{c} : \langle \texttt{pck} \mid 15, 0 \rangle \\
&\texttt{o} : \langle \texttt{pck} \mid 30, 0 \rangle \\
&\texttt{a} : \langle \texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \langle \texttt{pck} \mid \_, 0 \mid 10\ \textbf{div}\ \pi(\nu) \wedge 15\ \textbf{div}\ \pi(\nu) \rangle) \mid 10, 0 \rangle \\
&\texttt{b} : \left\langle \texttt{pck}\ \textbf{on}\ \texttt{true}\left(\texttt{c}, \left\langle \texttt{pck} \mid 10 * lcm\left(\frac{\pi_1(\texttt{a})}{10}, 3\right), 0 \right\rangle\right) \mid 30, 0 \right\rangle \\
&\texttt{x} : \langle \texttt{pck} \mid 90, 0 \rangle \\
&\texttt{y} : \langle \texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \langle \texttt{pck} \mid \_, 0 \mid 90\ \textbf{div}\ \pi(\nu) \wedge 15\ \textbf{div}\ \pi(\nu) \rangle) \mid 90, 0 \rangle \\
&\texttt{z} : \langle \texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \langle \texttt{pck} \mid \pi_1(\texttt{y}), 0 \mid \pi(\nu) = \pi_1(\texttt{b}) \rangle) \mid 30, 0 \rangle
\end{aligned}
$$

This is then lowered into the following request to the SMT solver:

$$
\begin{aligned}
&10\ \textbf{div}\ \texttt{a-1-period} \wedge 15\ \textbf{div}\ \texttt{a-1-period} \wedge \\
&\texttt{b-1-period} = 10 * lcm(\texttt{a-1-period}/10, 3) \wedge \\
&90\ \textbf{div}\ \texttt{y-1-period} \wedge 15\ \textbf{div}\ \texttt{y-1-period} \wedge \\
&\texttt{z-1-period} = \texttt{y-1-period} \wedge \texttt{z-1-period} = \texttt{b-1-period} \\
&minimize(\texttt{a-1-period}, \texttt{b-1-period}, \texttt{y-1-period}, \texttt{z-1-period})
\end{aligned}
$$

It has for solution:

$$
\begin{aligned}
\texttt{a-1-period} &= 90 \\
\texttt{b-1-period} &= 90 \\
\texttt{y-1-period} &= 90 \\
\texttt{z-1-period} &= 90
\end{aligned}
$$

The environment of the closed node is thus:

$$
\begin{aligned}
&\texttt{i} : \langle \texttt{pck} \mid 10, 0 \rangle && \texttt{j} : \langle \texttt{pck} \mid 45, 0 \rangle \\
&\texttt{c} : \langle \texttt{pck} \mid 15, 0 \rangle && \texttt{o} : \langle \texttt{pck} \mid 30, 0 \rangle \\
&\texttt{a} : \langle \texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \langle \texttt{pck} \mid 90, 0 \rangle) \mid 10, 0 \rangle && \texttt{b} : \langle \texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \langle \texttt{pck} \mid 90, 0 \rangle) \mid 30, 0 \rangle \\
&\texttt{x} : \langle \texttt{pck} \mid 90, 0 \rangle \\
&\texttt{y} : \langle \texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \langle \texttt{pck} \mid 90, 0 \rangle) \mid 90, 0 \rangle && \texttt{z} : \langle \texttt{pck}\ \textbf{on}\ \texttt{true}\,(\texttt{c}, \langle \texttt{pck} \mid 90, 0 \rangle) \mid 30, 0 \rangle
\end{aligned}
$$

Chapter 16

Evaluation

In this Chapter, we will tie everything back together with a discussion on the mode change protocol our state machines enable. This will allow us to take a step back and see a more complete image of our contribution.

16.1 Drone Case Study

To guide us through this section, we provide an implementation of the case study from [45, 50] in Figure 16.1. A transpilation in the core language is available in Appendix B. Note, that the transpilation process itself is unchanged compared to [28], our extension focuses on the clock semantics. It implements a large sized drone[1]. The system perceives its environment via a GPS and an Inertial Navigation System (INS). In addition, it receives via a wireless communication an enabling signal specifying in which mode it shall execute (`isEnabled`) and a destination point (`waypoint`). The system actuates via servo motors. Amplitude saturation on the computed servo commands protects them from extreme variations by ceiling the difference between two consecutive instructions.

The application has two modes. In the `Estimate` mode, the UAV preserves its previous course and measurements serve only to update the UAV position. In the `Actuate` mode, the UAV computes orders for the servo motors so as to reach the current waypoint. The automaton switches from mode `Estimate` to mode `Actuate` when expression `isEnabled` is true, and from `Actuate` to `Estimate` when `not(isEnabled)` is true. Thus, automaton transitions are emitted with rate $(10, 0)$ and by extension the dataflow holding the automaton state is on clock $(10, 0)$.

The application illustrates how our contribution allows for multi-mode multi-periodic systems: within a mode dataflows with different rates co-exist. For instance, within mode `Actuate`, `control` produces a dataflow with clock $(10, 0)$ **on** `Actuate(state,` $(10, 0)$`)` while the dataflow produced by `servo_driver` has clock $(20, 0)$ **on** `Actuate(state,` $(20, 0)$`)`. Not only do they produce values at a different rate, these dataflows also perceive the automaton state changes at different rates. This difference in views means that task `control` will respond to a change of `state` immediately, while `servo_driver` can in some cases respond with a delay of 10 time units. Thus, this automaton implements an *overlapping mode change protocol* (see below), i.e. nodes are not all impacted simultaneously by a mode change.

[1] In 2003, it would have been called a Unmanned Aerial Vehicle (UAV).

```
1  node main(GPS: GPSMessage rate(10,0); INS : INSMessage rate(10,0);
2            isEnabled: bool rate(10,0); waypoint: real[4] rate(10,0))
3  returns(servos: ServoMessage)
4  var pos, srvs;
5  let
6    servos = saturate(srvs, 0 fby servos);
7    automaton
8    | Estimate ->
9      unless isEnabled then Actuate;
10     var GPS_f, INS_f, pos_f;
11     GPS_f,INS_f,pos_f = h_f(GPS, INS, init_pos fby* pos);
12     pos = filter(GPS_f, INS_f, pos_f);
13     srvs = init_servos fby* srvs;
14
15   | Actuate ->
16     unless not(isEnabled) then Estimate;
17     var GPS_c, INS_c, pos_c, waypoint_c, controls;
18     GPS_c,INS_c,pos_c,waypoint_c = h_c(GPS, INS, init_pos fby* pos, waypoint);
19     pos,controls = control(GPS_c, INS_c, pos_c, waypoint_c);
20     srvs = servo_driver(controls/^2);
21   end
22 tel
```

Figure 16.1: Case Study in the Surface Language

16.2 Mode Change Protocols

Before discussing our contribution, let's recall some key elements from Chapter 5. First, recall that a mode change protocol describes how to transition a real-time system from one mode of execution into another, and that Mode Change Requests (MCRs) trigger this change. In [82] the criteria by which a mode change protocol can be classified are:

- *Overlapping*: when do the *new-mode tasks* start executing?
- *Periodicity*: are *unchanged tasks* impacted by mode changes?
- *Retirement*: what happens to *old-mode tasks* during a mode change?

Unchanged tasks In our work, unchanged tasks correspond to dataflows computed outside the automaton whenever a mode change occurs. For instance, in Figure 16.1, `saturate` at Line 6 is an unchanged task.

Retirement In our work, old-mode tasks continue their execution until their views perceive the state change. Since the period of the view cannot be shorter that the period of the task, this implies that we cannot implement early-retirement protocols. Supporting early-retirement would require to interrupt a task during its execution. Other synchronous languages, most notably Esterel, manage to do this with operators such as `abort`. However, in a dataflow context, implementing such an

```
@@ -4,8 +4,9 @@
-var pos;
+var pos, isEnabled_slow;
  let
   servos = saturate(srvs, 0 fby servos)
+  isEnabled_slow = isEnabled/^2;
   automaton
   | Estimate ->
-    unless isEnabled then Actuate;
+    unless isEnabled_slow then Actuate;
@@ -14,15 +15,16 @@
   | Actuate ->
-    unless isEnabled then Estimate;
+    unless isEnabled_slow then Actuate;
```

Figure 16.2: Changes for a non-overlapping

operator would raise serious semantics concerns and thus is out-of-scope of this work. Note, that clock views would still be usable in this context as they could be used to describe the rate at which a dataflow could be aborted.

Periodicity Our work only supports periodic protocols. Indeed, the execution of a node is triggered by the arrival of its inputs, it cannot be interrupted once it starts processing its inputs. As for early-retirement, supporting aperiodicity is antagonistic with dataflow semantics.

Overlapping We can implement both overlapping and non-overlapping protocols. As said earlier, our implementation in Figure 16.1 implements an overlapping protocol. Nodes `filter` and `control`, which compute `pos`, have the same view, $(10, 0)$. Thus, during a transition from one state to another, there is no overlap between nodes `filter` and `control`. However, in case of a transition from state `Actuate` to state `Estimate`, there can be an overlap, since `filter` (from the new mode) and `servo_driver` (from the old mode) may co-exist (because `servo_driver` has view $(20, 0)$).

An automaton implements a non-overlapping protocol iff all views are equal among dataflows within that automaton. To implement a non-overlapping version of the case study, one possibility is to slow down the period of transitions. Figure 16.2 shows the changes required. We define a new dataflow `isEnabled_slow`, a down-sampled version of `isEnabled`, and replace it everywhere inside the automaton. Now, `state` has clock $(20, 0)$ and the views of all expressions inside the automaton become $(20, 0)$. Thus, the automaton implements a non-overlapping mode change protocol.

Impact of programmer choices on views One point worth discussing is how the different choices of the programmer might impact views. Indeed, even in a simple program, small changes in periods could result in vastly different views.

For an expression `e when S1(s)`, assuming `s : rate (10,0)`, the clock of `e` has a great impact on the expression view. Below, we illustrate on some broad categories how the relation between the clock of `e` and `c` impacts the clock view.

Ideal mode changes If $\pi(\texttt{e})$ **div** $\pi(\texttt{s})$ (e.g. `e: rate (10,0)` or `e: rate (5,0)`), the view is $(10, 0)$ and thus the `when` expression observes mode changes as soon as possible. This represents in our opinion an *ideal* mode change. Most notably, at no point does `e when S1(s)` produce a value when the last value of `s` was different from `S1`.

Damaged ideal mode changes If $\pi(\texttt{s})$ **div** $\pi(\texttt{e})$ (e.g. `e: rate (20,0)`), the view is equal to the clock of `e` (e.g. `rate (20,0)`). We observe here what we call *view damage*. Indeed, because the period of `e` is greater, there is some loss of information about the state. Any further dataflow downstream will have no information about the state between ticks of the view. However, this scenario still meets the definition of an ideal mode change as above. The loss of information is in our opinion manageable.

Non-ideal mode changes with slow dataflows If $\pi(\texttt{e}) > \pi(\texttt{s}) \wedge \neg\pi(\texttt{s})$ **div** $\pi(\texttt{e})$ (e.g. `e: rate (15,0)`), the view becomes the least common multiple of the periods (e.g. `rate (30,0)`). As the view is less frequent than `s`, but `e` is more frequent than the view, some values of `e` do not observe the most recent value of `s`. In our example, if the state changes at date 10, `e when S1(s)` produces a value at date 15 because of the view.

Non-ideal mode changes with fast dataflows However, going faster does not always produce an ideal mode change. If $\pi(\texttt{e}) < \pi(\texttt{s}) \wedge \neg\pi(\texttt{e})$ **div** $\pi(\texttt{s})$ (e.g. `e: rate (6,0)`), the view still becomes the least common multiple of the periods (e.g. `rate (30,0)`). This mode change protocol behaves similar to the one above. Note that an even greater number of values of `e` do not observe the most recent value of `s`.

Asymptotic cases If $\pi(\texttt{e})$ and $\pi(\texttt{s})$ are coprime, i.e. their greatest common divisor is 1, the view grows exponentially. For instance with `e: rate (11,0)`, the view becomes `rate (110,0)`.

In most cases, the period of a clock view becomes the least-common multiple of the affected dataflows. Thus, under certain circumstances a certain trade-off has to be done. If for instance the period of `e` may not be lower than 15, the system designer will have to decide whether they prefer a higher degree of QoS (and thus assign the clock `rate (15,0)`), or a better mode change promptness (and thus assign the clock `rate (20,0)`.

Transparent mode changes Another point of note, is that with increasing view periods, *transparent* mode changes become more likely. When the state changes multiple times between ticks of the view, we call all but the last mode change transparent with respect to the dataflow and its view. Taking the asymptotic case above, if the state changes each time `s` is present, the values of `e` would only observe the state changes at dates which are multiples of 110, the state changes of dates 10, 20, 30, etc. could not be observed and are thus transparent. If this is of importance depends entirely on the system to be implemented.

Conclusion These examples illustrate the benefit of separating the execution rate of a flow (its strictly periodic clock) from the rate at which it perceives MCRs (its state transition view). This allows us to reason about mode change protocols, and avoids misinterpretations and ambiguities, while preserving the flexibility of the language.

16.3 Emergent Properties

Beyond the properties of overlap, retirement, and periodicity, one might be interested in determining other, derived properties, e.g. the Worst-Case Mode Change Promptness (WCMCP) of a protocol, i.e. the maximum amount of time that is required to transition from one state into another. From a point of view of real-time scheduling, calculating this value for most mode change protocols is tedious as MCRs may be emitted at any point. However, due to the formal nature of our language, MCRs are only emitted whenever a transition fires which is constrained by a specific clock. Thus, we propose the following conjecture to derive the WCMCP of a task from its clock:

Conjecture 1 (Worst-Case Mode Change Promptness)**.** *The Worst-Case Mode Change Promptness (WCMCP) of a dataflow s with respect to a condition c is defined as*

$$WCMCP(s) = n - \pi(\widehat{s}) + (p - \varphi(\widehat{s})) \qquad \text{where } \widehat{s} = ck \textbf{ on } C(c,(n,p))$$

Note that this is not an exact value, but an upper-bound whose resolution is the period of the task. For instance given a dataflow s with $\widehat{s} = (10,0)$ **on** $C(c,(20,0))$, we have $WCMCP(s) = 20 - 10 + (0 - 0) = 10$, i.e. it takes in the worst-case 10 time units, or one activation, for s to respond to a mode change. This corresponds to the case where c changes at dates 10, 30, 50, etc., but dataflow s still observes the value of c produced at dates 0, 20, 40, etc. Thus, from the point of view of synchronous Kahn semantics, there is at most a delay of one activation. This further illustrates how our approach leads to a system that is easier to reason about.

One might however argue, that an exact real-time scheduling analysis could determine that the worst-case response time of the task implementing s is some $n < 10$. This level of detail is of little importance in our opinion as it corresponds to the loss of detail entailed by the abstraction of logical time.

Part V

Conclusion

Chapter 17

Summary and Perspectives

In this Chapter, we will provide a summary of our contribution and provide perspectives where future work could improve our contribution.

17.1 Summary

In our work, we tackled two issues. First, the programming of real-time systems on multicore platforms. Second, the programming of real-time systems with multiple modes of execution.

17.1.1 Multicore Systems

In Part III, we presented a compilation process that translates synchronous dataflow programs into multi-task PREM-compliant C code. Tasks are divided into A-, E- and R-phases which respectively handle 1. Acquisition of task inputs from global and shared memory into local and private memory; 2. Execution of the task code using only the local memory; 3. Restitution of task outputs from local and private memory into global and shared memory. Data-dependencies are translated into precedence constraints, which are a adapted for describing the constraints on the execution order of tasks.

We evaluated our approach by producing code that targets a multicore platform with distributed memory using an industrial RTOS. The evaluation shows that we can easily configure our Prelude compiler to produce code adapted to different execution platforms and compare the performance of these variants with each other.

17.1.2 Multimode Systems

In Part IV, we presented an extension of the Prelude language semantics, more specifically of the ck **on** $C(c)$ clock operator (Chapter 12, Chapter 13). The extension allows to decouple the rate of the dataflow (ck), from how the condition is observed (c). This extension relies on the notion of *views*. In a conditional clock ck **on** $C(c, w)$, the view w is a strictly periodic clock that describes how the condition c is observed. This allows us to define a semantics for when ck and c do not have the same clock, and define a valid clock when applying a periodic clock operator ($*^{\wedge\#}$, $/^{\wedge\#}$, $\sim>^{\#}$, etc.) on a conditional clock.

This allowed us then to reuse the automata transpilation technique from [28] to define multi-periodic state machines (Chapter 14). Notably, our extension did not require any changes in the transpilation technique. Our work focused on the semantics of clocks and the transpilation technique integrated nicely.

To verify the consistency of clocks, we defined a clock calculus (Chapter 15) based on refinement typing. The properties of clocks such as periods and offsets are encoded as refinements, i.e. logical predicates, which are verified by a SMT solver. The periods of views are also computed by the SMT solver to reduce the burden of the programmer.

We validated our approach by a discussion centered around a simple case study. We compared the mode change protocol defined by our state machines with the literature on real-time scheduling, showed how the programmer can interact with the clock calculus to change the kind of mode change protocol that the automaton employs, and finally showed how our formal approach naturally leads to the emergence of properties about our state machines which are easy to reason about.

17.2 Perspectives

We identified the following avenues to improve our contribution or address limitations.

17.2.1 Refined Communications Semantics

A limitation of our work in Part III, is that there is very little way to describe how communications are performed. We simply distinguish between "private" memories which are predictable, and "shared" ones which are sources of hard-to-predict contentions.

However, on a multicore platform, communications might be more varied. For instance, a multicore platform might feature scratchpad memory which is private and thus predictable, SDRAM which is shared and thus a source of contentions, and flash memory which even when unshared is slow to write-to. Thus, it might be interesting to explore how to allow programmers to annotate such distinctions and influence the phase and code generation of the program, for instance by generating multiple R-phases for a task.

17.2.2 Compilation of Clock Views

Our work in Part IV focuses on the front end of the compiler. Once the clock calculus terminates, we have a fully annotated program. However, we do not consider the back end of the compiler, i.e. how to produce an executable from such a program. The original back end of Prelude as used in Part III is defined in [34, 67] and future work should identify how to adapt it to include clock views. Most notably, it should be studied if and how the communication protocol generation should be adapted.

17.2.3 Alternative Clock View Elaboration

Clock views are simply the least common multiple of the relevant dataflow periods. In pathological cases, this results in all view periods becoming the hyperperiod of the whole node which results in very poor promptness of mode change transitions. Future work should study whether this requirement could be lifted.

17.2.4 Polymorphism and Refinement Inference

Our extension of the Prelude language required us to mostly remove polymorphism from the language. Only imported nodes, which must be mono-periodic, can have polymorphic clocks in our version of the language. User-defined nodes must have their input rates annotated, which makes polymorphism impossible. The previous version of the language supported polymorphic user-defined nodes.

We introduced this restriction because this simplified the clock calculus. With the annotation of input clocks, the dataflow nature of our clock calculus guarantees that all dataflows ultimately depend upon the inputs, whose clocks are specified.

If we lifted this restriction, we would have to transition from a refinement clock calculus to a *liquid* clock calculus. The idea behind liquid typing is that in the absence of user annotations, the typer has access to refinement fragments and can construct refinements from conjunctions of these fragments. Fragments would be $\pi(\nu) = \pi(\star_{var})$, $\star_{cst}$ **div** $\pi(\nu)$, etc. where $\star_{var}$ is a placeholder for a program variable and $\star_{cst}$ would be a placeholder for a constant.

The main difficulty to introducing a liquid clock calculus is to guarantee that the types it synthesizes are *principal types* (or a sufficient approximation thereof), i.e. the most general type possible. Without such a guarantee, the liquid clock calculus would be in danger of systematically opting for the easy option of making everything mono-periodic. Thus, future work should explore a liquid clock calculus for our extended language.

17.2.5 Program Synthesis

Program synthesis is the discipline of constructing a valid program from a set of constraints (and potentially a partial program).

Clock views are a weak form of program synthesis. We express constraints on the views and use our SMT solver to find periods that satisfy them. Program synthesis for a subset of the Prelude programs has already been studied in [107]. This work however only focused on the `fby` operator. It introduced a special delay operator `dc`, "don't care", similar to `fby`. The compiler is able to decide to either keep the delay and replace the `dc` by a `fby`, or to drop it. User-provided maximal delay annotations for computation chains (e.g. there may be at most two delays in the computation chain `i => x => y => o`) guide the compiler in its choices.

Future work could expand upon this. It should be explored whether with more recent approaches to program synthesis [41], programmers could define programs with partial rate-annotations (e.g. `rate (5-10,0)` meaning $5 \leqslant \pi(\mathtt{o}) \leqslant 10 \wedge \varphi(\mathtt{o}) = 0$) and rate-transition operators (e.g. an operator `$^` indicating points where the compiler could introduce rate-transition operators). The program synthesis engine of the compiler would then substitute these annotations by concrete ones (e.g. `rate (8,0)` and `*^2/^3`).

Future work could explore, how to lower whole programs into the logic of the SMT solver. Currently, only clocks and the constraints produced during the clock calculus are lowered into SMT solver constraints. In this approach, partial annotations would simply be variables of the SMT program. A SAT result from the solver would then return a concrete program which satisfies these constraints.

Appendix A

Proof Program for Clock Deceleration

```
(set-logic QF_UFNIA)
(set-option :produce-unsat-cores true)
(set-option :produce-proofs true)

(declare-const t Int)
(declare-const t-before Int)
(declare-const t-after Int)

(declare-const pi Int)
(declare-const a Int)
(declare-const b Int)
(declare-const i Int)
(declare-const phi Int)
(declare-const r Int)
(declare-const p Int)
(declare-const s Int)

(declare-const k Int)
(declare-const pi-before Int)
(declare-const phi-before Int)
(declare-const pi-after Int)
(declare-const phi-after Int)

(assert (>= t 0))
(assert (>= t-before 0))
(assert (>= t-after 0))
(assert (>= pi 1))
(assert (>= a 1))
(assert (>= b 2))
(assert (>= i 0))
(assert (>= phi 0))
(assert (>= r 0))
(assert (>= p 0))
```

```
(assert (>= s 0))

(assert (= t (+ (* pi a b i) phi)))
(assert (= t-before (+ (* pi a r) p)))
(assert (= t-after (+ (* pi a b s)) p))

(assert (= (mod pi pi-before) 0))
(assert (= (mod pi pi-after) 0))
(assert (<= phi phi-before))
(assert (<= phi phi-after))

(assert (<= (+ (* pi a r) p) (+ (* pi a b i) phi)))
(assert (< (+ (* pi a b i) phi) (+ (* pi a (+ r 1)) p)))

(assert (<= (+ (* pi a b s) p) (+ (* pi a b i) phi)))
(assert (< (+ (* pi a b i) phi) (+ (* pi a b (+ s 1)) p)))

(assert (distinct r (* b s)))

(assert (= k (* a b)))
(assert (= pi-before (* pi a)))
(assert (= pi-after (* pi a b)))
(assert (= pi-after (* pi k)))
(assert (= phi-before p))
(assert (= phi-after p))

(check-sat)
(get-proof)
```

Appendix B

Drone Case Study in the Core Language

```
node main(GPS: GPSMessage rate(10,0); INS : INSMessage rate(10,0);
          isEnabled: bool rate(10,0); waypoint: real[4] rate(10,0))
returns(servos: ServoMessage)
var pos, srvs,
    state, previous_next_state, next_state,
    s_Estimate, s_Actuate,
    ns_Estimate, ns_Actuate,
    Estimate_isEnabled, Actuate_not_isEnabled, Actuate_isEnabled,
    Estimate_GPS, Estimate_INS, Estimate_pos_fbylast, pos_fbylast, srvs_fbylast,
    GPS_f, INS_f, pos_f, Estimate_pos, Estimate_srvs,
    Actuate_GPS, Actuate_INS, Actuate_waypoint_fbylast, waypoint_fbylast,
    GPS_c, INS_c, pos_c, waypoint_c, controls, controls_div_2,
    Actuate_pos, Actuate_srvs;
let
  srvs = merge(state, Estimate->Estimate_srvs, Actuate->Actuate_srvs);
  pos = merge(state, Estimate->Estimate_pos, Actuate->Actuate_pos);

  previous_next_state = Estimate fby next_state;
  state = merge(previous_next_state, Estimate->s_Estimate, Actuate->s_Actuate);
  next_state = merge(satte, Estimate->ns_Estimate, Actuate->ns_Actuate);

  s_Estimate = if Estimate_isEnabled then Actuate else Estimate;
  s_Actuate = if Actuate_not_isEnabled then Estimate else Actuate;

  ns_Estimate = Estimate;
  ns_Actuate = Actuate;

  Estimate_isEnabled = isEnabled when Estimate(previous_next_state);
  Actuate_not_isEnabled = not(Actuate_isEnabled);
  Actuate_isEnabled = isEnabled when Actuate(previous_next_state);
```

```
  GPS_f, INS_f, pos_f = h_f(Estimate_GPS, Estimate_INS, Estimate_pos_fbylast);
  Estimate_GPS = GPS when Estimate(state);
  Estimate_INS = INS when Estimate(state);
  Estimate_pos_fbylast = pos_fbylast when Estimate(state);
  pos_fbylast = init_pos fby pos;
  Estimate_pos = filter(GPS_f, INS_f, pos_f);
  Estimate_srvs = srvs_fby when Estimate(state);
  srvs_fbylast = init_servos fby srvs;

  GPS_c, INS_c, pos_c, waypoint_c =
    h_c(Actuate_GPS, Actuate_INS, Actuate_waypoint_fbylast);
  Actuate_GPS = GPS when Actuate(state);
  Actuate_INS = INS when Actuate(state);
  Actuate_waypoint_fbylast = waypoint_fby when Actuate(state);
  waypoint_fbylast = init_pos fby waypoint;
  Actuate_pos, controls = control(GPS_c, INS_c, pos_c, waypoint_c);
  controls_div_2 = controls/^2;
  Actuate_srvs = servo_driver(controls_div2);
tel
```

List of Figures

List of Tables

List of Definitions and Properties

List of Examples

Bibliography

[1] *The Cambridge Dictionary of Philosophy*. Cambridge University Press, 3 edition, 2015.

[2] A correct-by-construction methodology for supporting execution time variability in real-time systems, 2017.

[3] Kunal Agrawal, Sanjoy Baruah, and Alan Burns. Semi-clairvoyance in mixed-criticality scheduling. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 458–468. IEEE, 2019.

[4] Ahmed Alhammad and Rodolfo Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *Proceedings of the 14th International Conference on Embedded Software*, pages 1–10, 2014.

[5] Ahmed Alhammad and Rodolfo Pellizzoni. Time-predictable execution of multithreaded applications on multicore systems. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[6] Ahmed Alhammad, Saud Wasly, and Rodolfo Pellizzoni. Memory efficient global scheduling of real-time tasks. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 285–296. IEEE, 2015.

[7] Tochéou Amagbegnon, Loïc Besnard, and Paul Le Guernic. *Arborescent canonical form of boolean expressions*. PhD thesis, INRIA, 1994.

[8] Charles André. Syncharts: A visual representation of reactive behaviors. *I3S, Sophia-Antipolis, France, Tech. Rep. RR*, pages 95–52, 1996.

[9] CM Bailey. Hard real-time operating system kernel. investigation of mode change. *British Aerospace Systems Ltd*, 1993.

[10] Clément Ballabriga, Julien Forget, and Giuseppe Lipari. Symbolic wcet computation. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):1–26, 2017.

[11] Sanjoy Baruah and Pontus Ekberg. Graceful degradation in semi-clairvoyant scheduling. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[12] Sanjoy K Baruah, Aloysius K Mok, and Louis E Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *[1990] Proceedings 11th Real-Time Systems Symposium*, pages 182–190. IEEE, 1990.

[13] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24. IEEE, 2016.

[14] 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.

[15] Albert Benveniste, Paul Caspi, Paul Le Guernic, Hervé Marchand, Jean-Pierre Talpin, and Stavros Tripakis. A protocol for loosely time-triggered architectures. In *International Workshop on Embedded Software*, pages 252–265. Springer, 2002.

[16] Dominique Bertrand, Anne-Marie Déplanche, Sébastien Faucou, and Olivier H Roux. A study of the aadl mode change protocol. In *13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008)*, pages 288–293. IEEE, 2008.

[17] Frédéric Boussinot and Robert De Simone. The esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[18] Paolo Burgio, Andrea Marongiu, Paolo Valente, and Marko Bertogna. A memory-centric approach to enable timing-predictability within embedded many-core accelerators. In *2015 CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*, pages 1–8. IEEE, 2015.

[19] Alan Burns. System mode changes-general and criticality-based. In *Proc. of 2nd Workshop on Mixed Criticality Systems (WMC)*, pages 3–8, 2014.

[20] Alan Burns and Robert Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, pages 1–69, 2013.

[21] Alan Burns and Andrew J Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[22] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[23] Stefan Bygde, Andreas Ermedahl, and Björn Lisper. An efficient algorithm for parametric wcet calculation. *Journal of Systems Architecture*, 57(6):614–624, 2011.

[24] Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko Bertogna. Sigamma: Server based integrated gpu arbitration mechanism for memory accesses. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 48–57, 2017.

[25] Paul Caspi, Grégoire Hamon, and Marc Pouzet. Synchronous functional programming: The lucid synchrone experiment. *Real-Time Systems: Description and Verification Techniques: Theory and Tools. Hermes*, pages 28–41, 2008.

[26] Paul Caspi and Marc Pouzet. Synchronous kahn networks. *ACM SIGPLAN Notices*, 31(6):226–238, 1996.

[27] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Mixing signals and modes in synchronous data-flow systems. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 73–82, 2006.

[28] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, page 173–182, New York, NY, USA, 2005. Association for Computing Machinery.

[29] Jean-Louis Colaço and Marc Pouzet. Clocks as first class abstract types. In *International Workshop on Embedded Software*, pages 134–155. Springer, 2003.

[30] RTCA DO. 178 "software considerations in airborne systems and equipment certification," rtca. Technical report, Inc, Tech. Rep., 1st Dec. 1992. Last version DO-178C release in Jan, 2012.

[31] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), May 2021.

[32] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software (ERTS'14)*, 2014.

[33] Erika. Erika enterprise. http://erika.tuxfamily.org/drupal/.

[34] Julien Forget. *A synchronous language for critical embedded systems with multiple real-time constraints*. PhD thesis, Institut Supérieur de l'Aéronautique et de l'Espace, 2009.

[35] Julien Forget, Frédéric Boniol, Emmanuel Grolleau, David Lesens, and Claire Pagetti. Scheduling dependent periodic tasks without synchronization mechanisms. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 301–310. IEEE, 2010.

[36] Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. A multi-periodic synchronous data-flow language. In *2008 11th IEEE High Assurance Systems Engineering Symposium*, pages 251–260, 2008.

[37] Björn Forsberg, Luca Benini, and Andrea Marongiu. Heprem: Enabling predictable gpu execution on heterogeneous soc. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 539–544. IEEE, 2018.

[38] Björn Forsberg, Luca Benini, and Andrea Marongiu. Heprem: A predictable execution model for gpu-based heterogeneous socs. *IEEE Transactions on Computers*, 70(1):17–29, 2020.

[39] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, page 268–277, New York, NY, USA, 1991. Association for Computing Machinery.

[40] Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing mixed criticality applications on modern heterogeneous mpsoc platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[41] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[42] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[43] Reinhard Von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'brien, and Partha Roop. Sequentially constructive concurrency—a conservative extension of the synchronous model of computation. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):1–26, 2014.

[44] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[45] Thomas A Henzinger, Benjamin Horowitz, and Christoph M Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[46] Nicolas Hili, Alain Girault, and Eric Jenn. Worst-case reaction time optimization on deterministic multi-core architectures with synchronous languages. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–11. IEEE, 2019.

[47] Ranjit Jhala and Niki Vazou. Refinement types: A tutorial. *arXiv preprint arXiv:2010.07763*, 2020.

[48] Gilles Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[49] Yonit Kesten and Amir Pnueli. Timed and hybrid statecharts and their textual representation. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 591–620. Springer, 1992.

[50] H.Jin Kim and David H. Shim. A flight control system for aerial robots: algorithms and experiments. *Control Engineering Practice*, 11(12):1389–1400, 2003. Award winning applications-2002 IFAC World Congress.

[51] Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14. IEEE, 2019.

[52] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for system design. *Journal of Circuits, Systems, and Computers*, 12(03):261–303, 2003.

[53] Edward A Lee and Alberto Sangiovanni-Vincentelli. Comparing models of computation. In *Proceedings of International Conference on Computer Aided Design*, pages 234–241. IEEE, 1996.

[54] Paul LeGuernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[55] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[56] Daniel Lucas, Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Friedrich Gretz, and Franz-Josef Grosch. Extracting mode diagrams from blech code. In *2021 Forum on specification & Design Languages (FDL)*, pages 01–08. IEEE, 2021.

[57] Cláudio Maia, Geoffrey Nelissen, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. Schedulability analysis for global fixed-priority scheduling of the 3-phase task model. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10. IEEE, 2017.

[58] Cláudio Maia, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. A closer look into the aer model. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2016.

[59] Renato Mancuso, Roman Dudko, and Marco Caccamo. Light-prem: Automated software refactoring for predictable execution on cots embedded systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10. IEEE, 2014.

[60] Florence Maraninchi. The argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, volume 3. Citeseer, 1991.

[61] Paulo Martins, IG Hidalgo, MA Carvalho, A de Angelis, V Timóteo, R Moraes, E Ursini, and Udo Fritzke Jr. Schedulability analysis of mode changes with arbitrary deadlines. *Real-time systems, architecture, scheduling, and application*, 2012.

[62] Joel Matějka, Björn Forsberg, Michal Sojka, Zdeněk Hanzálek, Luca Benini, and Andrea Marongiu. Combining prem compilation and ilp scheduling for high-performance and predictable mpsoc execution. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 11–20, 2018.

[63] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. Memory-processor co-scheduling in fixed priority systems. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pages 87–96, 2015.

[64] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[65] Vincent Nelis, Bjorn Andersson, José Marinho, and Stefan M Petters. Global-edf scheduling of multimode real-time systems considering mode independent tasks. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 205–214. IEEE, 2011.

[66] Vincent Nelis, Joel Goossens, and Björn Andersson. Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 151–160. IEEE, 2009.

[67] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete event dynamic systems*, 21(3):307–338, 2011.

[68] Claire Pagetti, Julien Forget, Heiko Falk, Dominic Oehlert, and Arno Luppold. Automated generation of time-predictable executables on multicore. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pages 104–113, 2018.

[69] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. The rosace case study: From simulink specification to multi/many-core execution. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 309–318. IEEE, 2014.

[70] Paulo Pedro and Alan Burns. Schedulability analysis for mode changes in flexible real-time systems. In *Proceeding. 10th EUROMICRO Workshop on Real-Time Systems (Cat. No. 98EX168)*, pages 172–179. IEEE, 1998.

[71] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279. IEEE, 2011.

[72] Rodolfo Pellizzoni and Marco Caccamo. Impact of peripheral-processor interference on wcet analysis of real-time embedded systems. *IEEE Transactions on Computers*, 59(3):400–415, 2010.

[73] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 741–746. IEEE, 2010.

[74] Quentin Perret, Pascal Maurère, Éric Noulard, Claire Pagetti, Pascal Sainrat, and Benoît Triquet. Mapping hard real-time applications on many-core processors. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 235–244, 2016.

[75] Linh TX Phan, Samarjit Chakraborty, and PS Thiagarajan. A multi-mode real-time calculus. In *2008 Real-Time Systems Symposium*, pages 59–69. IEEE, 2008.

[76] Linh TX Phan, Insup Lee, and Oleg Sokolsky. Compositional analysis of multi-mode systems. In *2010 22nd Euromicro Conference on Real-Time Systems*, pages 197–206. IEEE, 2010.

[77] Linh TX Phan, Insup Lee, and Oleg Sokolsky. A semantic framework for mode change protocols. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 91–100. IEEE, 2011.

[78] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[79] Mojzesz Presburger. Über die vollstandigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchen die addition als einzige operation hervortritt. In *Comptes-Rendus du 1er Congres des Mathematiciens des Pays Slaves*, 1929.

[80] Wolfgang Puffitsch, Eric Noulard, and Claire Pagetti. Off-line mapping of multi-rate dependent task sets to many-core platforms. *Real-Time Systems*, 51(5):526–565, 2015.

[81] J Real. *Protocolos de cambio de modo para sistemas de tiempo real (mode change protocols for real time systems)*. PhD thesis, Ph. D. thesis, Universidad Politécnica de Valencia, 2000. In spanish, 2000.

[82] Jorge Real and Alfons Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-time systems*, 26(2):161–197, 2004.

[83] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. *SIGPLAN Not.*, 43(6):159–169, June 2008.

[84] Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–20, 2017.

[85] Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, and Marco Caccamo. Worst-case response time analysis of resource access models in multi-core systems. In *Design Automation Conference*, pages 332–337. IEEE, 2010.

[86] Lui Sha, Ragunathan Rajkumar, John Lehoczky, and Krithi Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1(3):243–264, 1989.

[87] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

[88] Irina M Smarandache, Thierry Gautier, and Paul Le Guernic. Validation of mixed signal-alpha real-time systems through affine calculus on clock synchronisation constraints. In *International Symposium on Formal Methods*, pages 1364–1383. Springer, 1999.

[89] Steven Smyth, Christian Motika, Karsten Rathlev, Reinhard Von Hanxleden, and Michael Mendler. Scest: sequentially constructive esterel. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):1–26, 2017.

[90] Muhammad R Soliman, Giovani Gracioli, Rohan Tabish, Rodolfo Pellizzoni, and Marco Caccamo. Segment streaming for the three-phase execution model: Design and implementation. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 260–273. IEEE, 2019.

[91] Muhammad Refaat Soliman and Rodolfo Pellizzoni. Wcet-driven dynamic data scratchpad management with compiler-directed prefetching. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[92] Nikolay Stoimenov, Simon Perathoner, and Lothar Thiele. Reliable mode changes in real-time systems with fixed priority or edf scheduling. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 99–104. IEEE, 2009.

[93] Rohan Tabish, Renato Mancuso, Saud Wasly, Ahmed Alhammad, Sujit S Phatak, Rodolfo Pellizzoni, and Marco Caccamo. A real-time scratchpad-centric os for multi-core embedded systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11. IEEE, 2016.

[94] Rohan Tabish, Renato Mancuso, Saud Wasly, Rodolfo Pellizzoni, and Marco Caccamo. A real-time scratchpad-centric os with predictable inter/intra-core communication for multi-core embedded systems. *Real-Time Systems*, 55(4):850–888, 2019.

[95] Rohan Tabish, Renato Mancuso, Saud Wasly, Sujit S Phatak, Rodolfo Pellizzoni, and Marco Caccamo. A reliable and predictable scratchpad-centric os for multi-core embedded systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 377–388. IEEE, 2017.

[96] Jean-Pierre Talpin, Christian Brunette, Thierry Gautier, and Abdoulaye Gamatié. Polychronous mode automata. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 83–92, 2006.

[97] Jean-Pierre Talpin, Pierre Jouvelot, and Sandeep Kumar Shukla. Towards refinement types for time-dependent data-flow networks. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 36–41. IEEE, 2015.

[98] Ken Tindell and Alejandro Alonso. A very simple protocol for mode changes in priority preemptive systems. Technical report, Technical report, Universidad Politécnica de Madrid, 1996.

[99] Ken Tindell, Alan Burns, and Andy J Wellings. Mode changes in priority pre-emptively scheduled systems. In *RTSS*, volume 92, pages 100–109. Citeseer, 1992.

[100] Marisol Garcia Valls, Alejandro Alonso, and A Juan. Mode change protocols for predictable contract-based resource management in embedded multimedia systems. In *2009 International Conference on Embedded Software and Systems*, pages 221–230. IEEE, 2009.

[101] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 269–282, 2014.

[102] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 239–243. IEEE, 2007.

[103] Michael Von der Beeck. A comparison of statecharts variants. In *Formal techniques in real-time and fault-tolerant systems*, pages 128–148. Springer, 1994.

[104] Reinhard Von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. Sccharts: sequentially constructive statecharts for safety-critical applications: Hw/sw-synthesis for a conservative extension of synchronous statecharts. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 372–383, 2014.

[105] Saud Wasly and Rodolfo Pellizzoni. A dynamic scratchpad memory unit for predictable real-time embedded systems. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 183–192. IEEE, 2013.

[106] Saud Wasly and Rodolfo Pellizzoni. Hiding memory latency using fixed priority scheduling. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 75–86. IEEE, 2014.

[107] Rémy Wyss, Frédéric Boniol, Julien Forget, and Claire Pagetti. A synchronous language with partial delay specification for real-time systems programming. In *Asian Symposium on Programming Languages and Systems*, pages 223–238. Springer, 2012.

[108] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48(6):681–715, 2012.

[109] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Heechul Yun, and Marco Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE Transactions on Computers*, 65(9):2739–2751, 2015.

[110] Eugene Yip, Alain Girault, Partha S Roop, and Morteza Biglari-Abhari. The forec synchronous deterministic parallel programming language for multicores. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pages 297–304. IEEE, 2016.