THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par **l'Institut Supérieur de l'Aéronautique et de l'Espace**
Spécialité : Informatique

Présentée et soutenue par **Julien FORGET**
le 19 novembre 2009

Un langage synchrone pour les systèmes embarqués critiques soumis à des contraintes temps réel multiples

---

JURY

M. Yves Sorel, président
M. Mario Aldea Rivas
M. Frédéric Boniol, directeur de thèse
M. Alain Girault, rapporteur
M. Paul Le Guernic
M. Nicolas Navet
Mme Claire Pagetti, co-directrice de thèse
M. Yvon Trinquet, rapporteur

École doctorale : **Mathématiques, informatique et télécommunications de Toulouse**

Unité de recherche : **Équipe d'accueil ISAE-ONERA MOIS**

Directeur de thèse : **M. Frédéric Boniol**
Co-directrice de thèse : **Mme Claire Pagetti**

Remerciements

Je tiens tout d'abord à remercier mon cher trio d'encadrants : Frédéric BONIOL pour sa sagesse, sa culture du domaine et pour son optimisme inébranlable, Claire PAGETTI pour sa rigueur, son investissement, sa tenacité et pour avoir été une co-bureau fort sympathique, David LESENS pour m'avoir transmis une part de sa connaissance de son milieu industriel, pour m'avoir réorienté quand nécessaire et pour sa constante modestie.

Merci à Alain GIRAULT et Yvon TRINQUET pour avoir accepté de rapporter cette thèse, à Yves SOREL pour avoir présidé mon jury et pour m'avoir mis sur la voie il y a sept ans, à Mario ALDEA RIVAS, Paul LE GUERNIC et Nicolas NAVET pour avoir accepté de faire partie de mon jury.

Un grand merci à mes anciens collègues, les éternels Nicolas, Arnaud, Cyril et Liliana, les experts Bruno et Jean-Louis, et bien sûr tous les chaleureux membres du DTIM. Ils ont supporté avec patience toutes mes questions, mes bavardages et sont tout simplement devenus des amis. Merci aussi à mes compagnons de tablées qui ont agréablement animé ces trois années.

Mes pensées affectueuses à ma deuxième famille croquante, soutien infaillible et inspiration, quel que soit l'éloignement géographique : Anne-Lise, Etienne, Ludovic, Nicolas, Sophie, Stève & Cécile et aussi le Will'Will.

Merci aux collocs de m'avoir fait une place à mi-temps dans leur maison et de m'avoir attendu à la fin des repas.

Je n'aurais rien fait de tout cela sans ma mère, mon père, mon frère et aussi mes neuveus, ma farandole adorée d'oncles, tantes, cousins et grands-parents.

Et bien sûr, surtout, à toi, Stéphanie, pour avoir donné tellement plus de saveur à tout ceci.

Contents

List of Figures **xi**

Résumé **xv**

Résumé étendu **xvii**

1 Etat de l'art xx
1.1 Langages de programmation bas-niveau xx
1.2 Les langages synchrones xxi
1.3 Matlab/Simulink xxvi
1.4 Les langages de description d'architecture (ADL) xxvi
1.5 Motivation de notre travail xxix
2 Horloges périodiques temps réel xxix
2.1 Un modèle reliant instants et temps réel xxix
2.2 Horloges Strictement Périodiques xxx
2.3 Horloges strictement périodiques et horloges booléennes xxxii
3 Présentation informelle du langage xxxiii
3.1 Typage polymorphe xxxiii
3.2 Primitives temps réel xxxiii
3.3 Retards xxxvii
3.4 Conditions d'activation xxxviii
4 Calcul d'horloges xxxix
4.1 Types d'horloges xxxix
4.2 Inférence d'horloges xl
4.3 Unification d'horloges xlii
4.4 Subsomption xliv
5 Mode de compilation xliv
5.1 Limitation de la génération de code classique en « boucle simple » xliv
5.2 Génération de code multi-tâche xlv
6 Extraction de tâches dépendantes xlvi

6.1 Expansion de programme xlvi
6.2 Graphe de tâches xlvi
6.3 Caractéristiques temps réel xlix
6.4 Conditions d'activation xlix
7 Traduction de tâches dépendantes en tâches indépendantes l
7.1 Encodage des précédences l
7.2 Analyse d'ordonnançabilité lvi
7.3 Communications inter-tâches lvi
8 Prototype lx
9 Conclusion lx

Introduction **1**

Chapter 1 State of the Art **5**
1.1 Low-level programming 5
1.1.1 Real-time API 5
1.1.2 Real-time Operating Systems 6
1.1.3 Towards a higher level of abstraction 6
1.2 Synchronous Languages 7
1.2.1 LUSTRE 7
1.2.2 SIGNAL 12
1.2.3 ESTEREL 13
1.2.4 SYNDEX and the AAA methodology 15
1.2.5 Synchronous Data-Flow (SDF) 16
1.2.6 Globally Asynchronous Locally Synchronous (GALS) Systems 18
1.3 Matlab/Simulink 18
1.4 Architecture description languages 19
1.4.1 AADL 19
1.4.2 REACT/CLARA 21
1.4.3 UML and CCSL 22
1.4.4 GIOTTO 23
1.5 Motivation of our work 24
1.6 Outline of the thesis 25

Part I Language Definition **27**

Chapter 2 The Synchronous Approach **29**

2.1 Logical time and instants 29
2.2 Verifying the synchronous hypothesis 29
2.3 Overview of synchronous languages 30
2.4 Data-flow synchronous languages, an example: LUSTRE 30
2.4.1 Basic operators 31
2.4.2 Clock operators 31
2.4.3 Nodes 31
2.4.4 Automated code generation 32

Chapter 3 Real-Time Periodic Clocks **35**
3.1 A model relating instants to real-time 35
3.1.1 Multiple logical time scales 35
3.1.2 Real-time flows 35
3.2 Strictly Periodic Clocks 36
3.2.1 Definition 36
3.2.2 Periodic clock transformations 36
3.2.3 Integer Strictly Periodic Clocks 37
3.3 Strictly periodic clocks and Boolean clocks 38
3.4 Summary 38

Chapter 4 Language Syntax and Semantics **39**
4.1 Types polymorphism 39
4.2 Real-time Primitives 40
4.2.1 Declaring real-time constraints 40
4.2.2 Rate transition operators 41
4.2.3 Phase offset operators 41
4.2.4 Rate transitions on unspecified rates 43
4.2.5 Sensors and actuators 43
4.3 Delays 44
4.4 Activation conditions 44
4.4.1 Boolean sampling operators 44
4.4.2 Combining Boolean and strictly periodic clock flow operators 44
4.5 Syntax 45
4.6 Synchronous Temporised Semantics 46
4.6.1 Kahn's semantics 46
4.6.2 Semantics of classic synchronous operators 46
4.6.3 Semantics of strictly periodic clock operators 47

4.6.4 Denotational semantics . . . 48
4.7 Summary . . . 49

Chapter 5 Multi-periodic Communication Patterns **51**
5.1 Sampling . . . 51
5.2 Queuing . . . 52
5.3 Mean value . . . 54
5.4 Non-harmonic periods . . . 55
5.5 A complete example . . . 55
5.6 Summary . . . 56

Chapter 6 Static Analyses **59**
6.1 Causality Analysis . . . 59
6.2 Typing . . . 60
6.2.1 Types . . . 60
6.2.2 Types inference . . . 60
6.2.3 Types unification . . . 62
6.2.4 Examples . . . 63
6.3 Clock Calculus . . . 64
6.3.1 Clock Types . . . 64
6.3.2 Clock inference . . . 65
6.3.3 Clocks unification . . . 68
6.3.4 Subsumption . . . 72
6.3.5 Example . . . 73
6.3.6 Correctness . . . 75
6.4 About Flows Initialisation . . . 78
6.5 Summary . . . 78

Chapter 7 Conclusion on language definition **79**

Part II Compiling into a Set of Real-Time Tasks **81**

Chapter 8 Overview of the Compilation Process **83**
8.1 Limitation of the classic "single-loop" code generation . . . 83
8.2 Multi-task code generation . . . 84
8.3 Compilation chain . . . 84

Chapter 9 Extracting Dependent Real-Time Tasks **87**
9.1 Program Expansion 87
9.2 Task Graph 88
9.2.1 Definition 88
9.2.2 Tasks 88
9.2.3 Precedences 89
9.2.4 Intermediate graph 89
9.2.5 Graph reduction 91
9.2.6 Task clustering 94
9.3 Real-time characteristics 94
9.4 Activation conditions 94
9.5 Summary 95

Chapter 10 From Dependent to Independent Tasks **97**
10.1 Scheduling Dependent Tasks 97
10.1.1 With/without preemption 97
10.1.2 Dynamic/static priority 98
10.1.3 Processing precedences before run-time/at run-time 99
10.2 Encoding Task Precedences 99
10.2.1 Adjusting real-time characteristics 99
10.2.2 Definition of task instance precedences 101
10.2.3 Release dates adjustment in the synchronous context 103
10.2.4 Deadline calculus 109
10.2.5 Optimality 116
10.2.6 Complexity 116
10.3 Schedulability analysis 117
10.4 Inter-Task Communications 118
10.4.1 Ensuring communication consistency 118
10.4.2 Task instances communications 119
10.4.3 Communication protocol 121
10.4.4 Example 124
10.5 Summary 125

Chapter 11 Code generation **127**
11.1 Communication buffers 127
11.2 Task functions 128
11.3 Task real-time attributes 130

11.4 Threads creation 131
11.5 Summary 131

Chapter 12 Prototype implementation 133
12.1 The compiler 133
12.2 Earliest-Deadline-First scheduler with deadline words 134
12.2.1 Application-defined schedulers in MARTE OS 134
12.2.2 EDF implementation 134
12.2.3 Deadline words support 135
12.3 Summary 135

Chapter 13 Conclusion on language compilation 137

Part III Case Study 139

Chapter 14 The Flight Application Software: Current Implementation 141
14.1 Hardware architecture 141
14.2 Software architecture 142
14.3 Bus communications 142
14.4 Summary 143

Chapter 15 Case study Implementation 145
15.1 System specification 145
15.2 Implementation overview 146
15.3 Specific requirements 147
15.3.1 Slow-to-fast communications 147
15.3.2 Producing outputs at exact dates 148
15.3.3 Output with a phase smaller than the related input 148
15.3.4 End-to-end latency longer than the period 148
15.4 Results 149
15.5 Summary 149

Conclusion 151
1 Summary 151
2 Limitations and perspectives 152
2.1 Task clustering 152
2.2 Static priority scheduling 152
2.3 Modes 152

Bibliography **155**

Appendix A Code for the FAS Case Study **163**

List of Figures

1 Le Flight Application Software . . . xix
2 Un programme LUSTRE . . . xxii
3 Comportement d'un programme LUSTRE . . . xxii
4 Rythme de base dans le cas multi-périodique . . . xxiii
5 Un programme ESTEREL . . . xxiv
6 Comportement d'un programme ESTEREL . . . xxiv
7 Un graphe SDF . . . xxv
8 Motifs de communication en AADL, pour une connexion de A vers B . . . xxvii
9 Une boucle de communication multi-rythme en CLARA . . . xxviii
10 Horloges strictement périodiques. . . . xxxi
11 Sous-échantillonnage périodique . . . xxxv
12 Sur-échantillonnage périodique . . . xxxv
13 Décalage de phase . . . xxxv
14 Eliminer la tête d'un flot . . . xxxvi
15 Concaténation de flots . . . xxxvi
16 Retarder un flot . . . xxxvii
17 Opérateurs d'horloges booléens . . . xxxviii
18 Application de transformations d'horloges booléennes à des horloges strictement périodiques . . . xxxviii
19 Règles d'inférence d'horloge . . . xli
20 Règles d'unification d'horloge . . . xliii
21 Exécution séquentielle du nœud `multi` . . . xlv
22 Caractéristiques temps réel de τ_i . . . xlv
23 Ordonnancement préemptif du nœud `multi` . . . xlv
24 Dépendances entre expressions . . . xlviii
25 Extraction d'un graphe de tâche intermédiaire . . . xlviii
26 Un graphe intermédiaire . . . xlviii
27 Graphe après réduction des variables . . . xlix
28 Un graphe de tâches réduit . . . xlix
29 Encodage des précédences étendues . . . li
30 Précédences entre instances de tâches, selon la définition de g_{ops} . . . lii
31 Illustration de Δ_{ops} . . . liii
32 Communications pour des précédences étendues . . . lvi
33 Dépendances de données . . . lx

1 The Flight Application Software . . . 3

1.1 Behaviour of a simple LUSTRE program . . . 8
1.2 Simplified communication loop between operations `FDIR` and `GNC US` . . . 8
1.3 Auxiliary LUSTRE nodes . . . 9
1.4 Behaviour of auxiliary nodes . . . 9
1.5 Programming a multi-rate communication loop in LUSTRE, with a fast base rate (10Hz) . 9
1.6 Programming a multi-rate communication loop in LUSTRE, with a fast base rate (10Hz), splitting the slow operation . . . 10
1.7 Programming a multi-ate communication loop in LUSTRE, with a slow base rate (1Hz) . 11
1.8 Programming a multi-rate communication loop in LUSTRE, with a fast base rate, using real-time extensions . . . 12
1.9 Behaviour of a simple ESTEREL program . . . 14
1.10 Programming a multi-rate communication loop in ESTEREL . . . 14
1.11 Programming a multi-rate communication loop in ESTEREL/TAXYS . . . 15
1.12 Programming a multi-rate communication loop in SYNDEX, with a slow base rate . . . 17
1.13 A SDF graph . . . 17
1.14 SDF graph for a multi-rate communication loop . . . 18
1.15 Communications schemes in AADL, for a connection from A to B . . . 20
1.16 A multi-rate communication loop in CLARA . . . 22
1.17 Communications in GIOTTO . . . 23
1.18 Programming a multi-rate communication loop in GIOTTO . . . 24

2.1 Behaviour of clock operators . . . 31
2.2 Nesting clock operators . . . 32

3.1 Strictly periodic clocks. . . . 37

4.1 Periodic under-sampling . . . 41
4.2 Periodic over-sampling . . . 41
4.3 Phase offset . . . 42
4.4 Keeping the tail of a flow . . . 42
4.5 Flow concatenation . . . 43
4.6 Delaying a flow . . . 44
4.7 Boolean clock operators . . . 45
4.8 Applying Boolean clock transformations to strictly periodic clocks . . . 45
4.9 Synchronous semantics of classical operators . . . 47
4.10 Synchronous temporised semantics of operators on strictly periodic clocks . . . 47
4.11 Denotational semantics . . . 48

5.1 A simple example to illustrate the different communication patterns . . . 51
5.2 Data sampling: choosing the place of the delay . . . 52
5.3 Data sampling, starting the sample on the second value . . . 53
5.4 Storing successive values of a flow ($n = 3$, $init = 0$) . . . 53
5.5 Computing the mean value of 3 successive “fast” values, (with $init = 0$) . . . 54
5.6 Flows of non-harmonic periods . . . 55
5.7 Programming the FAS with data-sampling . . . 56
5.8 Programming the FAS with data-queuing . . . 57

6.1 Type inference rules . . . 61
6.2 Types unification rules . . . 62

6.3 Clock inference rules . . . 66
6.4 Clock unification rules . . . 72

8.1 Sequential execution of node `multi` . . . 83
8.2 Real-time characteristics of task τ_i . . . 84
8.3 Preemptive scheduling of node `multi` . . . 84
8.4 Compilation chain . . . 85

9.1 A task graph . . . 89
9.2 Expression dependencies . . . 90
9.3 Extraction of the intermediate task graph . . . 90
9.4 An intermediate graph . . . 91
9.5 Graph after variable reduction . . . 91
9.6 Application of a graph transformation rule . . . 92
9.7 The graph transformation system . . . 93
9.8 A reduced task graph . . . 93

10.1 Encoding extended precedences . . . 100
10.2 Task instance precedences, as defined by g_{ops} . . . 103
10.3 Illustration of Δ_{ops} . . . 105
10.4 Scheduling the program requires the instances of A to have different deadlines . . . 110
10.5 Communications for extended precedences, with $prio(A) > prio(C) > prio(B)$ . . . 118
10.6 Data-dependencies between task instances . . . 119
10.7 Data-dependencies . . . 125

11.1 Reduced task graph . . . 128

12.1 The different parts of the prototype . . . 133

14.1 Hardware architecture of the ATV (FAS+MSU) . . . 142

Résumé

Ce travail porte sur la programmation de systèmes de Contrôle-Commande. Ces systèmes sont constitués d'une boucle de contrôle qui acquiert l'état actuel du système par l'intermédiaire de capteurs, exécute des algorithmes de contrôle à partir de ces données et calcule en réaction les commandes à appliquer sur les actionneurs du système dans le but de réguler son état et d'accomplir une mission donnée. Les logiciels de commandes de vol d'un aéronef sont des exemples typiques de systèmes de Contrôle-Commande ayant pour objectif de contrôler la position, la vitesse et l'attitude de l'aéronef durant le vol. Les systèmes considérés sont critiques, dans le sens où leur mauvais fonctionnement peut avoir des conséquences catastrophiques. Leur implantation doit donc être déterministe, non seulement sur le plan fonctionnel (produire les bonnes sorties en réponse aux entrées) mais aussi sur le plan temporel (produire les données aux bonnes dates).

Nous avons défini un langage formel et son compilateur permettant de programmer de tels systèmes. Le langage se situe à un niveau d'abstraction élevé et se présente comme un langage d'architecture logicielle temps réel. Il permet de spécifier avec une sémantique synchrone l'assemblage des composants fonctionnels d'un système multi-rythme. Un programme consiste en un ensemble d'opérations importées, implantées à l'extérieur du programme, reliées par des dépendances de données. Le langage permet de spécifier de multiples contraintes de périodicité ou d'échéance sur les opérations. Il définit de plus un ensemble réduit d'opérateurs de transition de rythme permettant de décrire de manière précise et non ambiguë des schémas de communication entre opérations de périodes différentes.

La sémantique du langage est définie formellement et la correction d'un programme, c'est-à-dire l'assurance que sa sémantique est bien définie, est vérifiée par un ensemble d'analyses statiques. Un programme correct est compilé en un ensemble de tâches temps réel implantées sous forme de threads C communicants. Le code d'un thread est constitué du code (externe) de l'opération correspondante, complété par un protocole de communication gérant les échanges de données avec les autres threads. Le protocole, basé sur des mécanismes de copies dans des mémoires tampons, est non bloquant et déterministe.

Le code généré s'exécute à l'aide d'un Système d'Exploitation Temps Réel classique disposant de la politique d'ordonnancement EDF. Les opérations du programme s'exécutent de manière concurrente et peuvent donc être préemptées en cours d'exécution en faveur d'une opération plus urgente. Le protocole de communication proposé permet d'assurer que, malgré les préemptions, l'exécution du programme généré est prédictible et correspond exactement à la sémantique formelle du programme d'origine.

Résumé étendu

Introduction

Le travail présenté dans ce mémoire est financé par EADS Astrium Space Transportation.

Systèmes réactifs

Un *système réactif* [HP85] est un type particulier de système embarqué qui maintient une interaction permanente avec son environnement physique. Il attend des entrées (stimuli), effectue des calculs et produits des sorties (commandes) en réaction aux entrées. Les systèmes de contrôle ou les systèmes de surveillance sont de bons exemples de systèmes réactifs.

Une caractéristique particulière des systèmes réactifs est que le temps de réponse du système doit respecter des contraintes temps réel induites par leur environnement. Si la réponse arrive trop tard, le système peut rater des entrées en provenance de son environnement et répondre à des entrées ne correspondant pas à l'état actuel de l'environnement. Par ailleurs, les systèmes réactifs sont souvent hautement critiques (par exemple, le système de pilotage d'un avion ou le système de contrôle d'une centrale atomique) et les erreurs ne peuvent donc pas être tolérées.

Systèmes événementiels contre systèmes échantillonnés

On distingue généralement deux classes de systèmes réactifs : les *systèmes événementiels* et les *systèmes échantillonnés*. La différence réside dans la manière dont leurs entrées sont acquises. Dans le cas d'un système événementiel (parfois qualifié de *système purement réactif*), le système attend l'arrivée d'un événement d'entrée et ensuite calcule sa réaction. A l'inverse, un système échantillonné acquiert ses entrées à intervalles de temps réguliers et calcule sa réaction pour chaque échantillon de ses entrées (et n'effectue éventuellement qu'une partie des calculs si seulement une partie des entrées est disponible pour l'échantillon courant). Bien que les systèmes événementiels correspondent à une plus grande classe de systèmes, les systèmes échantillonnés sont généralement plus simples à analyser et leur comportement est plus prédictible, en particulier en ce qui concerne les aspects temps réel. Dans la suite, nous nous consacrons aux systèmes échantillonnés.

Systèmes de contrôle-commande

Notre travail est plus particulièrement destiné aux *systèmes de contrôle-commande*. Ces systèmes consistent en une boucle de contrôle incluant capteurs, algorithmes de contrôle et actionneurs, ayant pour rôle de réguler l'état d'un système dans son environnement et d'accomplir une mission donnée. Les logiciels de commandes de vol d'un aéronef sont des exemples typiques de systèmes de contrôle-commande ayant pour objectif de contrôler la position, la vitesse et l'attitude de l'aéronef durant le vol à l'aide d'équipements physiques tels que les surfaces de contrôle pour un avion ou les propulseurs pour

un véhicule spatial. Le système opère en boucle fermée avec son environnement car les commandes appliquées sur le véhicule modifient son état (position, vitesse, attitude), ce qui affecte ensuite les entrées du système et nécessite de calculer de nouvelles commandes et ainsi de suite.

Les calculs effectués par ces systèmes sont très réguliers : par exemple, il n'y pas de création dynamique de processus et le nombre d'itérations des instructions de boucles est borné statiquement. Ceci permet de produire des implantations optimisées sur le plan logiciel et matériel dont le comportement est prédictible, une caractéristique essentielle pour des systèmes hautement critiques.

Les systèmes de contrôle-commande doivent respecter un certain nombre de contraintes temps réel liées à des contraintes physiques. Tout d'abord, le rythme auquel un équipement doit être contrôlé (et donc le rythme auquel les opérations exécutées pour contrôler cet équipement doivent s'effectuer) doit être supérieur à une fréquence minimale, liée à l'inertie et autres caractéristiques physiques du véhicule. Au-dessous de cette fréquence, la sécurité du véhicule n'est plus assurée. Ces fréquences minimales varient entre les équipements. Par exemple, les dispositifs de propulsion d'un véhicule spatial doivent être contrôlés à un rythme très élevé pour assurer la stabilité du véhicule, tandis que la position des panneaux solaires peut être contrôlée légèrement plus lentement car une petite perte d'énergie a un impact moins important sur le véhicule. Ensuite, le rythme de contrôle d'un équipement doit aussi être inférieur à une fréquence maximale au-dessus de laquelle l'équipement n'est pas capable d'appliquer les commandes suffisamment rapidement ou peut être endommagé. Ce paramètre varie lui aussi d'un équipement à l'autre. Le rythme de contrôle d'un équipement est généralement choisi le plus proche possible de la fréquence minimale, afin d'éviter les calculs inutiles et ainsi permettre d'utiliser un matériel moins puissant (qui sera donc moins coûteux, consommera moins d'énergie, ...)

Comme pour tout système réactif, le temps de réponse d'un système de contrôle-commande est borné. Le délai entre une observation (entrées) et la réaction correspondante (sorties) doit respecter une contrainte de latence maximale, c'est-à-dire une contrainte d'échéance relative à la date des observations. L'échéance d'une sortie n'est pas nécessairement égale à sa période et peut lui être inférieure afin de répondre rapidement à une entrée.

Cas d'étude : The Flight Application Software

Dans cette section, nous présentons le *Flight Application Software* (FAS), système de contrôle de vol du *Véhicule de Transfert Automatique* (ATV), un véhicule spatial réalisé par EADS Astrium Space Transportation pour le ravitaillement de la *Station Spatiale Internationale* (ISS). C'est un exemple typique de système de contrôle-commande. Nous présentons une version simplifiée et adaptée du FAS dans la Fig. 1. Cette version ne correspond pas exactement au système réel mais donne une bonne vision globale de ses caractéristiques principales.

Le FAS consiste principalement en un ensemble d'opérations d'acquisition, d'opérations de calcul et d'opérations de commande, chacune schématisée par une boîte dans la figure. Les flèches représentent des dépendances de données. Une flèche sans source correspond à une entrée du système et une flèche sans destination à une sortie du système. Les dépendances de données définissent un ordre partiel entre les opérations du systèmes, le consommateur d'une donnée devant s'exécuter après le producteur de la donnée. Le FAS commence par acquérir les entrées du système : l'orientation et la vitesse à partir des capteurs gyroscopiques (`GyroAcq`), la position à partir du GPS (`GPS Acq`) et du star tracker (`Str Acq`), et les télécommandes en provenance de la station sol (`TM/TC`). La fonction *Guidance Navigation and Control* (divisée en une partie flot montant, `GNC_US`, et flot descendant, `GNC_DS`) calcule ensuite les commandes à appliquer tandis que la fonction *Failure Detection Isolation and Recovery* (`FDIR`) vérifie l'état du FAS afin d'assurer l'absence de pannes. Pour finir, les commandes sont calculées et envoyées aux équipements de contrôle : les commandes propulseurs pour le *Propulsion Drive Electronics* (`PDE`), les commandes de distribution d'énergie pour le *Solar Generation System* (`SGS`), les commandes de po-

sitionnement des panneaux solaires pour le *Solar Generation System*, les télémétries à destination de la station sol (`TM/TC`), etc.

Chaque opération s'effectue à son propre rythme, situé entre 0.1Hz et 10Hz. On impose une contrainte d'échéance sur la donnée produite par le GNC Upstream (300ms au lieu de la période de 1s). Des opérations de rythmes différents peuvent communiquer, ce qui est une caractéristique essentielle de ce type de système et ce qui a un impact important sur la complexité des processus de spécification et d'implantation.

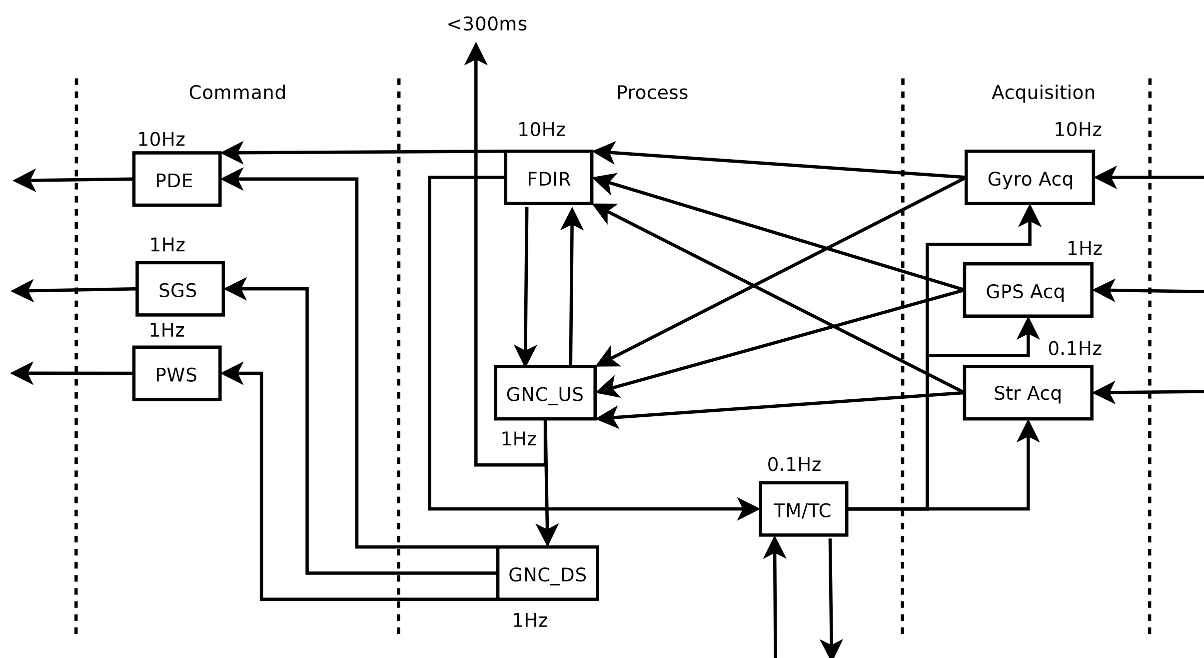


FIG. 1 – Le Flight Application Software

Contribution

La programmation d'un système de contrôle-commande est un processus complexe qui ne se résume pas à implanter les aspects fonctionnels d'un système. Ces systèmes étant hautement critiques, une exigence fondamentale sur le programme contrôlant le système est d'assurer que son comportement est prédictible. Ceci requiert un fort déterminisme fonctionnel, c'est-à-dire que le programme produise toujours la même séquence de sorties en réponse à une séquence donnée d'entrées. Le programme doit aussi être déterministe sur le plan temporel et respecter des contraintes temps réel dures. Le système doit de plus être optimisé, en termes de consommation mémoire, coût matériel, consommation énergétique ou poids par exemple.

La complexité du processus de développement et la criticité des systèmes motivent l'utilisation de langages de programmation haut-niveau, couvrant complètement le cycle de développement, de la spécification à l'implémentation. La génération de code automatique, à partir d'un programme haut-niveau vers un programme bas-niveau, constitue une bonne solution à ce problème, car elle procure une grande confiance dans le programme final du fait que chaque étape de la génération est définie formellement.

Dans cette thèse, nous proposons un tel langage haut-niveau pour la programmation de systèmes de contrôle-commande. Le langage est basé sur les langages synchrones [BB01] et hérite de leurs propriétés formelles. Il ajoute des primitives temps réel afin de permettre la spécification de systèmes multi-

périodiques. L'objectif du langage n'est pas de remplacer les langages synchrones existants, mais plutôt de fournir une couche d'abstraction supplémentaire. Il peut être considéré comme un langage de description d'architecture logicielle temps réel permettant d'assembler plusieurs systèmes synchrones localement mono-périodiques en un système global multi-périodique. Le langage est supporté par un compilateur générant un code C multi-tâches synchronisé. Le code généré s'exécute sur un système d'exploitation temps réel classique disposant d'une politique d'ordonnancement préemptive. Les synchronisations entre tâches sont assurées par un protocole de communication à base de mémoires tampons qui ne nécessite pas de primitives de synchronisation (telles que les sémaphores). Nous allons maintenant motiver la définition de notre langage en montrant que l'état de l'art du domaine ne traite pas complètement le problème exposé ci-dessus. Nous situons aussi notre langage par rapport à cet existant.

1 Etat de l'art

Dans ce chapitre, nous présentons un résumé des langages existant concernant la programmation de systèmes de contrôle-commande. Les caractéristiques souhaitables d'un tel langage sont :

- La possibilité de spécifier un ensemble d'opérations ;
- La possibilité de spécifier les échanges de données entre opérations et en particulier entre opérations de rythmes différents ;
- La possibilité de spécifier des contraintes temps réel, telles que des contraintes de périodicité ou d'échéance ;
- Une définition formelle du langage, afin d'assurer que la sémantique d'un programme est définie de manière non ambiguë ;
- Une génération de code automatique, afin de couvrir le processus de développement de la spécification jusqu'à l'implémentation ;
- Le déterminisme fonctionnel du code généré ;
- Le déterminisme temporel du code généré ;
- Pour finir : la simplicité, caractéristique essentielle permettant au programmeur de mieux comprendre et analyser un programme.

1.1 Langages de programmation bas-niveau

La technique probablement la plus répandue pour programmer un système temps réel est de le programmer à l'aide des langages impératifs traditionnels (C, ADA, JAVA). Ces langages n'étant à l'origine pas destinés à la programmation de systèmes temps réel, les aspects temps réel sont principalement gérés à l'aide d'interfaces de programmation dédiées (API), très proches du système d'exploitation temps réel (RTOS) sous-jacent.

API temps réel

Une API temps réel définit l'interface d'un ensemble de services (par exemple des fonctions ou des structures de données) liés à la programmation temps réel. L'objectif d'une telle API est de définir un standard permettant la portabilité d'une application entre différentes plates-formes d'exécution respectant ce standard. On peut par exemple citer les extensions temps réel de POSIX [POS93a, POS93b], pour les langages C et ADA, la Spécification Temps Réel pour Java (RTSJ), OSEK dans l'industrie automobile [OSE03] ou APEX dans le domaine avionique.

Ces API partagent les mêmes principes généraux : chaque opération du système est tout d'abord programmée dans une fonction séparée et les fonctions sont ensuite regroupées en threads sur lesquels

le programmeur peut spécifier des contraintes temps réel. Un programme est ensuite constitué d'un ensemble de threads ordonnancés de manière concurrente par le système d'exploitation. Les threads étant reliés par des dépendances de données, le programmeur doit ajouter des primitives de synchronisation assurant que le producteur de la donnée s'exécute avant le consommateur. Ces synchronisations sont essentielles dans le cas des systèmes critiques afin d'assurer la prédictibilité du système. Les synchronisations peuvent être implémentées soit en contrôlant précisément les dates d'exécution des différents threads (communications *time-triggered*), soit à l'aide de primitives de synchronisation bloquantes telles que les sémaphores (communications par *rendez-vous*).

Systèmes d'exploitation temps réel

Un système d'exploitation temps réel est un système d'exploitation multi-tâches destiné spécifiquement à l'exécution d'applications temps réel. Le système est capable d'exécuter plusieurs tâches de manière concurrente, alternant entre les différentes tâches. La (ou les) *politique d'ordonnancement* du système d'exploitation fournit un algorithme général permettant de déterminer l'ordre dans lequel les tâches doivent s'exécuter en fonction de leurs caractéristiques temps réel (période, échéance). L'ordonnancement est souvent *préemptif* (mais pas nécessairement), ce qui signifie qu'une tâche peut être suspendue en cours d'exécution, pour exécuter une tâche plus urgente, puis restaurée plus tard.

Un RTOS tente généralement de suivre une des APIs temps réel citées précédemment. On peut citer comme exemple VxWorks par Wind River Systems [VxW95], QNX Neutrino [Hil92] et OSE [OSE04] pour les systèmes commerciaux, RTLinux (`www.rtlinuxfree.com`) et RTAI (`www.rtai.org`) pour les systèmes open-source, ou encore SHARK [GAGB01], MaRTE [RH02] et ERIKA [GLA+00], des noyaux de recherche tentant de fournir un niveau d'abstraction plus élevé que les systèmes classiques.

Vers un plus haut niveau d'abstraction

Programmer des systèmes temps réel à l'aide de langages bas-niveau permet certes d'implémenter une grande classe de systèmes mais présente des inconvénients majeurs. Tout d'abord, dans le cas d'applications complexes, les mécanismes de synchronisation sont fastidieux à mettre en place et difficile à maîtriser. Plus généralement, le faible niveau d'abstraction rend la correction du programme difficile à prouver. Enfin, du fait de ce faible niveau d'abstraction, il est difficile de distinguer au sein d'un programme les éléments liés à de réelles exigences de conception de ceux liés à des soucis d'implémentation, ce qui rend la maintenance compliquée. Ces considérations ne peuvent bien entendue pas être évitées, cependant il est beaucoup plus simple de se reposer sur un langage de programmation avec un niveau d'abstraction plus élevé, pour lequel les problèmes de synchronisation et d'ordonnancement sont gérés par le compilateur du langage et non par le programmeur.

1.2 Les langages synchrones

Forts de ces considérations, les développeurs de systèmes de Contrôle-Commande s'orientent progressivement vers des langages avec un niveau d'abstraction plus élevé. Bien que ceux-ci soient pour l'instant principalement utilisés durant les premières phases du cycle de développement (spécification, vérification, simulation), plusieurs langages ont pour objectif de couvrir le cycle complet, de la spécification jusqu'à l'implémentation, à l'aide de processus de génération automatique de code. Une telle approche permet de réduire la durée du cycle de développement, d'écrire des programmes plus simples à comprendre et à analyser, et enfin de produire des programmes bas-niveau dont la correction est assurée par la compilation.

L'approche synchrone [BB01] propose un haut-niveau d'abstraction, basé sur des fondations mathématiques à la fois simples et solides, qui permettent de gérer la vérification et la compilation d'un

programme de manière formelle. L'approche synchrone simplifie la programmation des systèmes temps réel en faisant abstraction de ce temps réel pour le remplacer par un temps logique, défini comme une suite d'*instants*, où chaque instant représente l'exécution d'une réaction du système. Selon l'hypothèse synchrone, le développeur peut faire abstraction des dates auxquelles les calculs sont effectués durant un instant, à condition que ces calculs se terminent avant le début du prochain instant.

Cette approche a été implantée dans plusieurs langages, on distingue souvent les langages synchrones équationnels tels que LUSTRE [HCRP91] ou SIGNAL [BLGJ91], des langages synchrones impératifs tels que ESTEREL [BdS91].

Langages synchrones équationnels

Les variables et expressions manipulées par un langage synchrone équationnel correspondent à des séquences de valeurs infinies (appelées *flots* ou *signaux*) décrivant les valeurs que prennent ces variables ou expressions à chaque instant du programme. L'*horloge* d'une séquence définit les instants ou cette séquence porte une valeur (est *présente*) ou pas (est *absente*). Un programme consiste en un ensemble d'équations, structurées de manière hiérarchique (à l'aide de *nœuds* en LUSTRE ou *processus* en SIGNAL). Les équations définissent les séquences de sortie du programme en fonction de ses séquences d'entrées. Un exemple simple de programme LUSTRE est donné Fig. 2. Son comportement est illustré Fig. 3 où l'on donne la valeur de chaque flot à chaque instant. Les opérateurs arithmétiques et logiques sont simplement étendus point-à-point (`v1+1`). L'opérateur **`pre`** permet d'accéder à la valeur d'un flot à l'instant précédent. `x->y` vaut `x` au premier instant puis `y` aux instants suivants. L'expression `x` **`when`** `c` vaut `x` uniquement quand `c` vaut vrai et ne porte pas de valeur sinon. A l'inverse, **`current`**`(x)` permet de remplacer les valeurs absentes introduites par un opérateur **`when`** par la dernière valeur de `x`.

```
node N(c: bool; i:int) returns (o:int)
var v1: int; v2: int when c;
let
  v1=0->pre(i);
  v2=(v1+1) when (true->c);
  o=current(v2);
tel
```

FIG. 2 – Un programme LUSTRE

`i`	i_0	i_1	i_2	i_3	i_4	...
`c`	false	true	false	false	true	...
`pre``(i)`		i_0	i_1	i_2	i_3	...
`v1=0->`**`pre`**`(i)`	0	i_0	i_1	i_2	i_3	...
`v1+1`	1	i_0+1	i_1+1	i_2+1	i_3+1	...
`v2=(v1+1)` **`when`** `(true->c)`	1	i_0+1			i_3+1	...
`o=`**`current`**`(v2)`	1	i_0+1	i_0+1	i_0+1	i_3+1	...

FIG. 3 – Comportement d'un programme LUSTRE

Un programme synchrone équationnel décrit le comportement d'un système à chaque itération de base. Cependant, comment définir une itération de base dans le cas des systèmes multi-périodiques ? Dans la Fig. 4, on considère le cas d'un système constitué d'une opération rapide `F`, de rythme 10Hz, et d'une opération lente `S`, de rythme 2Hz. La durée d'une itération de base doit-elle être de 100ms

(10Hz) et comprendre une itération de l'opération rapide `F` ainsi qu'un fragment de l'opération lente `S` qui variera suivant les itérations, ou bien doit-elle être de 500ms (2Hz), et comprendre une itération complète de l'opération lente et 5 itérations de l'opération rapide ?

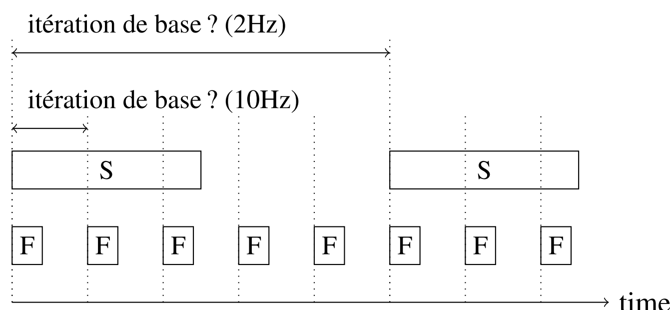


FIG. 4 – Rythme de base dans le cas multi-périodique

Ce type de problèmes a motivé l'introduction des aspects temps réel dans LUSTRE dans [Cur05]. Des *hypothèses* permettent de spécifier le rythme de base d'un programme (la durée temps réel d'un instant) ainsi que les durées d'exécution des nœuds. Des *exigences* permettent de contraindre la date à laquelle un flot doit être produit ainsi que la latence maximale entre deux flots. Enfin, ces extensions permettent de définir des horloges périodiques, qui valent vrai une fois toutes les n itérations de base. Le compilateur assure ensuite à l'aide de techniques d'ordonnancement statiques que les exigences seront satisfaites à condition que les hypothèses soient respectées. L'ordonnancement est très spécifique à la plateforme d'exécution considérée, TTA (Time-Triggered Architecture) [KB03].

Ce travail a été étendu dans [ACGR09], où sont proposés des opérateurs spécifiant des conditions d'activation périodiques et des primitives permettant de gérer plusieurs modes d'exécution. Les auteurs proposent aussi un processus de compilation plus simple et plus générique générant un ensemble de tâches temps réel concurrentes. Les communications inter-tâches sont gérées par le protocole proposé dans [STC06] qui assure la conservation de la sémantique synchrone de tâches concurrentes et communicantes, à l'aide de mécanismes de mémoires tampon basés sur les priorités des tâches. Le calcul des priorités des tâches n'est pas détaillé dans [ACGR09]. Notre travail s'intéresse particulièrement à ce problème et calcule les priorités des tâches de manière automatique.

Dans [SLG97], une extension du calcul d'horloge est proposée à travers la définition des *relations affines d'horloges*. Plus récemment, [MTGLG08] considère une restriction de ces relations aux systèmes multi-périodiques. Ce travail porte principalement sur la vérification formelle et pour l'instant la compilation d'un programme vers un ensemble de tâches temps réel ne semble pas avoir été étudiée.

Langages synchrones impératifs

Le principal représentant des langages synchrones impératifs est le langage ESTEREL [BdS91], qui s'intéresse principalement à la description du flot de contrôle d'un système. Un programme ESTEREL consiste en un ensemble de processus concurrents communiquant par l'intermédiaire de *signaux* et structurés en *modules*. Un signal émis par un processus est *instantanément* perçu par les tous les processus qui surveillent ce signal. Les processus exécutés par un module sont décrits comme une composition d'*instructions*. Les instructions peuvent être composées séquentiellement ou en parallèle (de manière concurrente). La plupart des instructions sont considérées immédiates, exceptées les instructions contenant des pauses explicites. A chaque réaction du programme, le flot de contrôle avance dans chaque processus jusqu'à atteindre une instruction de pause ou de terminaison. A la réaction suivante, les processus reprennent là où ils s'étaient arrêtés. Un exemple de programme est donné Fig. 5. Son comportement

est illustré Fig. 6 où nous détaillons les signaux reçus et émis par le programme à chaque réaction. Le programme répète indéfiniment le même comportement (`loop..end`). Il commence par effectuer deux branches en parallèle (`||`), la première branche reste en pause jusqu'à l'occurrence de `A` (`await A`) et la seconde jusqu'à l'occurrence de `B`. Cette instruction parallèle est ensuite composée en séquence avec une instruction émettant le signal `O` (`emit O`). Le programme reste ensuite en pause jusqu'à l'occurrence de `R`. Le programme revient ensuite au début de la boucle.

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

FIG. 5 – Un programme ESTEREL

Inputs		A	B	A,B	R	A,B	...
Outputs			O			O	...

FIG. 6 – Comportement d'un programme ESTEREL

Le projet TAXYS [BCP+01] introduit des aspects temps réel dans ESTEREL à l'aide de *pragmas* (des annotations de code), comme décrit ci-dessous :

```
loop
  await PulsePeriod %{PP:=age(PulsePeriod)}%;
  call C() %{(10,15), PP<=40}%;
emit Pulse;
end loop
```

Les pragmas sont démarqués par `%{` et `}%`. Le pragma $(10, 15)$ spécifie que la durée d'exécution de la procédure `C` est comprise entre 10 et 15. Le pragma`%{PP:=age(PulsePeriod)}%` déclare une variable de temps *continue*, représentant le temps écoulé depuis l'émission du signal `PulsePeriod`. Le pragma `PP<=40` spécifie une contrainte temps réel, qui exige que la procédure `C` s'achève au plus tard 40 unités de temps après l'émission du signal `PulsePeriod`.

L'outil TAXYS combine le compilateur ESTEREL SAXO-RT [WBC+00] avec l'outil de model-checking KRONOS [DOTY96]. Le compilateur traduit le programme annoté en un automate temporisé [AD94]. L'automate est ensuite analysé à l'aide du model-checker afin de valider les contraintes temps réel du programme. Le programme ESTEREL est traduit en un code C séquentiel et l'objectif de ce travail n'est pas d'utiliser un système d'exploitation temps réel préemptif.

SYNDEX et la méthodologie AAA

La méthodologie Adéquation-Algorithm-Architecture (AAA) [GLS99] et l'outil associé SYNDEX fournissent un cadre de développement pour les systèmes temps réel embarqués distribués. Elle repose sur une représentation sous forme de graphe des langages synchrones flots de données. Les aspects fonctionnels d'un système sont décrits dans la partie Algorithme (opérations et flots de données) tandis que les aspects matériels sont décrits dans la partie Architecture (opérateurs de calculs et média de communication). Le développeur spécifie ensuite le temps d'exécution des différentes opérations de

calcul et la durée nécessaire aux différents transferts de données. L'Adéquation consiste alors à chercher une implantation optimisée de l'Algorithme sur l'Architecture.

L'introduction des systèmes multi-périodiques dans la méthodologie AAA a tout d'abord été étudiée dans [Cuc04, CS02] puis dans [KCS06, Ker09]. Les systèmes multi-périodiques sont officiellement supportés par SYNDEX depuis sa version 7, diffusée en Août 2009. D'après des expérimentations encore incomplètes, le développeur ne peut spécifier des dépendances de données entre deux opérations de rythmes différents que si la période d'une opération est un multiple de la période de l'autre. Les motifs de communication multi-rythme sont uniquement sans perte. Par exemple, si le producteur d'une donnée est n fois plus rapide que le consommateur, chaque instance du consommateur consomme les données produites par n répétitions du producteur.

Synchronous Data-Flow (SDF)

Un graphe Synchrone Flot de Données (SDF) [LM87b] est un graphe flot de données constitué d'un ensemble de nœuds représentant des calculs, reliés par des dépendances de données. Chaque entrée ou sortie d'un nœud est annotée par un entier représentant le nombre de données consommées ou produites par le nœud. Les nombres de données n produites par la source d'une dépendance de donnée et n' consommées par la destination de la dépendance ne sont pas nécessairement identiques. L'exécution d'un graphe nécessite donc de répéter la source de la dépendance k fois pour chaque k' répétitions de la destination de la dépendance, de sorte que $kn = k'n'$.

La figure 7 montre un exemple de graphe SDF. Afin de respecter les rythmes d'échantillonnage de chaque nœud, une répétition du graphe correspond à une répétition de `C` et `OUT`, 2 répétitions de `B` (pour chaque `C`), 6 répétitions de `IN` et `A` (3 pour chaque répétition de `B`).

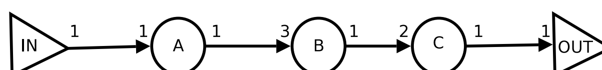


FIG. 7 – Un graphe SDF

Etant donné que les rythmes des différentes opérations sont déterminés par les ratios entre opérations dépendantes, les motifs de communication que l'on peut spécifier sont restreints à des communications sans perte (du même style que ceux décrits pour SYNDEX). De plus, les graphes SDF spécifient bien des systèmes *multi-rythmes* mais pas *multi-périodiques*. En effet, rien n'oblige les répétitions successives d'un nœud à s'exécuter à des intervalles de temps réguliers, donc d'un point de vue temps réel, les nœuds ne sont *pas périodiques*.

Les systèmes Globalement Asynchrones Localement Synchrones (GALS)

Alors que l'étude des systèmes multi-périodiques dans le cadre synchrone est relativement récente, la classe plus générale des systèmes Globalement Asynchrone Localement Synchrones (GALS) est nettement plus étudiée. Dans un système GALS, plusieurs systèmes localement synchrones sont assemblés à l'aide de mécanismes de communication asynchrones afin de produire un système complet globalement asynchrone. La classe générale des systèmes GALS est par exemple étudiée dans ESTEREL multi-horloge [BS01] et POLYCHRONY [LGTLL03]. L'approche Quasi-synchrone [CMP01], l'approche N-Synchrone [CDE+06, CMPP08] ou encore les Latency Insensitive Designs [CMSV01], ne s'intéressent qu'à des classes particulières de systèmes localement synchrones pouvant être composés de manière « quasiment » synchrone à l'aide de mécanismes de communication à mémoires bornées.

Il serait donc possible de considérer les systèmes de Contrôle-Commande comme un cas particulier de GALS et de réutiliser le travail existant. Cependant, les systèmes que nous considérons sont par

essence complètement synchrones et les traiter en tant que tels permet de mieux identifier leurs caractéristiques temps réel (lors de la spécification), de mieux les analyser (prouver la bonne synchronisation des opérations, effectuer des tests d'ordonnançabilité) et de produire un code plus optimisé (en utilisant des protocoles de communication spécifiques).

1.3 Matlab/Simulink

SIMULINK [Mat] est un langage de modélisation haut-niveau développé par The Mathworks et très utilisé dans de nombreux domaines d'application industriels. SIMULINK était à l'origine uniquement destiné à la spécification et à la simulation mais des générateurs de code commerciaux ont depuis été développés, notamment REAL-TIME WORKSHOP EMBEDDED CODER par The Mathworks, ou TARGET-LINK par dSpace. La difficulté majeure pour programmer des systèmes critiques à l'aide de SIMULINK est que SIMULINK et les outils associés ne sont pas définis de manière formelle. Par exemple, SIMULINK dispose de plusieurs sémantiques, qui varient suivant des options configurées par l'utilisateur et qui ne sont définies qu'informellement. De même, la conservation de la sémantique entre le modèle simulé et le code généré n'est pas clairement assurée.

Une traduction de SIMULINK vers LUSTRE a été proposée dans [TSCC05]. Cette traduction ne considère qu'un sous-ensemble bien défini des modèles SIMULINK et permet de bénéficier de la définition formelle de LUSTRE. Ceci permet d'effectuer des vérifications formelles et de profiter de la génération de code automatique de LUSTRE permettant de conserver la sémantique du modèle SIMULINK original.

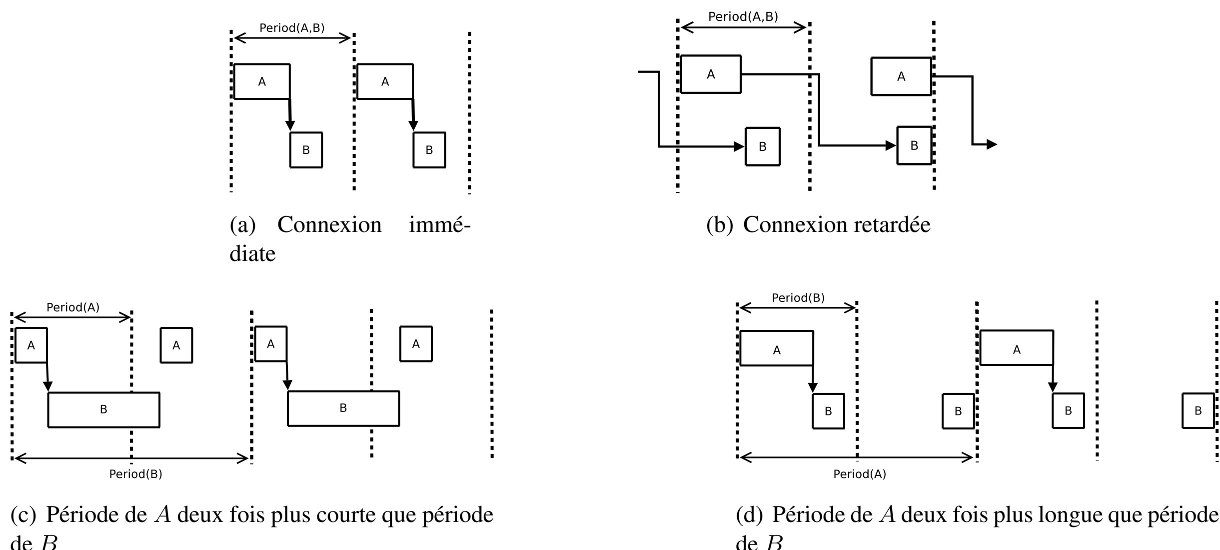
1.4 Les langages de description d'architecture (ADL)

Un langage de description d'architecture (ADL) fournit une couche d'abstraction supplémentaire en comparaison des langages présentés précédemment. Un tel langage se concentre sur la modélisation des interactions entre les composants haut-niveau d'un système. Typiquement, dans notre contexte, un ADL se concentre sur la description des interactions entre tâches plutôt qu'entre fonctions. Les fonctions doivent certes toujours être implémentées quelque part, mais ce niveau de détail est caché dans une spécification ADL. Une telle approche multi-niveau reflète l'organisation du processus de développement pour des systèmes complexes tels que les systèmes de Contrôle-Commande. Plusieurs ingénieurs développent séparément les différentes fonctionnalités du système et ensuite une personne différente, l'intégrateur, assemble ces différentes fonctionnalités pour produire le système complet. L'intégrateur a besoin d'un niveau d'abstraction élevé pour se coordonner avec les autres ingénieurs : les ADLs proposent ce niveau d'abstraction.

AADL

L'Architecture Analysis and Design Language (AADL) [FGH06] est un ADL destiné à la modélisation de l'architecture logicielle et matérielle d'un système embarqué temps réel. Une spécification AADL est constituée d'un ensemble de *composants*, qui interagissent par l'intermédiaire d'*interfaces*. Les composants incluent des composants logiciels tels que les threads mais aussi des composants matériels tels qu'un processeur. De nombreuses propriétés peuvent être spécifiées selon le type de composant, on retiendra la possibilité de spécifier pour un thread sont protocole de déclenchement (périodique, apériodique, sporadique), sa période, sa durée d'exécution et son échéance.

Les composants interagissent par des connexions entre les ports de leurs interfaces. Une connexion peut être immédiate ou retardée (de manière similaire aux langages synchrones). Des connexions peuvent aussi relier des threads de rythme différent, avec ou sans délai. Les motifs de communication obtenus pour les différents types de connexions sont illustrés Fig. 8.



(a) Connexion immédiate

(b) Connexion retardée

(c) Période de A deux fois plus courte que période de B

(d) Période de A deux fois plus longue que période de B

FIG. 8 – Motifs de communication en AADL, pour une connexion de A vers B

En comparaison d'AADL, notre langage permet de définir des motifs de communication plus riches et plus variés tout en reposant sur une sémantique synchrone définie formellement.

REACT/CLARA

Le projet REACT [FDT04] propose une suite d'outils dédiés à la conception de systèmes temps réel. Il combine la modélisation formelle, à l'aide de l'ADL CLARA [Dur98], avec des techniques de vérification formelle basées sur les Réseaux de Pétri Temporisés [BD91]. Le langage CLARA est dédié à la description de l'architecture fonctionnelle d'un système réactif.

Tout comme AADL, CLARA permet de décrire un système comme un ensemble de composants haut-niveau communicant appelés *activités*. Une activité est déclenchée de manière répétitive par la règle d'activation spécifiée sur son *port de début* et signale sa terminaison sur son *port de fin*. A chaque activation, une activité exécute une suite d'actions : acquisition d'entrées, appels de fonctions externes (dont la durée est spécifiée dans le programme), production de sorties. Des *générateurs d'occurrences* permettent de spécifier les données d'entrée ou signaux d'entrée d'un système. Il peuvent être soit périodiques, soit sporadiques.

Les ports des différentes activités son connectés par l'intermédiaire de *liens* sur lesquels le programmeur peut spécifier différents protocoles de communication : rendez-vous (protocole bloquant qui synchronise le producteur et le consommateur), tableau noir (protocole non-bloquant, le consommateur utilise simplement la dernière valeur produite) ou boîte aux lettres (communication par l'intermédiaire d'une FIFO). La taille d'une boîte aux lettres peut être bornée, auquel cas le producteur doit attendre lorsqu'elle est pleine et le consommateur doit attendre lorsqu'elle est vide. Il est aussi possible de spécifier des *opérateurs* sur des liens entre activités de rythmes différents. L'opérateur (n) spécifie que n répétitions du consommateur lisent la même valeur. L'opérateur $(1/n)$ spécifie que seule une valeur parmi n valeurs successives sera consommée.

Un exemple de programme est donné Fig. 9. Le système est constitué de deux activités `FDIR` et `GNC_US`, `GNC_US` étant 10 fois plus lente que `FDIR`. Le générateur d'occurrence `ck` émet un signal toutes les 100ms. Ce signal est communiqué à l'aide d'une boîte aux lettres de taille 1 vers le port de début `FDIR`, ce qui a pour effet de déclencher `FDIR` toutes les 100ms. L'opérateur $(1/10)$ situé sur le

lien du port de début de FDIR vers le port de fin de GNC_US permet de déclencher GNC_US une fois toutes les 10 terminaisons de FDIR. De même, GNC_US ne consomme qu'une donnée sur 10 produites par FDIR. A l'inverse, 10 itérations successives de FDIR consomment la même donnée produite par GNC_US (du fait de l'opérateur (10)).

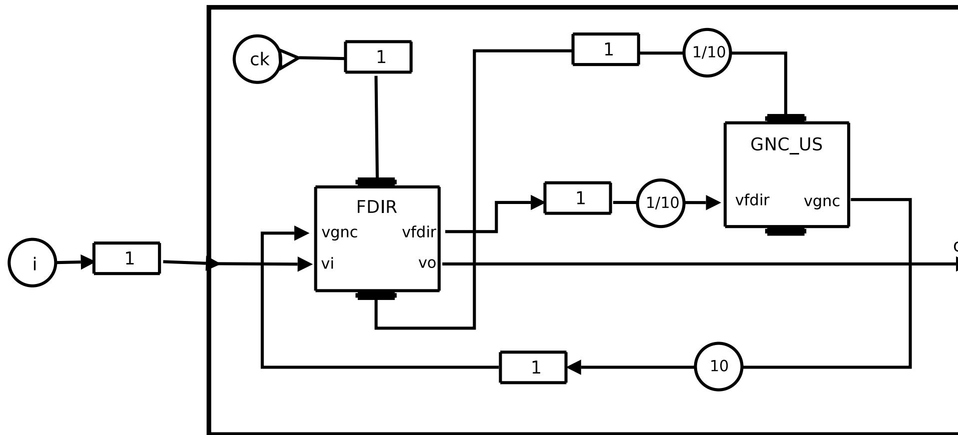


FIG. 9 – Une boucle de communication multi-rythme en CLARA

Notre langage retient certains des concepts présentés ici mais repose sur une sémantique synchrone. De plus, il semble que CLARA soit pour l'instant uniquement destiné à la vérification formelle et que la génération de code n'est pas encore supportée.

UML et CCSL

Le Unified Modeling Language (UML) [Obj07b] est un langage de modélisation à but général, constitué d'un ensemble de notations graphiques permettant de créer des modèles abstraits d'un système. Sa sémantique est sciemment lâche et peut être raffinée par la définition de *profils* spécifiques à un domaine. C'est le cas du profil UML pour la modélisation et l'analyse de systèmes temps réel embarqués (MARTE) [Obj07a]. Ce profil contient en particulier le *langage de spécification de contraintes d'horloges* (CCSL) [AM09] qui permet de spécifier les propriétés temps réel d'un système. Ce langage est inspiré des langages synchrones mais se destine à la classe plus générale des GALS.

CCSL fournit un ensemble de constructions basées sur les horloges et les contraintes d'horloges afin de spécifier des propriétés temporelles. Les horloges peuvent être soit *denses*, soit *discrètes*. Dans notre contexte, nous ne considérons que les horloges discrètes, qui sont constituées de séquences d'instants de durées abstraites. Le langage propose des contraintes d'horloges très riches, qui spécifient des contraintes sur l'ordre des instants de différentes horloges. Il est en particulier possible de définir des horloges périodiques et des échantillonnages d'horloges périodiques. CCSL est un langage très récent et n'est pour l'instant pas destiné à la génération de code mais à la vérification formelle. Comme il est très général, une génération de code efficace nécessiterait vraisemblablement de ne retenir qu'un sous-ensemble des contraintes proposées par le langage.

GIOTTO

GIOTTO [HHK03] est un ADL de type time-triggered destiné à l'implantation de systèmes temps réel. Un programme GIOTTO est constitué d'un ensemble de tâches multi-périodiques groupées en modes, les tâches étant implantées à l'extérieur de GIOTTO. Chaque tâche possède une fréquence re-

lative au mode la contenant : si le mode a pour période P et la tâche pour fréquence f, alors la tâche a pour période P/f.

Les données produites par une tâche sont considérées disponibles uniquement *à la fin de sa période*. A l'inverse, lorsqu'une tâche s'exécute, elle ne peut consommer que la dernière valeur de chaque entrée, produite *au début de sa période*. En pratique, cela revient à dire que les tâches communiquent toujours avec un délai à la LUSTRE. Ce motif de communication est très restrictif et implique des latences importantes entre l'acquisition d'une entrée et la production de la sortie correspondante. Ce type de communications n'est clairement pas adapté aux systèmes que nous considérons.

1.5 Motivation de notre travail

Comme nous pouvons le constater, les langages existants ne permettent que partiellement d'implanter des systèmes de Contrôle-Commande soumis à des contraintes temps réel multiples. Nous proposons un langage se présentant comme un langage d'architecture logicielle temps réel qui transpose la sémantique synchrone au niveau d'un ADL. Définir le langage en tant qu'ADL permet de disposer d'un haut niveau d'abstraction, tout en reposant sur la sémantique synchrone afin de profiter de propriétés formelles bien établies.

Un programme est constitué d'un ensemble de *nœuds* importés, implémentés à l'extérieur du programme à l'aide de langages existants (C ou LUSTRE par exemple), et de flots de données entre les nœuds importés. Le langage permet de spécifier de multiples contraintes d'échéance et de périodicité sur les flots et sur les nœuds. Il définit également un ensemble restreint d'opérateurs de transition de rythme permettant au développeur de définir de manière précise les motifs de communication entre nœuds de rythmes différents. Un programme est traduit automatiquement en un ensemble de tâches concurrentes implantées en C. Ces tâches peuvent être exécutées à l'aide de systèmes d'exploitation temps réel existants.

Le langage peut se comparer avec les langages existants comme suit :
- **ADLs** : Le niveau d'abstraction du langage est similaire à celui des ADLs mais le langage repose sur une sémantique différente (l'approche synchrone) et permet au développeur de définir ses propres motifs de communication entre tâches ;
- **Langages Synchrones** : Bien que le langage ait une sémantique synchrone, il possède un niveau d'abstraction plus élevé, ce qui permet une traduction efficace en un ensemble de tâches temps réel ;
- **Langages Impératifs** : Bien que le programme soit au final traduit en code impératif, le code généré est correct-par-construction. Les analyses peuvent donc être effectuées directement sur le programme d'origine plutôt que sur le code généré.

2 Horloges périodiques temps réel

A première vue, tenir compte du temps réel dans un langage synchrone peut sembler contradictoire, étant donné que l'hypothèse synchrone est souvent qualifiée d'hypothèse de temps zéro. Dans cette section, nous montrons que le temps réel peut néanmoins être introduit dans l'approche synchrone, tout en gardant sa sémantique claire et intuitive.

2.1 Un modèle reliant instants et temps réel

Nous avons vu précédemment que l'abstraction complète du temps réel par les langages synchrones rend la programmation de systèmes multi-périodiques difficile. Nous présentons dans cette section un modèle synchrone reliant les instants au temps réel, tout en conservant les bases de l'approche synchrone.

Echelles de temps logique multiples

L'idée centrale de notre modèle est qu'un système multi-périodique peut être considéré comme un ensemble de systèmes localement synchrones et mono-périodiques assemblés pour former un système global synchrone multi-périodique. Localement, chaque sous-système possède sa propre échelle de temps logique, sa propre notion d'instant et doit respecter l'hypothèse synchrone habituelle, qui nécessite de terminer les traitements avant la fin de l'instant courant de cette échelle de temps logique. Lorsque plusieurs sous-systèmes de rythmes différents sont assemblés, l'assemblage concerne des systèmes possédant des échelles de temps logique différentes. Ceci nécessite donc une référence commune afin de comparer des instants appartenant à des échelle de temps logique différentes. Le temps réel constitue cette référence commune.

Par exemple, dans le FAS, les opérations `Gyro Acq`, `FDIR` et `PDE` partagent la même échelle de temps logique, dans laquelle un instant a une durée de 100ms, tandis que `Str Acq` et `TM/TC` partagent une autre échelle de temps logique, dans laquelle les instants ont une durée de 10s. Le développeur peut décrire les opérations séparément à l'aide de l'approche synchrone habituelle mais lors de l'assemblage des opérations il doit pouvoir comparer des instants « rapides » avec des instants « lents ». Lorsqu'on considère la durée réelle d'un instant, il est naturel de considérer qu'un instant lent correspond à 100 instants rapides. Notre modèle formalise cette idée en reliant instants et temps réel.

Flots temps réel

Notre modèle synchrone temps réel repose sur le Tagged-Signal Model [LSV96]. Etant donné un ensemble de *valeurs* $\mathcal{V}$, un *flot* est une séquence de couples $(v_i, t_i)_{i\in\mathbb{N}}$ où v_i est une valeur dans $\mathcal{V}$ et t_i est une étiquette dans $\mathbb{Q}$, telle que pour tout i, $t_i < t_{i+1}$. L'horloge d'un flot f, notée $ck(f)$, est sa projection sur $\mathbb{Q}$. De même, $val(f)$ représente la séquence de valeurs de f (sa projection sur $\mathcal{V}$). Deux flots sont *synchrones* s'ils ont la même horloge. Une étiquette représente la quantité de temps écoulée depuis le début de l'exécution du programme. On dit qu'un flot est *présent* à la date t si t apparaît dans son horloge, *absent* sinon.

Selon *l'hypothèse synchrone relâchée* de [Cur05], à chaque activation la valeur d'un flot doit être calculée avant sa prochaine activation. Ainsi, chaque flot a sa propre notion d'instant et la durée de l'instant t_i est $t_{i+1} - t_i$. Deux flots sont synchrones si les durées de leurs instants sont identiques. Il est possible de comparer la durée des instants de deux flots pour déterminer si un flot est plus rapide que l'autre et, surtout, il est possible de déterminer *de combien* ce flot est plus rapide. Par exemple, dans le FAS, les sorties de `Gyro Acq` ont pour horloge $(0, 100, 200, \ldots)$, tandis que les sorties de `Str Acq` ont pour horloge $(0, 10000, 20000, \ldots)$. Ainsi les sorties de `Gyro Acq` sont 100 fois plus rapides.

2.2 Horloges Strictement Périodiques

Dans cette section nous introduisons une nouvelle classe d'horloges, les horloges strictement périodiques, ainsi que des transformations spécifiques permettant de gérer les transitions entre échelles de temps logique différentes.

Définition

Une horloge est une séquence d'étiquettes, nous définissons une classe particulière d'horloges appelées horloges strictement périodiques de la manière suivante :

Définition 1. *(*Horloge strictement périodique*). Une horloge $h = (t_i)_{i\in\mathbb{N}}$, $t_i \in \mathbb{Q}$, est* strictement périodique *si et seulement si :*

$$\exists n \in \mathbb{Q}^{+*},\ \forall i \in \mathbb{N},\ t_{i+1} - t_i = n$$

n est la période *de h, notée $\pi(h)$ et t_0 est la* phase *de h, notée $\varphi(h)$.*

Une horloge strictement périodique définit le rythme temps réel d'un flot. La notion de phase définie ici est un peu plus générale qu'habituellement étant donnée que nous n'imposons pas que $\varphi(h) < \pi(h)$. Une horloge strictement périodique est caractérisée de manière unique par sa phase et sa période :

Définition 2. *Le terme $(n, p) \in \mathbb{Q}^{+*} \times \mathbb{Q}$ représente l'horloge strictement périodique α telle que :*

$$\pi(\alpha) = n,\ \varphi(\alpha) = \pi(\alpha) * p$$

Transformations périodiques d'horloges

Nous définissons ensuite des transformations d'horloges spécifiques aux horloges strictement périodiques produisant de nouvelles horloges strictement périodiques :

Définition 3. *(*Division périodique d'horloge*). Soit α une horloge strictement périodique et $k \in \mathbb{Q}^{+*}$. "$\alpha/_{.}k$" est une horloge strictement périodique telle que :*

$$\pi(\alpha/_{.}k) = k * \pi(\alpha),\ \varphi(\alpha/_{.}k) = \varphi(\alpha)$$

Définition 4. *(*Multiplication périodique d'horloge*). Soit α une horloge strictement périodique et $k \in \mathbb{Q}^{+*}$. "$\alpha *_{.} k$" est une horloge strictement périodique telle que :*

$$\pi(\alpha *_{.} k) = \pi(\alpha)/k,\ \varphi(\alpha *_{.} k) = \varphi(\alpha)$$

Définition 5. *(*Décalage de phase*). Soit α une horloge strictement périodique et $k \in \mathbb{Q}$. "$\alpha \rightarrow_{.} k$" est une horloge strictement périodique telle que :*

$$\pi(\alpha \rightarrow_{.} k) = \pi(\alpha),\ \varphi(\alpha \rightarrow_{.} k) = \varphi(\alpha) + k * \pi(\alpha)$$

Les divisions réduisent et les multiplications augmentent la fréquence d'une horloge, tandis que les décalages de phase changent la phase d'une horloge. Ceci est illustré Fig. 10. A nouveau, nous n'exigeons pas que $\varphi(\alpha \rightarrow_{.} k) < \pi(\alpha)$.

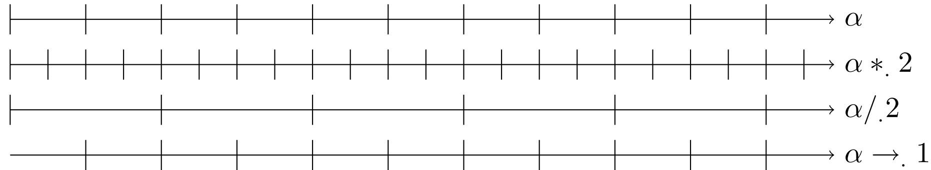


FIG. 10 – Horloges strictement périodiques.

Soit $\mathcal{P}$ l'ensemble des horloges strictement périodiques. Des définitions précédentes, nous déduisons les Propriétés 1, 2 et 3.

Propriété 1. $\forall \alpha, \beta \in \mathcal{P}$ *:*
- $\forall k \in \mathbb{Q},\ \alpha = \beta \rightarrow_{.} k \Leftrightarrow \beta = \alpha \rightarrow_{.} -k$ *;*
- $\forall k \in \mathbb{Q}^{+*},\ \alpha = \beta/_{.}k \Leftrightarrow \beta = \alpha *_{.} k$.

Propriété 2. $\forall (n, p) \in \mathbb{Q}^{+*} \times \mathbb{Q}$ *:*
- $\forall k \in \mathbb{Q}^{+*},\ (n, p) *_{.} k = (\frac{n}{k}, kp)$ *;*
- $\forall k \in \mathbb{Q}^{+*},\ (n, p)/_{.}k = (nk, \frac{p}{k})$ *;*
- $\forall k \in \mathbb{Q},\ (n, p) \rightarrow_{.} k = (n, p + k)$.

Propriété 3. *L'ensemble des horloges strictement périodiques $\mathcal{P}$ est fermé par les opérations $*_{.}$, $/_{.}$ et $\rightarrow_{.}$.*

Horloges strictement périodiques entières

Nous avons défini le modèle général des horloges strictement périodiques en utilisant des dates dans $\mathbb{Q}$. Cependant, afin de définir un langage compilable, nous devons nous restreindre à des dates dans $\mathbb{N}$. En effet, les systèmes d'exploitation reposent sur un modèle de temps discret et il existe toujours une borne inférieure du niveau de granularité pouvant être utilisée pour décrire le temps. Par exemple, la date $1/3$ n'existe pas dans un système réel, ou alors est une approximation. Pour les mêmes raisons, la théorie de l'ordonnancement, que nous utilisons pour implanter nos programmes, ne s'applique le plus souvent qu'à des dates et durées dans $\mathbb{N}$. Par conséquent, nous nous restreignons à des étiquettes dans $\mathbb{N}$ et nous considérons que les paramètres k utilisés dans les divisions périodiques d'horloges ou les multiplications périodiques d'horloges doivent être des éléments de $\mathbb{N}^*$. Le paramètre k utilisé dans les décalages de phase peut par contre être un élément de $\mathbb{Q}$. Dans la suite de ce mémoire, nous ne considérerons que les *horloges strictement périodiques entières* que nous qualifierons simplement d'horloges strictement périodiques.

Dans l'ensemble restreint des horloges strictement périodiques entières, l'horloge (n, p) n'est une horloge valide qu'à la condition suivante : $n \in \mathbb{N}^{+*}, n * p \in \mathbb{N}^+$. En effet, les horloges ne vérifiant pas cette propriété font référence à des dates n'étant pas dans $\mathbb{N}$. Cet ensemble restreint n'est plus clos par les opérations $*_.$ et $\rightarrow_.$ (mais reste clos par $/_.$). Notons $k|k'$ lorsque k divise k' (le reste de la division entière est 0). Tout d'abord, soit (n, p) une horloge strictement périodique, si $k|n$ alors $(n, p) *_. k$ est une horloge strictement périodique valide, sinon elle n'est pas valide, car elle fait référence à des dates qui ne sont pas dans $\mathbb{N}$. Ensuite, si $(p + k)n \in \mathbb{N}^+$ alors $(n, p) \rightarrow_. k$ est une horloge strictement périodique valide, sinon elle n'est pas valide, car elle fait référence à des dates négatives. Le calcul d'horloges, présenté Sect. 4, vérifie que de telles horloges invalides ne sont pas utilisées dans un programme.

2.3 Horloges strictement périodiques et horloges booléennes

Les horloges strictement périodiques et les transformations associées ne sont pas définies dans le but de remplacer les horloges booléennes classiques des langages synchrones. Notre langage se base d'ailleurs sur ces deux classes d'horloges car elles correspondent à des notions complémentaires : les horloges strictement périodiques définissent la fréquence temps réel d'un flot, tandis que les horloges booléennes définissent la condition d'activation d'un flot.

Afin d'ajouter les horloges booléennes dans notre langage, nous adaptons la définition de l'opérateur d'horloge booléen `on` de [CP03] à notre modèle. Le comportement de cet opérateur est défini inductivement sur la séquence des étiquettes d'une horloge. Le terme $t.s$ représente l'horloge dont la tête est l'étiquette t et dont la queue est la séquence s. De même, le terme $(v, t).s$ représente le flot dont la tête a la valeur v et l'étiquette t et dont la queue est la séquence s. Pour toute horloge s, pour tout flot booléen c, α `on` c est l'horloge définie inductivement de la manière suivante :

$$(t.ck) \text{ on } ((true, t).c) = t.(ck \text{ on } c)$$
$$(t.ck) \text{ on } ((false, t).c) = ck \text{ on } c$$

L'opération α `on` c produit une nouvelle horloge contenant uniquement les étiquettes de α pour lesquelles c vaut vrai. L'opération inverse α `on not` c ne conserve que les étiquettes pour lesquelles c vaut faux. Les opérations α `on` c ou α `on not` c ne sont valides que si c a pour horloge α. Le chapitre suivant illustre la manière dont les horloges booléennes et les horloges strictement périodiques peuvent être combinées dans un même programme.

3 Présentation informelle du langage

Ce chapitre présente le langage de manière informelle. Celui-ci est construit comme une extension de LUSTRE et emprunte certaines constructions à LUCID SYNCHRONE [Pou06] et aux versions récentes de SCADE. Il étend les langages synchrones flots de données avec des primitives temps réel haut-niveau basées sur les horloges strictement périodiques. La sémantique formelle du langage est disponible dans la partie anglaise du mémoire.

3.1 Typage polymorphe

Le langage fournit un typage polymorphe, ce qui est désormais une caractéristique relativement courante pour un langage de programmation. Les types des variables du programme peuvent être laissés non-spécifiés, auquel cas ils seront inférés par le compilateur. Par exemple :

```
node infer(i:int) returns (o)
let o=i; tel
```

Dans cet exemple, le compilateur infère que le type de `o` est `int`.

Un exemple de polymorphisme est donné ci-dessous :

```
node poly(i,j) returns (o,p)
let o=i; p=j; tel
```

Le compilateur peut seulement déduire que `o` et `i` ont le même type (disons α), et que `p` et `j` ont le même type (disons β). Le nœud `poly` peut être instancié avec différents types, par exemple le programme suivant est correct :

```
node inst(i:int; j:bool) returns (o,p,q,r)
let
  (o,p)=poly(i,j);
  (q,r)=poly(j,i);
tel
```

Le compilateur infère que `o` a le type `int`, `p` a le type `bool`, `q` a le type `bool` et `r` a le type `int`.

3.2 Primitives temps réel

Le langage a pour but de permettre de spécifier des contraintes temps réel avec un niveau d'abstraction élevé. Les primitives que nous introduisons sont destinées à spécifier des contraintes relatives à l'environnement du système (des contraintes physiques par exemple) plutôt que des contraintes liées à des considérations d'implémentation. Les entrées et sorties d'un nœud représentent son interface avec l'environnement. Les contraintes temps réel sont donc spécifiées sur les entrées et sorties d'un nœud : ce sont des contraintes venant de l'environnement du système.

La période d'une entrée ou d'une sortie d'un nœud peut être spécifiée de la manière suivante :

```
node periodic(i: int rate (10,0)) returns (o: int rate (5,0))
let
...
tel
```

`x: rate (n,p)` spécifie que l'horloge de x est l'horloge strictement périodique (n, p). Ainsi, i a pour période 10 et o a pour période 5.

Le développeur peut aussi spécifier la phase d'un flot périodique :

```
node phased(i: int rate (10,1/2)) returns (o: int rate (10,0))
let
...
tel
```

Dans cet exemple, i a pour période 10 et pour phase $\frac{1}{2}$, donc sa séquence d'horloge est $(5, 15, 25, 35, \ldots)$.

Une contrainte d'échéance définit une échéance à respecter pour le calcul d'un flot, relative au début de la période du flot. Les contraintes d'échéance peuvent être spécifiées sur des sorties de nœuds :

```
node deadline(i: int rate (10,0)) returns (o: int rate (10,0) due 8)
let
...
tel
```

`x: due d` spécifie que x a pour contrainte d'échéance d. Si x est présent à la date t, alors x doit être calculé avant la date $t + d$.

Des contraintes d'échéance peuvent aussi être spécifiées sur des entrées, mais leur signification est alors légèrement différente :

```
node deadline(i: int rate (10,0) before 2) returns (o: int)
let
...
tel
```

`x: before d` spécifie que l'entrée x doit être acquise par le programme avant l'échéance d (ie une échéance pour la « production » de l'entrée). En pratique, cela signifie qu'une opération *capteur* doit s'effectuer avant cette échéance. Les opérations effectuant des opérations à partir de cette entrée pourront lire la valeur de l'entrée à partir de la sortie du capteur.

Les contraintes d'échéance n'ont pas d'impact sur l'horloge d'un flot et donc sur les contraintes de synchronisation entre flots. Si un flot f a pour horloge (n, p) et pour échéance d et si un flot f' a pour horloge (n, p) et pour échéance d' (avec $d \neq d'$), les flots sont tout de même considérés synchrones et peuvent être combinés car ils sont produits durant les même instants logiques. Les échéances ne sont prises en compte que durant la phase d'ordonnancement du programme.

Opérateurs de transition de rythme

Nous définissons des *opérateurs de transition de rythme*, basés sur les transformations périodiques d'horloges, servant à gérer les transitions entre flots de rythmes différents. Ces opérateurs permettent de définir des motifs de communication entre nœuds de rythmes différents. Notre objectif est de fournir un ensemble réduit d'opérateur simples, qui peuvent être combinés afin de produire des motifs de communication complexes. La sémantique de ces opérateurs est définie formellement et leur comportement est parfaitement déterministe.

Tout d'abord, il est possible de sous-échantillonner un flot à l'aide de l'opérateur $/\hat{}$:

```
node under_sample(i: int rate (5,0))
returns (o: int rate (10,0))
let
  o=i/^2;
tel
```

$e/\hat{}k$ ne conserve que la première valeur parmi k valeurs successives de e. Si e a pour horloge α, alors $e/\hat{}k$ a pour horloge $\alpha/_{.}k$. Ce comportement est illustré Fig. 11.

A l'inverse, il est possible de sur-échantillonner un flot à l'aide de l'opérateur $*\hat{}$:

date	0	5	10	15	20	25	30	...
i	i_0	i_1	i_2	i_3	i_4	i_5	i_6	...
o	i_0		i_2		i_4		i_6	...

FIG. 11 – Sous-échantillonnage périodique

```
node over_sample(i: int rate (10,0))
returns (o: int rate (5,0))
let
  o=i*^2;
tel
```

$e *\hat{} k$ duplique chaque valeur de e, k fois. Si e a pour horloge α, alors $e *\hat{} k$ a pour horloge $\alpha *_. k$. Ce comportement est illustré Fig. 12.

date	0	5	10	15	20	25	30	...
i	i_0		i_1		i_2		i_3	...
o	i_0	i_0	i_1	i_1	i_2	i_2	i_3	...

FIG. 12 – Sur-échantillonnage périodique

Opérateurs de décalage de phase

Nous définissons trois opérateurs basés sur les décalages de phase. L'ensemble des valeurs d'un flot peut être décalée d'une fraction de sa période à l'aide de l'opérateur $\sim>$:

```
node offset(i: int rate (10,0))
returns (o: int rate (10,1/2))
let
  o=i~>1/2;
tel
```

Si e a pour horloge α, $e \sim> q$ retarde chaque valeur de e de $q*\pi(\alpha)$ (avec $q \in \mathbb{Q}^+$). L'horloge de $e \sim> q$ est $\alpha \rightarrow_. q$. Cet opérateur est le plus souvent utilisé avec q inférieur à 1, mais les valeurs supérieures à 1 sont autorisées. Le comportement de cet opérateur est illustré Fig. 13.

date	0	5	10	15	20	25	30	...
i	i_0		i_1		i_2		i_3	...
o		i_0		i_1		i_2		...

FIG. 13 – Décalage de phase

L'opérateur `tail` renvoie la queue d'un flot :

```
node tail_twice(i: int rate(10,0))
returns (o1: int rate(10,1); o2: int rate(10,2))
let
  o1=tail(i);
  o2=tail(o1);
tel
```

`tail` (e) élimine la première valeur de e. Si e a pour horloge α, alors `tail` (e) pour horloge $\alpha \rightarrow_{.} 1$, donc `tail` (e) est actif pour la première fois une période plus tard que e. Plusieurs `tail` éliminent plusieurs valeurs du flot. Ceci est illustré Fig. 14.

date	0	10	20	30	40	50	60	...
`i`	i_0	i_1	i_2	i_3	i_4	i_5	i_6	...
`o1`		i_1	i_2	i_3	i_4	i_5	i_6	...
`o2`			i_2	i_3	i_4	i_5	i_6	...

FIG. 14 – Eliminer la tête d'un flot

Le langage contient aussi un opérateur de concaténation « `::` » :

```
node init(i: int rate(10,0))
returns (o1: int rate(10,0); o2: int rate(10,0))
var v1,v2;
let
  (v1,v2)=tail_twice(i);
  o1=0::v1;
  o2=0::0::v2;
tel
```

cst `::` e produit cst une période avant la première valeur de e et produit ensuite e. Si e a pour horloge α, alors cst `::` e a pour horloge $\alpha \rightarrow_{.} -1$. Ceci est illustré Fig. 15.

date	0	10	20	30	40	50	60	...
`i`	i_0	i_1	i_2	i_3	i_4	i_5	i_6	...
`v1`		i_1	i_2	i_3	i_4	i_5	i_6	...
`v2`			i_2	i_3	i_4	i_5	i_6	...
`o1`	0	i_1	i_2	i_3	i_4	i_5	i_6	...
`o2`	0	0	i_2	i_3	i_4	i_5	i_6	...

FIG. 15 – Concaténation de flots

Transitions de rythmes sur des rythmes non spécifiés

Le développeur peut laisser les rythmes des entrées ou sorties d'un nœud non spécifiés, ils seront alors inférés par le calcul d'horloges. Par exemple :

```
node under_sample(i: int rate(10,0)) returns (o: int)
let
  o=i/^2;
tel
```

Le calcul d'horloges infère que `o` a pour horloge $(20, 0)$.

Les opérateurs de transition de rythme et les opérateurs de décalage de phase peuvent être appliqués sur des flots dont le rythme est inconnu. Par exemple, la définition suivante est tout à fait valide :

```
node under_sample(i: int) returns (o: int)
let
  o=i/^2;
tel
```

Ce programme spécifie simplement que le rythme de o est deux fois plus lent que celui de i, quel que soit le rythme de i. Le véritable rythme des flots ne sera calculé qu'à l'instanciation du nœud. Le nœud peut d'ailleurs être instancié à différents rythmes dans un même programme, par exemple :

```
node poly(i: int rate (10, 0); j: int rate (5, 0))
returns (o, p:  int)
let
  o=under_sample(i);
  p=under_sample(j);
tel
```

L'horloge inférée pour o est $(20, 0)$, tandis que l'horloge inférée pour p est $(10, 0)$. Ce polymorphisme d'horloges augmente le niveau de modularité avec lequel un système peut être programmé.

Capteurs et actionneurs

Le développeur doit spécifier le wcet de l'acquisition de chaque entrée du système (chaque entrée du nœud principal), appelée opération *capteur*, et le wcet de la production de chaque sortie du système (chaque sortie du nœud principal), appelée opération *actionneur*. Par exemple, le programme suivant spécifie que l'acquisition de i prend 1 unité de temps, l'acquisition de j prend 2 unités de temps et la production de o prend 2 unités de temps :

```
imported node A(i, j: int) returns (o: int) wcet 3;
sensor i wcet 1; sensor j wcet 2; actuator o wcet 2;

node N(i, j: int rate (10, 0)) returns (o: int rate (10, 0))
let
  o=A(i, j);
tel
```

3.3 Retards

Un flot peut être retardé à l'aide de l'opérateur fby défini dans LUCID SYNCHRONE :

```
node delay(i: int) returns (o: int)
let
  o=0 fby i;
tel
```

cst fby e produit tout d'abord cst puis les valeurs de e chacune retardée d'une période de e. L'horloge de cst fby e est égale à celle de e. Le comportement de cet opérateur est illustré Fig. 16. On peut noter que cst fby e est strictement équivalent à cst :: $(e \sim> 1)$ et peut être considéré comme du sucre syntaxique. Le processus de compilation ne tire cependant pas profit de cette possible réécriture car elle ne permet pas des simplifications significatives.

i	i_0	i_1	i_2	i_3	i_4	i_5	i_6	...
o	0	i_0	i_1	i_2	i_3	i_4	i_5	...

FIG. 16 – Retarder un flot

3.4 Conditions d'activation

Opérateurs d'échantillonnage booléens

Un flot peut être sous-échantillonné suivant une condition booléenne à l'aide de l'opérateur classique `when` présenté précédement. Si e a pour horloge α, alors e `when` c a pour horloge α `on` c.

Un flot peut être sur-échantillonné suivant une condition booléenne à l'aide de l'opérateur `merge`, défini dans LUCID SYNCHRONE. L'opérateur `merge` combine des flots sur des horloges booléennes complémentaires pour produire un flot plus rapide.

```
node boolean_clocks(c: bool;i, j: int) returns (o:  int)
let
  o=merge(c, i when c,  j whennot c);
tel
```

`merge` (c, e_1, e_2) produit la valeur de e_1 lorsque c vaut vrai puis la valeur de e_2 lorsque c vaut faux. Si c a pour horloge α, alors `merge` (c, e_1, e_2) a aussi pour horloge α. Le comportement de ce programme est décrit Fig. 17.

`i`	i_0	i_1	i_2	i_3	i_4	i_5	i_6	...
`j`	j_0	j_1	j_2	j_3	j_4	j_5	j_6	...
`c`	true	true	false	true	false	false	true	...
`i` **`when`** `c`	i_0	i_1		i_3			i_6	...
`j` **`whennot`** `c`			j_2		j_4	j_5		...
`o`	i_0	i_1	j_2	i_3	j_4	j_5	i_6	...

FIG. 17 – Opérateurs d'horloges booléens

Combiner opérateurs d'horloges booléens et strictement périodiques

L'opérateur `when` peut être appliqué à des flots dont l'horloge est booléenne mais aussi à des flots dont l'horloge est strictement périodique. Par exemple, le programme suivant est tout à fait valide :

```
node condperiodic(c: rate(5,0); i: rate (10,0)) returns (o)
let
  o=(i*^2) when c;
tel
```

L'horloge de la sortie o est : $(5, 0)$ `on` c. Le comportement de ce programme est illustré Fig. 18.

date	0	5	10	15	20	25	...
`c`	true	true	false	false	true	false	...
`i`	i_0		i_1		i_2		...
`i*^2`	i_0	i_0	i_1	i_1	i_2	i_2	...
`o=(i*^2)` **`when`** `c`	i_0	i_0			i_2		...

FIG. 18 – Application de transformations d'horloges booléennes à des horloges strictement périodiques

Cependant, l'inverse n'est pas autorisé, les opérateurs basés sur les horloges strictement périodiques ne peuvent pas être appliqués à des flots dont l'horloge contient des transformations d'horloges booléennes. Par exemple, le flot `(x` **`when`** `c)*^2` est rejeté par le compilateur. En effet, ceci nécessiterait d'appliquer une transformation périodique d'horloge à une horloge non périodique.

En combinant les deux classes d'horloges, le programmeur peut tout d'abord spécifier la fréquence de base d'une opération à l'aide des horloges strictement périodiques puis spécifier que l'opération n'est activé à ce rythme que si une certaines condition est vraie, à l'aide des horloges booléennes.

4 Calcul d'horloges

Le langage étant destiné à des systèmes embarqués critiques, un accent particulier est porté sur la vérification de la correction du programme à compiler. Une série d'analyses statiques est effectuée avant de compiler un programme vers du code bas-niveau. Ces analyses permettent d'assurer que la sémantique du programme à compiler est bien définie. Chaque analyse statique assure une propriété de correction différente du programme. L'analyse de causalité vérifie qu'il existe un ordre d'exécution pour l'évaluation des expressions du programme respectant toutes les dépendances de données du programme (c'est-à-dire que les dépendances de données ne sont pas cycliques). Le typage vérifie que seuls des flots de mêmes types sont combinés. Le calcul d'horloge vérifie que seuls des flots synchrones sont combinés. Lorsque qu'un programme passe toutes les analyses statiques, sa sémantique est bien définie. Le typage et l'analyse de causalité du langage sont relativement standards, nous ne détaillons donc ici que le calcul d'horloge (plus de détails sont disponibles dans la partie anglaise du mémoire).

Le calcul d'horloges vérifie qu'un programme ne combine que des flots de même horloge. Lorsque deux flots ont la même horloge, ils sont synchrones (toujours présents aux mêmes instants). Combiner des flots non-synchrones produit des programmes non-déterministes accédant à des valeurs non-définies. On dira qu'une expression est *bien-synchronisée* si elle ne combine que des flots de même horloge, *mal-synchronisée* sinon. L'objectif du calcul d'horloges est de vérifier qu'un programme n'utilise que des expressions bien-synchronisées.

Pour chaque expression du programme, le calcul d'horloges produit des contraintes devant être respectées pour que l'expression soit bien synchronisée. Comme cela a été montré dans [CP03], le calcul d'horloges peut utiliser un système de types pour générer ces contraintes. La structure globale du calcul d'horloges est donc très similaire à celle d'un système de type. Les horloges sont représentées par des types appelés *types d'horloges*. Des *règles d'inférence* permettent de représenter les contraintes que les horloges d'une expression doivent respecter pour être bien synchronisée. L'*unification d'horloges* tente ensuite de résoudre l'ensemble des contraintes générées pour un programme. Si une solution existe, le programme est bien-synchronisé, sinon il est mal-synchronisé.

4.1 Types d'horloges

Le calcul d'horloges produit des jugements de la forme $H \vdash e : cl$, signifiant « l'expression e a l'horloge cl dans l'environnement H ». La grammaire des types d'horloges est définie ci-dessous :

$$\begin{array}{lcl}
\sigma & ::= & \forall \alpha_1 <: C_1, ..., \alpha_m <: C_m.cl \\
cl & ::= & cl \rightarrow cl \mid cl \times cl \mid ck \mid c : ck \\
ck & ::= & ck \; \texttt{on} \; c \mid ck \; \texttt{on not} \; c \mid pck \\
pck & ::= & (n, p) \mid \alpha \mid pck *. \, k \mid pck /. \, k \mid pck \rightarrow. \, q \\
c & ::= & nm \mid X \\
H & ::= & [x_1 : \sigma_1, ..., x_m : \sigma_m]
\end{array}$$

Un type d'horloge peut être soit un schéma d'horloges (σ) quantifié sur un ensemble de variables d'horloges appartenant à différents sous-types d'horloges ($C_1, ..., C_m$), soit une expression d'horloge non-quantifiée (cl). Une expression d'horloge peut être soit une horloge fonctionnelle ($cl \rightarrow cl$), soit un produit d'horloges ($cl \times cl$), soit une séquence d'horloges (ck), soit une dépendance ($c : ck$).

Une séquence d'horloge est soit le résultat d'un sous-échantillonnage booléen (ck `on` c, ck `on not` c) soit une horloge strictement périodique (pck). Une horloge strictement périodique est soit une horloge strictement périodique constante ((n, p)), soit une variable d'horloge (α), soit une horloge strictement périodique sur-échantillonnée périodiquement ($pck *_. k$), soit une horloge strictement périodique sous-échantillonnée périodiquement ($pck/_. k$), soit le résultat d'un décalage de phase ($pck \rightarrow_. k$). En d'autres termes, une séquence d'horloges ck est une expression linéaire résultant d'une suite de transformations appliquées à une autre séquence d'horloge ck'. Dans la suite, ck est qualifiée d'*horloge abstraite* si ck' est une variable d'horloge. Elle est qualifiée d'*horloge concrète* si ck' est une horloge strictement périodique constante. Une porteuse (c) est soit un nom de flot (nm), soit une variable porteuse (X) (lorsqu'il s'agit d'une entrée du nœud).

Un environnement (H) associe des types d'horloges aux variables et définitions de nœuds. Un environnement peut donc être considéré comme une fonction associant un type d'horloge à un nom de variable ou de nœud.

Les horloges peuvent être généralisées (lors d'une définition de nœud) et instanciées (lors d'un appel de nœud) de la manière suivante :

$$inst(\forall \alpha <: C.cl) = cl[(cl' \in C)/\alpha]$$
$$gen_H(cl) = \forall \alpha_1 <: C_1, ..., \alpha_m <: C_m.cl, \text{ avec } \alpha_1, ..., \alpha_m = FTV(cl) \backslash FTV(H)$$

Un schéma d'horloges est instancié en remplaçant ses variables d'horloges par des horloges appartenant aux bons sous-types d'horloges. Une variable d'horloge peut être généralisée si elle n'apparaît pas libre dans l'environnement.

Une variable d'horloge α peut être contrainte à appartenir à un certain sous-type d'horloges C_i, ce qui est noté $\alpha <: C_i$. Une telle contrainte signifie que seules des horloges appartenant à C_i peuvent être substituées à α. Par exemple, si x a pour horloge α, alors $x *\hat{}\ k$ n'est bien synchronisée que si $k|\pi(\alpha)$, sinon la période de $x *\hat{}\ k$ n'est pas entière. Ces contraintes correspondent à la *quantification de types bornés* de [Pie02].

Soit $\mathcal{C}$ l'ensemble de tous les types d'horloges. Les deux principaux sous-types de $\mathcal{C}$ sont les horloges booléennes $\mathcal{B}$ (les horloges contenant l'opérateur `on`) et les horloges strictement périodiques $\mathcal{P}$. Le type d'horloge $\mathcal{P}$ est ensuite séparé en sous-types $\mathcal{P}_{k,\frac{a}{b}}$, $k \in \mathbb{N}^*$, $\frac{a}{b} \in \mathbb{Q}^+$, avec $a \wedge b = 1$ (a et b sont premiers entre eux) et $b|k$, avec :

$$\mathcal{P}_{k,\frac{a}{b}} = \{(n, p)|(k|n) \wedge p \geq \frac{a}{b}\}$$

En d'autres termes, l'ensemble $\mathcal{P}_{k,\frac{a}{b}}$ contient toutes les horloges strictement périodiques α pour lesquelles $\alpha *_. k$ et $\alpha \rightarrow_. -\frac{a}{b}$ produisent des horloges strictement périodiques valides. On note que $\mathcal{P} = \mathcal{P}_{1,0}$.

La relation de sous-typage $<:$ sur les types d'horloges se définit comme suite :

- $\mathcal{B} <: \mathcal{C}, \mathcal{P} <: \mathcal{C}$;
- $\mathcal{P}_{k,q} <: \mathcal{P}_{k',q'} \Leftrightarrow k'|k \wedge q \geq q'$.

4.2 Inférence d'horloges

Parent strictement périodique

Notre objectif étant d'ordonnancer un programme de manière efficace, il est nécessaire de calculer la période de chaque flot du programme. Lorsque l'horloge d'un flot contient des transformations booléennes, il faut donc approximer l'horloge à une horloge strictement périodique pour pouvoir calculer la période du flot. Par conséquent, nous définissons la sur-approximation d'une horloge à son plus proche parent strictement périodique de la manière suivante :

Définition 6. *L'*horloge strictement périodique parente *d'une horloge est définie récursivement comme suit :*

$$pparent(\alpha \text{ on } c) = pparent(\alpha)$$
$$pparent(pck) = pck$$

Nous imposons de plus que les horloges de toutes les entrées du nœud principal d'un programme soient strictement périodiques. Les horloges d'un programme dérivant toutes indirectement de ces horloges, ceci assure que $pparent(ck)$ est une horloge strictement périodique pour toute horloge ck.

Règles d'inférence

Les contraintes à vérifier pour qu'une expression soit bien synchronisée sont exprimées à l'aide de règles d'inférence. Une règle $\frac{A}{B}$ signifie que le jugement B, appelé la *conclusion*, est vrai si la condition A, appelée *prémisse*, est vraie. Les règles d'inférence d'horloge sont données Fig. 19.

$$\text{(SUB)}\ \frac{H \vdash e : ck \qquad H \vdash ck <: C}{H \vdash e : C} \qquad \text{(CONST)}\ \frac{}{H \vdash cst : ck} \qquad \text{(VAR)}\ \frac{x \in dom(H)}{H \vdash x : H(x)}$$

$$(\times)\ \frac{H \vdash e_1 : cl_1 \qquad H \vdash e_2 : cl_2}{H \vdash (e_1, e_2) : cl_1 \times cl_2} \qquad \text{(FBY)}\ \frac{H \vdash cst : cl \qquad H \vdash e : cl}{H \vdash cst \texttt{ fby } e : cl}$$

$$\text{(WHEN)}\ \frac{H \vdash e : ck \qquad H \vdash c : (c : ck)}{H \vdash e \texttt{ when } c : c \texttt{ on } ck} \qquad \text{(WHEN-NOT)}\ \frac{H \vdash e : ck \qquad H \vdash c : (c : ck)}{H \vdash e \texttt{ whennot } c : c \texttt{ on not } ck}$$

$$\text{(MERGE)}\ \frac{H \vdash c : (c : ck) \qquad H \vdash e_1 : ck \texttt{ on } c \qquad H \vdash e_2 : ck \texttt{ on not } c}{H \vdash \texttt{merge } (c, e_1, e_2) : ck}$$

$$(\hat{*})\ \frac{H \vdash e : pck \qquad pck <: \mathcal{P}_{k,0} \qquad k \neq 0}{H \vdash e *\hat{}\, k : pck *_. k} \qquad (/\hat{})\ \frac{H \vdash e : pck \qquad k \neq 0}{H \vdash e /\hat{}\, k : pck /_. k}$$

$$(\sim>)\ \frac{H \vdash e : pck \qquad pck <: \mathcal{P}_{b,0} \qquad (q = \frac{a}{b}), a \wedge b = 1}{H \vdash e \sim> q : pck \rightarrow_. q} \qquad (\texttt{tail})\ \frac{H \vdash e : pck}{H \vdash \texttt{tail } (e) : pck \rightarrow_. 1}$$

$$(::)\ \frac{H \vdash cst : pck \qquad H \vdash e : pck \qquad pck <: \mathcal{P}_{1,1}}{H \vdash cst \texttt{ :: } e : pck \rightarrow_. -1}$$

$$\text{(APP)}\ \frac{H \vdash N : \sigma \qquad cl'_1 \rightarrow cl'_2 = inst(\sigma) \qquad H \vdash e : cl'_1}{H \vdash N(e) : cl'_2} \qquad \text{(EQ)}\ \frac{H \vdash x : cl \qquad H \vdash e : cl}{H \vdash x = e}$$

$$\text{(EQS)}\ \frac{H \vdash eq_1 \qquad H \vdash eq_2}{H \vdash eq_1; eq_2}$$

$$\text{(NODE)}\ \frac{in : cl_1, out : cl_2, var : cl_3, H \vdash eqs}{H \vdash \texttt{ node } N(in) \texttt{ returns } (out) \texttt{ [var } var; \texttt{] let } eqs \texttt{ tel } : gen_H(cl_1 \rightarrow cl_2)}$$

$$\text{(IMP-NODE)}\ \frac{in : cl_1 \qquad out : cl_2 \qquad pparent(cl_1) = pparent(cl_2)}{H \vdash \texttt{ imported node } N(in) \texttt{ returns } (out) \texttt{ wcet } n : gen_H(cl_1 \rightarrow cl_2)}$$

FIG. 19 – Règles d'inférence d'horloge

4.3 Unification d'horloges

Equivalence d'expressions d'horloge strictement périodique

Les transformations périodiques d'horloges sont telles que différentes expressions d'horloges strictement périodiques peuvent correspondre à la même horloge. L'unification d'horloges nécessite donc de définir un test d'équivalence sur les expressions d'horloges strictement périodiques. Ce test repose sur un système de réécriture de termes. Nous montrons que le test d'équivalence de deux expressions horloges strictement périodiques peut se réduire à tester l'égalité des formes normales des expressions dans ce système de réécriture. La terminologie et les preuves de cette section se basent sur [BN98].

Soit $\mathcal{E}^{\mathcal{P}}$ l'ensemble des expressions sur horloges strictement périodiques. Cet ensemble correspond à l'ensemble des types d'horloges pck définis dans la grammaire de la Sect. 4.1. Les propriétés arithmétique suivantes se déduisent des définitions des transformations périodiques d'horloges :

Propriété 4. $\forall \alpha \in \mathcal{P}$:
- $\forall k, k' \in \mathbb{N}^{+*}, \alpha *_. k *_. k' = \alpha *_. kk'$;
- $\forall k, k' \in \mathbb{N}^{+*}, \alpha /_. k /_. k' = \alpha /_. kk'$;
- $\forall q, q' \in \mathbb{Q}, \alpha \rightarrow_. q \rightarrow_. q' = \alpha \rightarrow_. (q + q')$;
- $\forall k, k' \in \mathbb{N}^{+*}, \alpha /_. k *_. k' = \alpha *_. k' /_. k$;
- $\forall q \in \mathbb{Q}, \forall k \in \mathbb{N}^{+*}, \alpha \rightarrow_. q *_. k = \alpha *_. k \rightarrow_. qk$;
- $\forall q \in \mathbb{Q}, \forall k \in \mathbb{N}^{+*}, \alpha \rightarrow_. q /_. k = \alpha /_. k \rightarrow_. (q/k)$.

On définit ensuite un système de réécriture $\mathcal{R}_{\mathcal{P}}$ permettant de simplifier les expressions de $\mathcal{E}^{\mathcal{P}}$ basé sur ces propriétés :

Définition 7. *Le système de réécriture $\mathcal{R}_{\mathcal{P}}$ sur les expressions de $\mathcal{E}^{\mathcal{P}}$ se définit comme suit :*

$$\alpha *_. k *_. k' \mapsto \alpha *_. kk' \quad (1)$$
$$\alpha /_. k /_. k' \mapsto \alpha /_. kk' \quad (2)$$
$$\alpha \rightarrow_. q \rightarrow_. q' \mapsto \alpha \rightarrow_. (q + q') \quad (3)$$
$$\alpha /_. k *_. k' \mapsto \alpha *_. k' /_. k \quad (4)$$
$$\alpha \rightarrow_. q *_. k \mapsto \alpha *_. k \rightarrow_. qk \quad (5)$$
$$\alpha \rightarrow_. q /_. k \mapsto \alpha /_. k \rightarrow_. (q/k) \quad (6)$$

Propriété 5. *Le système $\mathcal{R}_{\mathcal{P}}$ est convergent.*

$\mathcal{R}_{\mathcal{P}}$ étant convergent, toute expression e de $\mathcal{E}^{\mathcal{P}}$ possède une unique forme normale, notée $NF_{\mathcal{R}_{\mathcal{P}}}(e)$. De plus :

Lemme 1. *Pour toute expression $e \in \mathcal{E}^{\mathcal{P}}$, $NF_{\mathcal{R}_{\mathcal{P}}}(e)$ est de la forme :*

$$NF_{\mathcal{R}_{\mathcal{P}}}(e) = \alpha *_. k /_. k' \rightarrow_. q$$

avec $k, k' \in \mathbb{N}$ and $q \in \mathbb{Q}$

Dans certains cas, il est possible de simplifier à nouveau la forme normale d'une expression d'horloge strictement périodique. On définit la forme canonique d'une expression d'horloge strictement périodique à partir de la propriété suivante :

Propriété 6. *Pour tout m, k, k' dans $\mathbb{N}^*$, pour tout q dans $\mathbb{Q}$, si $m|k$ et $m|k'$, alors :*

$$\alpha *_. k /_. k' \rightarrow_. q = \alpha *_. (k/m) /_. (k'/m) \rightarrow_. q$$

On en déduit la relation d'équivalence sur les expressions sous forme normale :

Propriété 7. *Soit* e_1, e_2 *des expressions d'horloges strictement périodiques sous forme normale, avec* $e_1 = \alpha *_. k_0 /_. k_1 \rightarrow_. q$ *et* $e_2 = \alpha *_. k'_0 /_. k'_1 \rightarrow_. q'$*. Alors* e_1 *et* e_2 *sont équivalentes si et seulement si :*

$$q = q' \wedge k'_0 * k_1 = k'_1 * k_0$$

Définition 8. *De cette relation d'équivalence, pour toute expression* $e = \alpha *_. k /_. k' \rightarrow_. q$*, l'élément représentatif* $e_{rep} = \alpha *_. k_r /_. k'_r \rightarrow_. q_r$ *de sa classe d'équivalence* $[e]$ *est tel que :*

$$q = q_r \wedge \exists g \in \mathbb{N}^*, k = gk_r, k' = gk'_r$$

Définition 9. *La forme canonique* $CF(e)$ *d'une expression d'horloge strictement périodique* e *sous forme normale est le représentant de sa classe d'équivalence. En d'autres termes :*

$$CF(\alpha *_. k /_. k' \rightarrow_. q) = \alpha *_. \frac{k}{gcd(k,k')} /_. \frac{k'}{gcd(k,k')} \rightarrow_. q$$

On obtient finalement le théorème suivant par construction :

Théorème 1. *Soit* e_1*,* e_2 *des expressions de* $\mathcal{E}^{\mathcal{P}}$*. Alors :*

$$(e_1 \Leftrightarrow e_2) \Leftrightarrow CF(NF_{\mathcal{R}_{\mathcal{P}}}(e_1)) = CF(NF_{\mathcal{R}_{\mathcal{P}}}(e_2))$$

Règles d'unification

Pour toute expression d'horloge ck, on note $concrete(ck)$ le prédicat retournant vrai si ck est une horloge concrète et $abstract(ck)$ le prédicat retournant vrai si ck est une horloge abstraite. Les règles d'unification, qui reposent en partie sur le test d'équivalence défini dans la section précédente, sont données Fig. 20.

$$(\rightarrow)\ \frac{cl_1 = cl'_1 \qquad cl_2 = cl'_2}{cl_1 \rightarrow cl_2 = cl'_1 \rightarrow cl'_2} \qquad (\times)\ \frac{cl_1 = cl'_1 \qquad cl_2 = cl'_2}{cl_1 \times cl_2 = cl'_1 \times cl'_2} \qquad (\text{ON})\ \frac{ck_1 = ck_2 \qquad c_1 = c_2}{ck_1 \texttt{ on } c_1 = ck_2 \texttt{ on } c_2}$$

$$(\text{VAR})\ \frac{\alpha \in FTV(pck) \qquad \alpha <: \mathcal{P}_{k,q} \cap \mathcal{P}_{k',q'} \qquad \alpha = CF(NF(pck))}{\alpha <: \mathcal{P}_{k,q} = ck <: \mathcal{P}_{k',q'}}$$

$$(\text{VAR'})\ \frac{\alpha \notin FTV(ck) \qquad ck <: \mathcal{P}_{k,q} \cap \mathcal{P}_{k',q'} \qquad \alpha \mapsto NF(ck)}{\alpha <: \mathcal{P}_{k,q} = ck <: \mathcal{P}_{k',q'}}$$

$$(\text{CONST})\ \frac{concrete(pck_1) \qquad concrete(pck_2) \qquad \pi(pck_1) = \pi(pck_2) \qquad \varphi(pck_1) = \varphi(pck_2)}{pck_1 = pck_2}$$

$$(*_.)\ \frac{abstract(pck) \qquad pck = pck' /_. k}{pck *_. k = pck'} \qquad (/_.)\ \frac{abstract(pck) \qquad pck = pck' *_. k \qquad pck' <: \mathcal{P}_{k,0}}{pck /_. k = pck'}$$

$$(\rightarrow_.)\ \frac{abstract(pck) \qquad pck = pck' \rightarrow_. (-\frac{a}{b}) \qquad pck' <: \mathcal{P}_{b,\frac{a}{b}}}{pck \rightarrow_. \frac{a}{b} = pck'}$$

FIG. 20 – Règles d'unification d'horloge

4.4 Subsomption

A plusieurs endroits, les règles d'inférence et d'unification nécessitent de vérifier qu'une expression d'horloge appartient à un certain sous-type. Afin de vérifier que ck est un sous-type de C_i (ie que ck subsume C_i), on utilise les propriétés suivantes :

Propriété 8. $\forall \alpha \in \mathcal{P}, \forall k, k' \in \mathbb{N}^{+*}, \forall q, q' \in \mathbb{Q}$:
- $\alpha *. k' <: \mathcal{P}_{k,q} \Leftrightarrow \alpha <: \mathcal{P}_{k*k',q/k'}$;
- $\alpha /. k' <: \mathcal{P}_{k,q} \Leftrightarrow \alpha <: \mathcal{P}_{k/gcd(k,k'),q*k'}$;
- $\alpha \rightarrow. q' <: \mathcal{P}_{k,q} \Leftrightarrow \alpha <: \mathcal{P}_{k,max(0,(q-q'))}$;
- $\alpha <: \mathcal{P}_{k,q} \wedge \alpha <: \mathcal{P}_{k',q'} \Leftrightarrow \alpha <: \mathcal{P}_{k,q} \cap \mathcal{P}_{k',q'} \Leftrightarrow \alpha <: \mathcal{P}_{lcm(k,k'),max(q,q')}$.

Lues de gauche à droite, ces propriétés forment des règles de subsomption pouvant être appliquées pour transférer des contraintes de sous-typage sur un type d'horloge vers la variable d'horloge apparaissant dans ce type (pour une horloge abstraite) ou vers l'horloge constante apparaissant dans ce type (pour une horloge abstraite). Dans le premier cas, la contrainte est simplement ajoutée aux contraintes portant déjà sur cette variable. Dans le second cas, on peut directement vérifier que l'horloge contrainte satisfait cette contrainte.

Nous allons maintenant expliquer comment, une fois les analyses statiques réalisées, un programme du langage est compilé en un programme bas-niveau.

5 Mode de compilation

Nous présentons ici le mode de compilation d'un programme vers un code bas-niveau. Ce code pourra ensuite être traduit en un code exécutable à l'aide d'un compilateur existant. Le processus de compilation est défini formellement, ce qui assure que le code généré se comporte comme spécifié par sa sémantique formelle.

5.1 Limitation de la génération de code classique en « boucle simple »

Comme pour les autres langages synchrones, notre compilateur traduit le programme d'entrée en un code de niveau intermédiaire (code C). Cependant, le mode de compilation diffère de la compilation classique de code séquentiel en « boucle simple », afin de mieux gérer les systèmes multi-périodiques. Considérons l'exemple ci-dessous :

```
imported node A(i: int) returns (o: int) wcet 1;
imported node B(i: int) returns (o: int) wcet 5;

node multi(i: rate(3, 0)) returns (o)
var v;
let
  v=A(i);
  o=B(v/^3);
tel
```

Comme indiqué Fig. 21, une exécution séquentielle du programme ne peut pas respecter toutes les contraintes d'échéance du programme, la seconde répétition de l'opération `A` rate son échéance (à la date 6).

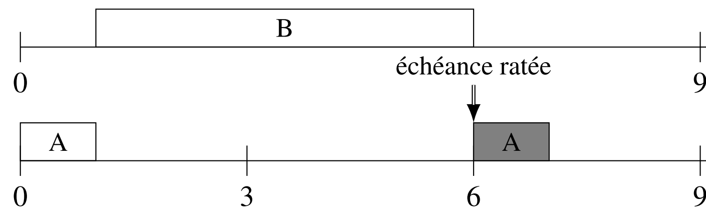


FIG. 21 – Exécution séquentielle du nœud `multi`

5.2 Génération de code multi-tâche

Si les opérations A et B sont implémentées comme deux tâches séparées et exécutées de manière concurrente par un système d'exploitation temps réel (c'est-à-dire avec préemption), il est possible de trouver un ordonnancement pour l'exécution des tâches respectant les contraintes de périodicité des deux opérations.

Chaque opération (chaque appel de nœud importé) est traduit en une tâche τ_i, possédant un ensemble d'attributs temps réel (T_i, C_i, r_i, d_i). τ_i est instanciée périodiquement avec la période T_i, $\tau_i[j]$ représente la $j^{ème}$ itération de la tâche τ_i. Une instance de tâche ne peut pas commencer son exécution avant que tous ses prédécesseurs, définis par les dépendances de données du programme, aient terminé leur exécution. C_i est le wcet de la tâche. r_i est la date de réveil de la première instance de la tâche. Les dates de réveil des instances suivantes sont r_i+T_i, r_i+2T_i, etc. d_i est l'échéance relative de la tâche. L'échéance absolue $D_i[j]$ de l'instance j de la tâche τ_i est égale à la date de réveil $R_i[j]$ de cette instance plus l'échéance relative de la tâche : $D_i[j] = R_i[j] + d_i$. Ces définitions sont illustrées Fig. 22.

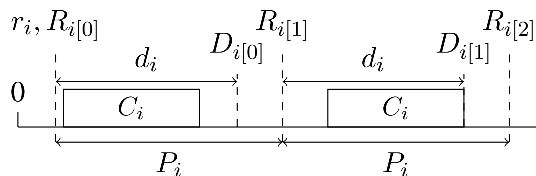


FIG. 22 – Caractéristiques temps réel de τ_i

Notre exemple se traduit donc en deux tâches τ_A d'attributs $(3, 1, 0, 3)$ et τ_B d'attributs $(9, 5, 0, 9)$, avec une précédence de τ_A vers τ_B. A l'aide de préemptions, il est ensuite possible de trouver un ordonnancement respectant les contraintes des deux tâches, comme indiqué Fig. 23 (les croix sur l'axe de temps de B représentent les préemptions/restaurations de tâches).

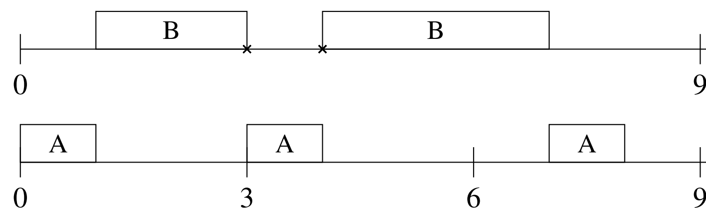


FIG. 23 – Ordonnancement préemptif du nœud `multi`

6 Extraction de tâches dépendantes

Une fois les analyses statiques terminées, le compilateur traduit donc le programme en un code bas-niveau constitué d'un ensemble de tâches temps réel concurrentes. La première étape de la traduction consiste à extraire le graphe de tâches correspondant au programme.

6.1 Expansion de programme

Un programme est constitué d'un hiérarchie de nœuds, dont les feuilles sont soit des opérateurs prédéfinis, soit des nœuds importés. Le processus de génération de code commence par remplacer récursivement chaque appel de nœud par l'ensemble d'équations de sa définition. Par exemple, le nœud suivant :

```
imported node A(i: int) returns (o: int) wcet 5;
imported node B(i: int) returns (o: int) wcet 10;

node sub (i, j) returns (o, p)
let
  o=A(i);
  p=B(j);
tel

node toexpand (i, j: rate (20, 0)) returns (o, p)
let
  (o, p)=sub(i, j);
tel
```

S'expanse en :

```
imported node A(i: int) returns (o: int) wcet 5;
imported node B(i: int) returns (o: int) wcet 10;

node expanded (i, j: rate (20, 0)) returns (o, p)
  var isub, jsub, osub, psub;
let
  isub=i; jsub=j;
  osub=A(isub);
  psub=B(jsub);
  (o, p)=(osub, psub);
tel
```

L'algorithme d'expansion est décrit dans l'Alg. 1.

6.2 Graphe de tâches

Le programme expansé est tout d'abord traduit en un *graphe de tâches*. Un graphe de tâches est constitué d'un ensemble de tâches et d'un ensemble de précédences entre les tâches.

Algorithm 1 Expansion du contenu du nœud N

```
1: for chaque instance de nœud M(args) dans le corps de N do
2:     for chaque argument arg de args do
3:         ajouter une variable locale v à N
4:         ajouter une équation v = arg à N
5:     end for
6:     for chaque variable locale ou chaque variable de sortie de la définition deM do
7:         ajouter une variable locale à N
8:     end for
9:     for chaque équation de la définition de M do
10:        ajouter l'équation à l'ensemble des équations de N
11:    end for
12:    remplacer l'instance de nœud dans N par le tuple constitué par ses sorties
13: end for
```

Définition

Soit $g = (V, E)$ un graphe de tâches, où V est l'ensemble des tâches du graphe et E l'ensemble des précédences du graphe (un sous-ensemble de $V \times V$). Les précédences sont annotées par des opérateurs prédéfinis du langage, qui décrivent le motif de communication utilisé pour communiquer. L'union de deux graphes $g = (V, E)$ et $g' = (V', E')$ est défini par $g \cup g' = (V \cup V', E \cup E')$. $last(g)$ représente l'ensemble des tâches de g n'ayant pas de successeurs, c'est-à-dire $last(V, E) = \{v \in V | \neg(\exists v' \in V, v \rightarrow v')\}$.

Chaque nœud importé du programme est traduit en une tâche. La fonction exécutée par la tâche est la fonction externe fournie pour le nœud importé. Cette fonction doit s'exécuter séquentiellement et ne pas contenir de point de synchronisation. Dans un premier temps, chaque variable du programme expansé et chaque appel à un opérateur prédéfini est aussi transformé en un sommet du graphe. Ces sommets particuliers seront ensuite réduits pour simplifier le graphe.

Une tâche doit s'exécuter après toutes les tâches produisant des données nécessaires à son exécution. Les dépendances de données entre tâches définissent ainsi des contraintes de précédence entre tâches. Ces précédentes sont déduites de la relation de dépendance entre les expressions du programme. On étend la notion de dépendance *syntaxique* de [HRR91] à notre langage. On note $e \prec e'$ lorsque e' dépend syntaxiquement de e. Par exemple, l'expression $N(e, e')$ dépend de e et e'. L'expression e `when` c dépend aussi syntaxiquement de e et c, bien qu'elle ne dépende *sémantiquement* de e que lorsque c vaut vrai.

La relation de dépendance syntaxique est surchargée pour les équations et définie Fig. 24.

Graphe intermédiaire

On extrait d'abord un graphe intermédiaire (contenant des variables et des opérateurs prédéfinis dans ses sommets) à partir d'un programme à l'aide de la fonction $\mathcal{G}(e)$ définie Fig. 25.

Prenons l'exemple suivant :

```
node sampling(i: rate (10, 0)) returns (o)
  var vf, vs;
let
  (o, vf)=F(i, (0 fby vs)*^3);
  vs=S(vf/^3);
```

$$
\begin{array}{lcl}
e & \prec & (e, e') \\
e' & \prec & (e, e') \\
e & \prec & (cst\ \texttt{fby}\ e) \\
e, c & \prec & e\ \texttt{when}\ c \\
e, c & \prec & e\ \texttt{whennot}\ c \\
c, e_1, e_2 & \prec & \texttt{merge}\ (c, e_1, e_2) \\
e & \prec & (e\ op\ k)\ \text{(avec } op \in \{/\hat{}, *\hat{}, \sim>\}\text{)} \\
e & \prec & (x = e) \\
(x = e) & \prec & x \\
e & \prec & (N(e))
\end{array}
$$

FIG. 24 – Dépendances entre expressions

$$
\begin{array}{lcl}
\mathcal{G}(cst) & = & (\emptyset, \emptyset) \\
\mathcal{G}(x) & = & (\{x\}, \emptyset) \\
\mathcal{G}(N(e)) & = & \mathcal{G}(e) \cup (\{N\}, \{l \rightarrow N, \forall l \in last(\mathcal{G}(e)) \\
\mathcal{G}(op(e)) & = & \mathcal{G}(e) \cup (\{op\}, \{l \rightarrow op, \forall l \in last(\mathcal{G}(e)) \\
 & & \text{(avec } op \in \{*\hat{}k,\ /\hat{}k,\ \sim> q, \texttt{tail}, cst\ \texttt{::}, cst\ \texttt{fby}\} \\
\mathcal{G}(e\ \texttt{when}\ c) & = & \mathcal{G}(e) \cup (\{c,\ \texttt{when}\ \}, \{c \rightarrow \texttt{when}, l \rightarrow \texttt{when}, \forall l \in last(\mathcal{G}(e))\} \\
\mathcal{G}(e\ \texttt{whennot}\ c) & = & \mathcal{G}(e) \cup \\
 & & (\{c,\ \texttt{whennot}\ \}, \{c \rightarrow \texttt{whennot}, l \rightarrow \texttt{whennot}, \forall l \in last(\mathcal{G}(e))\} \\
\mathcal{G}(\texttt{merge}\ (c, e_1, e_2)) & = & \mathcal{G}(e_1) \cup \mathcal{G}(e_2) \cup \\
 & & (\{c, \texttt{merge}\}, c \rightarrow \texttt{merge}, l \rightarrow \texttt{merge}, \forall l \in last(\mathcal{G}(e_1) \cup \mathcal{G}(e_2)) \\
\mathcal{G}(x = e) & = & \mathcal{G}(e) \cup (\{x\}, \{l \rightarrow x, \forall l \in last(\mathcal{G}(e)\}) \\
\mathcal{G}(eq; eq') & = & \mathcal{G}(eq) \cup \mathcal{G}(eq')
\end{array}
$$

FIG. 25 – Extraction d'un graphe de tâche intermédiaire

tel

Le graphe obtenu par la fonction $\mathcal{G}$ est donné Fig. 26.

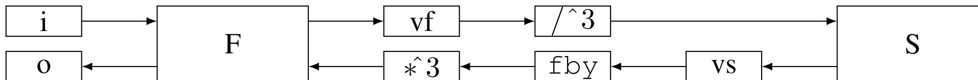


FIG. 26 – Un graphe intermédiaire

Réduction du graphe

Une suite de simplifications est ensuite appliquée sur ce graphe intermédiaire. Tout d'abord, chaque variable d'entrée du programme devient une opération capteur et chaque variable de sortie du programme devient une opération actionneur.

Ensuite chaque paire de précédence $N \rightarrow x \rightarrow M$, pour laquelle x est une variable locale, est remplacée par une seule précédence $N \rightarrow M$. Lorsque toutes les paires ont été remplacées, les variables sont supprimées du graphe. Pour l'exemple précédent, on obtient le graphe Fig. 27.

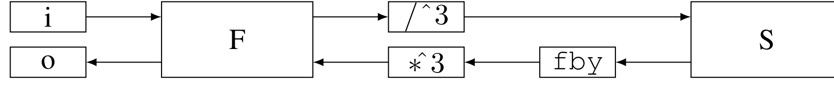


FIG. 27 – Graphe après réduction des variables

Les opérateurs prédéfinis du graphe sont ensuite transformés en *annotations de précédences*. Une précédence $\tau_i \stackrel{ops}{\rightarrow} \tau_j$ représente une *précédence étendue*, l'annotation *ops* est une liste d'opérateurs *op*, avec $op \in \{/\hat{}\,k, *\hat{}\,k, \sim> q, \texttt{tail}, cst\ \texttt{::}, cst\ \texttt{fby}, \texttt{when}\ c, \texttt{whennot}\ c\}$. L'annotation spécifie que la précédence relie des tâches de rythmes ou phases différents ($*\hat{}\,k, /\hat{}\,k, \sim> q$), est retardée ($cst$ `fby`) ou conditionnée (`when` c). Dans la suite, *op.ops* représente la liste dont la tête est *op* et la queue est *ops* et ϵ représente la liste vide.

Le graphe réduit de l'exemple précédent est donné Fig. 28.

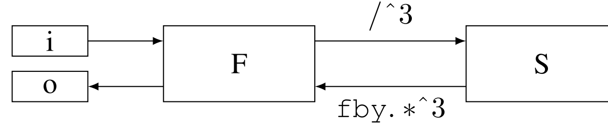


FIG. 28 – Un graphe de tâches réduit

6.3 Caractéristiques temps réel

Pour chaque tâche τ_i, nous devons extraire les caractéristiques temps réel (T_i, C_i, r_i, d_i). Le calcul d'horloges assure que les entrées et sorties d'un nœud importé ont toutes la même horloge strictement périodique parente. Par extension, on note pck_i l'horloge strictement périodique parente de τ_i. Les caractéristiques temps réel d'une tâche sont extraites de la manière suivante d'un programme :

Périodes La période d'une tâche se déduit de son horloge strictement périodique parente. On a $T_i = \pi(pck_i)$.

Echéances Par défaut, l'échéance d'une tâche est sa période ($d_i = T_i$). Des contraintes d'échéance peuvent aussi être spécifiées sur les capteurs ou actionneurs (ex : `o: due n`).

Dates de réveil La date de réveil initiale d'une tâche est la phase de son horloge strictement périodique parente : $r_i = \varphi(pck_i)$.

Durée d'exécution La durée d'exécution C_i d'une tâche est le wcet du nœud importé, capteur ou actionneur correspondant.

6.4 Conditions d'activation

La condition d'activation d'une tâche dépend des conditions d'activation de ses entrées/sorties. A chaque instance de tâche on teste la condition d'activation de la tâche. Si la condition est fausse, l'instance termine immédiatement, sinon elle se comporte normalement.

Définition 10. *La condition d'activation $cond(ck)$ d'une horloge ck se définit comme suit :*

$$cond(ck\ \texttt{on}\ c) = cond(ck)\ \texttt{and}\ c$$
$$cond(pck) = \texttt{true}$$

La condition d'activation d'une variable est la condition d'activation de son horloge. La condition d'activation d'une tâche se définit ensuite de la manière suivante :

Définition 11. *La condition d'activation $cond_i$ de τ_i est :*

$$cond_i = \bigvee_{x \in in_i} cond(x)$$

7 Traduction de tâches dépendantes en tâches indépendantes

Les tâches du graphe de tâche décrit précédemment sont dépendantes, car liées par des contraintes de précédences. Dans ce chapitre nous montrons comment traduire cet ensemble de tâches dépendantes en un ensemble de tâches indépendantes. Les précédences sont tout d'abord encodées par ajustement des caractéristiques temps réel des tâches. Un protocole de communication non bloquant basé sur des mémoires tampons assure ensuite la conservation de la sémantique formelle du programme transformé.

7.1 Encodage des précédences

Ajustement des attributs temps réel

Précédences simples La technique d'encodage sur laquelle nous nous basons est définie dans [CSB90]. Elle consiste à modifier les dates de réveil et les échéances des tâches de manière à refléter les contraintes de précédences, puis à utiliser EDF pour ordonnancer l'ensemble des tâches encodées. Cette technique est optimale, dans le sens où s'il existe un ordonnancement à priorité dynamique respectant les contraintes de l'ensemble de tâches d'origine, alors EDF appliqué à l'ensemble encodé trouvera un ordonnancement valide. Cette technique est initialement prévue pour des tâches apériodiques reliées par des contraintes de précédence. Soit $succ(\tau_i)$ les successeurs de τ_i et $pred(\tau_i)$ ses prédécesseurs. Les précédences peuvent être encodées de la manière suivante :

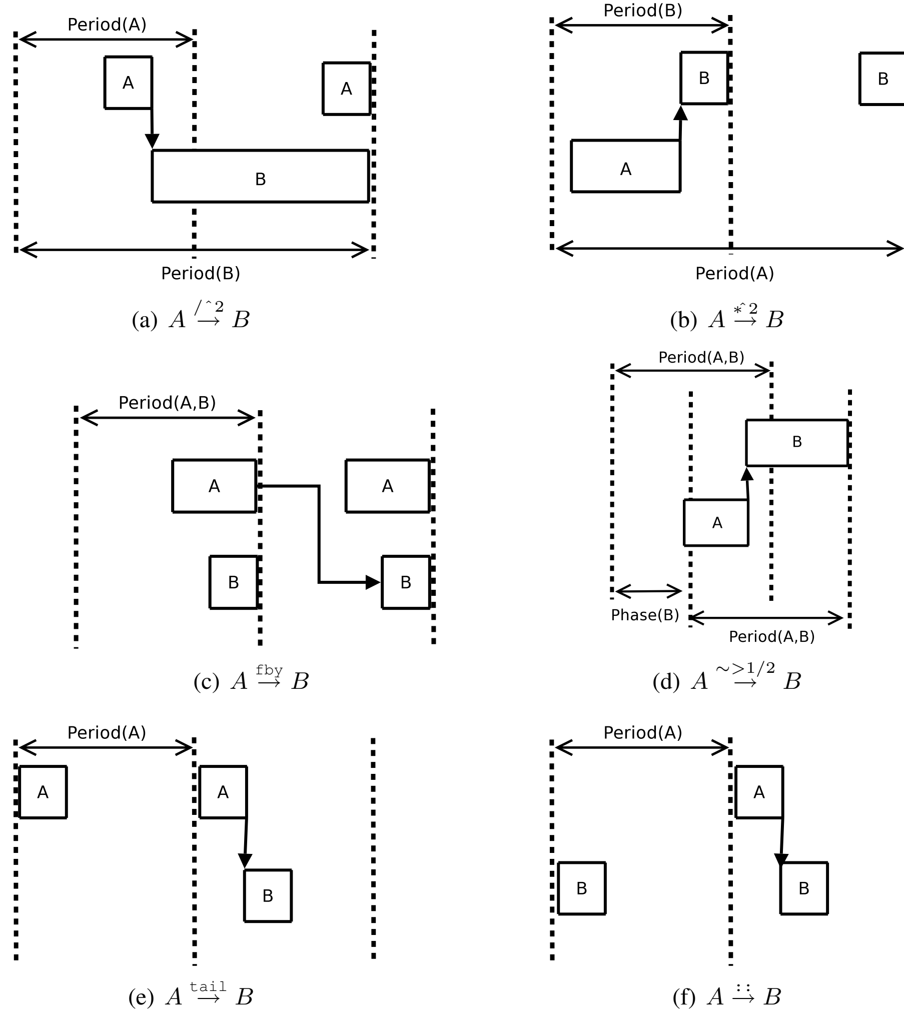
- Une tâche τ_i doit se terminer suffisamment tôt pour que ses successeurs puissent respecter leurs échéances, donc l'échéance absolue ajustée d'une tâche D_i^* est : $D_i^* = min(D_i, min(D_j^* - C_j))$ (pour tout $\tau_j \in succ(\tau_i)$) ;
- L'ajustement des dates de réveil proposé dans [CSB90] est : $R_i^* = max(R_i, max(R_j^* + C_j))$ (pour tout $\tau_j \in pred(\tau_i)$). Ceci peut se simplifier dans notre contexte en : $R_i^{*'} = max(R_i, max(R_j^{*'}))$ (pour tout $\tau_j \in pred(\tau_i)$). En effet, cette condition est suffisante pour qu'EDF affecte une priorité à la source d'une précédence supérieure à celle de la destination de la précédence.

Cette technique s'étend directement à des précédences entre tâches périodiques ayant la même période, que nous qualifierons de *précédences simples*.

Précédences étendues Dans notre cas, un graphe de tâches contient des précédences étendues entre tâches de rythmes différents. Comme illustré Fig. 29, l'encodage des précédences étendues nécessite d'ajuster les attributs d'une tâche de manière différente entre ses différentes instances. La suite de cette section étend donc la technique d'encodage au cas des précédences étendues

Précédences entre instances de tâches

Le principe de notre extension est que les précédences étendues correspondent à des motifs de précédences réguliers. Notre extension tient donc compte de cette régularité afin d'effectuer l'encodage de manière compacte et factorisée.



(a) $A \overset{/\hat{}2}{\rightarrow} B$ (b) $A \overset{*\hat{}2}{\rightarrow} B$

(c) $A \overset{\texttt{fby}}{\rightarrow} B$ (d) $A \overset{\sim>1/2}{\rightarrow} B$

(e) $A \overset{\texttt{tail}}{\rightarrow} B$ (f) $A \overset{::}{\rightarrow} B$

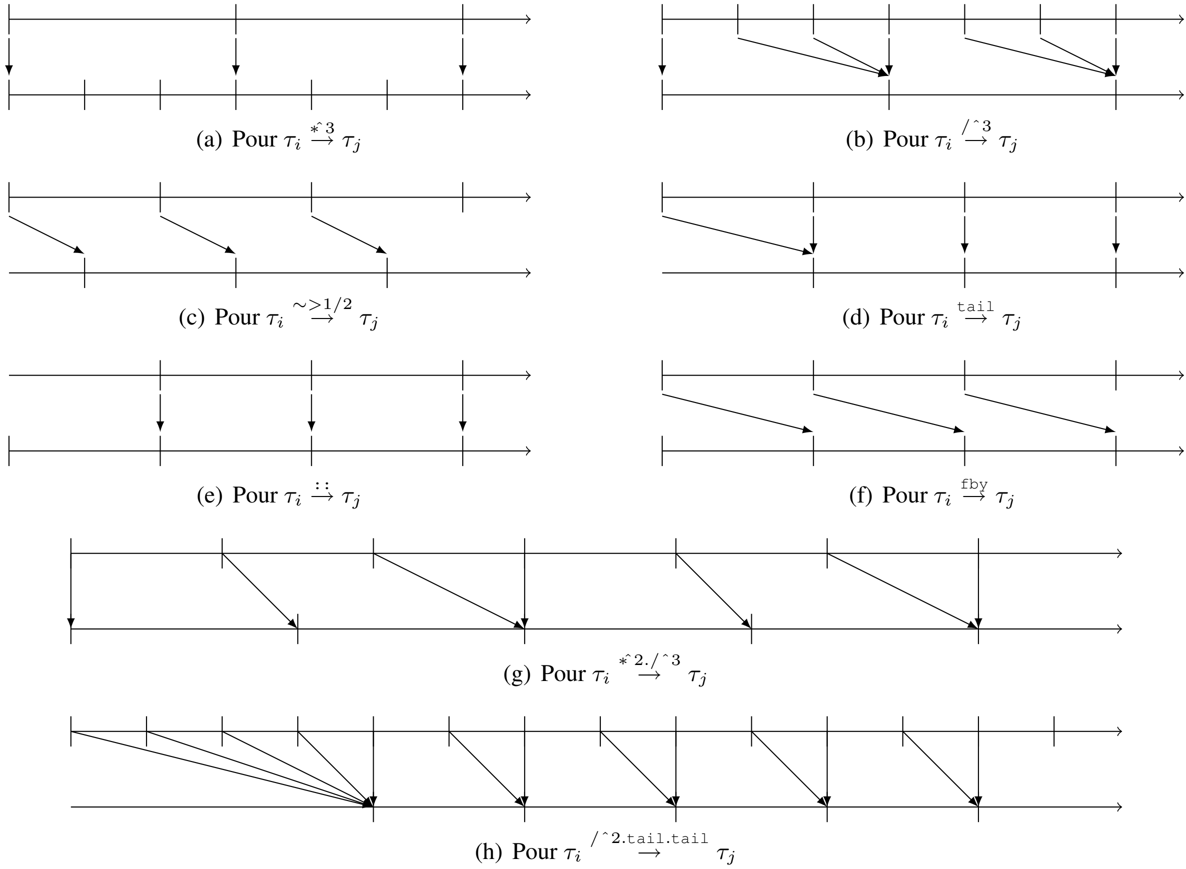
FIG. 29 – Encodage des précédences étendues

Nous commençons par définir l'ensemble des précédences entre instances de tâches : $\tau_i \overset{ops}{\rightarrow} \tau_j \Rightarrow \forall n,\ \tau_i[n] \rightarrow \tau_j[g_{ops}(n)]$, avec g_{ops} défini comme suit :

$$
\begin{aligned}
g_{*\hat{}k.ops}(n) &= g_{ops}(kn) \\
g_{/\hat{}k.ops}(n) &= g_{ops}(\lceil n/k \rceil) \\
g_{\sim>q.ops}(n) &= g_{ops}(n) \\
g_{\texttt{tail}.ops}(n) &= \begin{cases} g_{ops}(0) & \text{si n=0} \\ g_{ops}(n-1) & \text{sinon} \end{cases} \\
g_{::.ops} &= g_{ops}(n+1) \\
g_{\texttt{fby}.ops}(n) &= g_{ops}(n+1) \\
g_{\texttt{when}.ops}(n) &= g_{ops}(n) \\
g_{\texttt{whennot}.ops}(n) &= g_{ops}(n) \\
g_{\epsilon}(n) &= n
\end{aligned}
$$

Le terme $\lceil r \rceil$, avec $r \in \mathbb{Q}^+$, désigne le plus petit entier supérieur ou égal à r (par exemple 3 pour $\frac{5}{2}$). La

fonction g_{ops} est illustrée Fig. 30 où l'on précise les dates de réveil de deux tâches τ_i et τ_j reliées par une précédence $\tau_i \stackrel{\tau}{\rightarrow}_j$. Les flèches décrivent quelle instance de τ_i précède quelle instance de τ_j, d'après la définition de g_{ops}.



(a) Pour $\tau_i \stackrel{*\hat{}3}{\rightarrow} \tau_j$
(b) Pour $\tau_i \stackrel{/\hat{}3}{\rightarrow} \tau_j$
(c) Pour $\tau_i \stackrel{\sim>1/2}{\rightarrow} \tau_j$
(d) Pour $\tau_i \stackrel{\texttt{tail}}{\rightarrow} \tau_j$
(e) Pour $\tau_i \stackrel{::}{\rightarrow} \tau_j$
(f) Pour $\tau_i \stackrel{\texttt{fby}}{\rightarrow} \tau_j$
(g) Pour $\tau_i \stackrel{*\hat{}2./\hat{}3}{\rightarrow} \tau_j$
(h) Pour $\tau_i \stackrel{/\hat{}2.\texttt{tail.tail}}{\rightarrow} \tau_j$

FIG. 30 – Précédences entre instances de tâches, selon la définition de g_{ops}

On note que $\tau_i[n] \rightarrow \tau_j[n']$ n'implique pas nécessairement que la donnée produite par $\tau_j[n]$ est consommée par $\tau_i[n']$. Les dépendances de données entre instances de tâches seront définies de manière plus précise Sect. 7.3.

Ajustement des dates de réveil dans le contexte synchrone

Nous montrons dans cette section que la sémantique synchrone du langage permet de ne pas ajuster les dates de réveil des tâches.

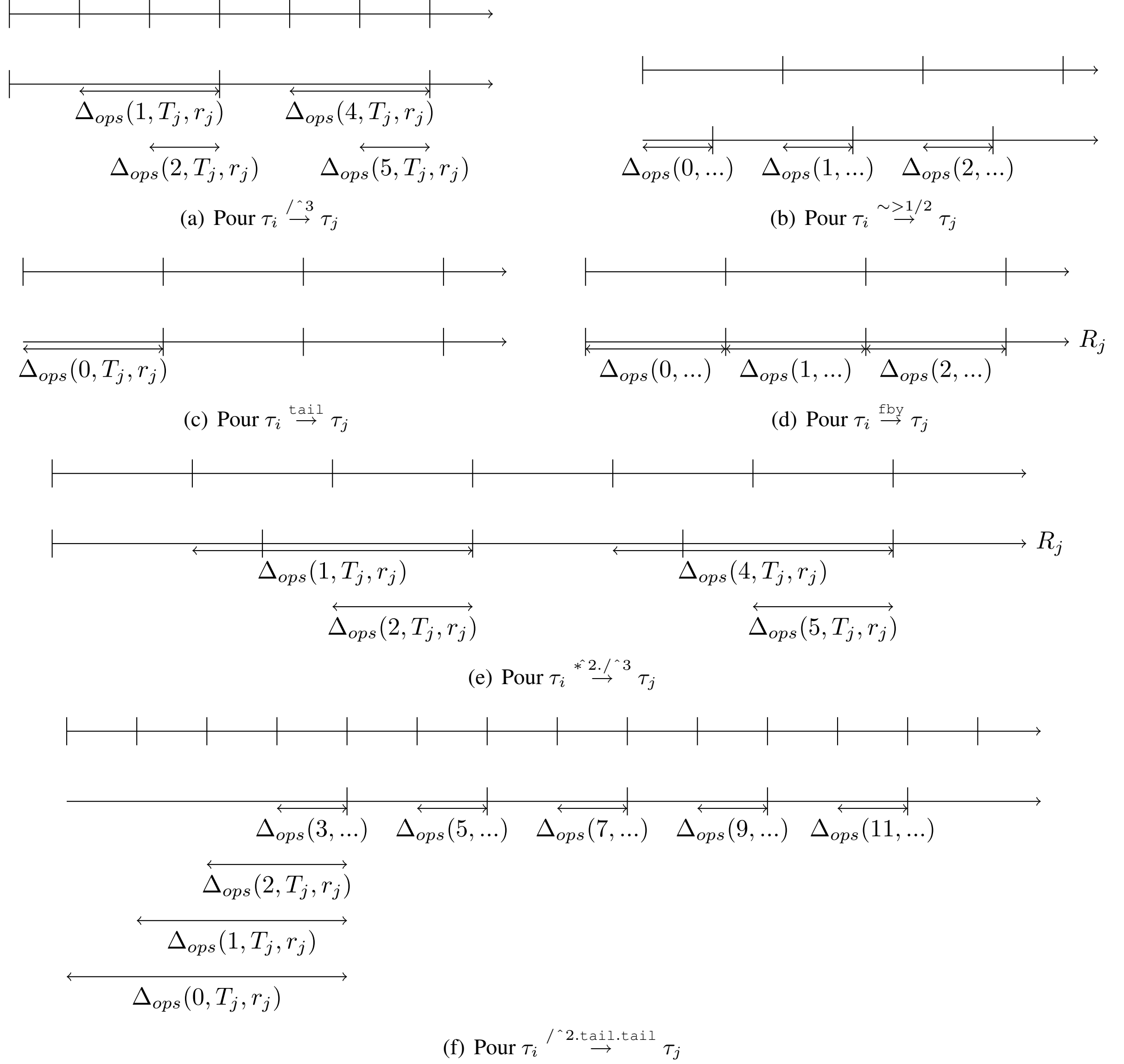
Nous définissons tout d'abord la fonction $\Delta_{ops}(n, T_i, T_j, r_i, r_j)$ qui joue un rôle essentiel dans notre technique d'encodage des précédences :

Définition 12. *Soit $\Delta_{ops}(n, T_i, T_j, r_i, r_j) = g_{ops}(n)T_j - nT_i + r_j - r_i$. D'après la définition de $R_i[n]$, on a :*

$$R_j[g_{ops}(n)] = R_i[n] + \Delta_{ops}(n, T_j, r_j)$$

Δ_{ops} représente donc la différence entre les dates de réveil de deux instances de tâches reliées par une relation de précédence. Ceci est illustré Fig.31, où nous montrons les dates de réveil des instances

successives de deux tâches τ_i et τ_j reliées par une précédence $\tau_i \stackrel{ops}{\rightarrow} \tau_j$. On ne représente pas les cas pour lesquels $\Delta_{ops}(n, T_i, T_j, r_i, r_j) = 0$.



(a) Pour $\tau_i \stackrel{/\hat{}3}{\rightarrow} \tau_j$

(b) Pour $\tau_i \stackrel{\sim>1/2}{\rightarrow} \tau_j$

(c) Pour $\tau_i \stackrel{\texttt{tail}}{\rightarrow} \tau_j$

(d) Pour $\tau_i \stackrel{\texttt{fby}}{\rightarrow} \tau_j$

(e) Pour $\tau_i \stackrel{*\hat{}2./\hat{}3}{\rightarrow} \tau_j$

(f) Pour $\tau_i \stackrel{/\hat{}2.\texttt{tail}.\texttt{tail}}{\rightarrow} \tau_j$

FIG. 31 – Illustration de Δ_{ops}

On note que la séquence des valeurs $\Delta_{ops}(n, T_i, T_j, r_i, r_j)$ est ultimement périodique (elle suit un motif répétitif après un préfixe initial). Notre technique d'ajustement repose sur cette propriété essentielle qui nous permet de représenter la séquence des échéances ajustées des instances d'une tâche sous forme d'un motif ultimement périodique.

A partir des définitions ci-dessus, il est possible de prouver la propriété suivante :

Théorème 2. *Pour toute précédence $\tau_i \stackrel{ops}{\rightarrow} \tau_j$, pour tout n :*

$$R_{\tau_j[g_{ops}(n)]} \geq R_i[n]$$

Par conséquence :

Corollaire 1. *Pour tout $\tau_j \in pred(\tau_i)$:*

$$R_i \geq max(R_i, max(R_j))$$

Ainsi, l'encodage des dates de réveil des tâches est directement assuré par les contraintes de synchronisation imposées par le calcul d'horloges. Par ailleurs, on note la propriété suivante qui nous permettra d'ignorer les précédences contenant un retard dans leurs annotations durant la phase d'encodage :

Propriété 9. *Pour une précédence $\tau_i \stackrel{ops}{\rightarrow} \tau_j$ telle que ops contient un délai, $R_j[g_{ops}(n)] \geq R_i[n] + T_i$.*

Calcul d'échéances

Nous passons maintenant à l'encodage des échéances des tâches appelé *calcul d'échéances*. Le principe de ce calcul vient de l'observation que la contrainte reliant $d_i[n]$ à $d_j[g_{ops}(n)]$ est périodique. Nous allons montrer qu'il existe toujours deux entiers b, p tels que pour tout $n > b$, $d_i^*[n] = d_i^*[n+p]$. La séquence des échéances relatives d'une tâche est donc ultimement périodique et peut se représenter sous la forme d'un motif répétitif appelé *mot d'échéances*. Le calcul d'échéances effectue l'encodage des précédences en calculant le mot d'échéances de chaque tâche du graphe.

Transposition de l'ajustement aux échéances relatives Tout d'abord, nous transposons la formule d'ajustement des échéances absolues aux échéances relatives :

Propriété 10. *Pour toute précédence $\tau_i \stackrel{ops}{\rightarrow} \tau_j$:*

$$D_i[n] \leq D_j[g_{ops}(n)] - C_j \Leftrightarrow d_i[n] \leq d_j[g_{ops}(n)] + \Delta_{ops}(n, T_i, T_j, r_i, r_j) - C_j$$

D'après la propriété 9, on peut déduire que pour une précédence $\tau_i \stackrel{ops}{\rightarrow} \tau_j$, telle que *ops* contient un retard, il n'est pas nécessaire d'ajuster $d_i[n]$ (quel que soit n).

Mots d'échéances Une *échéance unitaire* spécifie l'échéance relative d'une unique instance de tâche (un entier d). Un *mot d'échéances* définit la séquence des échéances unitaires d'une tâche. L'ensemble des mots d'échéances $\mathcal{W}_{\text{d}}$ se définit par la grammaire suivante :

$$\begin{array}{rcl} w & ::= & u.(u)^{\omega} \\ u & ::= & d \mid d.u \end{array}$$

Pour un mot $w = u.(v)^{\omega}$, u est le préfixe de w et $(v)^{\omega}$ représente la répétition infinie du mot v, appelée le motif répétitif de w.

De plus :
- $|u|$ représente la taille du mot fini u ;
- $w[n]$ représente la $n^{ème}$ échéance unitaire du mot w ($n \in \mathbb{N}$). Si $w = u.(v)^{\omega}$ alors :

$$w[n] = \begin{cases} u[n] & \text{si } n < |u| \\ v[(n - |u|) \bmod |v|] & \text{sinon} \end{cases}$$

- La relation d'ordre sur les mots d'échéances est l'ordre lexicographique ;
- Le minimum de deux mots d'échéances est le minimum point-à-point des deux mots (ce qui nécessite éventuellement de déplier les deux mots pour obtenir des mots de même taille) ;
- La somme de deux mots d'échéances est la somme point-à-point des deux mots ;
- Le mot $w = w_i + k$, avec $w_i \in \mathcal{W}_{\text{d}}$ et $k \in \mathbb{N}$ est tel que pour tout n, $w[n] = w_i[n] + k$.

Calcul des mots d'échéances Soit w_i le mot d'échéances représentant la séquence des échéances unitaires de τ_i. D'après la Propriété 10, chaque précédence $\tau_i \stackrel{ops}{\rightarrow} \tau_j$ peut s'encoder par une contrainte reliant w_i à w_j de la forme :

$$w_i \leq W_{ops}(w_j) + \Delta_{ops}(T_i, T_j, r_i, r_j) - C_j$$

avec $W_{ops}(w_j)$ le mot tel que $W_{ops}(w_j)[n] = w_j[g_{ops}(n)]$ pour tout n et $\Delta_{ops}(T_i, T_j, r_i, r_j)$ le mot tel que $\Delta_{ops}(T_i, T_j, r_i, r_j)[n] = \Delta_{ops}(n, T_i, T_j, r_i, r_j)$ pour tout n.

Il est possible de montrer à partir des définitions de Δ_{ops} et g_{ops} que ces deux mots sont bien *ultimement périodiques*. Ils peuvent donc simplement être calculés en calculant leur valeur en chaque point de leur préfixe et de leur motif répétitif. La valeur de w_i se calcule ensuite comme la combinaison de toutes les contraintes le concernant :

$$w_i = min(w_i, min(\Delta_{ops}(T_j, r_j) + W_{ops}(w_j) - C_j))(\text{pour tout } \tau_j, \tau_i \stackrel{ops}{\rightarrow} \tau_j)$$

On en déduit l'Alg. 2 permettant de calculer les mots d'échéances d'un graphe de tâches. Les précédences contenant un délai sont ignorées dans le cadre de cet algorithme.

Algorithm 2 Calcul des mots d'échéances d'un graphe de tâches

1: **for** Tout τ_i **do**
2:     **if** τ_i est un capteur ou un actionneur possédant une contrainte d'échéance **then**
3:         $w_i \leftarrow (n)^\omega$
4:     **else** $w_i \leftarrow (T_i)^\omega$
5:     **end if**
6: **end for**
7: $S \leftarrow$ Liste des tâches sans successeurs
8: **while** S n'est pas vide **do**
9:     Supprimer la tête τ_j de S
10:     **for** Chaque τ_i tel que $\tau_i \stackrel{ops}{\rightarrow} \tau_j$ et `fby` $\notin$ *ops* **do**
11:         $w_i \leftarrow min(w_i, \Delta_{ops}(T_j, r_j) + W_{ops}(w_j) - C_j)$
12:         Supprimer la précédence $\tau_i \stackrel{ops}{\rightarrow} \tau_j$ du graphe
13:         **if** τ_i n'a pas d'autre successeur **then**
14:             Insérer τ_i dans S
15:         **end if**
16:     **end for**
17: **end while**

Optimalité

Par construction, notre algorithme d'encodage respecte la propriété suivante :

Lemme 2. *Pour tout $\tau_i \stackrel{ops}{\rightarrow} \tau_j$, les mots d'échéances w_i et w_j calculés par l'Alg. 2 vérifient la propriété suivante :*

$$\forall n, n', \tau_i[n] \rightarrow \tau_j[n'] \Rightarrow w_i[n] \leq W_{ops}(w_j)[n] + \Delta_{ops}(w_j, r_j)[n] - C_j \wedge n' = g_{ops}(n)$$

Ceci, combiné aux propriétés démontrées dans [CSB90], nous permet d'obtenir le théorème suivant

Théorème 3. *Soit $g = (\{\tau_i(T_i, C_i, r_i, d_i), 1 \leq i \leq n\}, E)$ un graphe de tâches réduit. Soit ensuite $s = \{\tau'_i(T_i, C_i, r_i, w_i), 1 \leq i \leq n\}$ un ensemble de tâches tel que pour tout τ'_i, w_i est le mot d'échéances calculé par l'Alg. 2 pour τ_i. Alors :*

g est ordonnançable si et seulement si s est ordonnançable par EDF.

7.2 Analyse d'ordonnançabilité

En ce qui concerne l'analyse d'ordonnançabilité, étant donné que le graphe de tâches est ordonnançable si et seulement si l'ensemble encodé est ordonnançable par EDF, nous pouvons réutiliser les résultats existants pour EDF. On cherche donc un test d'ordonnançabilité pour un ensemble de tâches périodiques, ayant des dates de réveil différentes de 0, des échéances inférieures à leur période, ordonnancé par EDF. [LM80] donne une condition nécessaire et suffisante pour ce problème qui consiste à calculer l'ordonnancement du système sur l'intervalle de temps $[0, 2HP + max(r_i)]$ (où HP désigne l'hyperpériode du système) et vérifier que toutes les contraintes temps réel sont satisfaites sur cet intervalle.

7.3 Communications inter-tâches

Dans cette section nous proposons un protocole de communication assurant que les communications inter-tâches respectent la sémantique synchrone du programme.

Assurer la consistance des communications

Un mécanisme de communication simple à mettre en place consiste à allouer une mémoire tampon dans un espace mémoire global, le producteur de la donnée écrit dans ce tampon lors de son exécution et le consommateur de la donnée lit à partir de ce même tampon lors de son exécution. Afin d'assurer le respect de la sémantique synchrone, il faut assurer pour chaque dépendance que :

1. Le producteur écrit avant que le consommateur lise.
2. Etant donné que les tâches sont périodiques il faut aussi assurer que l'instance courante du consommateur lit avant que l'instance suivante du producteur recouvre la donnée produite par son instance précédente.

L'encodage des précédences présenté dans la section précédente assure le respect du premier point. Cependant il n'assure pas le second point. En effet, pour une précédence $\tau_i \stackrel{ops}{\rightarrow} \tau_j$, la donnée produite par $\tau_i[n]$ peut éventuellement être consommée par $\tau_j[g_{ops}(n)]$ après le début de $\tau_i[n+1]$. Ceci est illustré Fig. 32, où l'on montre l'ordonnancement de deux tâches reliées par des précédences étendues. Les croix sur les axes de temps représentent des préemptions. Une flèche de A à la date t vers B à la date t' signifie que la tâche B peut lire à la date t' la valeur produite par A à la date t.

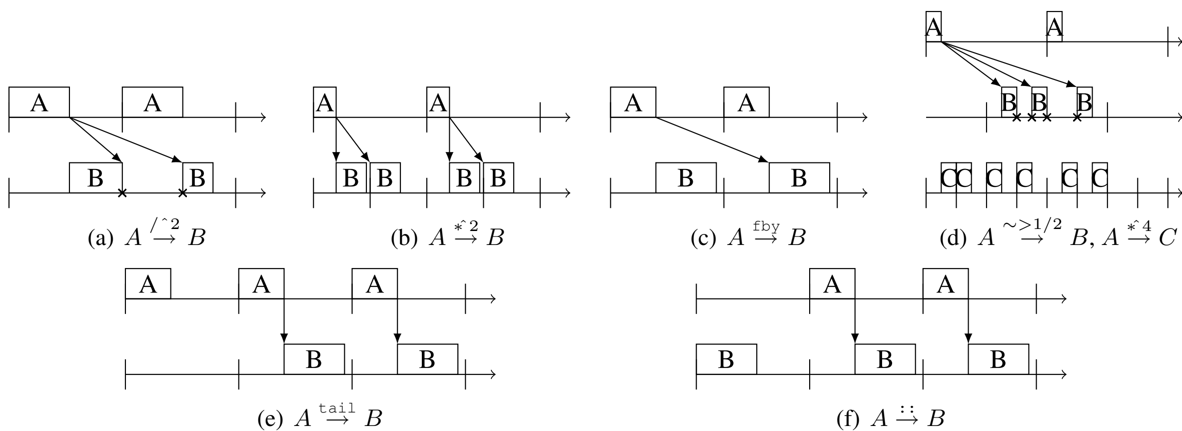


FIG. 32 – Communications pour des précédences étendues

Communications entre instances de tâches

Définition Les dépendances de données entre instances de tâches peuvent être définies comme suit :

Propriété 11. *Notons $\tau_i[n] \rightarrow_d \tau_j[n']$ une dépendance de donnée de $\tau_i[n]$ vers $\tau_j[n']$. Pour toute précédence $\tau_i \stackrel{ops}{\rightarrow} \tau_j$:*

$$\tau_i[n] \rightarrow_d \tau_j[n'] \Leftrightarrow n' \in [g_{ops}(n), g_{ops}(n+1)[$$

D'après cette définition, $\tau_i[n]$ peut donc :
- Ne pas être consommée du tout : lorsque $g_{ops}(n+1) = g_{ops}(n)$;
- Etre consommée exactement une fois : lorsque $g_{ops}(n+1) = g_{ops}(n) + 1$;
- Etre consommée n fois : lorsque $g_{ops}(n+1) = g_{ops}(n) + n$.

Mots de dépendances Tout comme pour les mots d'échéances, les dépendances de données peuvent se représenter sous forme de motifs répétitifs. On définit $conso_{ops}(n) = g_{ops}(n+1) - g_{ops}(n)$ pour tout $n \in \mathbb{N}$. Pour toute précédence, $\tau_i \stackrel{ops}{\rightarrow} \tau_j$, $conso_{ops}(n)$ représente le nombre d'instances de τ_j consommant la donnée produite par $\tau_i[n]$ (éventuellement 0). La séquence des valeurs de $conso_{ops}$ peut être représentée par des motifs répétitifs appelés *mots de dépendances*, définis de manière similaire aux mots d'échéances :

$$\begin{array}{rcl} w & ::= & u.(u)^{\omega} \\ u & ::= & i \mid i.u \end{array}$$

A nouveau, d'après la définition de g_{ops} on peut prouver que $conso_{ops}$ est ultimement périodique et peut donc être représenté sous forme de mot de dépendances.

Durées de vie La *durée de vie* d'une donnée produite par une instance de tâche est l'intervalle de temps séparant sa date de production de sa date de consommation. Plus formellement :

Définition 13. *Pour toute précédence $\tau_i \stackrel{ops}{\rightarrow} \tau_j$, pour tout n, la durée de vie de $\tau_i[n]$ est :*

$$sp_{i,j}(n) = \begin{cases} [R_i[n], R_i[n][& \textit{si } conso_{ops}(n) = 0 \\ [R_i[n], R_j[g_{ops}(n)] + conso_{ops}(n) * T_j[& \textit{sinon} \end{cases}$$

Pour un intervalle $I = [a, b[$, on note $|I| = b - a$ la taille de I. On peut de nouveau prouver que la séquence des valeurs $|sp_{i,j}(n)|$ est ultimement périodique.

Protocole de communication

Nous proposons un protocole de communication basé sur les mots de dépendances, qui assure que les entrées d'une tâche restent disponibles jusqu'à l'échéance de la tâche, afin de satisfaire l'exigence 2. de la section précédente. Ce protocole ne nécessite pas de primitives de synchronisation bloquantes (telles que des sémaphores) et repose sur des copies dans des mémoires tampons. Le protocole assure le respect de la sémantique du langage.

Un tampon de communication est alloué pour chaque précédence $\tau_i \stackrel{ops}{\rightarrow} \tau_j$. Les mots de dépendances permettent de savoir quelles instances de τ_i écrivent dans le tampon et quelles instances de τ_j lisent la valeur correspondante. Les durées de vie permettent de déterminer quelles instances peuvent écrire dans la même case du tampon.

Nombre de cases de la mémoire de communication Soit $cell_i[n]$ la case du tampon dans laquelle $\tau_i[n]$ écrit. Le nombre de cases du tampon doit être tel que pour tout n, n' tels que $sp_{i,j}(n)$ et $sp_{i,j}(n')$ ont une intersection, $cell_i[n]$ et $cell_i[n']$ peuvent être choisis de manière à ce que $cell_i[n] \neq cell_i[n']$. Pour une durée de vie donnée, il est possible de borner le nombre de données dont les durées de vie ont une intersection avec celle-ci :

Propriété 12. *Pour tout $\tau_i \overset{ops}{\rightarrow} \tau_j$, pour tout n :*

$$|\{sp_{i,j}(n'), sp_{i,j}(n') \cap sp_{i,j}(n) \neq \{\}\}| \leq \lceil sp_{i,j}(n)/T_i \rceil$$

La taille du tampon est donc déterminée en parcourant le mot ultimement périodique formé par $|sp_{i,j}(n)|$ et en prenant le nombre d'intersections maximum d'une durée de vie avec les autres durées de vie.

Le protocole Nous définissons maintenant le protocole de communication pour une précédence étendue $\tau_i \overset{ops}{\rightarrow} \tau_j$. Soit $dep_{i,j}$ le mot de dépendances de cette précédence. Soit $buf_{i,j}$ la mémoire de communication de cette précédence. On commence par calculer quelles instances de τ_i doivent écrire dans $buf_{i,j}$ à l'aide de l'Alg. 3. Le protocole de communication pour le producteur d'une donnée est ensuite décrit Alg. 4.

Algorithm 3 Calcul des instances du producteur devant écrire, pour $\tau_i \overset{ops}{\rightarrow} \tau_j$

```
1: write ← ε;
2: for k = 0 to max(0, pref(ops)) + P(ops) − 1 do
3:     if dep_{i,j}[k] ≥ 1 then
4:         write ← write.true
5:     else
6:         write ← write.false
7:     end if
8: end for
```

Algorithm 4 Protocole de communication pour $\tau_i \overset{ops}{\rightarrow} \tau_j$, côté producteur

```
1: inst ← 0; cell ← 0;
2: while true do
3:     if write[inst] then
4:         Ecrire dans buf_{i,j}[cell]
5:         cell ← (cell + 1) mod |buf_{i,j}|
6:     end if
7:     inst ← inst + 1
8: end while
```

On calcule pour chaque instance du consommateur la case où elle doit lire à l'aide de l'Alg. 5. Le protocole de communication du consommateur est décrit Alg. 6.

Exemple

Ce protocole de communication est illustré sur l'exemple de la Fig. 33 (les flèches représentent des dépendances de données). Tout d'abord, pour la précédence $\tau_i \overset{*3./\hat{}2}{\rightarrow} \tau_j$, avec $T_i = 3, T_j = 2$:

Algorithm 5 Calcul des motifs de lecture $\tau_i \overset{ops}{\rightarrow} \tau_j$

1: $change_read \leftarrow \epsilon$;
2: **for** $k = 0$ à $max(0, pref(ops)) + P(ops) - 1$ **do**
3:     **if** $dep_{i,j}[k] \geq 1$ **then**
4:         **for** $i = 0$ à $dep_{i,j}[k] - 2$ **do**
5:             $change \leftarrow change.false$
6:         **end for**
7:         $change \leftarrow change.true$
8:     **end if**
9: **end for**

Algorithm 6 Protocole de communication pour $\tau_i \overset{ops}{\rightarrow} \tau_j$, côté consommateur

1: $inst \leftarrow 0$; $cell \leftarrow 0$
2: **while** true **do**
3:     Lire à partir de $buf_{i,j}[cell]$
4:     **if** $change[inst]$ **then**
5:         $cell \leftarrow (cell + 1) \bmod |buf_{i,j}|$
6:     **end if**
7:     $inst \leftarrow inst + 1$
8: **end while**

- $dep_{i,j}(n) = (2.1)^\omega$;
- $sp_{i,j}(n) = [0,4], [3,6], [6,10], [9,12], \ldots$;
- $|buf_{i,j}| = 2$;
- $write = (true)^\omega$;
- $change = (false.true.true)^\omega$.

Donc :
- La séquence des cases dans lesquelles τ_i écrit est : 0, 1, 0, 1, 0, 1, ...
- La séquence des cases à partir desquelles τ_j lit est : 0, 0, 1, 0, 0, 1, ...

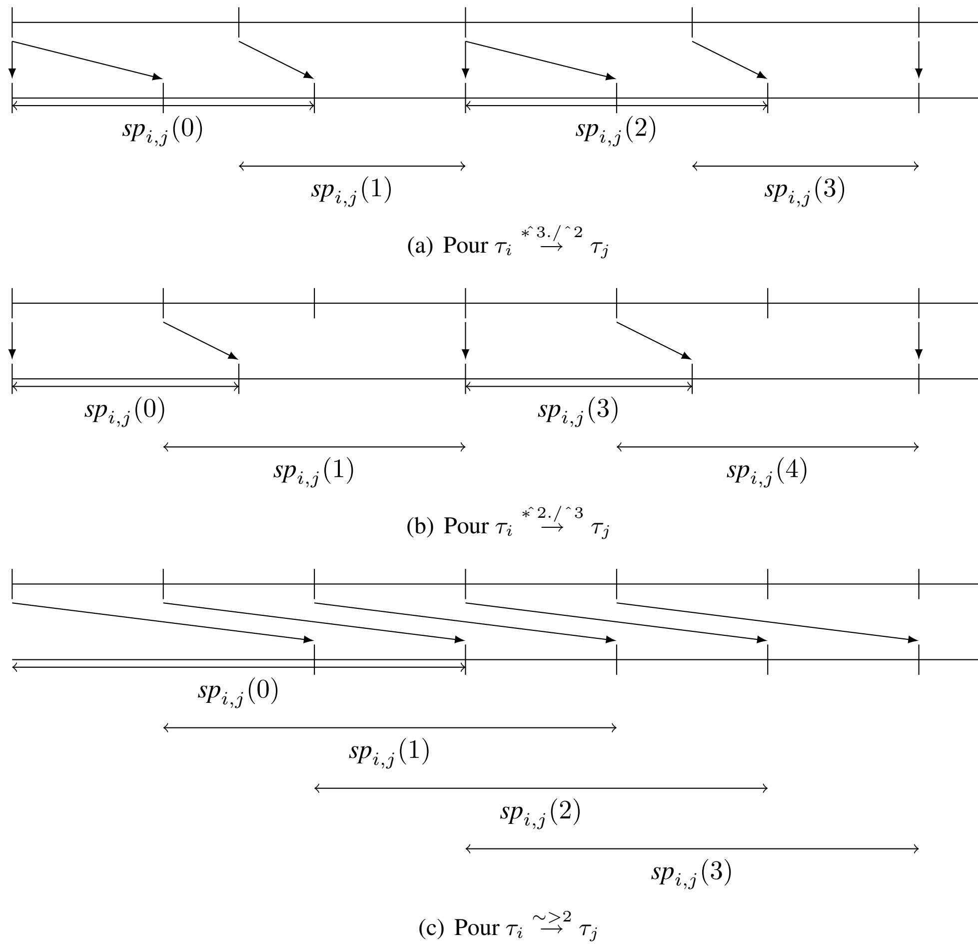
Ensuite, pour la précédence $\tau_i \overset{*\hat{}2./\hat{}3}{\rightarrow} \tau_j$, avec $T_i = 2, T_j = 3$:
- $dep_{i,j}(n) = (1.1.0)^\omega$;
- $sp_{i,j}(n) = [0,3[, [2,6[, [4,4[, [6,9[, [8,12[, \ldots$;
- $|buf_{i,j}| = 2$;
- $write = (true.true.false)^\omega$;
- $change = (true)^\omega$.

Donc :
- La séquence des cases dans lesquelles τ_i écrit est : 0, 1, none, 0, 1, none, ...
- La séquence des cases à partir desquelles τ_j lit est : 0, 1, 0, 1, 0, 1, ...

Enfin, pour la précédence $\tau_i \overset{\sim>2}{\rightarrow} \tau_j$, avec $T_i = T_j = 2$:
- $dep_{i,j}(n) = (1)^\omega$;
- $sp_{i,j}(n) = [0,6[, [2,8[, [4,10[, [6,12[, [8,14[, \ldots$;
- $|buf_{i,j}| = 3$;
- $write = (true)^\omega$;
- $change = (true)^\omega$.

Donc :
- La séquence des cases dans lesquelles τ_i écrit est : 0, 1, 2, 0, 1, 2, ...

– La séquence des cases à partir desquelles τ_j lit est : 0, 1, 2, 0, 1, 2, ...



(a) Pour $\tau_i \overset{*\hat{}3./\hat{}2}{\rightarrow} \tau_j$

(b) Pour $\tau_i \overset{*\hat{}2./\hat{}3}{\rightarrow} \tau_j$

(c) Pour $\tau_i \overset{\sim>2}{\rightarrow} \tau_j$

FIG. 33 – Dépendances de données

8 Prototype

Une fois que l'ensemble des tâches indépendantes correspondant à un programme est généré, il est ensuite très simple de le traduire en code C. Un prototype du compilateur a été implanté en OCAML. Il génère un code C utilisant les extensions temps réel de POSIX [POS98]. Le programme généré consiste en un ensemble de threads communicant. Chaque thread est constitué d'une boucle infinie qui empaquette le code de la fonction du nœud importé correspondant avec le protocole de communication défini ci-dessus. La politique EDF modifiée afin de supporter les mot d'échéances à été implantée à l'aide de MARTE OS [RH02]. Une étude de cas réaliste, basée sur une version étendue du FAS décrit en introduction, a été implantée et a permis de valider le langage et son compilateur.

9 Conclusion

Nous avons proposé un langage pour la programmation de systèmes de contrôle-commande. Il se présente comme un langage d'architecture logicielle temps réel possédant une sémantique synchrone.

Il fournit un haut niveau d'abstraction et une sémantique formelle, ce qui permet de spécifier de manière non ambiguë l'intégration de plusieurs systèmes synchrones localement périodiques en un système synchrone global multi-périodique.

Le langage repose sur une classe particulière d'horloges, les horloges strictement périodiques, qui permettent de relier le temps logique (la séquence d'instants d'un programme) au temps réel. Une horloge strictement périodique se caractérise de manière unique par sa période et par sa phase et définit le rythme temps réel d'un flot. Les transformations périodiques d'horloges permettent de créer de nouvelles horloges strictement périodiques à partir d'horloges strictement périodiques existantes, soit en accélérant l'horloge, soit en la ralentissant, soit en appliquant un décalage de phase. Le langage définit des primitives temps réel basées sur les horloges strictement périodiques et les transformations périodiques d'horloges. Ces primitives permettent de spécifier des contraintes de périodicité, des contraintes d'échéance et des opérateurs de transition de rythme, afin de définir de manière précise des motifs de communication entre opérations de rythmes différents. Un programme consiste ensuite en un ensemble de nœuds importés (implantés à l'extérieur du programme) assemblés à l'aide d'opérations de transition de rythme. La sémantique du langage est définie formellement. Les analyses statiques vérifient que la sémantique du programme est bien définie avant de le traduire vers un code bas niveau.

Lorsqu'un programme est accepté par les analyses statiques, sa sémantique est bien définie. Il est alors traduit en un ensemble de tâches temps réel concurrentes et communicantes, programmé en C. La première étape de la traduction consiste à extraire un ensemble de tâches temps réel à partir du programme, les tâches étant reliées par des contraintes de précédence dûes aux dépendances de données. Les précédences sont ensuite encodées dans les attributs temps réel des tâches, en ajustant les échéances des tâches reliées par des contraintes de précédence. Lorsqu'une précédence relie deux tâches de rythmes différents, elle ne relie pas toutes les instances des deux tâches et il est donc nécessaire d'affecter différentes échéances pour différentes instances d'une même tâche. L'encodage repose alors sur l'utilisation de mots d'échéances, un mot d'échéances définissant la séquence des échéances d'une tâche sous la forme d'un motif répétitif fini. L'ensemble de tâches obtenu après ajustement des échéances est ordonnancé à l'aide de la politique EDF. Cette technique d'ordonnancement (encodage + EDF) est optimale. Un protocole de communication inter-tâches est ensuite défini afin d'assurer le respect de la sémantique formelle du programme d'origine. Ce protocole est basé sur des mécanismes de mémoire tampons et ne nécessite pas de primitives de synchronisations (telles que des sémaphores).

Introduction

The work presented in this dissertation was funded by EADS Astrium Space Transportation.

Reactive Systems

A *reactive system* [HP85] is a particular type of embedded system that maintains permanent interaction with its physical environment. The system waits for stimuli, then performs computations and finally produces outputs (commands) as a reaction to the input stimuli. Control systems, monitoring systems or signal processing systems are classical examples of reactive systems.

A distinctive characteristic of reactive systems is that the response time of the system must respect real-time constraints induced by the environment. If the response comes too late the system may miss environment inputs and respond to inputs that may not correspond to the current state of the environment. In addition, reactive systems are often highly critical (e.g. aircraft piloting system, nuclear plant control system) and faults cannot be tolerated.

Event-driven systems vs sampled systems

Reactive systems can be separated into two sub-classes: *event-driven systems* and *sampled systems*. The difference stands in the way inputs are acquired. In an event-driven system (sometimes referred to as purely reactive system), the system waits for an input event to occur and then computes its reaction. On the opposite, a sampled system acquires its inputs at regular intervals of time and computes its reaction for each sample of its inputs (possibly doing only part of the computations if only part of the inputs are present for the current sample). While event-driven systems correspond to a larger class of systems, sampled systems are usually easier to analyze and their behaviour is more predictable, especially concerning the real-time aspects. In the following we only consider sampled systems.

Embedded control systems

Our work is specifically targeted for *embedded control systems*. By embedded control systems, we mean linear feedback systems, that is to say control loops including sensors, control algorithms and actuators that regulate the state of the system in its environment. Spacecraft and aircraft flight control systems are good examples of control systems and their study originally motivated our work. The aim of these systems is to control the position, speed and attitude of the vehicle thanks to physical devices, such as control surfaces for an aircraft or thrusters for a spacecraft. The system operates in a closed loop with its environment as commands applied to the vehicle modify its state (position, speed and attitude), which modifies the inputs of the system and require to compute new commands and so on.

The computations performed by such systems are highly regular: for instance, there is no dynamic creation of processes and only statically bounded loops. This enables the production of optimized software and hardware, the behaviour of which is predictable, an essential feature for highly critical system.

Control systems must respect several real-time constraints related to physical constraints. First, the control rate of a device (and thus the rate of the operations executed to control this device) must be higher than a minimal frequency, related to the inertia and physical characteristics of the vehicle. Under this frequency, the safety of the vehicle is not ensured. The minimal frequencies are not the same for all the physical devices of the controlled system, for instance the propulsion devices of a space vehicle must be controlled very fast to ensure the stability of the vehicle, while the position of its solar panels can be controlled a little slower as a little energy loss has less impact on the vehicle. Second, the control rate of a device must be lower than a maximal frequency requirement, over which the device will not be able to apply the commands fast enough or may get damaged. This parameter again changes from one device to another. The actual control rate of a device is usually chosen as close to the minimal frequency as possible, in order to spare unnecessary computations and to enable the use of less powerful hardware (which is less expensive, consumes less energy, ...).

As for any reactive system, the response time of a control system is bounded. The delay between an observation (inputs) and the corresponding reaction (outputs) must respect a maximum latency, ie must be lower than a deadline relative to the date of the observation. The deadline of an output is not necessarily equal to its period, it can be smaller, so as to follow closely the corresponding inputs.

Case study: The Flight Application Software

In this section we present the *Flight Application Software* (FAS), the flight control system of the *Automated Transfer Vehicle* (ATV), a space vehicle designed by EADS Astrium Space Transportation for resupplying the *International Space Station* (ISS). It is a typical example of control system. A simplified and adapted version of the FAS is described in Fig. 1. It does not exactly correspond to the real system, but gives a good overview of its characteristics.

The FAS consists mainly of acquisition operations, processing operations and command operations, each represented by a different box in the figure. Arrows between boxes represent data-dependencies. Arrows without sources represent system inputs and arrows without destination represent system outputs. Data-dependencies define a partial order between the different operations of the system, as the consumer of a data-flow must execute after the producer of the data-flow. The FAS first acquires data: the orientation and speed from gyroscopic sensors (`GyroAcq`), the position from the GPS (`GPS Acq`) and from the star tracker (`Str Acq`) and telecommands from the ground station (`TM/TC`). The *Guidance Navigation and Control* function (divided into an upstream part, `GNC_US`, and a downstream part, `GNC_DS`) then computes the commands to apply while the *Failure Detection Isolation and Recovery* function (`FDIR`) verifies the state of the FAS and looks for possible failures. Finally commands are computed and sent to the control devices: thrusters orders for the *Propulsion Drive Electronics* (`PDE`), power distribution order for the *Power System* (`PWS`), solar panel positioning orders for the *Solar Generation System* (`SGS`) and telemetry towards the ground station (`TM/TC`), etc.

Each operation has its own rate, ranging from 0.1Hz to 10Hz. An intermediate deadline constraint is imposed on data produced by the GNC Upstream (300ms while the period is 1s). Operations of different rate can communicate, which is an essential feature of such systems and has very important impacts on the complexity of the design and implementation process.

Contribution

As we can see, programming an embedded control system is a complex task since this does not only consist in implementing the functional aspects of the system. As the system is highly critical, the main

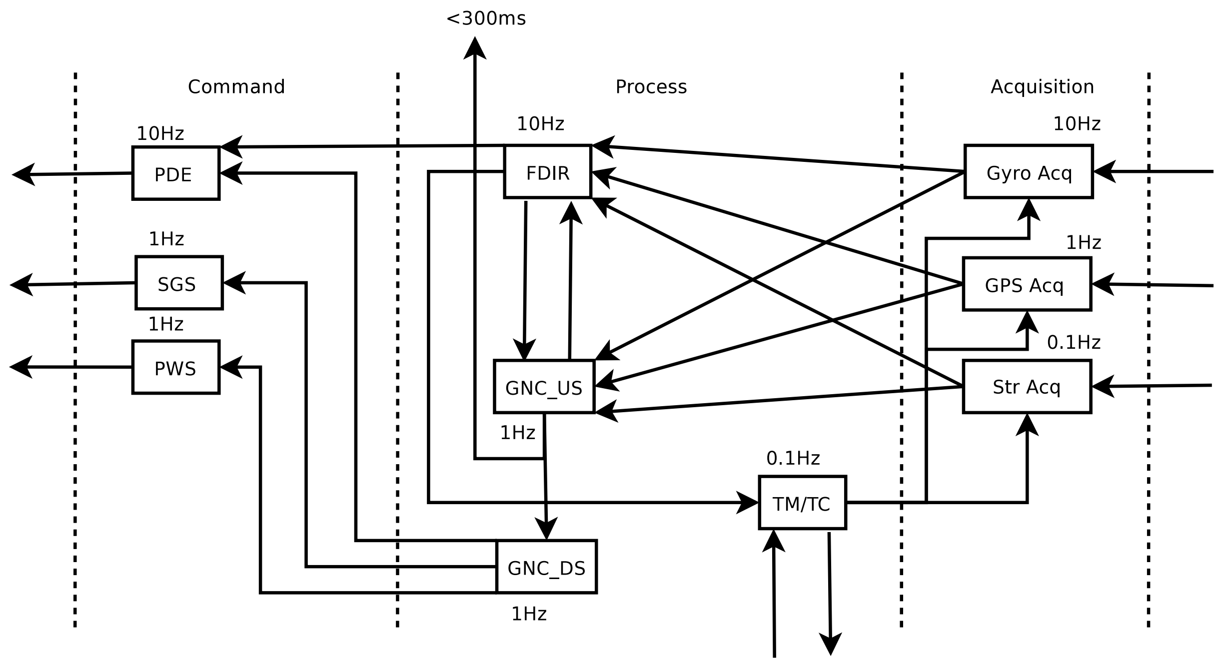


Figure 1: The Flight Application Software

concern for the program supervising the system is a need for predictability. This requires strong functional determinism, which means that the program will always produce the same output sequence with respect to the same input sequence. The program must be temporally deterministic as well, always having the same temporal behaviour and respecting multiple hard real-time constraints. While respecting these constraints, the system still needs to be optimized, in terms of memory consumption, hardware cost, power consumption or weight for instance.

The complexity of the development process and the criticity of the systems call for high-level programming languages, which cover the complete development process from design to implementation. Automated code generation process is the key, it provides high confidence in the final program as every step of the generation can be defined formally.

In this Ph. D. thesis, we propose such a high-level language for programming embedded control systems (extending our work in [FBLP08]). The language is built upon Synchronous Languages [BB01] and inherits their formal properties. It adds real-time primitives to enable the programming of multi-periodic systems. The objective of the language is not to replace other synchronous languages but instead to provide a higher layer of abstraction, on top of classic synchronous languages. It can be considered as a real-time software architecture language that enables to assemble locally mono-periodic synchronous systems into a globally multi-periodic synchronous system. The language is supported by a compiler that generates synchronized multi-task C code. The generated code executes on a standard real-time operating system with a preemptive scheduling policy. Synchronization is achieved by a buffering communication protocol (a refined version of [BCFP09]) that requires no explicit synchronization primitives (like semaphores). The compilation is defined formally and produces completely deterministic code, which respects the real-time semantics of the original program (period, deadlines, release dates and precedences) as well as its functional semantics (respect of variables consumption). In the next chapter we motivate the definition of this language by showing that the state of the art on the domain does not completely address the problems emphasized above. We also detail the distinctive features of the language.

Chapter 1

State of the Art

In this chapter we present a survey of existing languages related to the programming of embedded control systems (this was in part presented in [FBL+08]). The desirable features of the language are:

- The ability to specify a set of operations;
- The ability to specify data-communications between operations, in particular between operations of different rates;
- The ability to specify multiple real-time constraints, such as periods, release dates and deadlines;
- A formal definition of the language, to ensure that the semantics of a program is unambiguously defined;
- Automated code generation, to cover the process from design to implementation;
- Functional determinism of the generated code;
- Temporal determinism of the generated code;
- And last: simplicity. This is essential for the programmer to better understand and analyze a program.

1.1 Low-level programming

Probably the most widely used technique for programming real-time systems is to directly program the system with traditional imperative languages. These languages were not originally designed for real-time systems and the real-time aspects are mostly supported by Application Programming Interfaces (API) for real-time, which are very close to the underlying Real-Time Operating System (RTOS).

1.1.1 Real-time API

A real-time API defines the interface for a set of services (for instance functions and data-structures) related to real-time programming. The objective of a real-time API is to define a standard, which should allow the portability of an application between different execution platforms that respect this standard. Real-time APIs are not necessarily related to a single language. For instance, the widely adopted POSIX extensions for real-time [POS93a, POS93b], have been implemented for C as well as for ADA. On the opposite, the Real-Time Specification for Java (RTSJ) for instance is specifically defined for JAVA and

is a combination of APIs and modifications to apply to the JAVA virtual machine. Other popular real-time APIs include OSEK [OSE03], for the automotive industry, and APEX, for avionics systems. The general principles are similar for the different APIs: each operation of the system is first programmed as a separate function and functions are grouped into threads, on which the programmer can specify real-time constraints. The program then consists of several threads, scheduled concurrently by the OS.

The different threads are related by data-dependencies, as data produced in one thread may be needed in another thread. In critical systems, the programmer must control the order in which data is produced and consumed (otherwise the behaviour of the system might not be predictable), which requires to introduce synchronization mechanisms between threads. Synchronizations can be implemented either by precisely controlling the dates at which threads execute so that data is produced and consumed in the right order (*time-triggered* communications) or by using blocking synchronization primitives like semaphores (*rendez-vous* communications).

1.1.2 Real-time Operating Systems

A real-time operating system is an operating system with multi-tasking capabilities designed specifically for executing real-time applications. Multi-tasking means that the OS is able to execute multiple tasks concurrently, alternating between the different tasks. The *scheduling policy* of the RTOS provides a general algorithm for deciding the order in which tasks must be executed, according to their real-time characteristics (periods and deadlines). Scheduling is often *preemptive* (but not always), meaning that the OS can suspend the execution of the currently executing task to execute a more urgent one and restore the execution of the suspended task later.

A RTOS tries to follow one of the aforementioned real-time APIs and usually provides (among other things):

- Structures to define tasks (threads);
- One or several scheduling policies;
- Data-buffers shared between communicating tasks (global memory);
- Locking mechanisms (semaphores), when a task needs exclusive access to a resource (data-buffer or peripheral access for instance);
- Explicit time manipulation (accurate timestamps, timers, ...).

Popular commercial real-time operating systems include VxWorks by Wind River Systems [VxW95] and QNX Neutrino [Hil92], which both conform to the POSIX standard, and OSE [OSE04], which defines four different APIs, depending on the number of features required for a given system. Several extensions for Linux provide real-time features, including RTLinux (`www.rtlinuxfree.com`) and RTAI (`www.rtai.org`), which have the major advantage of being open-source projects. Finally, a number of research kernels try to provide a new generation of operating systems with a higher level of abstraction, including more explicit timing constraints (task deadlines instead of task priorities for instance), better guarantees that allow to test in advance whether real-time constraints can be met or not and so on. This includes SHARK [GAGB01], MaRTE [RH02] and ERIKA [GLA+00].

1.1.3 Towards a higher level of abstraction

Programming real-time systems with low-level languages allows to implement a very large class of applications but has important drawbacks. First, in the case of large applications, synchronizing threads

is a tedious and error-prone task. Time-triggered communications are hard to implement as it amounts to manually scheduling part of the system. Rendez-vous communications may lead to deadlocks where two tasks are indirectly waiting for each other to continue their execution. Second, the correctness of the program is hard to verify (either manually or using automated verification tools) due to the low-level of abstraction with which the system was programmed. Finally, the low-level of abstraction also makes it hard to analyze which parts of the program correspond to real design constraints and which parts only correspond to implementation concerns and thus during maintenance phases it is hard to know which part of the program can be modified without risk and which cannot. While dealing with such considerations cannot be avoided, it is much easier to rely on programming languages with a higher level of abstraction, where synchronization and scheduling concerns are handled by the compiler of the language.

1.2 Synchronous Languages

The difficulty for programming embedded systems directly with low-level languages progressively leads programmers to consider languages with a higher-level of abstraction. Such languages are currently still mainly used only for the early phases of the development process, namely specification, verification and simulation. Some languages though aim at covering the whole process, from specification to implementation, relying on automated code generation where the input program is automatically translated into lower level code. This approach offers important advantages. First, the duration of the development process is reduced, as low-level implementation concerns are handled directly by the code generator. Second, a higher-level program is usually easier to understand (by other programmers) and to analyze (by verification tools). Finally, the correctness of the lower-level program into which the program is translated is ensured by the translation process.

The synchronous approach [BB01] proposes such a high level of abstraction. It is based on solid, elegant and yet simple mathematical foundations, which make it possible to handle the compilation and the verification of a program in a formal way. The synchronous approach simplifies the programming of real-time systems by abstracting from real-time and reasoning instead on a logical-time scale, defined as a succession of *instants*, where each instant corresponds to a reaction of the system. The synchronous hypothesis says that the programmer does not need to consider the dates at which computations occur during an instant, as long as the computations finish before the beginning of the next instant. This approach led to the definition of three synchronous languages approximately simultaneously: LUSTRE, SIGNAL and ESTEREL.

1.2.1 LUSTRE

Overview

LUSTRE [HCRP91] is a data-flow synchronous language. Each variable or expression of a program denotes a *flow*, an infinite sequence of values. The *clock* of a flow defines the instants when the flow is present or absent (bears no value). A LUSTRE program consists of a set of equations, structured into *nodes*, which define the values of the different variables of the program at each instant. The example below illustrates the different features of the language:

```
node N(c: bool; i:int) returns (o:int)
var v1: int; v2: int when c;
let
  v1=0->pre(i);
  v2=(v1+1) when (true->c);
  o=current(v2);
```

```
tel
```

The behaviour of the program is illustrated in Fig. 1.1, where we give the value of each flow at each instant. The node computes one output (`o`) from its two inputs (`c`, `i`). The variables `v1` and `v2` are used for intermediate computations. The equation `v1=0->pre(i)` means that `v1` equals 0 at the first instant and for the subsequent instants it equals the value of `i` at the previous instant. The equation defining `v2` says that it is present (bears a value) only when `true->c` is true and that for the instants when it is present, its value is the value of `v1` at this instant plus 1. The output `o` is finally defined as the current value of `v2`: the value of `o` is the value of `v2` when it was last present.

`i`	i_0	i_1	i_2	i_3	i_4	...
`c`	false	true	false	false	true	...
`pre(i)`		i_0	i_1	i_2	i_3	...
`v1=0->pre(i)`	0	i_0	i_1	i_2	i_3	...
`v1+1`	1	i_0+1	i_1+1	i_2+1	i_3+1	...
`v2=(v1+1) when (true->c)`	1	i_0+1			i_3+1	...
`o=current(v2)`	1	i_0+1	i_0+1	i_0+1	i_3+1	...

Figure 1.1: Behaviour of a simple LUSTRE program

Programming multi-periodic systems

LUSTRE usually only considers systems where all the processes share the same logical time scale, that is systems where all the processes can be sampled and executed at the same real-time periodic rate. Some processes can be deactivated at some instants using clocks, but when they execute they still need to complete before the end of the instant, so within the same period. Clocks serve as activation conditions but not as rate modifiers.

Multi-rate systems can be specified using a single logical time scale but this is quite tedious in practice. First, the programmer needs to choose the base sample rate of his program among the different rates of the system. In the following, we will consider two opposite choices: programming the system using the fastest rate as the base rate or programming the system using the slowest rate as the base rate. We could choose any rate in between, but this would not change the problem much. For simplification, in the following we take an extract of the case study of the FAS presented previously in Fig. 1 as example: we only consider the communication loop between operations `FDIR` and `GNC US`, as shown in Fig. 1.2.

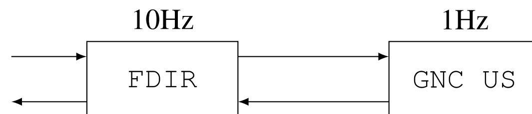


Figure 1.2: Simplified communication loop between operations `FDIR` and `GNC US`

Programming with a fast base rate

We first define a few auxiliary nodes in Fig. 1.3. Their behaviour is detailed in Fig. 1.4. The node $countN$ is a counter modulo n. The output of the node $everyN$ is true once every n instants. The node $i_current$ is an initialised current. It returns the $init$ value as long as its input x has never been present. It returns the current value of x afterwards.

```
node countN(n: int) returns(o: int)
let
  o=(0->(pre(o)+1)) mod n;
tel

node everyN(n: int) returns (reached: bool)
let
  reached=(countN(n)=n-1);
tel

node never(c: bool) returns (not_ever: bool)
let
  not_ever=true->if c then false
                  else pre(not_ever);
tel

node i_current(c: bool; init: int; x: int when c)
returns (o: int)
let
  o = if never(c) then init
      else current(x);
tel
```

Figure 1.3: Auxiliary LUSTRE nodes

cpt=countN(3)	0	1	2	0	1	2	...
c=everyN(3)	false	false	true	false	false	true	...
v=cpt **when** c			2			2	...
never(c)	true	true	false	false	false	false	...
i_current(c,0,v)	0	0	2	2	2	2	...

Figure 1.4: Behaviour of auxiliary nodes

We give a first version of the program with a fast base rate in Fig. 1.5. This program assumes that the main node `FAS_fast` is activated with a sample rate of 100ms. The node `FDIR` corresponds to a fast operation, it is performed at each instant, thus at 10Hz. The node `GNC_US` is activated only when the condition `clock10` is true, that is every 10 instants, so at 1Hz. The data communication from the fast operation to the slow operation is performed instantaneously (flow `vfdir`) and only one out of 10 successive produced values are consumed (due to **`when`** `clock10`). Data communication from the slow operation to the fast operation (flow `vgnc`) is delayed using the **`pre`** operator. This avoids a causality loop. Indeed, the node instance `FDIR` depends on a value produced by the node instance `GNC_US` (`cur_vgnc`), which itself depends on a value produced by `FDIR` (`vfdir`). Thus the dependencies cannot both be instantaneous and one of them must be delayed.

```
node FAS_fast(i: int) returns (o: int)
var clock10: bool;
    vfdir, vgnc: int;
    ds, us1, us2: int when clock5;
let
  (o, vfdir)=FDIR(i, cur_vgnc);
  vgnc=GNC_US(vfdir when clock10);
  clock10=everyN(10);
  cur_vgnc=i_current(clock10, 0, pre(vgnc));
tel
```

Figure 1.5: Programming a multi-rate communication loop in LUSTRE, with a fast base rate (10Hz)

The main default of this version is that it assumes that it is possible to execute the two operations in less time than the sample period (100ms). Indeed, during instants when both the slow and fast operations are activated, they must all finish within the instant.

This assumption is far too restrictive and leads us to a new version given Fig. 1.6. This time we split the processes performed by the slow operation into several nodes (`GNC_US0`, ..., `GNC_US9`) and distribute these nodes between 10 successive instants, which correspond to one slow period. Each of these nodes is activated only once every 10 instants, with an offset of one instant between the different nodes. Consequently, the complete slow operation is executed with a frequency of 1Hz and can spread over more than one instant.

```
node FAS_fast2(i: int) returns (o: int)
var clock0, clock1, clock2, clock3, clock4, clock5, clock6, clock7, clock8 ,clock9: bool;
    count, vfdir: int;
    vgnc0: int when clock0; vgnc1: int when clock1; vgnc2: int when clock2;
    vgnc3: int when clock3; vgnc4: int when clock4; vgnc5: int when clock5;
    vgnc6: int when clock6; vgnc7: int when clock7; vgnc8: int when clock8;
    vgnc9: int when clock9;
let
  count=countN(10);
  clock0=(count=0); clock1=(count=1); clock2=(count=2); clock3=(count=3); clock4=(count=4);
  clock5=(count=5); clock6=(count=6); clock7=(count=7); clock8=(count=8); clock9=(count=9);
  vgnc0=GNC_US0(vfdir when clock0);
  vgnc1=GNC_US1(current(vgnc0) when clock1);
  vgnc2=GNC_US2(current(vgnc1) when clock2);
  vgnc3=GNC_US3(current(vgnc2) when clock3);
  vgnc4=GNC_US4(current(vgnc3) when clock4);
  vgnc5=GNC_US5(current(vgnc4) when clock5);
  vgnc6=GNC_US6(current(vgnc5) when clock6);
  vgnc7=GNC_US7(current(vgnc6) when clock7);
  vgnc8=GNC_US8(current(vgnc7) when clock8);
  vgnc9=GNC_US9(current(vgnc8) when clock9);
  (o,vfdir)=FDIR(i,i_current(clock9, 0, pre(vgnc9)));
tel
```

Figure 1.6: Programming a multi-rate communication loop in LUSTRE, with a fast base rate (10Hz), splitting the slow operation

For a complete system, manually distributing processes performed by slow operations between successive fast instants can be tedious and error-prone. First, this requires the programmer to determine a fair distribution between instants in terms of execution times, which amounts to manual scheduling and should be automatized. Second, this assumes that the software architecture of the processes allows such a distribution. For instance, if a slow operation is split into two sub-nodes, as side effects are forbidden in synchronous languages, data produced by the first sub-node at the end of the first instant explicitly needs to be passed on to the next sub-node of the operation at the next instant (flow `vgnc0` between `GNC_US0` and `GNC_US1`). Such data communications can be numerous if the computations performed by one operation are very interdependent. This leads to high memory consumption and again is a tedious task that should be automatized.

Programming with a slow base rate

We now consider a third version using the slow rate as the base rate of the program. The corresponding code is given in Fig. 1.7. This program assumes that the main node `FAS_slow` is activated with a sample period of 1s. The node corresponding to the slow operation (`GNC_US`) is executed at each instant, thus with a frequency of 1Hz. The node corresponding to the fast operation (`FDIR`) is instantiated 10 times in order to be executed with a frequency of 10Hz. The inputs and outputs of the program are respectively

consumed and produced by the fast operation, thus they also need to be duplicated 10 times, as they are acquired and produced 10 times in an instant. For the communication from `GNC_US` to `FDIR`, we choose to consume the value produced by the third iteration of `GNC_US`.

```
node FAS_slow(i0, i1, i2, i3, i4, i5, i6, i7, i8, i9: int)
     returns (o0, o1, o2, o3, o4, o5, o6, o7, o8, o9: int)
var vgnc: int;
    vfdir0, vfdir1, vfdir2, vfdir3, vfdir4, vfdir5, vfdir6, vfdir7, vfdir8, vfdir9: int;
let
  vgnc=GNC_US(vfdir3);
  (o0, vfdir0)=FDIR(i,0->pre(vgnc));
  (o1, vfdir1)=FDIR(i,0->pre(vgnc));
  (o2, vfdir2)=FDIR(i,0->pre(vgnc));
  (o3, vfdir3)=FDIR(i,0->pre(vgnc));
  (o4, vfdir4)=FDIR(i,0->pre(vgnc));
  (o5, vfdir5)=FDIR(i,0->pre(vgnc));
  (o6, vfdir6)=FDIR(i,0->pre(vgnc));
  (o7, vfdir7)=FDIR(i,0->pre(vgnc));
  (o8, vfdir8)=FDIR(i,0->pre(vgnc));
  (o9, vfdir9)=FDIR(i,0->pre(vgnc));
tel
```

Figure 1.7: Programming a multi-ate communication loop in LUSTRE, with a slow base rate (1Hz)

This version is incorrect because even if the fast operations are performed 10 times during each slow period, nothing forces the different repetitions to execute at the right time during one instant. Let t_0 be the start date of the instant. The duration of the instant is 1s. We expect operation `FDIR` to execute once during each of the 10 time intervals $[t_0, t_0+100[$, $[t_0+100, t_0+200[$, ..., $[t_0+900, t_0+1000[$. This behaviour is in no way specified in this program. Similarly, the inputs of the system will all be consumed simultaneously at the beginning of the instant and the outputs of the system will all be produced simultaneously at the end of the instant (thus with a rate of 1Hz), which is not the expected behavior. Finally, instantiating a node 10 times does not produce the same result as repeating the same node instance 10 times, if the node contains memories (**`pre`**). 10 instances of the same node use 10 different memories, while 10 repetitions of the same node use the same memories. For instance, repeating a counter 10 times does not produce the same result (the value 10) as repeating one time 10 different counter instances (the value 1 for each counter).

Extensions for real-time

As they completely abstract from real-time, the synchronous languages are not well-adapted to handling systems with multiple real-time constraints such as the FAS. Recent work proposed to introduce real-time aspects in LUSTRE through the use of *assumptions* made about the program environment and *requirements* about the program behaviour [Cur05]. Assumptions specify the base rate of the program (for instance `basic_period=5`) as well as node execution times (for instance `exec_time N in[3,4]`). Requirements constrain the availability date of a flow (for instance `date(x)<5`) or the latency between two flows (for instance `date(x)-date(y)>4`). In addition, the primitive `periodic_clock(k,p)` defines a clock of period k and of phase p. This clock is false during the $p-1$ first instants and then true once every k instants.

The compiler ensures, using static scheduling techniques, that if the assumptions hold the requirements will be satisfied at execution. It compiles a program into a set of communicating tasks for a Time Triggered Architecture (TTA) [KB03]. The scheduling is based on a LP solver and specific to TTA. The synchronous hypothesis is relaxed for flows the clock of which is periodic and the constraint becomes

that the computation of a periodic flow must end before the next activation of the flow instead of before the end of the instant.

Using these extensions, we give an improved version of our example programmed on a fast base rate in Fig. 1.8. The assumptions specify the base rate of the program and the duration of each node. The slow operations are activated once every 10 instants using a periodic clock.

```
node FAS_fast(i: int) returns (o: int)
(hyp)
  basic_period=100;
  exec_time (FDIR) in [10, 15];
  exec_time (GNC_US) in [180, 210];
var vfdir, cur_gnc: int;
    clock10: bool;
    vgnc: int when periodic_clock(10, 1);
let
  clock10=periodic_clock(10,1);
  (o, vfdir)=FDIR(i, cur_vgnc);
  vgnc=GNC_US(vfdir when clock10);
  cur_vgnc=i_current(clock10,0,pre(vgnc));
tel
```

Figure 1.8: Programming a multi-rate communication loop in LUSTRE, with a fast base rate, using real-time extensions

This new version is better adapted to our case study and our work retains some of the concepts introduced here, in particular the relaxed synchronous hypothesis. The main drawback of this approach is that the compilation scheme is very specific to the TTA architecture.

This work was extended in [ACGR09], which proposes periodic activation condition operators based on periodic clocks and primitives to handle exclusive execution modes. The authors also propose a simpler, more generic compilation scheme, which generates a set of concurrent real-time tasks. A compiler was implemented for the Xenomai RTOS (`www.xenomai.org`). Task communications are handled using the protocol proposed by [STC06]. This protocol ensures the conservation of the synchronous semantics of communicating tasks thanks to data-buffering mechanisms based on task priorities. The computation of task priorities is not detailed in [ACGR09]. Our work focuses especially on this point and automates the computation of task priorities.

1.2.2 SIGNAL

SIGNAL [BLGJ91, ABLG95, LGTLL03] is a data-flow synchronous language in many points similar to LUSTRE but also with important differences. It retains the declarative style of LUSTRE, specifying systems by equations over synchronized streams (*signals* instead of flows). The first important difference with respect to LUSTRE lies in the fact that in SIGNAL the user can manipulate clocks explicitly as first-class objects: the programmer can declare clock variables (variables of type `event`), extract the clock of a signal (`event1=`**`when`**` s`), specify clock relations (for instance `x^=y^+z` says that the clock of `x` is the union of the clocks of `y` and `z`), ... In LUSTRE, a clock is not a first-class object, it is manipulated through the flow that contains it.

An important consequence of this clock model is that clocks can be synthesized by the compiler. Behind a SIGNAL program lies a system of clock equations (Boolean equations) that drives this synthesis. The synthesis, called *clock calculus*, transforms the set of equations into a set of equalities where every clock is defined with respect to the free clocks of the system. The free clocks correspond to clocks that will be provided by the environment of the program as input variables. Though this clock synthesis is very powerful, the synthesis problem is NP-hard, so the SIGNAL compiler does not seek completeness

but instead efficiency for commonly encountered systems, meaning that some correct programs may be rejected.

The second important difference, which is in part a consequence of the first, is that SIGNAL programs can be *polychronous*. This means that the clocks hierarchy of a program can form a forest (the free clocks of the program are the roots of the different trees of the forest). In other words, the different clocks of a program do not necessarily have a common ancestor. The clocks hieraracy in LUSTRE forms a single tree, LUSTRE programs are thus called *endochronous*.

For the most part, programming the FAS example with SIGNAL faces the same difficulties as when programming it with LUSTRE, so we do not illustrate this in details here. In [SLG97], an extension of clock relations called *affine clock relations* was proposed. An *affine transformation* of parameters (n, φ, d) applied to a clock H produces a clock K, obtained by inserting $n-1$ instants between two successive instants of H to produce an intermediate clock H' and then by taking each dth instant on H', starting from d. It allows to specify multi-rate systems more explicitly than with core SIGNAL and the clock calculus problem can be solved efficiently for this class of clock relations. This work is not specifically targeted for multi-periodic systems. More recently, [MTGLG08] studied this model on the particular case of periodic systems. This work focuses on formal verification and, as far as we know, the compilation of a program into a set of concurrent real-time tasks has not been studied yet.

1.2.3 ESTEREL

Overview

ESTEREL [BdS91] is an imperative synchronous language and focuses mainly on the description of the control-flow in a system. An ESTEREL program consists of several concurrent processes communicating by means of broadcasted *signals* and structured into *modules*. When a signal is emitted by a process, it is *instantaneously* perceived by all other processes that listen to it. The processes executed by a module are described as a composition of *statements*. Statements can either be composed sequentially or in parallel (concurrently). Most statements are considered to be instantaneous, except for explicit pause statements. At each reaction of a program, control flows through its processes until it arrives to a pause or termination instruction. At the next reaction, processes resume where they stopped. The example below illustrates the basic features of the language:

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

The program consists of a single module `ABRO`, which has three input signals `A`, `B` and `R` and one output signal `O`. The body of the module consists of an infinite loop statement (`loop..end`). The body of the loop first consists of two statements composed in parallel (using `||`). The first statement (`await A`) causes the control to pause in this branch of the composition until `A` is emitted. The second statement (`await B`) causes the control to pause in the other branch of the composition until `B` is emitted. The control flows to the rest of the loop only after both `A` and `B` have been emitted (not necessarily simultaneously). This parallel statement is then composed sequentially (using `;`) with statement `emit O`, which causes signal `O` to be emitted. The control then continues to flow (instantaneously) to the next statement

(`await R`), which causes it to pause until `R` is emitted. The control then immediately loops back to the begining of the `loop..end` statement. The behaviour of this program is illustrated in Fig. 1.9, in which we detail the input received by the program and the output signals emitted by the program.

Inputs		A	B	A,B	R	A,B	...
Outputs			O			O	...

Figure 1.9: Behaviour of a simple ESTEREL program

The ESTEREL program for the example of Fig. 1.1 is given in Fig.1.10. Implicitly, the program is activated at the rate of its input `i`, which is assumed to be 10Hz. The `FDIR` operation is declared as an external procedure. The first group of parameters of the procedure is the outputs and the second group is the inputs. As operation `GNC_US` can take longer than 100ms to execute (the duration of a program reaction), it is declared as an external task. The task can then be executed *asynchronously* using the statement `exec`. The statement `exec` is said asynchronous because it can last longer than a reaction. The reaction in which it will complete is unknown a priori. When the statement completes, the process simply resumes to the next statement. The two operations `FDIR` and `GNC_US` are executed in parallel. Operation `FDIR` executes every time an input occurs (`every i`), while operation `GNC_US` executes once every 10 inputs (`loop .. each 10 i`). In the first branch of the parallel composition, task `GNC_US` produces the value `xv1`, which is then emitted via signal `s1` (statement `emit s1(xv1)`). The procedure `FDIR` can then consume the value contained by the signal (`?s1`). Similarly, `FDIR` produces values `xo` and `xv2`, emits the output signal `o` with value `xo` and the signal `v2` with value `xv2`. The statement `await immediate s2` in the first branch forces task `GNC_US` to execute after `FDIR`. The statement `await s2` requires the control flow to pause for at least one instant, even if `s2` is present when the control flow reaches this statement, while `await immediate s2` allows the control flow to directly continue if `s2` is present. On the opposite, `FDIR` does not need to wait for the completion of `GNC_US` to start its execution, as the signal `s1` is emitted before executing `GNC_US`. This way, `FDIR` consumes the previous value produced by `GNC_US`.

```
module FAS :
input i;
output o;
procedure FDIR(integer, integer) (integer);
task GNC_US(integer) (integer);
signal s1: integer, s2 : integer in
var xv1:=0 : integer,xv2,xo : integer in

loop
  emit s1(xv1);
  await immediate s2;
  exec GNC_US(xv1) (?s2)
each 10 i
||
every i do
  call FDIR(xo, xv2) (?s1);
  emit o(xo);
  emit s2(xv2)
end
```

Figure 1.10: Programming a multi-rate communication loop in ESTEREL

The main concern with this program is that the absence of explicit real-time information does not allow automated real-time analyses of the program (schedulability analysis) or concurrent scheduling based on task real-time properties.

Real-time extensions

The project TAXYS [BCP+01] introduced real-time aspects in ESTEREL by means of *pragmas* (code annotations) as shown in the following program:

```
loop
  await PulsePeriod %{PP:=age(PulsePeriod)}%;
  call C() %{(10,15), PP<=40}%;
emit Pulse;
end loop
```

Pragmas appear between `%{` and `}%`. The pragma $(10, 15)$ next to statement `call C()` specifies that the execution time of `C` is 10 in the best case (BCET) and 15 in the worst case (wcet). The pragma `%{PP:=age(PulsePeriod)}%` declares a *continuous* time variable, which value is the time elapsed since the emission of signal `PulsePeriod`. The pragma `PP<=40` specifies a real-time constraints, which requires procedure `C` to complete no later than 40 units of time after the emission of signal `PulsePeriod`.

Using these extensions, we can add real-time characteristics and real-time constraints to our previous example, as shown in Fig. 1.11.

```
module FAS :
input i;
output o;
procedure FDIR(integer, integer) (integer);
task GNC_US(integer) (integer);
signal s1: integer, s2 : integer in
var xv1:=0 : integer,xv2,xo : integer in

loop
  %{SPeriod:=age(i)}%
  emit s1(xv1);
  await immediate s2;
  exec GNC_US(xv1) (?s2) %{(180,210), SPeriod<1000}%;
each 10 i
||
every i do
  %{FPeriod:=age(i)}%
  call FDIR(xo, xv2) (?s1); %{(10,15), FPeriod<100}%;
  emit o(xo);
  emit s2(xv2)
end
```

Figure 1.11: Programming a multi-rate communication loop in ESTEREL/TAXYS

The tool TAXYS integrates the ESTEREL SAXO-RT compiler [WBC+00] and the model checking tool KRONOS [DOTY96]. The compiler translates the annotated program into a timed automaton [AD94] and the automaton is then analyzed by the model checker to validate the real-time constraints of the program. As far as we know, the ESTEREL program is translated into sequential C code and the objective of this work is not to take advantage of a preemptive RTOS.

1.2.4 SYNDEX and the AAA methodology

The Algorithm-Architecture Adequation methodology (AAA) [GLS99] and the associated tool SYNDEX provide a framework for programming distributed real-time embedded systems, based on a graphs formalism with synchronous semantics. The functional aspects of a system are described by the Algorithm part, representing computations as a data-flow graph repeated infinitely. For the most part, the

Algorithm part transposes synchronous languages to a graphs formalism (close to SIGNAL or LUSTRE). In addition, this methodology allows to specify the hardware Architecture on which the Algorithm must be implemented. The Architecture is a graph where nodes correspond to either computation Operators or communication Media and edges correspond to connections between Operators and Media. The programmer must also provide execution times for the computation of the operations of the Algorithm on the different Operators of the Architecture and the transfer times for the different data-types that can be transferred by the media. The Adequation then consists in searching for an optimized implantation of the Algorithm on the Architecture. As the distributed scheduling problem is NP-complete, the Adequation is an heuristic and not an exact algorithm (it may not always find the optimal implantation but only a good implantation).

Programming our example with SYNDEX is very similar to what we did with LUSTRE. We can however give a better version of the example programmed with a slow base rate as shown in Fig. 1.12. The Algorithm part retains the structure of the LUSTRE version (Fig. 1.7): during a graph iteration (a slow period), the input is acquired 10 times (`i0`, `i1`, ..., `i9`), the output is produced 10 times (`o0`, `o1`, ..., `o9`) and the `FDIR` is repeated 10 times. Slow to fast communications are performed using a delay (**`pre`**). Operations `t100`, `t200`, ..., `t900` are "timing operations". They have a duration of 100ms and are executed on a specific timer processor (`timerProc` in the Architecture part), while the other operations are all executed on the main processor (`mainProc`). By adding a precedence from timing operations to acquisition operations (`i0`, `i1`, ...), we force fast operations to execute after a certain date. For instance, the acquisition `i1` cannot be performed before date 100ms. This gives a better control on the date at which fast operations take place and thus improves the previous LUSTRE version.

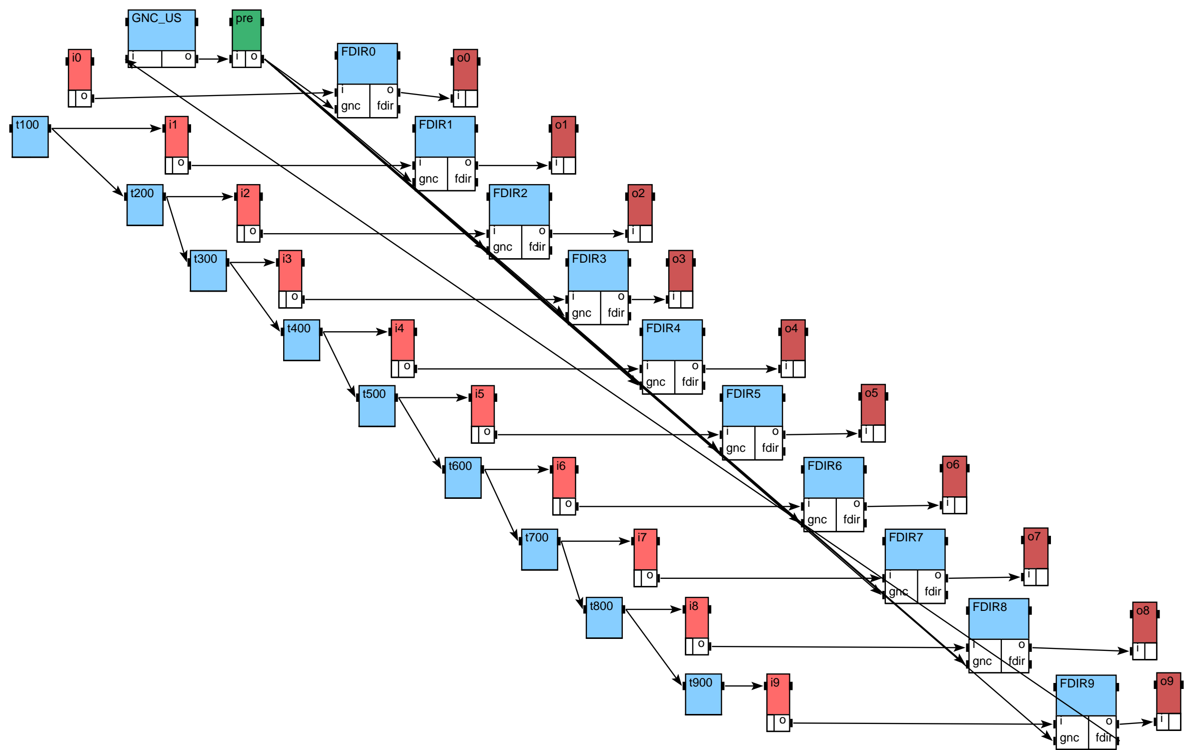
This version still does not completely respect the requirements. Indeed, while we ensure that fast operations do not execute too early (before their period), nothing in the program ensures that they will not execute too late (after their period), so the Adequation may choose a schedule that does not respect these constraints. Furthermore, implementing the timing operator and timing operations may not be simple in practice.

The support of multi-periodic systems for the AAA methodology has first been studied in [Cuc04, CS02] and then in [KCS06, Ker09]. Multi-periodic systems are officially supported by SYNDEX since the version 7 of the software, which was released in August 2009. We had very little time to experiment, according to these early experiments, the programmer can specify a data-dependency between two operations of different periods if the period of one task is a multiple of the other. Communication schemes are restricted to communications without loss. For instance, if the producer is n times faster than the consumer, each instance of the consumer consumes data produced by n repetitions of the producer.

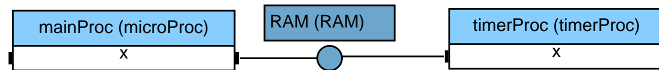
1.2.5 Synchronous Data-Flow (SDF)

Synchronous Data Flow [LM87b] is a special case of data-flow graphs where the number of data-samples produced and consumed by each node are known a priory, originally designed for programing concurrent Digital Signal Processing applications. A SDF graph consists of a set of nodes representing computations, related by edges representing data-dependencies. As for synchronous languages, edges represent infinite streams of values and nodes represent computations repeated infinitely. Each input or output of a node is annotated with a number specifying the number of data-samples the node must consume before performing its computations and the number of data-samples it produces at the end of the computation. The number of data-samples n produced by the source node of an edge and the number of data-samples n' consumed by the destination node of the edge are not necessarily the same, thus the execution of the graph requires to repeat the source node k times for each k' repetitions of the destination node, such that $kn = k'n'$.

The figure 1.13 gives a simple example of SDF graph. To satisfy data-sample rates, a repetition of



(a) The Algorithm



(b) The Architecture

Figure 1.12: Programming a multi-rate communication loop in SYNDEX, with a slow base rate

the SDF graph consists of a single repetition of `C` and `OUT`, 2 repetitions of `B` (for each `C`), 6 repetitions of `IN` and `A` (3 for each repetition of `B`).

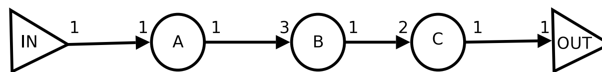


Figure 1.13: A SDF graph

Compiling a SDF graph consists in finding a schedule for the execution of the nodes such that the data-dependency order and data-sample rates are respected. There are several possible schedules for the example of Fig. 1.13:

- `IN`, `A`, `IN`, `A`, `IN`, `A`, `B`, `IN`, `A`, `IN`, `A`, `IN`, `A`, `B`, `C`, `OUT`;
- `IN`, `A`, `IN`, `A`, `IN`, `A`, `IN`, `A`, `IN`, `A`, `IN`, `A`, `B`, `B`, `C`, `OUT` (we changed the order for `B`).
- And many others.

The SDF graph for the example of Fig. 1.2 is given in Fig. 1.14. Each time the node GNC_US executes, it consumes 10 data-samples produced by the node FDIR. As the node FDIR only produces one data-sample at each execution, there must be 10 repetitions of FDIR before each execution of GNC_US. GNC_US produces 10 data-samples at each execution, one for each repetition of FDIR. Communication from GNC_US to FDIR is delayed (the D on the edge) to prevent a causality loop.

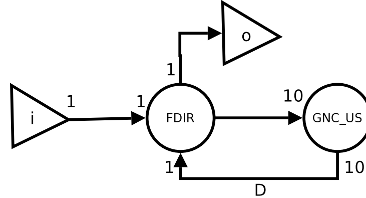


Figure 1.14: SDF graph for a multi-rate communication loop

As node rates are determined by data-sample ratios between related nodes, the communication patterns we can specify are quite restricted. It is for instance not possible to specify a program where GNC_US samples only one out of ten successive values produced by FDIR and where FDIR still executes 10 times more often than GNC_US. More importantly, while SDF graphs specify *multi-rate* systems, they do not necessarily specify *multi-periodic* systems. Indeed, nothing requires the successive repetitions of a node to be executed at regular intervals of time, so from the real-time point of view nodes are *not periodic*. For instance, in the second schedule we gave for the example of Fig. 1.13, nodes clearly do not execute periodically.

1.2.6 Globally Asynchronous Locally Synchronous (GALS) Systems

While extending synchronous languages to support multi-periodic systems is quite a recent concern, the more general class of globally asynchronous and locally synchronous systems is a well-studied topic. In a GALS system, several locally synchronous systems are assembled using asynchronous communication mechanisms to produce a globally asynchronous complete system. Synchronous languages targeted for GALS systems are also often referred to as multi-clock/polychronous languages. For instance, multi-clock ESTEREL [BS01] and POLYCHRONY [LGTLL03] study GALS systems in general. The Quasi-Synchronous approach [CMP01], the N-Synchronous approach [CDE$^+$06, CMPP08] or Latency Insensitive Designs [CMSV01] focus on more specific classes of locally synchronous systems that can be composed "nearly" synchronously using bounded memory communication mechanisms. While these systems are not completely synchronous, they retain a globally predictable behaviour, which makes them suitable for formal verification and/or efficient compilation.

A possible approach to our problem would be to consider multi-periodic systems as a special case of GALS systems to reuse existing work. However, multi-periodic systems are in essence completely synchronous, treating them as such allows to more clearly identify the real-time properties of a system (during the specification), provides better analysis possibilities (proving the correct synchronization of the operations, performing schedulability analysis) and allows the production of more optimized code (specific communication protocols).

1.3 Matlab/Simulink

SIMULINK [Mat] is a high-level modelling language developed by The Mathworks, widely used in many industrial application domains. SIMULINK was originally designed for specification and simulation pur-

poses but commercial code generators have been developed since, such as REAL-TIME WORKSHOP EMBEDDED CODER from Mathworks, or TARGETLINK from dSpace. The major difficulty for using SIMULINK for programming critical systems is that SIMULINK and the associated tools lack formal definition. For instance, SIMULINK has several semantics, which depend on user-configurable options and are only informally defined. Similarly, the conservation of the semantics between the model simulation and the generated code (either for RT WORKSHOP or TARGETLINK) is unclear.

As SIMULINK models are mainly data-flow graphs, a translation from SIMULINK to LUSTRE has been proposed in [TSCC05]. This translation only considers a well-defined subset of SIMULINK models and allows to benefit from the formal definition of LUSTRE. This enables formal verification and automated code generation that preserves the semantics of the original SIMULINK model.

1.4 Architecture description languages

An Architecture Description Language (ADL) provides an additional layer of abstraction compared to the languages studied previously, by focusing on modelling the interactions between high-level components of a system. Typically, in our context, ADLs focus on the specification of interactions between tasks instead of interactions between functions or nodes. Function implementations still need to be provided at some level but this level of detail is hidden in an ADL specification. Tasks specified in an ADL specification are considered as “black-boxes”, implemented outside the ADL specification. Such a multi-layer approach reflects the organization of the development process for complex systems such as embedded control systems. Different engineers first develop separately the different functionalities of the system and then a different person, the integrator, assembles the different functionalities to produce the complete system. The integrator needs a high level of abstraction to coordinate with other engineers: ADLs can address this issue.

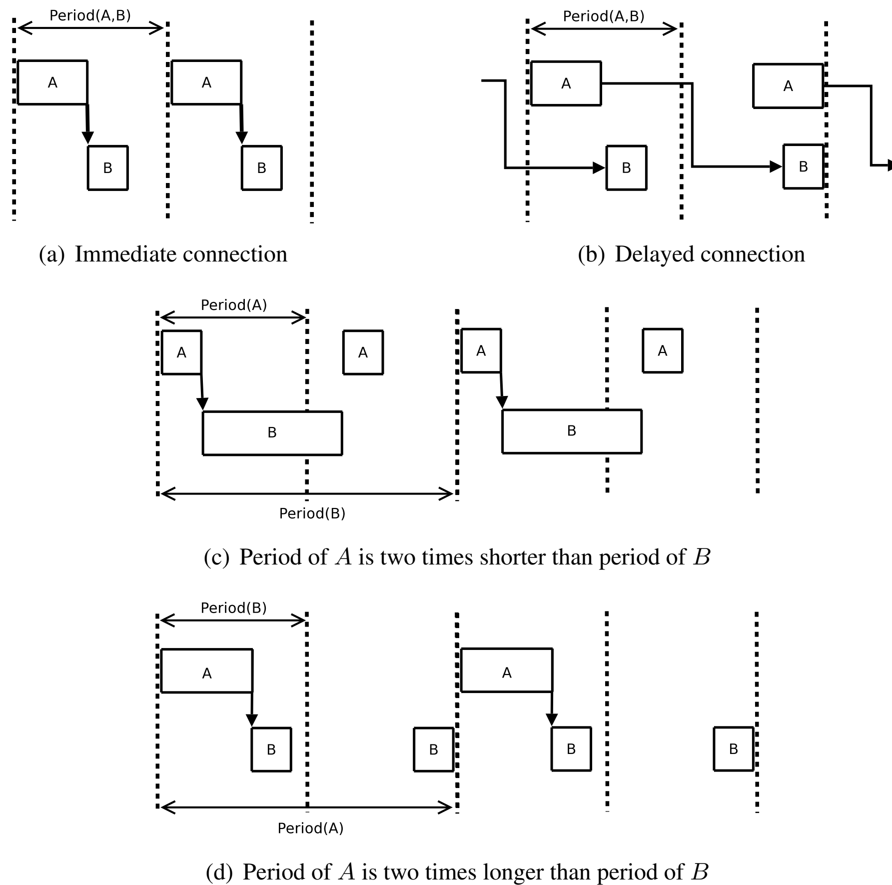
1.4.1 AADL

The Architecture Analysis and Design Language (AADL) [FGH06] is an architecture description language, used for modelling the software and hardware architecture of an embedded real-time system. An AADL design consists of a set of *components*, which interact through *interfaces*. Components can be separated into three categories: application software, execution platform (hardware) or composite components (composition of other components). A large variety of distinguishing properties can be specified on each component. We will only detail the software components as handling the hardware aspects of a system is out of the scope of our work.

A software component can be a thread (a schedulable unit of concurrent execution), a thread group (a compositional unit for organizing threads), a process (a protected address space), data (data types and static data in source text) or a subprogram (callable sequentially executable code). In our case, the different operations of a system can be modelled as threads grouped in a single process. A *component type* defines the externally visible characteristics of the component while a *component implementation* specifies the internal structure of a component in terms of sub-components, interactions between sub-components and component properties. Component properties allow to specify component real-time characteristics, including dispatch protocol (periodic, aperiodic, sporadic), periods, worst-case execution times and deadlines.

Components can interact through connections between elements defined in their interface, in particular data-communications can be specified through connections between component *ports*. Connections can either be immediate (between simultaneous dispatches of the communicating threads) or delayed (from the current dispatch of the producer to the next dispatch of the consumer). Immediate connections

impose a precedence order from the producer to the consumer of the data, while in the case of delayed connections the two threads can be scheduled concurrently. Connections can also relate threads of different periods, either immediately or with a delay. These different communications schemes are illustrated in Fig. 1.15. Finally, end-to-end latency constraints can be imposed on *flows*, where a flow is a path through sub-components of a component.



(a) Immediate connection (b) Delayed connection

(c) Period of A is two times shorter than period of B

(d) Period of A is two times longer than period of B

Figure 1.15: Communications schemes in AADL, for a connection from A to B

The AADL program for our example is given below:

```
package FAS_DATA
public
  data int
  end int;
end FAS_DATA;

package FAS_SW
public
  thread GNC_US
    features
      from_fdir: in data port; to_fdir: out data port;
    properties
      Dispatch_Protocol => Periodic;
      Period => 1000 Ms;
      Compute_Execution_Time => 200 Ms .. 220 Ms;
  end GNC_US;

  thread FDIR
    features
```

```
      i: in data port; from_gnc: in data port;
      o: out data port; to_gnc: out data port;
    properties
      Dispatch_Protocol => Periodic;
      Period => 100 Ms;
      Compute_Execution_Time => 10 Ms .. 20 Ms;
  end FDIR;

  process FAS
    features
      i: in data port; o: out data port;
  end FAS;

  process implementation FAS.FAS_Impl
    subcomponents
      gnc_us: thread GNC_US;
      fdir: thread FDIR;
    connections
      i_cnx: data port i -> fdir.i;
      o_cnx: data port fdir.o -> o;
      gnc_fdir_cnx: data port gnc_us.to_fdir ->> fdir.from_gnc;
      fdir_gnc_cnx: data port fdir.to_gnc -> gnc_us.from_fdir;
   end FAS.FAS_Impl;
end FAS_SW;
```

The package `FAS_DATA` declares the data types used in the program. The package `FAS_SW` specifies the software architecture of the program. It consists of one thread declaration for each operation of the program (`GNC_US` and `FDIR`), the declaration of the global system as a process (`FAS`) and its implementation (`FAS.FAS_impl`). Each thread declaration specifies the inputs and outputs of the corresponding operation along with its period and execution time (declared with an upper and a lower bound). The implementation `FAS.FAS_Impl` specifies the connections (data-dependencies) between the inputs and outputs of the system and the inputs and outputs of the threads. All connections are immediate except the connection `gnc_fdir_cnx`, which is delayed (this is denoted using the double arrow `->>`).

The AADL approach enables to design complex real-time systems with a high-level of abstraction. The main restriction of this approach is that the programmer only has a limited choice of communication schemes and cannot define its own schemes.

1.4.2 REACT/CLARA

The REACT project [FDT04] proposes a toolkit dedicated to real-time system design. It combines formal modelling, using the Architecture Design Language CLARA [Dur98], with formal verification techniques based on Timed Petri Nets [BD91]. The language CLARA is dedicated to the description of the functional architecture of a reactive system.

Similarly to AADL, CLARA describes a system as a set of high level interacting components. A *system* component consists of a set of communicating *activity* components. An activity is triggered repeatedly by the activation law defined for its *start* port and signals its completion on its *end* port. An activity communicates with other activities by means of input or output exchange ports. Each time it is triggered, the activity executes a finite sequence of actions. An action either corresponds to an external function call (the execution duration of which is specified in the action), to an input acquisition or to an output production.

Occurrence generator components specify the input data or input signals of the system. They occur either periodically or sporadically. Activities can also access to *shared resources* or *shared variables* components.

A *link* connects output ports to input ports. To simplify, here we consider that a link connects an input port to an output port using only a *protocol* block and possibly an *operator* block. CLARA actually

allows more complex link definitions connecting more than two ports together. The protocol of a link specifies the policy used for transferring data or signals between the ports of the link. The protocol can be rendez-vous (the consumer and the producer must wait for each other before executing the transfer), blackboard (the activities are not synchronized, the consumer simply uses the last value produced by the producer) or mailbox (the producer and the consumer communicate through a FIFO). The size of the mailbox can be bounded or unbounded. When the mailbox is bounded, the producer must wait if the mailbox is full (until the consumer consumes a value) and the consumer must wait if the mailbox is empty (until the producer produces a value). Operators can be combined with protocols to modify their behaviour when a link connects two activities of different rates. The operator (n) appears on the producer side and specifies that n repetitions of the consumer read the same value. The operator $(1/n)$ appears on the consumer side and specifies that only one out of n successive produced values will be consumed.

Our example can be programmed as shown in Fig. 1.16. The system consists of two activities `FDIR` and `GNC_US`. It reads one data source from the environment (`i`) and produces one output value (`o`). The start port of an activity is represented on its top and the end port on its bottom. The other ports are data exchange ports. The occurrence generator `ck` emits a signal every 100ms. This signal is consumed by `FDIR` using a bounded mailbox of size 1. This triggers `FDIR` each 100ms. The end port of `FDIR` is linked to the start port of `GNC_US` using a bounded mailbox of size 1. As there is an operator (1/10) on the link, `GNC_US` is triggered only once every 10 repetitions of `FDIR`. Similarly, only one out of 10 values produced by `FDIR` (port `vfdir`) are consumed by `GNC_US`. On the opposite, the operator (10) on the link from `GNC_US` to `FDIR` specifies that 10 repetitions of `FDIR` consume the same value produced by `GNC_US`.

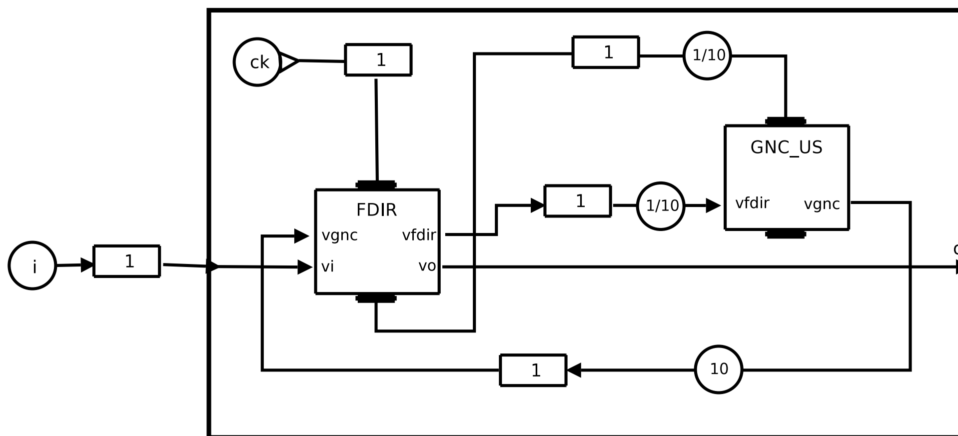


Figure 1.16: A multi-rate communication loop in CLARA

This approach enables the specification of multi-periodic systems. Our language retains some of the concept introduced here but relies on a different semantics (the synchronous approach). As far as we know, the tools developed in the REACT context use CLARA for formal verification but automated code generation is not supported yet.

1.4.3 UML and CCSL

The Unified Modeling Language (UML) [Obj07b] is a general-purpose modeling language, consisting of a set of graphical notations for the creation of abstract models of a system. Its semantics is purposely loose and can be refined through the definition of domain specific *profiles*. Such a profile has recently

been defined for real-time embedded systems: the UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) [Obj07a]. This profile includes the *Clock Constraint Specification Language* (CCSL) [AM09] for specifying the timing properties of the system. This language is inspired from Synchronous Languages but is targeted for the more general context of Globally Asynchronous and Locally Synchronous (GALS) systems.

CCSL provides constructs based on clocks and clock constraints to specify timing properties. Clocks can either be dense or discrete. In our context, we only consider discrete clocks, which consist of a sequence of instants with abstract durations. The language supports a rich variety of clock constraints, which specify constraints on the ordering of instants of different clocks. This includes in particular the possibility to define periodic clocks and periodic clock sampling. CCSL is very recent and for now it is targeted for system specification and system verification, not for automated code generation. As it allows to specify very general clock constraints, efficient code generation would most likely require to only consider a subset of the clock constraints provided by the language.

1.4.4 GIOTTO

GIOTTO [HHK03] is a time-triggered architecture language designed for the implementation of real-time embedded systems. A GIOTTO program consists of a set a multi-periodic tasks grouped into modes, where each task is implemented outside GIOTTO.

A task is defined by a set of input and output ports, by the external function implementing it and by a frequency, relative to the mode containing the task. A mode contains a set of tasks and data-dependencies between the ports of the tasks. A period P is specified for each mode and a frequency f is specified for each task of the mode. The period of a task is then P/f. A GIOTTO program finally consists of a set of modes and mode switches (Boolean conditions).

Data produced by a task is considered available only at the end of its period. On the opposite, when a task executes it can only consume the last values of its inputs produced before the beginning of its period, not the values produced during the current period. This is illustrated in Fig. 1.17. In this example, `B` consumes data produced by `A` and the two tasks have the same period (depicted by vertical dashed lines). Data production and consumption dates are depicted by vertical arrows. We can see that `B` consumes the value produced by `A` during the previous period, not during the current period. This communication pattern amounts to always using a delay in LUSTRE.

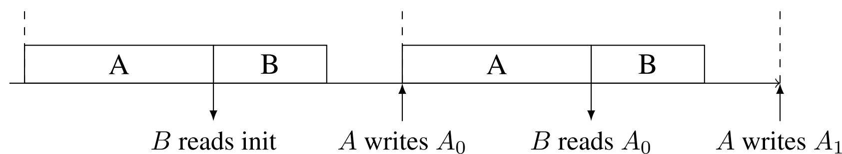


Figure 1.17: Communications in GIOTTO

The GIOTTO program for our example is given in Fig. 1.18, it consists of two tasks (`FDIR` and `GNC_US`) and a single mode. The period of the mode is 1s, the frequency of `GNC_US` is 1 so its period is 1s, the frequency of `FDIR` is 10, so its period is 100ms. Task communications are cyclic but this does not imply a causality loop as all communications are delayed.

The implicitly delayed communications of GIOTTO are very restrictive and imply high latencies between an input acquisition and the corresponding output production. In the example of Fig. 1.17, if a system input `i` is acquired by `A` and processed by the sequence of tasks `A`, `B`, to produce a system output `o`, the latency between `i` and `o` equals two periods of `A` (or `B`) instead of the sum of the execution times of `A` and `B`. Such communication patterns are clearly not adapted to the systems we consider.

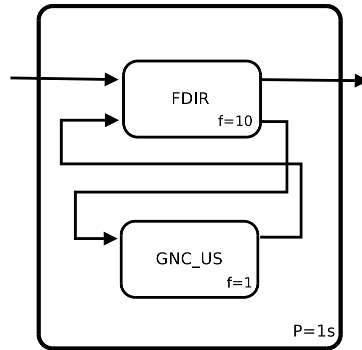


Figure 1.18: Programming a multi-rate communication loop in GIOTTO

1.5 Motivation of our work

As we can see, existing programming languages only partially address the implementation of embedded control systems with multiple real-time constraints. Programming with traditional imperative languages coupled with real-time APIs allows to implement a very large class of applications but the low-level of abstraction makes the task tedious and error-prone. Synchronous languages provide a higher level of abstraction with formal semantics, which makes them well suited for critical systems. They support automated and formally defined low-level code generation, which greatly simplifies the development process and produces correct-by-construction low-level code. The level of abstraction of synchronous languages is unfortunately not very well-adapted for systems with multiple real-time constraints. SIMULINK provides a data-flow model similar to synchronous languages but lacks formal semantics and automated code generation tools do not ensure that the generated code preserves the semantics of the original model. Compared to synchronous languages, architecture design languages propose an additional level of abstraction, by focusing on the description of the real-time software architecture of a system in terms of tasks and task interactions. ADLs combined with lower-level programming languages to implement the system tasks are well-suited for the design of embedded control systems with multiple real-time constraints. The description of task interactions however remains quite limited in existing ADLs, which calls for a more expressive language.

Based on these observations, we propose a language built as a real-time software architecture design language that transposes the synchronous semantics to the architecture design level. Defining the language as an ADL provides a high-level of abstraction, while relying on a synchronous semantics benefits from well-established formal properties.

A program of this language consists of a set of *imported nodes*, implemented outside the program using other existing languages (C or LUSTRE for instance), and data-flows between the imported nodes. The language allows to specify multiple deadline and periodicity constraints on flows and nodes. It also defines a small set of rate transition operators that allow the programmer to precisely define the communication patterns between nodes of different rates. A program is automatically translated into a set of concurrent tasks implemented in C, which can be executed with existing real-time operating systems.

The language can be compared with other languages as follows:

- **ADLs**: The level of abstraction of the language is the same as that of existing ADLs but the language relies on a different semantics (the synchronous approach) and allows the programmer to define its own communication patterns between tasks;

- **Synchronous languages**: Though the language has a synchronous semantics, it addresses a differ-

ent phase of the development (the integration) compared to existing synchronous languages, and enables efficient translation into a set of real-time tasks;

- **Imperative languages**: Though a program is in the end translated into imperative code, the generated code is correct-by-construction. Thus, analyses can directly be performed on the original program rather than on the generated code.

1.6 Outline of the thesis

The first part of the thesis focuses on the language definition. First, we present the synchronous approach in details in Chapter 2, as it is the fundamental basis of our work. Then, in Chapter 3 we introduce real-time aspects in the synchronous approach by defining a particular class of clocks called *strictly periodic clocks*, which enable the definition of multi-periodic systems. Chapter 4 presents the language in detail, with its complete syntax and formal semantics. The real-time primitives of the language are based on strictly periodic clocks. The Chapter 5 illustrates the expressiveness of the language by providing a series of examples of different multi-periodic communication patterns. Finally, the static analyses of the language, which ensure that the semantics of a program is well-defined, are detailed in Chapter 6.

The second part of the thesis details the compilation of the language into a set of real-time tasks. Chapter 9 first describes how dependent real-time tasks (task related by data-dependencies) are extracted from a program. In Chapter 10, we propose to translate the set of dependent tasks into an equivalent set of independent tasks, where precedences are encoded in task real-time attributes and where a buffering communication protocol ensures communication consistency. The independent task set can then easily be translated into low-level code as explained in Chapter 11. We present a prototype implementation of the compiler in Chapter 12.

The third part of the thesis studies the implementation of the complete FAS.

Part I

Language Definition

Chapter 2

The Synchronous Approach

In this chapter we present the basic notions of the synchronous approach and focus more particularly on data-flow synchronous languages. We take LUSTRE [HCRP91] as example because it is the fundamental basis of the commercial development environment SCADE [Est], often used for programming aeronautic and aerospace systems.

2.1 Logical time and instants

The key feature of the synchronous approach [BB01] is that it abstracts from real-time and instead describes the behaviour of a system on a *logical* time scale. The computations performed by the system are split into a succession of *instants*. At each instant, the system repeats the same behaviour: acquire inputs and perform computations on them to produce outputs (a *reaction* of the system). For the system to respect the *synchronous hypothesis*, at each instant computations must complete before the current instant ends and the next one begins. The exact real-time dates at which computations take place during the instant are irrelevant, as long as computations respect this hypothesis. As far as logical time is concerned, computations are performed *instantaneously* (literally during the instant). This is however only a conceptual abstraction and does of course not require computations to have no duration in practice.

By abstracting from real-time, the synchronous approach greatly simplifies system design. The designer does not need to specify when computations occur but only the order in which they must be performed (during each instant). This order is specified by the description of the data-dependencies between the different operations the system is composed of. The formal semantics of synchronous languages along with a series of static analysis performed by their compilers require the designer to give a precise and non-ambiguous description of the system (in particular of the data-dependencies), which allows the compilers to generate completely deterministic code.

2.2 Verifying the synchronous hypothesis

Though the synchronous approach aims at abstracting from real-time for system design, real-time must still be taken into account at some point of the development process. The programmer has to verify that his program respects the synchronous hypothesis. This means that he must ensure that computations always complete before the end of the instant.

The duration of an instant is defined differently depending on the class of systems we consider:

- In *purely reactive systems*, the duration of an instant is determined by the arrival of some event that triggers the beginning of the instant. The next instant starts at the next arrival of an event. In

practice the durations of instants in a purely reactive system are hard to predict and can vary from one instant to the next.

- In *sampled reactive systems*, the duration of an instant is fixed to a given sampling period and is the same for every instant. A new instant is triggered at each period.

In any case, the programmer needs to determine the worst case execution time (wcet) of the computations performed during each instant and to check that it is lower than the duration of this instant. Computing this wcet may not be a trivial task and can be simplified as follows:

- Synchronous languages do not allow computations with unbound execution times such as loops or recursion (with the exception of loops the size of which can be determined statically [MM04]).

- If some processes are activated only during certain instants, according to a condition that cannot be evaluated statically, the process is considered as always active and its execution time must be taken into account for every instant.

- The duration of the different processes of the system are over-approximated.

These restrictions may not always be negligible, but are acceptable for critical systems as they enable to produce a deterministic implementation of the system.

2.3 Overview of synchronous languages

The synchronous approach has been implemented in several programming languages. A complete survey can be found in [BCE+03]. To summarize, synchronous languages can be separated in two different classes:

- Imperative synchronous languages, among which ESTEREL [BdS91], mainly focus on the description of the control-flow in the system. A program consists of a set of nested and concurrent threads, synchronized by a single global clock and communicating by means of broadcasted signals. At each reaction, control flows through each thread, until it arrives to a pause or termination instruction. At the next reaction, threads resume were they previously stopped.

- Declarative synchronous languages, among which LUSTRE [HCRP91] and SIGNAL [BLGJ91], mainly focus on the description of the data-flow in the system. A program consists of a set of equations that define infinite sequences of values called flows. At each reaction, the program computes the current value in the sequence of each output flow, depending on the current or previous values of the input flows.

2.4 Data-flow synchronous languages, an example: LUSTRE

All the variables and expressions used in a LUSTRE program are *flows*. A flow consists of a sequence of values, potentially infinite, and of a clock. The clock of a flow defines the instants when the flow is present (defined). Each time the flow is present, it takes the next value in its sequence of values. For instance, the constant value 0 stands for an infinite constant sequence. Each time it is present, it has value 0. A LUSTRE program consists of a set of equations, which define the value of flows at each instant.

2.4.1 Basic operators

Classic arithmetical and logical operators are extended point wisely over flows. For instance, let x and y be two flows of the same clock. Let $(x_1, x_2, \ldots, x_n)$ denote the sequence of values of x and $(y_1, y_2, \ldots, y_n)$ denote the sequence of values of y. The sum of the two flows `x+y` has the following sequence of values: $(x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n)$.

In the same way, **`if-then-else`** is the point wise conditional operator. For instance, the absolute value of a flow x can be defined by the following equation:

```
abs=if x<0 then -x else x;
```

The reader should not mistake this operator for its homonym instruction of imperative languages like C. This operator always requires a **`then`** and a **`else`** branch and both branches are *always activated*, whether the condition is true or not. The condition only decides whether the current value of the flow is the value of the **`then`** or of the **`else`** branch. Therefore, this operator cannot be used to activate or deactivate flow computations, instead clock operators are designed for this purpose (see below).

The **`pre`** operator is an instant delay operator, it stands for the previous value of a flow. It is often used along with the `->` initialisation operator, as the value of **`pre`** `x` at its first activation is undefined. The flow `x->y` takes the value of x at its first activation and then the value of y for the remaining activations. For instance, a counter can be defined as follows (its sequence of values is $0, 1, 2, 3, \ldots$):

```
count=0->(pre(count)+1);
```

Notice that a flow can depend on itself, as long as the dependency is not instantaneous (a **`pre`** is required).

2.4.2 Clock operators

The clock operator **`when`** under-samples a flow using a Boolean condition. The flow `x` **`when`** `c` is present and takes the value of x only when c is true. When c is false, the flow is absent, its value is undefined and cannot be accessed. Conversely, the **`current`** operator over-samples a flow, replacing the absent values introduced by **`when`** by the last present value of the flow. This is illustrated in Fig. 2.1, in which we give the value of each flow at each instant. Notice that in general, `x`≠**`current`**`(x` **`when`** `c)`. The operator **`when`** is often used as an *activation condition* to activate or deactivate some processes of the system. Indeed, on the contrary to the **`if-then-else`** operator, in the expression `x` **`when`** `c`, the flow x is not computed at all if c is false.

`count`	0	1	2	3	4	...
`c`	true	false	true	true	false	...
`count` **`when`** `c`	0		2	3		...
`current``(count` **`when`** `c)`	0	0	2	3	3	...

Figure 2.1: Behaviour of clock operators

A flow can be under-sampled several times and then over-sampled several times. The over-sampling of a flow produces a flow on the clock it had before its last under-sampling. This is illustrated in Fig. 2.2. If the flow is already always present, applying a **`current`** is considered as an error.

2.4.3 Nodes

LUSTRE programs are structured into *nodes*. A node contains a set of equations, which defines how its outputs are computed from its inputs. An example is given below with the node $threshold$. The output $reached$ of the node is true when more than N instants have occurred since the last time $reset$ was true:

count	0	1	2	3	4	...
c1	true	false	true	true	false	...
c2	true		false	true		...
x=count **when** c1;	0		2	3		...
y=x **when** c2;	0			3		...
x2=**current**(y);	0		0	3		...
count2=**current**(x2);	0	0	0	3	3	...

Figure 2.2: Nesting clock operators

```
node threshold(const N: int; reset: bool)
returns (reached: bool)
var count: int;
let
  count=0->if reset then 0 else pre(count)+1;
  reached=count>=N;
tel
```

The set of equations of a node can contain calls to other nodes, so a LUSTRE program consists in a hierarchy of nodes. Among those nodes, one is chosen as the main node, the entry point of the program.

2.4.4 Automated code generation

Synchronous languages compilers do not directly generate assembly code, but instead intermediate level code (C code) [Hal93]. This generated code can then be compiled for a variety of embedded systems as micro-processor manufacturers usually provide a C compiler for their processors.

Single-loop

The most intuitive way to generate imperative code from a synchronous program is to translate the program into a single loop, the body of which corresponds to the computations performed during an instant. To do so, the compiler must translate the set of equations of the input program into a sequence of instructions.

The first step is to find an order for the evaluation of the expressions used in the equations that respects data-dependencies between variables. For instance, for the set of equations: `o=v*2;v=i+1;`, we need to evaluate v before evaluating o. This often requires to introduce some auxiliary variables and to split complex expressions into several simpler expressions.

Once expressions are ordered, the compiler generates a variable allocation for each variable of the program and translates each expression into a C instruction. The translation of LUSTRE operators into C is fairly straightforward:

- Arithmetic and logic operators are translated into their C equivalent.
- The **pre** operator introduces an additional variable to store the previous value of a flow. This variable copies the current value of the flow at the end of the loop.
- The -> operator requires to test if the program is in its initial phase.
- The **when** operator is translated into an **if**.
- The **current** operator does not generate code at all.

For instance, the following node programs a reinitialisable counter:

```
node counter(reset:bool) returns (count:int)
let
  count=0->if reset then 0 else pre(count)+1;
tel
```

The C code generated for this program is the following (function $counter_step$ corresponds to one step of the infinite loop):

```
bool reset;
int count;
int _pcount;
bool _init=true;

void counter_step()
{
  if (_init) count=0;
  else
    if (reset) count=0;
    else count=_pcount+1;
  _pcount=count;
  _init=false;
}
```

Node expansion

Node expansion replaces a node call appearing in an equation by the body of the node definition. It is performed as a preliminary step of the code generation. Expanding node calls enables to accept programs that would otherwise be rejected by the syntactic causality analysis. For instance, without expansion the following program is rejected:

```
node invert(i1, i2: int) returns (o1, o2: int)
let
  (o1, o2) = (i2, i1);
tel

node N(i: int) returns (o1, o2)
let
  (o1, o2)=invert(i, o2);
tel
```

However, after expansion (and a few simplifications) this program is clearly causal:

```
node N(i: int) returns (o1, o2)
let
  (o1, o2)=(o2, i);
tel
```

The Verimag LUSTRE compiler expands recursively each node call, while this is only performed on demand in SCADE as it supports modular compilation of nodes. The SCADE compiler generates a C procedure for each non-expanded node definition and the same procedure can be reused if the program

contains several calls to the same node. This generates shorter code, though the trade-off is that causality analysis may reject some correct programs (which can be expanded to solve the problem).

Compiling into automata

The Verimag LUSTRE compiler supports a second compilation technique, which translates a program into automata. This technique borrows from the ESTEREL compiler. It has gradually been abandoned and will not be detailed here. The basic idea is that each state of the automaton generated for a program corresponds to a different valuation for the *state variables* of the program, while transitions, from one program state to another, are associated with the code corresponding to a program reaction. State variables represent the current state of the program.

Chapter 3

Real-Time Periodic Clocks

At first glance, taking real-time into account in a synchronous language may seem contradictory, given that the synchronous hypothesis is often qualified as "zero-time" hypothesis. In this chapter we show that real-time can however be introduced in the synchronous approach, while still retaining its clean and intuitive semantics.

3.1 A model relating instants to real-time

We have seen in Chap. 1 that completely abstracting from real-time renders the programming of multi-periodic systems difficult. Therefore, in this section we present a synchronous model that relates instants to real-time, while retaining the basis of the synchronous approach.

3.1.1 Multiple logical time scales

The main idea of our model is that a multi-periodic system can be considered as a set of locally mono-periodic synchronous processes assembled together to form a globally multi-periodic synchronous system. Locally, each process has its own logical time scale, its own notion of instant and must respect the usual synchronous hypothesis, which requires it to complete its computations before the end of the current instant of this logical time scale. When we assemble processes of different rates, we assemble processes of different logical time scales so we need a common reference to compare instants belonging to different logical time scales. This common reference is the real-time scale.

For instance, for the FAS, operations $GyroAcq$, `FDIR` and `PDE` share the same logical time scale, where instants have a duration of 100ms, while `Str Acq` and `TM/TC` share another one, where instants have a duration of 10s. The programmer can describe the behaviour of the different operations separately with the usual synchronous approach, but when he assembles the operations he needs to be able to compare "fast" instants with "slow" instants. When we consider the real-time duration of instants, it is natural to say that one slow instant corresponds to 100 fast instants. Our model formalizes this idea by relating instants to real-time.

3.1.2 Real-time flows

Our synchronous real-time model relies on the Tagged-Signal Model [LSV96]. Given a set of *values* $\mathcal{V}$, a *flow* is a sequence of pairs $(v_i, t_i)_{i\in\mathbb{N}}$ where v_i is a value in $\mathcal{V}$ and t_i is a tag in $\mathbb{Q}$, such that for all i, $t_i < t_{i+1}$. The clock of a flow f, denoted $ck(f)$, is its projection on $\mathbb{Q}$. Similarly, $val(f)$ denotes the sequence of values of f (its projection on $\mathcal{V}$). Two flows are *synchronous* if they have the same clock. A

tag represents an amount of time elapsed since the beginning of the execution of the program. We say that a flow is *present* at date t if t appears in its clock, *absent* otherwise.

Following the *relaxed synchronous hypothesis* of [Cur05], at each activation a flow is required to be computed before its next activation. Thus, each flow has its own notion of instant and the duration of the instant t_i is $t_{i+1} - t_i$. Two flows are synchronous if the durations of their instants are the same. We can compare the durations of the instants of two flows to determine if a flow is faster than the other and more importantly, we can determine *how much faster* this flow is. For instance, in the example of the FAS, the outputs of `Gyro Acq` have clock $(0, 100, 200, \ldots)$, while the outputs of `Str Acq` have clock $(0, 10000, 20000, \ldots)$, so the outputs of `Gyro Acq` are 100 times faster.

3.2 Strictly Periodic Clocks

In this section we introduce a new class of clocks, strictly periodic clocks, along with specific transformations to handle transitions between different logical time scales.

3.2.1 Definition

A clock is a sequence of tags, we define a particular class of clocks called strictly periodic clocks as follows:

Definition 1. *(*Strictly periodic clock*). A clock $h = (t_i)_{i \in \mathbb{N}}$, $t_i \in \mathbb{Q}$, is* strictly periodic *if and only if:*

$$\exists n \in \mathbb{Q}^{+*},\ \forall i \in \mathbb{N},\ t_{i+1} - t_i = n$$

n is the period *of h, denoted $\pi(h)$ and t_0 is the* phase *of h, denoted $\varphi(h)$.*

A strictly periodic clock defines the real-time rate of a flow. The notion of phase defined above is a little more general than usual as we do not impose that $\varphi(h) < \pi(h)$. A strictly periodic clock is uniquely characterized by its phase and by its period:

Definition 2. *The term $(n, p) \in \mathbb{Q}^{+*} \times \mathbb{Q}$ denotes the strictly periodic clock α such that:*

$$\pi(\alpha) = n,\ \varphi(\alpha) = \pi(\alpha) * p$$

3.2.2 Periodic clock transformations

We then define clock transformations specific to strictly periodic clocks, which produce new strictly periodic clocks:

Definition 3. *(*Periodic clock division*). Let α be a strictly periodic clock and $k \in \mathbb{Q}^{+*}$. "$\alpha /_. k$" is a strictly periodic clock such that:*

$$\pi(\alpha /_. k) = k * \pi(\alpha),\ \varphi(\alpha /_. k) = \varphi(\alpha)$$

Definition 4. *(*Periodic clock multiplication*). Let α be a strictly periodic clock and $k \in \mathbb{Q}^{+*}$. "$\alpha *_. k$" is a strictly periodic clock such that:*

$$\pi(\alpha *_. k) = \pi(\alpha)/k,\ \varphi(\alpha *_. k) = \varphi(\alpha)$$

Definition 5. *(*Phase offset*). Let α be a strictly periodic clock and $k \in \mathbb{Q}$. "$\alpha \rightarrow_. k$" is a strictly periodic clock such that:*

$$\pi(\alpha \rightarrow_. k) = \pi(\alpha),\ \varphi(\alpha \rightarrow_. k) = \varphi(\alpha) + k * \pi(\alpha)$$

Divisions and multiplications respectively decrease or increase the frequency of a clock while phase offsets shift the phase of a clock. This is illustrated in Fig. 3.1. Again, we do not constrain that $\varphi(\alpha \rightarrow_. k) < \pi(\alpha)$.

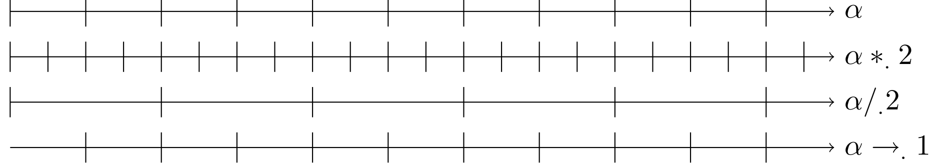


Figure 3.1: Strictly periodic clocks.

Let $\mathcal{P}$ be the set of strictly periodic clocks. From previous definitions we directly obtain Properties 1, 2 and 3.

Property 1. $\forall \alpha, \beta \in \mathcal{P}$*:*

- $\forall k \in \mathbb{Q},\ \alpha = \beta \rightarrow_. k \Leftrightarrow \beta = \alpha \rightarrow_. -k$*;*
- $\forall k \in \mathbb{Q}^{+*},\ \alpha = \beta /_. k \Leftrightarrow \beta = \alpha *_. k$*.*

Property 2. $\forall (n,p) \in \mathbb{Q}^{+*} \times \mathbb{Q}$*:*

- $\forall k \in \mathbb{Q}^{+*},\ (n,p) *_. k = (\frac{n}{k}, kp)$*;*
- $\forall k \in \mathbb{Q}^{+*},\ (n,p) /_. k = (nk, \frac{p}{k})$*;*
- $\forall k \in \mathbb{Q},\ (n,p) \rightarrow_. k = (n, p+k)$*.*

Property 3. *The set of strictly periodic clocks* $\mathcal{P}$ *is closed under operations* $*_.$*,* $/_.$ *and* $\rightarrow_.$*.*

3.2.3 Integer Strictly Periodic Clocks

We defined the general model of strictly periodic clocks using dates in $\mathbb{Q}$. However, to define a compilable language, we need to restrain to dates in $\mathbb{N}$. Indeed operating systems rely on a discrete time model and there is always a lower bound on the level of granularity that can be used to describe time. For instance the date $1/3$ does not exist in a real system, or is approximated. For the same reasons, scheduling theory, needed to implement our systems, only applies to dates and durations in $\mathbb{N}$. We consequently constrain flow tags to be elements of $\mathbb{N}$ and the parameter k used in periodic clock divisions and periodic clock multiplications to be an element of $\mathbb{N}^*$. The parameter k used in phase offsets remains an element of $\mathbb{Q}$. In the rest of the document we will only consider such *integer strictly periodic clocks* and we will refer to them simply as strictly periodic clocks.

In the set of integer strictly periodic clocks, clock (n,p) is a valid strictly periodic clock if and only if: $n \in \mathbb{N}^{+*}, n * p \in \mathbb{N}^+$. Indeed, clocks that do not verify this property refer to dates that are not in $\mathbb{N}$. This restricted set is not closed under operations $*_.$ and $\rightarrow_.$ anymore (but is still closed under $/_.$). We use notation $k|k'$ to say that k divides k' (the rest of the integer division is 0). First, let (n,p) be a strictly periodic clock, if $k|n$ then $(n,p) *_. k$ is a valid strictly periodic clock, but otherwise it is not, as this clock refers to dates which value is not in $\mathbb{N}$. Second, if $(p+k)n \in \mathbb{N}^+$ then $(n,p) \rightarrow_. k$ is a valid strictly periodic clock, but otherwise it is not, as this clock refers to negative dates. The clock calculus, presented in Chap. 6.3, will check that no such invalid clock is used in a program.

3.3 Strictly periodic clocks and Boolean clocks

Strictly periodic clocks and their associated transformations are not meant for replacing the usual Boolean clocks of the synchronous languages. Our language actually uses both classes of clocks. These two classes correspond to complementary notions: strictly periodic clocks define the real-time frequency of a flow, while Boolean clocks define the activation condition of a flow.

To support Boolean clocks in our language, we adapt the definition of the Boolean clock operator `on` of [CP03] to our model. The behaviour of operator `on` is defined inductively on the sequence of tags of a clock. Let the term $t.s$ denotes the clock whose head is the tag t and whose tail is the sequence s. Similarly, let the term $(v, t).s$ denotes the flow whose head has value v and tag t and whose tail is sequence s. For all clock α, for all Boolean flow c, α on c is a clock defined inductively as follows:

$$(t.ck) \texttt{ on } ((true, t).c) = t.(ck \texttt{ on } c)$$
$$(t.ck) \texttt{ on } ((false, t).c) = ck \texttt{ on } c$$

The operation α on c produces a new clock containing only the tags of α at which c is true. The opposite operation α on not c only keeps tags when c is false. The operations α on c or α on not c are valid only if c has clock α. The next chapter will illustrate how Boolean clocks and strictly periodic clocks can be combined in the same program.

3.4 Summary

In this chapter, we have defined:

- A synchronous model that explicitly relates instants to real-time;
- The class of strictly periodic clocks, which allow to define the real-time rate of a flow;
- Periodic clock transformations, which allow to compare the rates of two flows.

This enables the specification of real-time properties in a synchronous context. These new concepts serve as the basis for the real-time operators defined in the next chapter.

Chapter 4

Language Syntax and Semantics

This chapter details the syntax and the semantics of our language. It is built as an extension of LUSTRE and borrows some constructions from LUCID SYNCHRONE [Pou06] and from recent versions of the SCADE language. It extends data-flow synchronous languages with high-level real-time primitives based on strictly periodic clocks.

4.1 Types polymorphism

The language supports types polymorphism, which is now a pretty common feature in many programming languages and in particular in recent synchronous languages. The type of the variables of the program can be left unspecified, in which case they will be inferred by the compiler (see Chap. 6). For instance:

```
node infer(i:int) returns (o)
let o=i; tel
```

In this example, the compiler infers that the type of `o` is `int`.

The following example introduces types polymorphism:

```
node poly(i,j) returns (o,p)
let o=i; p=j; tel
```

All the compiler can say is that `o` and `i` have the same type (say α), and that `p` and `j` have the same type (say β). The node `poly` can then be instantiated with different types, for instance the following program is correct:

```
node inst(i:int; j:bool) returns (o,p,q,r)
let
  (o,p)=poly(i,j);
  (q,r)=poly(j,i);
tel
```

The compiler infers that `o` has type `int`, `p` has type `bool`, `q` has type `bool` and `r` has type `int`.

4.2 Real-time Primitives

4.2.1 Declaring real-time constraints

The language aims at specifying real-time constraints with a high-level of abstraction. The primitives we introduce are targeted for specifying constraints related to the environment of the system (physical constraints for instance) rather than constraints related to implementation concerns. The inputs and outputs of a node represent its interface with the node environment. Therefore, real-time constraints are specified on node inputs and outputs: they represent environmental constraints.

Periodicity constraints

The period of a node input or output can be specified as follows:

```
node periodic(i: int rate (10,0)) returns (o: int rate (5,0))
let
...
tel
```

`x: rate (n,p)` specifies that the clock of x is the strictly periodic clock (n, p). Thus, i has period 10 and o has period 5.

The user can also specify the phase of a periodic flow:

```
node phased(i: int rate (10,1/2)) returns (o: int rate (10,0))
let
...
tel
```

In this example, i has period 10 and phase $\frac{1}{2}$, so its clock sequence is $(5, 15, 25, 35, \ldots)$.

Deadline constraints

A deadline constraint defines a deadline for the computation of a flow relative to the beginning of the period of the flow. Deadline constraints can be specified on node outputs:

```
node deadline(i: int rate (10,0)) returns (o: int rate (10,0) due 8)
let
...
tel
```

`x: due d` specifies that x has a deadline constraint of d. If x is present at date t, then x must computed before date $t + d$.

Deadline constraints can also be specified on node inputs, though they have a slightly different meaning in this case:

```
node deadline(i: int rate (10,0) before 2) returns (o: int)
let
...
tel
```

`x: before d` specifies that x should be acquired by the program before the deadline d (ie a deadline for the "production" of this input). In practice, this means that a sensor operation will have to fetch the input before this deadline. Operations that perform computations that require the input will read from the output of this sensor.

Deadline constraints have no impact on the clock of a flow and thus on synchronization constraints between two flows. If a flow f has clock (n, p) and deadline d and a flow f' has clock (n, p) and deadline d' (with $d \neq d'$), the flows are still considered synchronous and can be combined as they are produced during the same logical instants. Deadlines are only taken into account during the scheduling of the program.

4.2.2 Rate transition operators

We define a series of *rate transition operators*, based on periodic clock transformations, that handle transitions between flows of different rates. These operators enable the definition of user-specified communication patterns between nodes of different rates. Our objective is to provide a small set of simple operators, which can be combined to produce complex communication patterns. Furthermore, the semantics of these operators is defined formally and their behaviour is completely deterministic.

First, we can under-sample a flow using operator $/\hat{}$:

```
node under_sample(i: int rate (5,0))
returns (o: int rate (10,0))
let
  o=i/^2;
tel
```

$e/\hat{}k$ only keeps the first value out of each k successive values of e. If e has clock α then $e/\hat{}k$ has clock $\alpha/_{.}k$. This behaviour is illustrated in Fig. 4.1.

date	0	5	10	15	20	25	30	...
i	i_0	i_1	i_2	i_3	i_4	i_5	i_6	...
o	i_0		i_2		i_4		i_6	...

Figure 4.1: Periodic under-sampling

On the opposite, we can over-sample a flow using operator $*\hat{}$:

```
node over_sample(i: int rate (10,0))
returns (o: int rate (5,0))
let
  o=i*^2;
tel
```

$e *\hat{} k$ duplicates each value of e, k times. If e has clock α, then $e *\hat{} k$ has clock $\alpha *_{.} k$. This behaviour is illustrated in Fig. 4.2.

date	0	5	10	15	20	25	30	...
i	i_0		i_1		i_2		i_3	...
o	i_0	i_0	i_1	i_1	i_2	i_2	i_3	...

Figure 4.2: Periodic over-sampling

4.2.3 Phase offset operators

We define three different operators based on clock phase offsets. We can offset all the values of a flow by a fraction of its period using operator $\sim>$:

```
node offset(i: int rate (10,0))
returns (o: int rate (10,1/2))
let
  o=i~>1/2;
tel
```

If e has clock α, $e \sim> q$ delays each value of e by $q * \pi(\alpha)$ (with $q \in \mathbb{Q}^+$). The clock of $e \sim> q$ is $\alpha \rightarrow_. q$. This operator will most often be used with q lower than 1, but values greater than 1 are accepted. The behaviour of this operator is illustrated in Fig. 4.3.

date	0	5	10	15	20	25	30	...
i	i_0		i_1		i_2		i_3	...
o		i_0		i_1		i_2		...

Figure 4.3: Phase offset

The operator `tail` returns the tail of a flow:

```
node tail_twice(i: int rate(10,0))
returns (o1: int rate(10,1); o2: int rate(10,2))
let
  o1=tail(i);
  o2=tail(o1);
tel
```

`tail` (e) drops the first value of e. If e has clock α, then `tail` (e) has clock $\alpha \rightarrow_. 1$, so `tail` (e) becomes active one period later than e. Several calls to `tail` will drop several values of the flow. This is illustrated in Fig. 4.4.

date	0	10	20	30	40	50	60	...
i	i_0	i_1	i_2	i_3	i_4	i_5	i_6	...
o1		i_1	i_2	i_3	i_4	i_5	i_6	...
o2			i_2	i_3	i_4	i_5	i_6	...

Figure 4.4: Keeping the tail of a flow

The language also contains a concatenation operator "`::`":

```
node init(i: int rate(10,0))
returns (o1: int rate(10,0); o2: int rate(10,0))
var v1,v2;
let
  (v1,v2)=tail_twice(i);
  o1=0::v1;
  o2=0::0::v2;
tel
```

cst `::` e produces cst one period earlier than the first value of e and then produces e. If e has clock α, then cst `::` e has clock $\alpha \rightarrow_. -1$. This is illustrated in Fig. 4.5.

date	0	10	20	30	40	50	60	...
i	i_0	i_1	i_2	i_3	i_4	i_5	i_6	...
v1		i_1	i_2	i_3	i_4	i_5	i_6	...
v2			i_2	i_3	i_4	i_5	i_6	...
o1	0	i_1	i_2	i_3	i_4	i_5	i_6	...
o2	0	0	i_2	i_3	i_4	i_5	i_6	...

Figure 4.5: Flow concatenation

4.2.4 Rate transitions on unspecified rates

The programmer can leave the rates of the inputs or outputs of a node unspecified, they will be inferred by the clock calculus. For instance:

```
node under_sample(i: int rate(10,0)) returns (o: int)
let
  o=i/^2;
tel
```

The clock calculus infers that `o` has clock $(20, 0)$.

Furthermore, rate transition and phase offset operators can be applied to flows the rate of which is unknown. For instance, the following node definition is perfectly valid:

```
node under_sample(i: int) returns (o: int)
let
  o=i/^2;
tel
```

This program simply specifies that the rate of `o` is two times slower than the rate of `i`, regardless of what the rate of `i` may be. The actual rate of the flows will only be computed when instantiating the node. The node can even be instantiated with different rates in the same program, for instance:

```
node poly(i: int rate (10, 0); j: int rate (5, 0))
returns (o, p:  int)
let
  o=under_sample(i);
  p=under_sample(j);
tel
```

The clock inferred for `o` is $(20, 0)$, while the clock inferred for `p` is $(10, 0)$. Such clock polymorphism increases the modularity with which systems can be programmed.

4.2.5 Sensors and actuators

The programmer must specify the wcet for the acquisition of each system input (each input of the main node of the program), called a *sensor* operation, and for the production of each system output (each output of the main node of the program), called an *actuator* operation. For instance, the following program specifies that the acquisition of `i` takes 1 unit of time, the acquisition of `j` takes 2 units of time and the production of `o` takes 2 units of time:

```
imported node A(i, j: int) returns (o: int) wcet 3;
sensor i wcet 1; sensor j wcet 2; actuator o wcet 2;
```

```
node N(i, j: int rate (10, 0)) returns (o: int rate (10, 0))
let
  o=A(i, j);
tel
```

4.3 Delays

A flow can be delayed using the operator `fby` introduced in LUCID SYNCHRONE:

```
node delay(i: int) returns (o: int)
let
  o=0 fby i;
tel
```

cst `fby` e first produces cst and then the values of e delayed by the period of e. The clock of cst `fby` e is the same as the clock of e. This operator was borrowed from LUCID SYNCHRONE. Its behaviour is illustrated in Fig. 4.6. Notice that cst `fby` e is strictly equivalent to cst `::` $(e \sim> 1)$ and could be considered as syntactic sugar. We do however not use this rewriting duration compilation as it does not simplify the compilation much.

i	i_0	i_1	i_2	i_3	i_4	i_5	i_6	...
o	0	i_0	i_1	i_2	i_3	i_4	i_5	...

Figure 4.6: Delaying a flow

4.4 Activation conditions

4.4.1 Boolean sampling operators

A flow can be under-sampled according to a Boolean condition using the classic operator `when` presented in Sect. 2.4. If e has clock α, then e `when` c has clock α `on` c.

A flow can be over-sampled according to a Boolean condition using the `merge` operator, introduced in LUCID SYNCHRONE. Operator `merge` combines flows on complementary Boolean clocks to produce a faster flow.

```
node boolean_clocks(c: bool;i, j: int) returns (o:  int)
let
  o=merge(c, i when c,  j whennot c);
tel
```

`merge` (c, e_1, e_2) produces the value of e_1 when c is true and the value of e_2 when c is false. If c has clock α, then `merge` (c, e_1, e_2) also has clock α. The behaviour of the previous program is given in Fig. 4.7.

4.4.2 Combining Boolean and strictly periodic clock flow operators

The operator `when` can be applied to flows the clock of which is Boolean but also to flows the clock of which is strictly periodic. For instance, the following program is perfectly valid:

i	i_0	i_1	i_2	i_3	i_4	i_5	i_6	...
j	j_0	j_1	j_2	j_3	j_4	j_5	j_6	...
c	true	true	false	true	false	false	true	...
i **when** c	i_0	i_1		i_3			i_6	...
j **whennot** c			j_2		j_4	j_5		...
o	i_0	i_1	j_2	i_3	j_4	j_5	i_6	...

Figure 4.7: Boolean clock operators

```
node condperiodic(c: rate(5,0); i: rate (10,0)) returns (o)
let
  o=(i*^2) when c;
tel
```

The clock of output o is: $(5,0)$ on c. The behaviour of this program is illustrated in Fig. 4.8.

date	0	5	10	15	20	25	...
c	true	true	false	false	true	false	...
i	i_0		i_1		i_2		...
i*^2	i_0	i_0	i_1	i_1	i_2	i_2	...
o=(i*^2) **when** c	i_0	i_0			i_2		...

Figure 4.8: Applying Boolean clock transformations to strictly periodic clocks

However, the opposite is not allowed, flow operators based on strictly periodic clocks cannot be applied to flows the clock of which contains a Boolean clock transformation. For instance the flow `(x when c)*^2` is rejected by the compiler. Indeed, this would require to apply a periodic clock transformation to a clock that is not strictly periodic, the semantics of which is undefined.

By combining both classes of clocks, the programmer can first specify the base frequency of a process using strictly periodic clocks and then specify that on this frequency the process is activated only if a certain condition holds, using Boolean clocks.

4.5 Syntax

This section details the complete syntax of the language. It is close to LUSTRE, however we do not impose to declare types and clocks, which are computed automatically, similarly to LUCID SYNCHRONE (see Chap. 6). The language grammar is given below:

$$
\begin{array}{lcl}
cst & ::= & \texttt{true} \mid \texttt{false} \mid 0 \mid ... \\
var & ::= & x \mid var, var \\
e & ::= & cst \mid x \mid (e,e) \mid cst \; \texttt{fby} \; e \mid N(e) \mid e \; \texttt{when} \; e \mid \texttt{merge} \; (e,e,e) \\
 & & \mid e/\hat{}k \mid e*\hat{}k \mid e \sim> q \mid \texttt{tail} \; (e) \mid cst \; \texttt{::} \; e \\
eq & ::= & var = e \mid eq; eq \\
typ & ::= & \texttt{int} \mid \texttt{bool} \\
in & ::= & x \; [: [typ] \; [rate(n,p)] \; [\; \texttt{before} \; n']] \mid in; in \\
out & ::= & x \; [: [typ] \; [rate(n,p)] \; [\; \texttt{due} \; n']] \mid out; out \\
decl & ::= & \texttt{node} \; N(in) \; \texttt{returns} \; (out) \; [\; \texttt{var} \; var;] \; \texttt{let} \; eq \; \texttt{tel} \\
 & & \mid \; \texttt{imported node} \; N(in) \; \texttt{returns} \; (out) \; \texttt{wcet} \; n; \\
 & & \mid \; \texttt{sensor} \; x \; \texttt{wcet} \; n \mid \; \texttt{actuator} \; x \; \texttt{wcet} \; n
\end{array}
$$

A program consists of a list of declarations ($decl$). A declaration can either be a node, an actuator or a sensor declaration. Nodes can either be defined in the program (`node`) or implemented outside (`imported node`), for instance by a C function. Node durations must be provided for each imported node, more precisely the worst case execution times ($wcet$). The external code provided for imported nodes can itself be generated by a standard synchronous language compiler (like LUSTRE), in case developers want to program the whole system using synchronous languages. Sensor and actuator declarations specify the wcet of the actuators and sensors of the program (the main node inputs and outputs, see Sect. 9.2.5 for details).

The clock of an input/output parameter (in/out) can be declared strictly periodic ($x : rate(n, p)$, x then has clock (n, p)) or left unspecified. The type of an input/output parameter can be integer (int), Boolean ($bool$) or unspecified. A deadline constraint can be imposed on input/output parameters (x : `before` n or x : `due` n, the deadline is n, relatively to the beginning of the period of the flow).

The body of a node consists of an optional list of local variables (var) and a list of equations (eq). Each equation defines the value of one or several variables using an expression on flows ($var = e$). Expressions may be immediate constants (cst), variables (x), pairs ((e, e)), initialised delays (cst `fby` e), applications ($N(e)$), Boolean sub-sampling (e `when` e), combination of flows on complementary Boolean clocks (`merge` (e, e, e)) or expressions using strictly periodic clocks. $e/\hat{}\,k$ under-samples e using a periodic clock division and $e *\hat{}\,k$ over-samples e using a periodic clock multiplication ($k \in \mathbb{N}^*$). $e \sim> q$ offsets the values of e by a factor q ($q \in \mathbb{Q}^+$). $tail(e)$ drops the first value of e. cst `::` e concatenates cst to e. Values k, n, q must be statically evaluable.

4.6 Synchronous Temporised Semantics

The section gives the formal semantics of the language. The compilation process is defined so as to respect this semantics. Thus, the behaviour of the code generated for a program is completely predictable (it is the one defined by the present semantics), it is known *before executing the actual program*. This is an essential feature of formal languages.

4.6.1 Kahn's semantics

We provide a semantics based on Kahn's semantics on sequences [Kah74]. It is a direct adaptation of the synchronous semantics presented in [CP03] to the Tagged-Signal model. For any operator $\diamond$, $\diamond^{\#}(s_1, ..., s_n) = s'$ means that the operator $\diamond$ applied to sequences $s_1, \ ..., \ s_n$ produces the sequence s'. Term $(v, t).s$ denotes the flow whose head has value v and tag t and whose tail is sequence s. Operator semantics is defined inductively on the argument sequences. We omit rules on empty flows, which all return an empty flow.

We will see that the semantics of some operators (for instance operator `fby` or `when`) is defined only for synchronous flows. This implies that if the flows are not synchronous, the semantics of the program is ill-defined. Therefore, we will later define a verification, called the *clock calculus* (Sect. 6.3), which will check that the program only combines synchronous flows and consequently that the semantics of the program is well-defined. Verifying that two flows are synchronous amounts to verifying that their clocks correspond to the same sequence of tags. As the sequence of tags of a clock is increasing, we can express the constraint that the arguments of an operator must be synchronous by requiring that at each induction step, the tags of the heads of the arguments are the same.

4.6.2 Semantics of classic synchronous operators

We start by redefining the semantics of classic synchronous operators in Fig. 4.9.

$$\begin{aligned}
\texttt{fby}^{\#}(v,(v',t).s) &= (v,t).\ \texttt{fby}^{\#}(v',s)\\
\texttt{when}^{\#}((v,t).s,(T,t).cs) &= (v,t).\ \texttt{when}^{\#}(s,cs)\\
\texttt{when}^{\#}((v,t).s,(F,t).cs) &= \texttt{when}^{\#}(s,cs)\\
\texttt{whennot}^{\#}((v,t).s,(T,t).cs) &= \texttt{whennot}^{\#}(s,cs)\\
\texttt{whennot}^{\#}((v,t).s,(F,t).cs) &= (v,t).\ \texttt{whennot}^{\#}(s,cs)\\
\texttt{merge}^{\#}((T,t).s,(v,t).s_1,s_2)) &= (v,t).\texttt{merge}^{\#}(s,s_1,s_2))\\
\texttt{merge}^{\#}((F,t).s,s_1,(v,t).s_2)) &= (v,t).\texttt{merge}^{\#}(s,s_1,s_2)
\end{aligned}$$

Figure 4.9: Synchronous semantics of classical operators

- x `fby` y concatenates the head of x to y, delaying the values of y by one tag;
- x `when` c produces a sub-sequence of x, only producing values of x when c is true;
- `merge` (c, x, y) combines flows x and y which are on complementary Boolean clocks: when c is true, x is present and y is absent. When c is false, x is absent and y is present. The result is present every time c is present (i.e. on the clock of c). It replaces the operator `current` of LUSTRE.

4.6.3 Semantics of strictly periodic clock operators

The synchronous semantics of operators on strictly periodic clocks is given in Fig. 4.10. These semantic rules are temporised in addition to being synchronous. Operator $*\hat{}$ produces values on tags that do not appear in the flow on which they are applied. These new tags must respect some specific temporal properties relating them to the tags of the parameter flow. Function π is extended to flows, $\pi(f) = \pi(\alpha)$ where α is the clock of flow f.

$$*\hat{}^{\#}((v,t).s,k) = \prod_{i=0}^{k-1}(v,t'_i).*\hat{}^{\#}(s,k)$$
$$\text{(where } t'_0 = t \text{ and } t'_{i+1} - t'_i = \pi(s)/k)$$

$$/\hat{}^{\#}((v,t).s,k) = \begin{cases}(v,t)./\hat{}^{\#}(s,k) & \text{if } k*\pi(s)|t\\ /\hat{}^{\#}(s,k) & \text{otherwise}\end{cases}$$

$$\sim>^{\#}((v,t).s,q) = (v,t').\sim>^{\#}(s,q)$$
$$\text{(where } t' = t + q\pi(s))$$

$$\texttt{tail}^{\#}((v,t).s) = s$$

$$::^{\#}(cst,s) = (cst,\varphi(s)-\pi(s)).s$$

Figure 4.10: Synchronous temporised semantics of operators on strictly periodic clocks

- $x *\hat{}\, k$ produces a flow k times faster than x. Each value of x is duplicated k times in the result. The time interval between two successive duplicated values is k times shorter than the interval between two successive values in x;

- $x/\hat{}k$ produces a flow k times slower than x, dropping part of the values of x;
- $x \sim> q$ delays each value of x by $q\pi(x)$ temporal units;
- $tail(x)$ produces a flow equal to x except that it drops the first value of x;
- cst `::` e produces a flow resulting of the concatenation of value cst to e. The value cst is produced at date $\varphi(e) - \pi(e)$.

4.6.4 Denotational semantics

The flows of a program are defined with respect to some flow variables, so we need to define how a program behaves when flow values (ie sequences of pairs (v_i, t_i)) are assigned to flow variables. To this intent, we define a denotational semantics [Rey98] for the rest of the constructions of the language. This denotational semantics is very close to that of [CP03].

Let ρ denote an assignment mapping flow values to variable names. For instance, $\rho(v)$ denotes the flow mapped to variable v. We define the *denotation* of an expression e by $S_\rho(e)$. Intuitively, $S_\rho(e)$ denotes the sequence obtained for e under assignment ρ. The notation is overloaded for equations. $S_\rho(e)$ is defined inductively in Fig. 4.11.

$$\begin{aligned}
S_\rho(x) &= \rho(x) \\
S_\rho(cst \texttt{ fby } e) &= \texttt{fby}^\#(cst, S_\rho(e)) \\
S_\rho(e_1 \texttt{ when } e_2) &= \texttt{when}^\#(S_\rho(e_1), S_\rho(e_2)) \\
S_\rho(e_1 \texttt{ whennot } e_2) &= \texttt{whennot}^\#(S_\rho(e_1), S_\rho(e_2)) \\
S_\rho(\texttt{merge }(e_1, e_2, e_3)) &= \texttt{merge}^\#(S_\rho(e_1), S_\rho(e_2), S_\rho(e_3)) \\
S_\rho(e *\hat{}k) &= *\hat{}^\#(S_\rho(e), k) \\
S_\rho(e/\hat{}k) &= /\hat{}^\#(S_\rho(e), k) \\
S_\rho(e \sim> k) &= \sim>^\#(S_\rho(e), k) \\
S_\rho(\texttt{tail }(e)) &= \texttt{tail }^\#(S_\rho(e)) \\
S_\rho(cst \texttt{ :: } e) &= \texttt{ :: }^\#(cst, S_\rho(e)) \\
S_\rho(e_1, e_2) &= (S_\rho(e_1), S_\rho(e_2)) \\
S_\rho(x = e) &= \rho[x \leftarrow x^\infty] \text{ where } x^\infty = fix\ \lambda y.S_{\rho[x \leftarrow y]}(e) \\
S_\rho(x_1 = e_1; x_2 = e_2) &= S_{S_\rho(x_1 = e_1[e_2/x_2])}(x_2 = e_2) \\
S_\rho(\texttt{ node } N(in) \texttt{ returns } (out) \texttt{ let } eqs \texttt{ tel }) &= \lambda y.S_{\rho[in \leftarrow y]}(eqs) \\
S_\rho(e_1(e_2)) &= S_\rho(e_1)(S_\rho(e_2))
\end{aligned}$$

Figure 4.11: Denotational semantics

- The denotation of a variable is the flow mapped to the variable in the assignment;
- The denotation of an operator (`fby`, `when`, `whennot`, `merge`, $*\hat{}$, $/\hat{}$, $\sim>$, `tail`, `::`) is the sequence obtained when applying the operator to the denotation of its arguments;

- The denotation of a tuple is the tuple made up of the denotations of its different members;

- The denotation of an equation is defined as an assignment. The denotation of equation $x = e$ for assignment ρ replaces x by x^∞ in ρ. We cannot directly replace x by $S_\rho(e)$ because x may appear in e (for instance $x = 0$ `fby` $plus(x, 1)$) so we need to compute a fix point;

- The denotation of two equations is also an assignment. We first compute the assignment ρ' obtained for equation $x_1 = e_1[e_2/x_2]$, where $e_1[e_2/x_2]$ is the expression obtained by substituting expression e_2 for variable x_2 in e_1. Then we compute $S_{\rho'}(x_2 = e_2)$, that is to say the assignment obtained for equation $x_2 = e_2$ in assignment ρ');

- The denotation of a node definition is the assignment obtained for its equations. This assignment is parameterized (λy) by the values assigned to the inputs of the node ($in \leftarrow y$);

- The denotation of a node application is the denotation of the node definition applied to the denotation of the arguments of the application.

4.7 Summary

In this chapter, we have defined high-level real-time synchronous primitives based on strictly periodic clocks. Using these primitives, the programmer can specify:

- Periodicity constraints;
- Deadline constraints;
- Phase offsets for periodic flows;
- Rate transitions between flows of different rates.

We have also shown that the existing delay and Boolean clock sampling operators easily fit in our language.

In the next chapter, we illustrate the expressiveness of this language and show how it can be used to program multi-periodic systems.

Chapter 5

Multi-periodic Communication Patterns

A key feature of the language is that it gives the programmer the freedom to choose the pattern used for communicating between operations of different rates, while most languages only allow a small set of predefined patterns. Rate transition operators apply very simple flow transformations but can be combined to produce various communication patterns. The formal definition of the language and the static analyses performed by the compiler ensure that only deterministic patterns are used. For instance the informal communication pattern "each operation consumes the last value produced before its execution" can lead to undeterministic program behaviours, as values produced by the program can vary depending on the actual scheduling of the program. It is thus not supported by our language.

This chapter illustrates the expressiveness of the language through a series of communication patterns and shows that common patterns can easily be defined as generic library operators that can be reused in different programs. We illustrate the different communication patterns on the simple example of Fig. 5.1, where a fast operation `F` (of period 10ms), exchanges data with a slow operation `S` (of period 30ms). In addition, operation `F` exchanges data with the environment of the program.

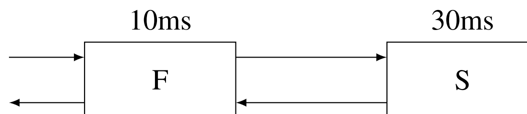


Figure 5.1: A simple example to illustrate the different communication patterns

5.1 Sampling

The simplest communication pattern that can be implemented with our language is based on *data sampling*. For instance, for the example of Fig. 5.1:

```
node sampling(i: rate (10, 0)) returns (o)
  var vf, vs;
let
  (o, vf)=F(i, (0 fby vs)*^3);
  vs=S(vf/^3);
tel
```

The slow operation under-samples the data produced by the fast operation, consuming only the first out of 3 successive values of the flow. On the opposite, the fast operation over-samples the data produced

by the slow operation, each group of 3 successive repetitions of the fast operation consumes the same value. The slow to fast communication is delayed for two reasons. First, we need a delay either from `S` to `F` or from `F` to `S`, otherwise there is a causality loop between `F` and `S`, the compiler cannot find an order for the execution of the program that respects all the data-dependencies. Second, the delay avoids the reduction of the deadline of operation `S`. Indeed, let us consider an alternative implementation of the previous example:

```
node sampling2(i: rate (10, 0)) returns (o)
  var vf, vs;
let
  (o, vf)=F(i, vs*^3);
  vs=S((0 fby vf)/^3);
tel
```

This version is also correct but an important difference with the previous version is that `S` must end before a relative deadline of 10ms, as it must end early enough for `F` to end before its own deadline (10ms). In the previous version, the delay avoids this restrictive deadline. The behaviour of these two examples is illustrated in Fig. 5.2.

date	0	10	20	30	40	50	60	70	80	...
`vf`	vf_0	vf_1	vf_2	vf_3	vf_4	vf_5	vf_6	vf_7	vf_8	...
`vf/^3`	vf_0			vf_3			vf_6			...
`vs`	vs_0			vs_1			vs_2			...
`0 fby vs`	0			vs_0			vs_1			...
`(0 fby vs)*^3`	0	0	0	vs_0	vs_0	vs_0	vs_1	vs_1	vs_1	...
`0 fby vf`	0	vf_0	vf_1	vf_2	vf_3	vf_4	vf_5	vf_6	vf_7	...
`(0 fby vf)/^3`	0			vf_2			vf_5			...
`vs*^3`	vs_0	vs_0	vs_0	vs_1	vs_1	vs_1	vs_2	vs_2	vs_2	...

Figure 5.2: Data sampling: choosing the place of the delay

Using the operator `tail`, it is possible to choose which of the successive values of the fast operation should be consumed (ie not necessarily the first). For instance, we can consume the second value instead of the first:

```
node sampling_tail(i: rate (10, 0)) returns (o)
  var vf, vs;
let
  (o, vf)=F(i, (0::((0 fby vs)*^3)));
  vs=S((tail(vf))/^3);
tel
```

The behaviour of this program is illustrated in Fig. 5.3.

5.2 Queuing

Combining rate transition operators and LUSTRE arrays, we can implement different communication patterns based on *data queuing*. Though arrays are not directly present in the syntax we gave in Sect. 4.5, they can be implemented using a pre-processing phase, which expands arrays into independent variables [RH91] (an array of size n is expanded into n independent variables). This pre-processing is performed

date	0	10	20	30	40	50	60	70	80	...
vf	vf_0	vf_1	vf_2	vf_3	vf_4	vf_5	vf_6	vf_7	vf_8	...
tail(vf)/^3		vf_1			vf_4			vf_7		...
vs		vs_0			vs_1			vs_2		...
(0 **fby** vs)*^3		0	0	0	vs_0	vs_0	vs_0	vs_1	vs_1	...
0::((0 **fby** vs)*^3)	0	0	0	0	vs_0	vs_0	vs_0	vs_1	vs_1	...

Figure 5.3: Data sampling, starting the sample on the second value

at the very beginning of the compilation and the program is then compiled normally. A more efficient implementation would take advantage of the work of [MM04] on array iterators but we do not consider it here, for simplification.

First, we define a node that stores n successive values of a flow:

```
node store_n(const n: int;i,init) returns (A: int^n)
let
  A[n-1]=i;
  A[0..(n-2)]=(init^(n-1)) fby (A[1..(n-1)]));
tel
```

`int^n` denotes an integer array of size n. `A[n]` denotes the nth value of array A. `init^(n-1)` denotes an array of size $n-1$ where each value of the array is set to $init$. `A[1..(n-1)]` denotes a slice of array A, from cell 1 to cell $n-1$. The behaviour of this node is the following. The n successive values of flow i are stored in the array A of size n. At each iteration, the node queues the current value of i at the end of A, moving the previous values of i by one cell towards the beginning of the array. At each instant, the array A contains the current value of i along with its $n-1$ previous values. The previous values are initialised to the value `init`. This is illustrated in Fig. 5.4.

i	i_0	i_1	i_2	i_3	i_4	...
A	$[0,0,i_0]$	$[0,i_0,i_1]$	$[i_0,i_1,i_2]$	$[i_1,i_2,i_3]$	$[i_2,i_3,i_4]$	...

Figure 5.4: Storing successive values of a flow ($n=3$, $init=0$)

Then we define two more complex rate transition nodes:

```
node split(const n: int;i) returns(o)
var ifast;
let
  ifast = i*^ n;
  o = ifast[count_n(n)];
tel

node join(const n: int; i,init) returns(o)
var ofast;
let
  ofast=store_n(n,i,init);
  o=ofast /^n;
tel
```

The node `count_n` used in this example is a counter modulo n (its sequence of values is $0,1,\cdots,n-1,0,1,\cdots,n-1,\cdots$). The node `split` handles communications from slow to fast operations. It takes

an array of size n as input and splits it into n successive values. The outputs are produced n times faster than the input. The node `join` handles communications from fast to slow operations. It joins n successive values of its input into an array. The output is n times slower than the input. This behaviour is illustrated below:

i	i_0	i_1	i_2	i_3	i_4	i_5	i_6	...
i'=join(3,i,0)	$[0, 0, i_0]$			$[i_1, i_2, i_3]$			$[i_4, i_5, i_6]$	...
split(3,i')	0	0	i_0	i_1	i_2	i_3	i_4	...

We can then define a different version of the example of Fig. 5.1 that uses a queuing communication pattern:

```
node queuing(i: rate (10, 0)) returns (o)
  var vf, vs;
let
  (o, vf)=F(i, split(3, 0 fby vs));
  vs=S(join(3, vf, 0));
tel
```

This version requires flows `vf` and `vs` to be arrays of integers instead of simple integers and thus node `S` must return an array, not just an integer.

5.3 Mean value

We give a third example of communication pattern, which is an intermediate between sampling and queuing. Instead of queuing the successive values produced by the fast operations, we take their mean value:

```
node mean_n(const n: int;i, init) returns (m: int)
  var n_values, n_sum: int^n;
let
  n_values=store_n(n, i, init);
  n_sum[0]=n_values[0];
  n_sum[1..n-1]=n_values[1..n-1]+n_sum[0..n-2];
  m=(n_sum[n-1]/n)/^n;
tel
```

Notice that there is no causality loop in the third equation as each `n_sum[i]` depends on `n_sum[i-1]`, which are independent variables. The behaviour of this node is illustrated in Fig. 5.5.

i	1	2	3	4	5	6	7	...
n_values	$[0, 0, 1]$	$[0, 1, 2]$	$[1, 2, 3]$	$[2, 3, 4]$	$[3, 4, 5]$	$[4, 5, 6]$	$[5, 6, 7]$	...
n_sum	$[0, 0, 1]$	$[0, 1, 3]$	$[1, 3, 6]$	$[2, 5, 9]$	$[3, 7, 12]$	$[4, 9, 15]$	$[5, 11, 18]$	...
m	0			3			6	...

Figure 5.5: Computing the mean value of 3 successive "fast" values, (with $init = 0$)

We can then define a different version of the example of Fig. 5.1 that takes the mean value of 3 successive fast values (flow `vf`) and over-samples fast values (flow `vs`):

```
node sampling(i: rate (10, 0)) returns (o)
  var vf, vs;
let
  (o, vf)=F(i, (0 fby vs)*^3);
  vs=S(mean_n(3, vf, 0);
tel
```

5.4 Non-harmonic periods

The language is not restricted to systems where all the processes have harmonic periods. For instance, the following node is perfectly valid:

```
node non_harmonic(i: rate (60, 0)) returns (o1, o2, o3)
let
  o1=A(i*^3);
  o2=B(i*^2);
  o3=C(i*^3/^4);
tel
```

A has clock $(20, 0)$, B has clock $(30, 0)$ and C has clock $(80, 0)$. The values consumed by the different nodes are given in Fig. 5.6.

date	0	10	20	30	40	50	60	70	80	90	100	110	120	...
`i`	i_0						i_1						i_2	...
`i*^2`	i_0			i_0			i_1			i_1			i_2	...
`i*^3`	i_0		i_0		i_0		i_1		i_1		i_1		i_2	...
`i*^3/^4`	i_0								i_1					...

Figure 5.6: Flows of non-harmonic periods

The only constraints on task periods are those imposed by the clock calculus, ie that only flows of the same clock can be combined. For instance, we would not be allowed to directly compute the sum of $o1$ and $o2$, we would first have to use rate transition operators to bring the flows to the same clock.

5.5 A complete example

Fig. 5.7 gives a program for the FAS of Fig. 1 using communication patterns based on data sampling. Each operation of the system is first declared as an imported node. The program then specifies the wcet of each sensor and each actuator of the system (for each input/output of the node `FAS`).

The node `FAS` is the main node of the program. It first specifies the inputs (`gyro, gps, str, tc`) and outputs (`pde, sgs, gnc, pws, tm`) of the system with their rates (for instance `gyro` has period 100) and deadline constraints (for instance `gnc` has deadline 300). The body of the node then specifies the real-time architecture of the system. For instance, the first equation says that the node `Gyro_Acq` computes the variable `gyro_acq` from the input `gyro` and from the variable `tm`. As `tm` is produced with a period of 10s while `Gyro_Acq` has a period of 100ms, we over-sample `tm` by a factor 100. We use a delay before performing the over-sampling, to avoid reducing the deadline of node `TM_TC`. Some flows are over-sampled without using a delay, for instance when the outputs of the acquisition nodes are consumed by faster nodes (`gps_acq*^10` in the equation of the `FDIR` for instance).

This implies that the acquisition nodes will have a deadline much shorter than their period, which means that they execute less frequently but they must respond fast. Communications from fast to slow nodes are simply handled using the under-sampling operator `/^`. For instance, the node `TM_TC` has a period of 10s and consumes the flow `fdir_tm`, which is produced with a period of 100ms, so we under-sample `fdir_tm` by a factor 100. Finally, in the equation of node `PWS` we apply a phase offset of one half to its input `gnc_pws`, which causes the node `PWS` to execute with a phase of half its period (it has period 1000 and phase 500).

```
imported node Gyro_Acq(gyro, tc: int) returns (o: int) wcet 3;
imported node GPS_Acq(gps, tc: int) returns (o: int) wcet 3;
imported node Str_Acq(str, tc: int) returns (o: int) wcet 3;
imported node FDIR(gyr, gps, str, gnc: int) returns (to_pde, to_gnc, to_tm: int) wcet 15;
imported node GNC_US(fdir, gyr, gps, str: int) returns (o: int) wcet 210;
imported node GNC_DS(us: int) returns (pde, sgs, pws: int) wcet 300;
imported node TM_TC(from_gr, fdir: int) returns (cmd: int) wcet 1000;
imported node PDE(fdir, gnc: int) returns (pde_order: int) wcet 3;
imported node SGS(gnc: int) returns (sgs_order: int) wcet 3;
imported node PWS(gnc: int) returns (pws_order: int) wcet 3;
sensor gyro wcet 1; sensor gps wcet 1; sensor str wcet 1; sensor tc wcet 1;
actuator pde wcet 1; actuator sgs wcet 1; actuator gnc wcet 1;
actuator pws wcet 1; actuator tm wcet 1;

node FAS(gyro: rate (100, 0); gps: rate (1000, 0); str:  rate (10000, 0); tc: rate (10000, 0))
returns (pde, sgs;gnc: due 300; pws, tm)
var gyro_acq, gps_acq, str_acq, fdir_pde, fdir_gnc, fdir_tm, gnc_pde, gnc_sgs, gnc_pws;
let
  gyro_acq = Gyro_Acq(gyro, (0 fby tm)*^100);
  gps_acq = GPS_Acq(gps, (0 fby tm)*^10);
  str_acq = Str_Acq(str, 0 fby tm);
  (fdir_pde, fdir_gnc, fdir_tm) = FDIR(gyro_acq, gps_acq*^10, str_acq*^100, (0 fby gnc)*^10);
  gnc=GNC_US(fdir_gnc/^10, gyro_acq/^10, gps_acq, str_acq*^10);
  (gnc_pde, gnc_sgs, gnc_pws)=GNC_DS(gnc);
  pde = PDE(fdir_pde, (0 fby gnc_pde)*^10);
  sgs = SGS(gnc_sgs);
  pws=PWS(gnc_pws~>1/2);
  tm = TM_TC(tc, fdir_tm/^100);
tel
```

Figure 5.7: Programming the FAS with data-sampling

Fig. 5.8 gives a different version of the program, using communication patterns based on data queuing. This version assumes different signatures for the imported nodes of the system. Basically, nodes executing at 10Hz are assumed to consume and produce simple integer values, nodes executing at 1Hz are assumed to consume and produce arrays of 10 integers and nodes executing at 0.1Hz are assumed to consume and produce arrays of 100 integers. Then, over-sampling operators ($*\hat{}$) are replaced by calls to the node `split`, while under-sampling operators ($/\hat{}$) are replaced by calls to the node `join`. Of course, we could have mixed different communications patterns in the same program (for instance, mixing data-queuing and data-sampling).

5.6 Summary

In this chapter, we have illustrated the expressiveness of the language and emphasized the ability to define various communication patterns instead of being restricted to a limited set of predefined patterns. Complex patterns (such as data-sampling for instance) could easily be defined as library nodes and reused in different program. This reusability relies on types polymorphism and more importantly on

```
imported node Gyro_Acq(gyro, tc: int) returns (o: int) wcet 3;
imported node GPS_Acq(gps, tc: int^10) returns (o: int^10) wcet 3;
imported node Str_Acq(str, tc: int^100) returns (o: int^100) wcet 3;
imported node FDIR(gyr, gps, str, gnc: int) returns (to_pde, to_gnc, to_tm: int) wcet 15;
imported node GNC_US(fdir, gyr, gps, str: int^10) returns (o: int^10) wcet 210;
imported node GNC_DS(us: int^10) returns (pde, sgs, pws: int^10) wcet 300;
imported node TM_TC(from_gr, fdir: int^100) returns (cmd: int^100) wcet 1000;
imported node PDE(fdir, gnc: int) returns (pde_order: int) wcet 3;
imported node SGS(gnc: int^10) returns (sgs_order: int^10) wcet 3;
imported node PWS(gnc: int^100) returns (pws_order: int^100) wcet 3;
sensor gyro wcet 1; sensor gps wcet 1; sensor str wcet 1; sensor tc wcet 1;
actuator pde wcet 1; actuator sgs wcet 1; actuator gnc wcet 1;
actuator pws wcet 1; actuator tm wcet 1;

node FAS(gyro: rate (100, 0); gps: rate (1000, 0); str: rate (10000, 0); tc: rate (10000, 0))
returns (pde, sgs;gnc: due 300; pws, tm)
var gyro_acq, gps_acq, str_acq, fdir_pde, fdir_gnc, fdir_tm, gnc_pde, gnc_sgs, gnc_pws;
let
  gyro_acq = Gyro_Acq(gyro, split(100,split(100,(0 fby tm));
  gps_acq = GPS_Acq(gps, split(10,(0 fby tm));
  str_acq = Str_Acq(str, 0 fby tm);
  (fdir_pde, fdir_gnc, fdir_tm) = FDIR(gyro_acq, split(10,gps_acq), split(100,str_acq),
                                  split(10,(0 fby gnc));
  gnc=GNC_US(join(10,fdir_gnc,0), join(10,gyro_acq,0), gps_acq, split(10,str_acq));
  (gnc_pde, gnc_sgs, gnc_pws)=GNC_DS(gnc);
  pde = PDE(fdir_pde, split(10,(0 fby gnc_pde));
  sgs = SGS(gnc_sgs);
  pws=PWS(gnc_pws~>1/2);
  tm = TM_TC(tc, join(100,fdir_tm,0);
tel
```

Figure 5.8: Programming the FAS with data-queuing

clocks polymorphism. The next chapter details the static analyses of the language and in particular the typing and the clock calculus, which enable this polymorphism.

Chapter 6

Static Analyses

The language is targeted for critical embedded systems, therefore strong emphasis is put on the verification of the correctness of the program that must be compiled. Before compiling the input program into lower-level code, a series of static analyses is performed, which ensure that the semantics of the program to compile is well-defined. Each static analysis ensures different correctness properties for the program. The causality analysis verifies that there exists an order for the evaluation of the expressions of the program that respects all the data-dependencies of the program (ie that data-dependencies are not cyclic). The typing verifies that only flows of the same type are combined. The clock calculus verifies that only synchronous flows are combined. Finally, the initialisation analysis verifies that the program does not access to values that are not initialised. If all the static analyses succeed, then the program has a well-defined semantics, specified by the semantic rules of Sect. 4.6.

6.1 Causality Analysis

The order in which the variables of a program must be computed depends on the data-dependencies between the variables. For instance, the variable defined by the equation $x = e$ cannot be evaluated before the variables appearing in e have been evaluated. Equations are ordered according to the *syntactic* data-dependencies of the program [HRR91]: an expression is considered to depend on all the variables appearing in it, except if the variable appears as an argument of a **`fby`** operator. Thus, an equation of the form $x = e$ can only be evaluated after the equations defining the variables e syntactically depends on have been evaluated.

The causality analysis verifies that the program does not contain cyclic definitions: a variable cannot instantaneously depend on itself (i.e. not without a **`fby`** in the dependencies). For instance, if `imp` is an imported node, the equation `x=imp(x,1);` is incorrect. It is similar to a deadlock since we need to evaluate `imp(x,1)` to evaluate `x` and we need to evaluate `x` to evaluate `imp(x,1)`.

The causality analysis is based on *syntactic* data-dependencies instead of *semantic* data-dependencies. This restriction may reject correct programs such as the following, which is actually causal:

```
x=merge(c,y when c,3);
y=merge(c,2,x whennot c);
```

Unfortunately, in the general case, a semantic causality analysis is undecidable.

6.2 Typing

The language is strongly typed in the sense of [Car89], that is the execution of a program cannot produce a run-time type error. Each flow has a single, well-defined type and only flows of the same type can be combined. The typing of the language is fairly standard, it relies on well-known type-inference techniques that can be found in [Pie02]. We present the typing in details as the same techniques will be used for the clock calculus in a more complex setting.

6.2.1 Types

The typing produces type judgements of two different forms. First, $E \vdash e : t$, means that "the expression e has type t in environment E". Second, $E \vdash eq$ means that "the equation eq is well-typed", but no type is associated to it. This second form is also used for sets of equations, untyped declarations (sensors and actuators) and sequences of declarations. The grammar of types is given below:

$$\begin{array}{lll} \sigma & ::= & \forall \alpha_1, ..., \alpha_m.t \\ t & ::= & t \rightarrow t \mid t \times t \mid base \mid \alpha \\ base & ::= & int \mid bool \mid rat \\ E & ::= & [x_1 : \sigma_1, ..., x_m : \sigma_m] \end{array}$$

Type schemes (σ) are type expressions (t) quantified over a set of type variables (α). Type expressions can be functional types ($t \rightarrow t$), type products ($t \times t$), base types ($base$), or type variables (α). Base types are either integers (int), Booleans ($bool$) or rationals (rat). Typing environments (E) associate types to flow variables and node definitions. They can be considered as functions, which associate a type to variable names or node names. Thus, $E(x)$ denotes the type associated to x in environment E and $Dom(E)$, the domain of E, is the set of names that are defined in E (that have an associated type).

Types may be generalized (at a node definition) and instantiated (at a node call) as follows:

$$inst(\forall \alpha.t) = t[t'/\alpha]$$
$$gen_E(t) = \forall \alpha_1, ..., \alpha_m.t, \text{ where } \alpha_1, ..., \alpha_m = FTV(t) \backslash FTV(E)$$

This states that a type scheme is instantiated by replacing type variables by type expressions and that any type variable can be generalized if it does not appear free in the environment (FTV stands for “free type variables”).

For instance, the type scheme $\forall \alpha.\alpha \times \alpha \rightarrow \alpha$ represents the type of a polymorphic node that takes two arguments of the same type and returns a value of this type. If this node is called with a first argument of type int, then the type scheme is instantiated to $\alpha \times \alpha \rightarrow \alpha[int/\alpha] = int \times int \rightarrow int$, so the two arguments of the node call must be integers and it returns an integer. The same node can later be called with arguments of a different type, which will produce a different type instance (*bool* $\times$ *bool* $\rightarrow$ *bool* for example). This type scheme is the type of operator `fby` for instance.

6.2.2 Types inference

For each expression of the program, the typing produces constraints that have to be met for the expression to be well-typed. These constraints are produced by *inference rules*. A rule $\frac{A}{B}$ states that the type judgement B, called the *conclusion*, holds if the condition A, called the *premise* holds. The inference rules of the language are given in Fig. 6.1. Programs are typed in an initial environment E_{init} that contains the constants of the language with their types.

$$\text{(CONST)}\ \frac{cst \in dom(E)}{E \vdash cst : E(cst)} \quad \text{(VAR)}\ \frac{x \in dom(E)}{E \vdash x : E(x)} \quad (\times)\ \frac{E \vdash e_1 : t_1 \quad E \vdash e_2 : t_2}{E \vdash (e_1, e_2) : t_1 \times t_2}$$

$$\text{(FBY)}\ \frac{E \vdash cst : t \quad E \vdash e : t}{E \vdash cst\ \texttt{fby}\ e : t} \quad \text{(WHEN)}\ \frac{E \vdash e_1 : t \quad E \vdash e_2 : bool}{E \vdash e_1\ \texttt{when}\ e_2 : t}$$

$$\text{(MERGE)}\ \frac{E \vdash e_1 : bool \quad E \vdash e_2 : t \quad E \vdash e_3 : t}{E \vdash \texttt{merge}\ (e_1, e_2, e_3) : t} \quad (/\hat{}\,)\ \frac{E \vdash e : t \quad E \vdash k : int}{E \vdash e/\hat{}\,k : t}$$

$$(*\hat{}\,)\ \frac{E \vdash e : t \quad E \vdash k : int}{E \vdash e *\hat{}\,k : t} \quad (\sim>)\ \frac{E \vdash e : t \quad E \vdash q : rat}{E \vdash e \sim> q : t} \quad (\texttt{tail})\ \frac{E \vdash e : t}{E \vdash \texttt{tail}\ (e) : t}$$

$$(\,::\,)\ \frac{E \vdash cst : t \quad E \vdash e : t}{E \vdash cst\ ::\ e : t} \quad \text{(APP)}\ \frac{E \vdash N : \sigma \quad t_1' \rightarrow t_2' = inst(\sigma) \quad E \vdash e : t_1'}{E \vdash N(e) : t_2'}$$

$$\text{(EQ)}\ \frac{E \vdash x : t \quad E \vdash e : t}{E \vdash x = e} \quad \text{(EQS)}\ \frac{E \vdash eq_1 \quad E \vdash eq_2}{E \vdash eq_1; eq_2}$$

$$\text{(NODE)}\ \frac{in : t_1, out : t_2, var : t_3, E \vdash eqs}{E \vdash\ \texttt{node}\ N(in)\ \texttt{returns}\ (out)\ [\,\texttt{var}\ var;]\ \texttt{let}\ eqs\ \texttt{tel}\ : gen_E(t_1 \rightarrow t_2)}$$

$$\text{(IMP-NODE)}\ \frac{in : t_1 \quad out : t_2 \quad gen_\emptyset(t_1) = t_1 \quad gen_\emptyset(t_2) = t_2}{E \vdash\ \texttt{imported node}\ N(in)\ \texttt{returns}\ (out)\ \texttt{wcet}\ n : t_1 \rightarrow t_2}$$

$$\text{(SENSOR)}\ E \vdash\ \texttt{sensor}\ x\ \texttt{wcet}\ n \quad \text{(ACTUATOR)}\ E \vdash\ \texttt{actuator}\ x\ \texttt{wcet}\ n$$

$$\text{(DECLS-NODE)}\ \frac{E \vdash [\,\texttt{imported}\,]\ \texttt{node}\ N[...] : \sigma \quad E, N : \sigma \vdash decl_2}{E \vdash [\,\texttt{imported}\,]\ \texttt{node}\ N[...]\ \ decl_2}$$

$$\text{(DECLS-OTHER)}\ \frac{E \vdash decl_1 \quad E \vdash decl_2}{E \vdash decl_1\ \ decl_2}$$

Figure 6.1: Type inference rules

- A constant is well-typed if it is declared in the (initial) typing environment (rule *(CONST)*);
- The same is true for variables (rule *(VAR)*). Variables are added to the typing environment by rule *(NODE)* (see below);
- The type of a pair is the product of the types of its components (rule *(×)*);
- Operator `fby` takes two arguments of the same type and the result is a value of this type;
- The first argument of operator `when` can have any type t, while the second must have type *bool*. The result has type t;
- The first argument of operator `merge` must have type *bool*. Its second and third arguments must have the same type t and the result also has type t;
- The first argument of operator $/\hat{}$ can have any type t, while the second must have type int. The result has type t. The same goes for operator $*\hat{}$;
- Operator $\sim>$ has the same type as $/\hat{}$ or $*\hat{}$, except that its second argument must have type rat;
- The result of operator `tail` has the same type as the argument;
- Operator `::` takes two arguments of the same type and the result is a value of this type;

- The rule (*(APP)*) states that the arguments of a node call must have the type expected for the inputs of the node. The result of the node call has the type of the outputs of the node. The type of the node is instantiated during the node call, in case it is polymorphic (quantified);
- An equation is well-typed if its left hand side and its right hand side have the same type (rule *(EQ)*);
- A set of equations is well-typed if each equation of the set is well-typed;
- The rule (*NODE*) states that to type a node declaration, we first add the input, output and local variables of the node in the typing environment. The type of each variable in the environment is the type declared in the node if any, or a fresh type variable otherwise. Then, we type the set of equations of the node in this new typing environment. Finally, the type of the node is a functional type, from the type of its inputs to the type of its outputs. This type is generalized to support types polymorphism;
- The rule (*IMP-NODE*) states that an imported node cannot have a polymorphic type. This is imposed by the premises $gen_{\emptyset}(t_1) = t_1$ and $gen_{\emptyset}(t_2) = t_2$, which imply that t_1 and t_2 contain no type variable. The type of the node is a functional type, from the type of its inputs to the type of its outputs;
- Sensor (*SENSOR*) and actuator (*ACTUATOR*) declarations are always well-typed (no premisse);
- The rule (*DECLS-NODE*) states that a sequence of declarations consisting of the declaration of a node N (imported or not) followed by another declaration is well-typed if the node declaration is well-typed, has type σ, and if the second declaration is well-typed in an environment where the node N has type σ;
- For other kinds of declarations (*DECLS-OTHER*), a sequence of declarations is well-typed if the different declarations of the sequence are well-typed.

6.2.3 Types unification

The inference rules produce constraints for each expression of the program. The set of constraints produced for a program is then solved following the technique of [Rém92], where typing judgements are treated as unificands, i.e. typing problems are reduced to unification problems. The resolution algorithm consists in trying to find substitutions σ such that $\sigma(E) \vdash e : \sigma(t)$, ie substitutions that unify $E \vdash e : t$. Unification rules are given in Fig. 6.2. A rule $\frac{A}{B}$ means that to solve B we must solve A. $t_1 \sim t_2$ means that t_1 and t_2 are *unifiable*, that is to say, there exists a substitution σ such that $\sigma(t_1) = \sigma(t_2)$.

$$(\rightarrow)\ \frac{t_1 \sim t_1' \qquad t_2 \sim t_2'}{t_1 \rightarrow t_2 \sim t_1' \rightarrow t_2'} \qquad (\times)\ \frac{t_1 \sim t_1' \qquad t_2 \sim t_2'}{t_1 \times t_2 \sim t_1' \times t_2'} \qquad \text{(VAR)}\ \frac{\alpha \notin FTV(t) \qquad t \mapsto \alpha}{\alpha \sim t}$$

Figure 6.2: Types unification rules

- Rules *(→)* and *(×)* state that unification is performed recursively;
- Rule *(VAR)* states that we can unify a type variable with a type t if α does not appear (free) in t. The unification then consists in substituting t for α ($t \mapsto \alpha$);

- Of course, $int = int$ is unifiable and the same goes for $bool$ and rat.

The typing fails if we cannot find a substitution that unifies all the constraints produced by the inference rules. Otherwise, the result of the substitution produces the validated types of the program.

6.2.4 Examples

We consider the simple example below:

```
imported node imp(i, j: int) returns (o: bool) wcet 5;

node works(i, j) returns (o: bool)
let
  o=imp(i, j);
tel
```

The rule ***(IMP-NODE)*** can be applied to infer the type of node `imp`:

$$\text{(IMP-NODE)}\ \frac{i,j : int \times int \qquad o : bool \qquad gen_{\emptyset}(int \times int) = int \times int \qquad gen_{\emptyset}(bool) = bool}{E_{init} \vdash \texttt{imported node}\ imp(i,j)\ \texttt{returns}\ (o)\ \texttt{wcet}\ n : \quad int \times int \rightarrow bool}$$

This produces a new typing environment $E_0 = imp : int \times int \rightarrow bool, E_{init}$. Then, for node `works`, the rule *(NODE)* says that we must type the set of equations of the node in environment $E_1 = i : t_1, j : t_2, o : bool, E_0$, where t_1 and t_2 are fresh type variables:

$$\text{(EQ)}\ \frac{E_1 \vdash o : bool \qquad \text{(APP)}\ \dfrac{E_1 \vdash imp : int \times int \rightarrow bool \qquad E_1 \vdash i,j : int \times int}{E_1 \vdash imp(i,j) : bool}}{E_1 \vdash o = imp(i,j)}$$

When we apply rule *(APP)*, we unify type $t_1 \times t_2$ with type $int \times int$ (the type of (i,j)), which produces the substitutions $\{int/t_1, int/t_2\}$. The typing of the body of node `works` succeeds and the type of the node after applying substitutions is $int \times int \rightarrow bool$.

We consider the slightly different node definition:

```
node fails(i, j) returns (o)
var v;
let
  v=imp(i, j);
  o=imp(v, i);
tel
```

The equations must be typed in environment $E_2 = i : t_1, j : t_2, o : t_3, v : t_4, E_0$. For the first equation:

$$\text{(EQ)}\ \frac{E_2 \vdash v : bool \qquad \text{(APP)}\ \dfrac{E_2 \vdash imp : int \times int \rightarrow bool \qquad E_2 \vdash i,j : int \times int}{E_2 \vdash imp(i,j) : bool}}{E_2 \vdash v = imp(i,j)}$$

This produces substitutions $\{int/t_1, int/t_2, bool/t_4\}$. So, we type the second equation in environment $E_3 = i : int, j : int, o : t_3, v : bool, E_0$:

$$\text{(EQ)}\ \frac{E_3 \vdash o : bool \qquad \text{(APP)}\ \dfrac{E_3 \vdash imp : int \times int \rightarrow bool \qquad E_3 \vdash v,j :?}{E_3 \vdash imp(v,j) : bool}}{E_3 \vdash o = imp(v,j)}$$

The top-right premise cannot be proved. It requires v, j to have type $int \times int$, while E_3 states that v, j has type $bool \times int$. The typing fails because types $int \times int$ and $bool \times int$ are not unifiable.

As a last example, the type inferred for the main node of the FAS program of Fig. 5.7 is:

$$(int \times int \times int \times int) \rightarrow (int \times int \times int \times int \times int)$$

6.3 Clock Calculus

The clock calculus verifies that a program only combines flows that have the same clock. When two flows have the same clock, they are synchronous (always present at the same instants). Combining non-synchronous flows leads to non-deterministic programs that access to undefined values. We say that an expression is *well-synchronized* if it only combines flows on the same clocks, *ill-synchronized* otherwise. The clock calculus verifies that a program only uses well-synchronized expressions.

For each expression of the program, the clock calculus produces constraints that have to be met for the expression to be well-synchronized. As shown in [CP03], the clock calculus can use a type system to generate these constraints. The overall structure of the clock calculus is therefore very similar to that of the typing presented in Sect. 6.2. Clocks are represented by types, called *clock types*. Inference rules represent the constraints the clocks of an expression need to respect for the expression to be well-synchronized. Finally, clocks unification tries to solve the set of all the constraints generated for the program. If a solution exists, the program is well-synchronized, otherwise it is ill-synchronized.

6.3.1 Clock Types

The clock calculus produces type judgements of two different forms. First, $H \vdash e : cl$, means that "the expression e has clock type cl in environment H". Second, $H \vdash eq$ means that "the equation eq is well-synchronized", but no clock type is associated to it. This second form is also used for sets of equations, untyped declarations (sensors and actuators) and sequences of declarations. The grammar of clock types is given below:

$$\begin{array}{lcl}
\sigma & ::= & \forall \alpha_1 <: C_1, ..., \alpha_m <: C_m.cl \\
cl & ::= & cl \rightarrow cl \mid cl \times cl \mid ck \mid c : ck \\
ck & ::= & ck \texttt{ on } c \mid ck \texttt{ on not } c \mid pck \\
pck & ::= & (n, p) \mid \alpha \mid pck *. \; k \mid pck /. k \mid pck \rightarrow. \; q \\
c & ::= & nm \mid X \\
H & ::= & [x_1 : \sigma_1, ..., x_m : \sigma_m]
\end{array}$$

Clock types are clock schemes (σ) quantified over a set of clock variables, which belong to different subtypes of clocks ($C_1, ..., C_m$). A clock expression can be a functional clock ($cl \rightarrow cl$), a clock product ($cl \times cl$), a clock sequence (ck) or a dependency ($c : ck$).

A clock sequence can be the result of a Boolean clock under-sampling (ck `on` c, ck `on not` c) or a strictly periodic clock (pck). A strictly periodic clock can be a constant strictly periodic clock ((n, p)), a clock variable (α), a strictly periodic clock over-sampled periodically ($pck *. \; k$), a strictly periodic clock under-sampled periodically ($pck/. k$), or the result of a phase offset ($pck \rightarrow. \; k$). In other words, a clock sequence ck is a linear expression resulting of a succession of clock transformations (either strictly periodic or Boolean) on another clock sequence ck'. In the following, ck is called an *abstract clock* if ck' is a clock variable. It is called a *concrete clock* if ck' is a constant strictly periodic clock. The values n and p are constant integer values. Integer values k and rational values q must be computable statically.

A carrier (c) can be a name (nm) or a carrier variable (X). For instance, in the node declaration **node** `N(c: bool; i: (10,0)` **when** `c)` **returns** `(o)`, `c` is a carrier variable, it de-

notes a clock condition that will be known only when instantiating the node `N`. In the set of equations `c=and(c1,c2); o=i when c`, `c` is a carrier name, it denotes the clock condition `and(c1,c2)`. This approach, used in [CP03], allows to simplify the clock calculus by considering clock conditions as mere names instead of arbitrary complex Boolean expressions.

Environments (H) associate clock types to flow variables and node definitions. As for typing environments, they can be considered as functions, which associate a clock type to variable names or node names.

Finally, clocks can be generalized (at a node definition) and instantiated (at a node call) as follows:

$$inst(\forall\alpha <: C.cl) = cl[(cl' \in C)/\alpha]$$
$$gen_H(cl) = \forall\alpha_1 <: C_1, ..., \alpha_m <: C_m.cl, \text{ where } \alpha_1, ..., \alpha_m = FTV(cl)\backslash FTV(H)$$

This states that a clock scheme is instantiated by replacing clock variables by clocks belonging to the correct clock subtypes and that any clock variable can be generalized if it does not appear free in the environment.

A clock variable α can be constrained to belong to some clock subtype C_i, which is denoted by $\alpha <: C_i$. Such a clock subtyping constraint states that only clock types belonging to C_i can be substituted for α. For instance, if x has clock α, then $x *\hat{} k$ is well-synchronized only if $k|\pi(\alpha)$, because otherwise the period of $x *\hat{} k$ is not an integer (see Sect. 3.2.3). Such sub-typing constraints are called *bounded types quantifications* in [Pie02].

Let $\mathcal{C}$ denote the set of all clock types. The two main clock subtypes of $\mathcal{C}$ are Boolean clocks $\mathcal{B}$ (clocks containing the operator `on`) and strictly periodic clocks $\mathcal{P}$. Clock type $\mathcal{P}$ contains subtypes $\mathcal{P}_{k,\frac{a}{b}}$, $k \in \mathbb{N}^*$, $\frac{a}{b} \in \mathbb{Q}^+$, with $a \wedge b = 1$ (a and b are co-prime) and $b|k$, where:

$$\mathcal{P}_{k,\frac{a}{b}} = \{(n,p)|(k|n) \wedge p \geq \frac{a}{b}\}$$

In other words, the set $\mathcal{P}_{k,\frac{a}{b}}$ contains all the strictly periodic clocks α for which transformations $\alpha *. k$ and $\alpha \rightarrow. -\frac{a}{b}$ produce valid strictly periodic clocks. Notice that $\mathcal{P} = \mathcal{P}_{1,0}$.

The sub-typing relation $<:$ on clock types is defined as follows:

- $\mathcal{B} <: \mathcal{C}$, $\mathcal{P} <: \mathcal{C}$;
- $\mathcal{P}_{k,q} <: \mathcal{P}_{k',q'} \Leftrightarrow k'|k \wedge q \geq q'$.

The set of all subsets of $\mathcal{P}$ ordered by the subset inclusion $<:$ forms a lattice. This allows to solve subtyping constraints efficiently, by replacing two constraints $t <: C_1$ and $t <: C_2$ by $t <: C_1 \cap C_2$ ie the least upper bound of the sets C_1 and C_2 (see Sect. 6.3.4 for details).

6.3.2 Clock inference

Strictly periodic clock parent

The language allows to mix freely Boolean and strictly periodic clocks. As our objective is to schedule programs efficiently, we need to be able to compute a period for each flow of the program. When the clock of a flow contains Boolean transformations, we need to approximate the clock to a strictly periodic clock to be able to compute the period of the flow. To this intent, we first define the over-approximation of a clock to its closest strictly periodic clock parent:

Definition 6. *The strictly periodic clock parent of a clock is defined recursively as follows:*

$$pparent(\alpha \text{ on } c) = pparent(\alpha)$$
$$pparent(pck) = pck$$

Then, we need to ensure that the clocks of a program are always derived, at least indirectly, from a strictly periodic clock. All the clocks of a program are derived indirectly from the clocks of the inputs of the main node, thus, in addition to the inference rules below, the clock calculus requires the clocks of the inputs of the main node of a program to be strictly periodic clocks. This ensures that for any clock ck, $pparent(ck)$ always returns a strictly periodic clock. So, we are able to approximate the clock of any flow to a strictly periodic clock and we can compute a period for each flow of the program.

Inference rules

We give the clock inference rules in Fig. 6.3.

$$\text{(SUB)}\ \frac{H \vdash e : ck \qquad H \vdash ck <: C}{H \vdash e : C} \qquad \text{(CONST)}\ \frac{}{H \vdash cst : ck} \qquad \text{(VAR)}\ \frac{x \in dom(H)}{H \vdash x : H(x)}$$

$$(\times)\ \frac{H \vdash e_1 : cl_1 \qquad H \vdash e_2 : cl_2}{H \vdash (e_1, e_2) : cl_1 \times cl_2} \qquad \text{(FBY)}\ \frac{H \vdash cst : cl \qquad H \vdash e : cl}{H \vdash cst \ \texttt{fby}\ e : cl}$$

$$\text{(WHEN)}\ \frac{H \vdash e : ck \qquad H \vdash c : (c : ck)}{H \vdash e\ \texttt{when}\ c : c\ \texttt{on}\ ck} \qquad \text{(WHEN-NOT)}\ \frac{H \vdash e : ck \qquad H \vdash c : (c : ck)}{H \vdash e\ \texttt{whennot}\ c : c\ \texttt{on not}\ ck}$$

$$\text{(MERGE)}\ \frac{H \vdash c : (c : ck) \qquad H \vdash e_1 : ck\ \texttt{on}\ c \qquad H \vdash e_2 : ck\ \texttt{on not}\ c}{H \vdash \texttt{merge}\ (c, e_1, e_2) : ck}$$

$$(\hat{*})\ \frac{H \vdash e : pck \qquad pck <: \mathcal{P}_{k,0} \qquad k \neq 0}{H \vdash e *\hat{}\, k : pck *_. k} \qquad (/\hat{})\ \frac{H \vdash e : pck \qquad k \neq 0}{H \vdash e /\hat{}\, k : pck /_. k}$$

$$(\sim>)\ \frac{H \vdash e : pck \qquad pck <: \mathcal{P}_{b,0} \qquad (q = \frac{a}{b}), a \wedge b = 1}{H \vdash e \sim> q : pck \rightarrow_. q} \qquad (\texttt{tail})\ \frac{H \vdash e : pck}{H \vdash \texttt{tail}\ (e) : pck \rightarrow_. 1}$$

$$(\texttt{::})\ \frac{H \vdash cst : pck \qquad H \vdash e : pck \qquad pck <: \mathcal{P}_{1,1}}{H \vdash cst\ \texttt{::}\ e : pck \rightarrow_. -1}$$

$$\text{(APP)}\ \frac{H \vdash N : \sigma \qquad cl'_1 \rightarrow cl'_2 = inst(\sigma) \qquad H \vdash e : cl'_1}{H \vdash N(e) : cl'_2} \qquad \text{(EQ)}\ \frac{H \vdash x : cl \qquad H \vdash e : cl}{H \vdash x = e}$$

$$\text{(EQS)}\ \frac{H \vdash eq_1 \qquad H \vdash eq_2}{H \vdash eq_1; eq_2}$$

$$\text{(NODE)}\ \frac{in : cl_1, out : cl_2, var : cl_3, H \vdash eqs}{H \vdash\ \texttt{node}\ N(in)\ \texttt{returns}\ (out)\ [\ \texttt{var}\ var;]\ \texttt{let}\ eqs\ \texttt{tel}\ : gen_H(cl_1 \rightarrow cl_2)}$$

$$\text{(IMP-NODE)}\ \frac{in : cl_1 \qquad out : cl_2 \qquad pparent(cl_1) = pparent(cl_2)}{H \vdash\ \texttt{imported node}\ N(in)\ \texttt{returns}\ (out)\ \texttt{wcet}\ n : gen_H(cl_1 \rightarrow cl_2)}$$

$$\text{(SENSOR)}\ H \vdash\ \texttt{sensor}\ x\ \texttt{wcet}\ n \qquad \text{(ACTUATOR)}\ H \vdash\ \texttt{actuator}\ x\ \texttt{wcet}\ n$$

$$\text{(DECLS-NODE)}\ \frac{H \vdash [\ \texttt{imported}\ ]\ \texttt{node}\ N[...] : \sigma \qquad H, N : \sigma \vdash decl_2}{H \vdash [\ \texttt{imported}\ ]\ \texttt{node}\ N[...]\ \ decl_2}$$

$$\text{(DECLS-OTHER)}\ \frac{H \vdash decl_1 \qquad H \vdash decl_2}{H \vdash decl_1\ \ decl_2}$$

Figure 6.3: Clock inference rules

- The *(SUB)* rule is the classical subsumption rule of sub-typing [Pie02] applied to clock types. It states that if ck is a clock subtype of C then any expression of clock ck also has clock type C;

- A constant can have any clock (rule *(CONST)*);
- A variable must be declared in the environment (rule *(VAR)*);
- The clock of a pair is the product of the clock of its components (rule *($\times$)*);
- Operator `fby` takes two arguments of the same clock and the result also has this clock;
- The two arguments of operator `when` must be present at the same instants (on clock ck). The judgement $H \vdash c : (c : ck)$ states that the expression c defines a clock condition c and that this condition has clock ck. This notation makes the clock condition transported by the expression appear clearly in its clock, which allows to use this condition in other clock expressions (ck `on` c for instance). The result of e `when` c is present only at the dates defined by ck at which c is true, which is denoted ck `on` c;
- The constraints are the same for operator `whennot`, except that the result is present only at the dates defined by ck at which c is false, which is denoted ck `on not` c;
- The first argument of operator `merge` specifies a sampling condition c that has clock ck. The two other arguments have complementary clocks: the second argument is present at the dates defined by clock ck for which c is true and the third is present at the dates defined by clock ck for which c is false. The result is present on clock ck;
- Operator `*^` can only be applied to an expression the clock of which is a subtype of $\mathcal{P}_{k,0}$. This ensures that the period of the resulting clock is an integer. The result is k times faster than the argument;
- Operator `/^` can only be applied to any expression the clock of which is strictly periodic. The result is k times slower than the argument;
- The parameter q of operator $\sim>$ denotes a rational $\frac{a}{b}$ for which $a \wedge b = 1$. The constraint $pck <: \mathcal{P}_{b,0}$ ensures that the phase of $pck \rightarrow_{.} q$ is an integer;
- Operator `tail` can be applied to any expression the clock of which is strictly periodic. The phase of the result is increased by 1;
- Operator `::` can only be applied to an expression the clock of which is a subtype of $\mathcal{P}_{1,1}$. This ensures that the phase of the result is positive as the phase of the result is decreased by 1;
- The rule *(APP)* states that the arguments of a node call must have the clock expected for the inputs of the node. The result of the node call has the clock of the outputs of the node. The clock of the node is instantiated during the node call, in case it is polymorphic (quantified). During the instantiation, the clock calculus verifies that the clocks of the inputs provided for the node belong to the correct clock subtype (see below for details);
- An equation is well-synchronized if its left hand side and its right hand side have the same clock (rule *EQ*);
- A set of equations is well-synchronized if each equation of the set is well-synchronized;
- The rule (*NODE*) states that to perform the clock calculus of a node declaration, we first add the input, output and local variables of the node in the environment. The clock of each variable in the environment is the clock declared in the node if any, or a fresh clock variable otherwise. Then, we

perform the clock calculus of the set of equations of the node in this new environment. Finally, the clock of the node is a functional clock, from the clock of its inputs to the clock of its outputs;

- The rule (*IMP-NODE*) requires the strictly periodic clock parent of all the inputs and outputs of an imported node to be the same. This implies that the inputs and outputs can have different clock activation conditions but if we do not consider clock conditions, they have the same strictly periodic clocks. For instance:
  `imported node I(c: bool; i: int when c) returns (o: int)`
  has clock $(c : \alpha) \times \alpha \texttt{ on } c \rightarrow \alpha$, which is valid as the strictly periodic clock parent of every input/output is α. On the opposite:
  `imported node I(i: int rate(10,0)) returns (o: int rate (5,0))`
  has clock $(10,0) \times (5,0)$, which is invalid. This restriction allows to compute a single activation rate (period and phase) for each imported node instantiation;

- Sensor (*SENSOR*) and actuator (*ACTUATOR*) declarations are always well-synchronized (no premisse);

- The rule (*DECLS-NODE*) states that a sequence of declarations consisting of the declaration of a node N (imported or not) followed by another declaration is well-synchronized if the node declaration is well-synchronized, has clock σ, and if the second declaration is well-synchronized in an environment where the node N has clock σ;

- For other kinds of declarations (*DECLS-OTHER*), a sequence of declarations is well-synchronized if the different declarations of the sequence are well-synchronized.

6.3.3 Clocks unification

Equivalence of strictly periodic clock expressions

Periodic clock transformations are such that different strictly periodic clock expressions may correspond to the same clock. Thus, clocks unification requires the definition of an equivalence test on strictly periodic clock expressions. This test relies on a term rewriting system. We show that testing the equivalence of two strictly periodic clock expressions can be reduced to testing the equality of their normal forms. The terminology and proof techniques we use in this section can be found in [BN98].

Let $\mathcal{E}^{\mathcal{P}}$ denote the set of expressions on strictly periodic clocks, that is expressions obtained by applying a succession of periodic clock transformations on a strictly periodic clock. The set $\mathcal{E}^{\mathcal{P}}$ is the set of clocks types pck defined in the clock types grammar of Sect. 6.3.1. The following arithmetical properties are derived from the definition of periodic clock transformations:

Property 4. *$\forall \alpha \in \mathcal{P}$:*

- $\forall k, k' \in \mathbb{N}^{+*}, \alpha *_. k *_. k' = \alpha *_. kk'$;
- $\forall k, k' \in \mathbb{N}^{+*}, \alpha /_. k /_. k' = \alpha /_. kk'$;
- $\forall q, q' \in \mathbb{Q}, \alpha \rightarrow_. q \rightarrow_. q' = \alpha \rightarrow_. (q + q')$;
- $\forall k, k' \in \mathbb{N}^{+*}, \alpha /_. k *_. k' = \alpha *_. k' /_. k$;
- $\forall q \in \mathbb{Q}, \forall k \in \mathbb{N}^{+*}, \alpha \rightarrow_. q *_. k = \alpha *_. k \rightarrow_. qk$;
- $\forall q \in \mathbb{Q}, \forall k \in \mathbb{N}^{+*}, \alpha \rightarrow_. q /_. k = \alpha /_. k \rightarrow_. (q/k)$.

We can define a rewriting system $\mathcal{R}_\mathcal{P}$ to simplify expressions in $\mathcal{E}^\mathcal{P}$ based on these properties:

Definition 7. *The rewriting system $\mathcal{R}_\mathcal{P}$ on expressions in $\mathcal{E}^\mathcal{P}$ is defined as:*

$$\alpha *_. k *_. k' \mapsto \alpha *_. kk' \tag{6.1}$$
$$\alpha /_. k /_. k' \mapsto \alpha /_. kk' \tag{6.2}$$
$$\alpha \rightarrow_. q \rightarrow_. q' \mapsto \alpha \rightarrow_. (q + q') \tag{6.3}$$
$$\alpha /_. k *_. k' \mapsto \alpha *_. k' /_. k \tag{6.4}$$
$$\alpha \rightarrow_. q *_. k \mapsto \alpha *_. k \rightarrow_. qk \tag{6.5}$$
$$\alpha \rightarrow_. q /_. k \mapsto \alpha /_. k \rightarrow_. (q/k) \tag{6.6}$$

Property 5. *The rewriting system $\mathcal{R}_\mathcal{P}$ is convergent.*

For $\mathcal{R}_\mathcal{P}$ to be convergent, it must *terminate* and it must be *confluent*.

Termination. The principle of the proof is to define a strict order $<_{\mathcal{E}^\mathcal{P}}$ on $\mathcal{E}^\mathcal{P}$ and to prove that for all rewriting rule $l \mapsto r$, $r <_{\mathcal{E}^\mathcal{P}} l$. This is a sufficient condition for $\mathcal{R}_\mathcal{P}$ to terminate (see [BN98] for details). To this intent, we introduce the following notations, for e in $\mathcal{E}^\mathcal{P}$:

- $|e|_{*_.}$ denotes the number of operations $*_.$ appearing in e;
- $|e|_{/_.}$ denotes the number of operations $/_.$ appearing in e;
- $|e|_{\rightarrow_.}$ denotes the number of operations $\rightarrow_.$ appearing in e;
- $[e]_{*_.}$ denotes the number of operations preceding the first operation $*_.$ appearing in e. For instance, $[\alpha /_. n \rightarrow_. q *_. n' /_. n'' *_. n''']_{*_.} = 2$;
- Similarly, $[e]_{/_.}$ denotes the position of the first operation $/_.$ appearing in e.

$<_{\mathcal{E}^\mathcal{P}}$ is then the order defined as:

$$\begin{aligned} e <_{\mathcal{E}^\mathcal{P}} e' \Leftrightarrow & |e|_{*_.} < |e'|_{*_.} \vee \\ & (|e|_{*_.} = |e'|_{*_.} \wedge |e|_{/_.} < |e'|_{/_.}) \vee \\ & (|e|_{*_.} = |e'|_{*_.} \wedge |e|_{/_.} = |e'|_{/_.} \wedge |e|_{\rightarrow_.} < |e'|_{\rightarrow_.}) \vee \\ & (|e|_{*_.} = |e'|_{*_.} \wedge |e|_{/_.} = |e'|_{/_.} \wedge |e|_{\rightarrow_.} = |e'|_{\rightarrow_.} \wedge [e]_{*_.} < [e']_{*_.}) \vee \\ & (|e|_{*_.} = |e'|_{*_.} \wedge |e|_{/_.} = |e'|_{/_.} \wedge |e|_{\rightarrow_.} = |e'|_{\rightarrow_.} \wedge [e]_{*_.} = [e']_{*_.} \wedge [e]_{/_.} < [e']_{/_.}) \end{aligned}$$

It is then trivial to prove that for all rewriting rule $l \mapsto r$ of $\mathcal{R}_\mathcal{P}$, we have $r <_{\mathcal{E}^\mathcal{P}} l$. Thus, $\mathcal{R}_\mathcal{P}$ terminates. □

Confluence. As $\mathcal{R}_\mathcal{P}$ terminates, to prove its confluence we simply prove the local confluence of its *critical pairs* (see [BN98] for details). Two rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ form a critical pair if they overlap, that is to say if there exists a sub-term s_1 of l_1 (resp. s_2 of l_2) that is not a variable, and a unifier (a term substitution) θ such that $\theta(s_1) = \theta(r_2)$ (resp. $\theta(s_2) = \theta(r_1)$). Unification is applied after renaming variables such that $Vars(l_1, r_1) \cap Vars(l_2, r_2) = \{\}$. There are only two critical pairs in the system $\mathcal{R}_\mathcal{P}$:

- Rule 6.3 and Rule 6.5 overlap. Indeed, we can substitute $\beta \rightarrow_. q' \rightarrow_. q''$ for $\alpha \rightarrow_. q$ in the second rule. We must then verify that the different possible rewritings for $\beta \rightarrow_. q' \rightarrow_. q'' *_. k$ lead to the same final result. If we start by applying the first rule:

$$\begin{aligned}\beta \rightarrow_. q' \rightarrow_. q'' *_. k &\mapsto \beta \rightarrow_. (q' + q'') *_. k \\ &\mapsto \beta *_. k \rightarrow_. (q' + q'')k\end{aligned}$$

  If we start by applying the second rule:

$$\begin{aligned}\beta \rightarrow_. q' \rightarrow_. q'' *_. k &\mapsto \beta \rightarrow_. q' *_. k \rightarrow_. (q''k) \\ &\mapsto \beta *_. k \rightarrow_. (q'k) \rightarrow_. (q''k) \\ &\mapsto \beta *_. k \rightarrow_. (q' + q'')k\end{aligned}$$

  So this critical pair is locally confluent.

- Similarly, Rule 6.3 and Rule 6.6 overlap by substituting $\beta \rightarrow_. q' \rightarrow_. q''$ for $\alpha \rightarrow_. q$ in the second rule. We must check the different rewritings for $\beta \rightarrow_. q' \rightarrow_. q''/_.k$. If we start by applying the first rule:

$$\begin{aligned}\beta \rightarrow_. q' \rightarrow_. q''/_.k &\mapsto \beta \rightarrow_. (q' + q'')/_.k \\ &\mapsto \beta/_.k \rightarrow_. (q' + q'')/k\end{aligned}$$

  If we start by applying the second rule:

$$\begin{aligned}\beta \rightarrow_. q' \rightarrow_. q''/_.k &\mapsto \beta \rightarrow_. q'/_.k \rightarrow_. (q''/k) \\ &\mapsto \beta/_.k \rightarrow_. (q'/k) \rightarrow_. (q''/k) \\ &\mapsto \beta/_.k \rightarrow_. (q' + q'')/k\end{aligned}$$

All the critical pairs of $\mathcal{R}_\mathcal{P}$ are locally confluent, so $\mathcal{R}_\mathcal{P}$ is confluent. □

As $\mathcal{R}_\mathcal{P}$ is convergent, any expression e of $\mathcal{E}^\mathcal{P}$ has a unique normal form, denoted $NF_{\mathcal{R}_\mathcal{P}}(e)$. Furthermore:

Lemma 1. *For all clock expression $e \in \mathcal{E}^\mathcal{P}$, $NF_{\mathcal{R}_\mathcal{P}}(e)$ is of the form:*

$$NF_{\mathcal{R}_\mathcal{P}}(e) = \alpha *_. k/_.k' \rightarrow_. q$$

with $k, k' \in \mathbb{N}$ and $q \in \mathbb{Q}$

Proof. This amounts to saying that for any expression e in $\mathcal{E}^\mathcal{P}$, we can always apply a rewriting rule of $\mathcal{R}_\mathcal{P}$ if the following condition holds:

$$|e|_{*_.} > 1 \vee |e|_{/_.} > 1 \vee |e|_{\rightarrow_.} > 1 \vee [e]_{*_.} > 0 \vee [e]_{/_.} > 1$$

We consider each possibility separately:

- $|e|_{*_.} > 1$: then either two operations $*_.$ appear next to each other in the expression and we can apply rule 6.1 or no operation $*_.$ appear next to each other in the expression and we can apply rules 6.5 or 6.4;

- $|e|_{/.} > 1$: then either two operations $/.$ appear next to each other in the expression and we can apply rule 6.2 or no operation $/.$ appear next to each other in the expression and we can apply rules 6.6;
- $[e]_{*.} > 0$: then we can apply rule 6.2, 6.1, 6.3, 6.4 or 6.5;
- $[e]_{/.} > 1$: then we can apply rule 6.2, 6.1, 6.3 or 6.6;

□

In some cases, we can further simplify the normal form of a strictly periodic clock expression. We use the following property to define the canonical form of a strictly periodic clock expression in normal form:

Property 6. *For all m, k, k' in $\mathbb{N}^*$, for all q in $\mathbb{Q}$, if $m|k$ and $m|k'$, then:*

$$\alpha *. \ k/.k' \rightarrow. \ q = \alpha *. \ (k/m)/.(k'/m) \rightarrow. \ q$$

Proof. Using the definitions of periodic clock transformations and Property. 4, we have:

$$\begin{aligned} \alpha *. \ (k/m)/.(k'/m) \rightarrow. \ q &= \alpha *. \ k/.m/.k' *. \ m \rightarrow. \ q \\ &= \alpha *. \ k/.m *. \ m/.k' \rightarrow. \ q \\ &= \alpha *. \ k/.k' \rightarrow. \ q \end{aligned}$$

□

From this property, we deduce an equivalence relation on clocks in normal forms:

Property 7. *Let e_1, e_2 be strictly periodic clock expression in normal forms, with $e_1 = \alpha *. \ k_0/.k_1 \rightarrow. \ q$ and $e_2 = \alpha *. \ k'_0/.k'_1 \rightarrow. \ q'$. Then e_1 and e_2 are equivalent if and only if:*

$$q = q' \wedge k'_0 * k_1 = k'_1 * k_0$$

Definition 8. *From this equivalence relation, for any expression $e = \alpha *. \ k/.k' \rightarrow. \ q$, the representative element $e_{rep} = \alpha *. \ k_r/.k'_r \rightarrow. \ q_r$ of its equivalence class $[e]$ is such that:*

$$q = q_r \wedge \exists g \in \mathbb{N}^*, k = gk_r, k' = gk'_r$$

Definition 9. *The canonical form $CF(e)$ of a strictly periodic clock expression e in normal form is the representative of its equivalence class. In other words:*

$$CF(\alpha *. \ k/.k' \rightarrow. \ q) = \alpha *. \ \frac{k}{gcd(k,k')}/. \frac{k'}{gcd(k,k')} \rightarrow. \ q$$

Let $\equiv$ denote the (semantic) equivalence of strictly periodic clocks. Then, we obtain the following theorem by construction:

Theorem 1. *Let e_1, e_2 be expressions of $\mathcal{E}^{\mathcal{P}}$. Then:*

$$(e_1 \equiv e_2) \Leftrightarrow CF(NF_{\mathcal{R}_{\mathcal{P}}}(e_1)) = CF(NF_{\mathcal{R}_{\mathcal{P}}}(e_2))$$

$$(\rightarrow)\ \frac{cl_1 \sim cl'_1 \qquad cl_2 \sim cl'_2}{cl_1 \rightarrow cl_2 \sim cl'_1 \rightarrow cl'_2} \quad (\times)\ \frac{cl_1 \sim cl'_1 \qquad cl_2 \sim cl'_2}{cl_1 \times cl_2 \sim cl'_1 \times cl'_2} \quad (\text{ON})\ \frac{ck_1 \sim ck_2 \qquad c_1 = c_2}{ck_1 \texttt{ on } c_1 \sim ck_2 \texttt{ on } c_2}$$

$$(\text{VAR})\ \frac{\alpha \in FTV(pck) \qquad \alpha <: \mathcal{P}_{k,q} \cap \mathcal{P}_{k',q'} \qquad \alpha = CF(NF(pck))}{\alpha <: \mathcal{P}_{k,q} \sim ck <: \mathcal{P}_{k',q'}}$$

$$(\text{VAR'})\ \frac{\alpha \notin FTV(ck) \qquad ck <: \mathcal{P}_{k,q} \cap \mathcal{P}_{k',q'} \qquad \alpha \mapsto CF(NF(ck))}{\alpha <: \mathcal{P}_{k,q} \sim ck <: \mathcal{P}_{k',q'}}$$

$$(\text{CONST})\ \frac{concrete(pck_1) \qquad concrete(pck_2) \qquad \pi(pck_1) = \pi(pck_2) \qquad \varphi(pck_1) = \varphi(pck_2)}{pck_1 \sim pck_2}$$

$$(*.)\ \frac{abstract(pck) \qquad pck \sim pck' /. k}{pck *. k \sim pck'} \quad (/.)\ \frac{abstract(pck) \qquad pck \sim pck' *. k \qquad pck' <: \mathcal{P}_{k,0}}{pck /. k \sim pck'}$$

$$(\rightarrow.)\ \frac{abstract(pck) \qquad pck \sim pck' \rightarrow. (-\frac{a}{b}) \qquad pck' <: \mathcal{P}_{b,\frac{a}{b}}}{pck \rightarrow. \frac{a}{b} \sim pck'}$$

Figure 6.4: Clock unification rules

Unification rules

For two clocks ck_1, ck_2, we denote $ck_1 \sim ck_2$ when ck_1 and ck_2 are unifiable. For any clock ck, let $concrete(ck)$ denote the predicate that returns true if ck is a concrete clock, false otherwise and let $abstract(ck)$ denote the predicate that returns true if ck is abstract, false otherwise. Now that we defined this equivalence test, clock unification rules are given in Fig. 6.4. An example of clocks unification is detailed in Sect. 6.3.5.

- We use structural unification for clocks $\rightarrow$ and $\times$;
- Two clocks $ck_1 \texttt{ on } c_1$ and $ck_2 \texttt{ on } c_2$ can be unified if ck_1 and ck_2 can be unified and if c_1 and c_2 are *syntactically* equal. This implies that sometimes two *semantically* equivalent clocks will be considered non-unifiable. For instance if $c_1 = a \wedge b$ and $c_2 = b \wedge a$, the clocks $\alpha \texttt{ on } c_1$ and $\alpha \texttt{ on } c_2$ are not unifiable. This is however a reasonable restriction in practice as the programmer should simply replace any expression x `when` c_2 by x `when` c_1 to solve the problem;
- The unification of a clock variable α with a clock expression ck has two variants. If α appears in ck and ck is a strictly periodic clock (rule *VAR*), then the normal form of ck must be α for the two clocks to be unifiable. If α does not occur in ck (rule *VAR'*), $NF(ck)$ can be substituted for α (denoted $\alpha \mapsto NF(ck)$). In both cases, ck and α must be subtypes of the intersection of their subtyping constraints. Rule (*VAR'*) is the only rule in which substitutions take place;
- Rule (*CONST*) says that two concrete periodic clocks can be unified if their periods and phases are equal. The period and phase of a concrete strictly periodic clock is easily computed using the definition of periodic clock transformations given in Sect. 3.2;
- Rules $(*.)$, $(/.)$ and $(\rightarrow.)$ rely on Property. 1 to rewrite the unification problem. Rules $(/.)$ and $(\rightarrow.)$ also introduce sub-typing constraints, the verification of which relies on Property. 8.

Unification is illustrated in Sect. 6.3.5.

6.3.4 Subsumption

At several points, clock inference rules and clocks unification require to verify that a clock expression belongs to some subtype. To verify that ck is a subtype of C_i (ie that ck subsumes C_i), we use the

following properties:

Property 8. $\forall \alpha \in \mathcal{P}, \forall k, k' \in \mathbb{N}^{+*}, \forall q, q' \in \mathbb{Q}$:

- $\alpha *. k' <: \mathcal{P}_{k,q} \Leftrightarrow \alpha <: \mathcal{P}_{k*k',q/k'}$;
- $\alpha /. k' <: \mathcal{P}_{k,q} \Leftrightarrow \alpha <: \mathcal{P}_{k/gcd(k,k'),q*k'}$;
- $\alpha \rightarrow. q' <: \mathcal{P}_{k,q} \Leftrightarrow \alpha <: \mathcal{P}_{k,max(0,(q-q'))}$;
- $\alpha <: \mathcal{P}_{k,q} \wedge \alpha <: \mathcal{P}_{k',q'} \Leftrightarrow \alpha <: \mathcal{P}_{k,q} \cap \mathcal{P}_{k',q'} \Leftrightarrow \alpha <: \mathcal{P}_{lcm(k,k'),max(q,q')}$.

Proof. Let $\alpha = (n, p)$. Then, from Property. 2 and from the definition of $\mathcal{P}_{k,q}$:

$$\begin{aligned} \alpha *. k' <: \mathcal{P}_{k,q} &\Leftrightarrow (\frac{n}{k'}, k'p) <: \mathcal{P}_{k,q} \Leftrightarrow k|\frac{n}{k'} \wedge k'p \geq q \\ &\Leftrightarrow k * k'|n \wedge p \geq q/k' \Leftrightarrow \alpha <: \mathcal{P}_{k*k',q/k'} \end{aligned}$$

Similarly:

$$\begin{aligned} \alpha /. k' <: \mathcal{P}_{k,q} &\Leftrightarrow (nk', \frac{p}{k'}) <: \mathcal{P}_{k,q} \Leftrightarrow k|nk' \wedge \frac{p}{k'} \geq q \\ &\Leftrightarrow k/gcd(k,k')|n \wedge p \geq q * k' \Leftrightarrow \alpha <: \mathcal{P}_{k/gcd(k,k'),q*k'} \end{aligned}$$

Similarly:

$$\begin{aligned} \alpha \rightarrow. q' <: \mathcal{P}_{k,q} &\Leftrightarrow (n, p + q') <: \mathcal{P}_{k,q} \Leftrightarrow k|n \wedge p + q' \geq q \\ &\Leftrightarrow k|n \wedge p \geq q - q' \wedge p \geq 0 \Leftrightarrow \alpha <: \mathcal{P}_{k,max(0,(q-q'))} \end{aligned}$$

And last:

$$\begin{aligned} \alpha <: \mathcal{P}_{k,q} \wedge \alpha <: \mathcal{P}_{k',q'} &\Leftrightarrow k|n \wedge k'|n \wedge p \geq q \wedge p \geq q' \Leftrightarrow lcm(k,k')|n \wedge p \geq max(q,q') \\ &\Leftrightarrow \alpha <: \mathcal{P}_{lcm(k,k'),max(q,q')} \end{aligned}$$

□

When read from left to right, these properties form subsumption rules that can be applied to transfer sub-typing constraints on a clock type to the clock variable appearing in it (for an abstract clock) or to the constant clock appearing in it (for a concrete clock). In the former case, the constraints are just added to the constraints already existing on this clock variable. In the latter case, we can directly check that the constant clock verifies the constraints. This is illustrated in the next section.

6.3.5 Example

We consider the following example:

```
imported node imp(i, j: int) returns (o: int) wcet 10;

node N(i, j, c) returns (o)
let
  o=imp((i*^2) when c, (j/^4*^3) when c);
tel
```

The rule *(IMP-NODE)* can be applied to infer the type of node `imp`:

$$\text{(IMP-NODE)}\ \frac{i,j : cl_1 \times cl_1 \qquad o : cl_1}{H_{init} \vdash \texttt{imported node}\ imp(i,j)\ \texttt{returns}\ (o)\ \texttt{wcet}\ 10 : gen_H(cl_1 \times cl_1 \rightarrow cl_1) = \forall\alpha.\alpha \times \alpha \rightarrow \alpha}$$

As the clock parent of the node inputs and outputs must be unifiable, the clocks of the inputs and outputs are all equal to the same clock variable (cl_1). This clock variable can then be generalized (to clock α). This produces a new clock environment $H_0 = imp : \forall\alpha.\alpha \times \alpha \rightarrow \alpha, H_{init}$, which states that the node `imp` can only be applied to two arguments of the same clock and that the result of the application has the clock of the arguments.

Then, for node `N`, the rule *(NODE)* says that we must perform the clock calculus of the equations of the node in environment $H_1 = i : cl_2, j : cl_3, c : cl_4, o : cl_5, H_0$, where cl_2, cl_3, cl_4, cl_5 are fresh clock type variables:

$$\text{(EQ)}\ \frac{H_1 \vdash o : cl_5 \qquad \dfrac{\begin{array}{c} H_1 \vdash imp : \forall\alpha.\alpha \times \alpha \rightarrow \alpha \qquad cl_5 \times cl_5 \rightarrow cl_5 = inst(\forall\alpha.\alpha \times \alpha \rightarrow \alpha) \\ H_1 \vdash (i *\hat{}\, 2)\ \texttt{when}\ c : cl_5 \qquad H_1 \vdash (j /\hat{}\, 4 *\hat{}\, 3)\ \texttt{when}\ c : cl_5 \end{array}}{H_1 \vdash imp((i *\hat{}\, 2)\ \texttt{when}\ c, (j /\hat{}\, 4 *\hat{}\, 3)\ \texttt{when}\ c) : cl_5}\ \text{(APP)}}{H_1 \vdash o = imp((i *\hat{}\, 2)\ \texttt{when}\ c, (j /\hat{}\, 4 *\hat{}\, 3)\ \texttt{when}\ c);}$$

Now, we must compute the clock of the two arguments of `imp` and verify that they are unifiable, as they must both be equal to the clock type of `o`, ie cl_5:

$$\text{(WHEN)}\ \frac{(*\hat{})\ \dfrac{H_1 \vdash i : pck_1 \qquad pck_1 <: \mathcal{P}_{2,0}}{H_1 \vdash i *\hat{}\, 2 : pck_1 *.\, 2} \qquad H_1 \vdash c : (c : pck_1 *.\, 2)}{H_1 \vdash (i *\hat{}\, 2)\ \texttt{when}\ c : pck_1 *.\, 2\ \texttt{on}\ c}$$

And:

$$\text{(WHEN)}\ \frac{(*\hat{})\ \dfrac{(/\hat{})\ \dfrac{H_1 \vdash j : pck_2}{H_1 \vdash j /\hat{}\, 4 : pck_2 /.\, 4} \qquad pck_2 /.\, 4 <: \mathcal{P}_{3,0}}{H_1 \vdash j /\hat{}\, 4 *\hat{}\, 3 : pck_2 /.\, 4 *.\, 3} \qquad H_1 \vdash c : (c : pck_2 /.\, 4 *.\, 3)}{H_1 \vdash (j /\hat{}\, 4 *\hat{}\, 3)\ \texttt{when}\ c : pck_2 /.\, 4 *.\, 3\ \texttt{on}\ c}$$

We can see that cl_2 and cl_3 are actually strictly periodic clocks, so they are replaced by pck_1 and pck_2. Furthermore, we have $pck_2 /.\, 4 <: \mathcal{P}_{3,0}$, so subsumption rules tell us that $pck_2 <: \mathcal{P}_{3,0}$.

Now we must unify clocks $pck_1 *.\, 2\ \texttt{on}\ c$ and $pck_2 /.\, 4 *.\, 3\ \texttt{on}\ c$. From the unification rules, we have:

$$\begin{aligned} pck_1 *.\, 2\ \texttt{on}\ c = pck_2 /.\, 4 *.\, 3\ \texttt{on}\ c &\Leftrightarrow pck_1 *.\, 2 = pck_2 /.\, 4 *.\, 3 \wedge c = c \\ &\Leftrightarrow pck_1 *.\, 2 /.\, 3 = pck_2 /.\, 4 && \text{(keep most ancient variable)} \\ &\Leftrightarrow pck_1 *.\, 2 /.\, 3 *.\, 4 = pck_2 \\ &\Leftrightarrow pck_1 *.\, 8 /.\, 3 = pck_2 && \text{(after normalisation)} \end{aligned}$$

So we can substitute $pck_1 *.\, 8 /.\, 3$ for pck_2. As $pck_2 <: \mathcal{P}_{3,0}$, when applying the substitution we have the constraint $pck_1 *.\, 8 /.\, 3 <: \mathcal{P}_{3,0}$. Using subsumption rules, we have:

$$\begin{aligned} pck_1 *.\, 8 /.\, 3 <: \mathcal{P}_{3,0} &\Leftrightarrow pck_1 *.\, 8 <: \mathcal{P}_{gcd(3,3),0} \\ &\Leftrightarrow pck_1 *.\, 8 <: \mathcal{P}_{1,0} \\ &\Leftrightarrow pck_1 <: \mathcal{P}_{8,0} \end{aligned}$$

We already had the constraint $pck_1 <: \mathcal{P}_{2,0}$, so we must take the intersection between the two constraints, which is $pck_1 <: \mathcal{P}_{8,0}$.

We can now give the type of node N:

$$H_0 \vdash N : gen_{H_0}(pck_1 \times pck_1 *. 8/.3 \times (c : pck_1 *. 2) \rightarrow pck_1 *. 2 \texttt{ on } c)$$

And after generalisation:

$$H_0 \vdash N : \forall \alpha <: \mathcal{P}_{8,0}.\alpha \times \alpha *. 8/.3 \times (c : \alpha *. 2) \rightarrow \alpha *. 2 \texttt{ on } c$$

The subtyping constraint tells us that when applying the node N, the period of the clock of the first argument of the node must be a multiple of 8.

Now, the following node correctly instantiates the node N:

```
node inst1(i: rate (16, 0); j: rate(6, 0); c) returns (o)
let
  o=N(i, j, c);
tel
```

The clock of inst1 is: $(16, 0) \times (6, 0) \times (8, 0) \rightarrow (8, 0)$ on c.

The following instantiation is incorrect:

```
node inst2(i: rate (16, 0); j: rate(4, 0); c: rate(8, 0)) returns (o)
let
  o=N(i, j, c);
tel
```

The clock calculus cannot unify clock $(4, 0)$ with clock $(16, 0) *. 8/.3 = (6, 0)$ (when computing the clock of the second argument of N).

Finally, the following instantiation is also incorrect:

```
node inst3(i: rate (7, 0); j; c) returns (o)
let
  o=N(i, j, c);
tel
```

Subsumption fails as $(7, 0)$ is not a subtype of $\mathcal{P}_{8,0}$.

As a last example, the clock inferred for the main node of the FAS program of Fig. 5.7 is:

$$((100, 0) \times (1000, 0) \times (10000, 0) \times (10000, 0)) \rightarrow$$
$$((100, 0) \times (1000, 0) \times (1000, 0) \times (1000, 1/2) \times (10000, 0))$$

6.3.6 Correctness

As we stated before, the aim of the clock calculus is to ensure that the semantics of a program is well-defined. To prove this property, we need to relate clock types to the semantics given in Sect. 4.6. To this intent, we define interpretation functions $\mathcal{I}(.)$ relating clock types and sets of flow values (each flow value is a sequence of pairs (v_i, t_i)). Let $s[i]$ denote the ith value of a sequence s (with $i \in \mathbb{N}$). The

interpretation $\mathcal{I}(t)$ of a clock type t is the set of values defined inductively as follows:

$$
\begin{aligned}
v \in \mathcal{I}((n,p)) &\Leftrightarrow \forall i,\ ck(v)[i] = p * n + i * n \\
v \in \mathcal{I}(cl_1 \rightarrow cl_2) &\Leftrightarrow \forall v_1 \in \mathcal{I}(cl_1),\ v(v_1) \in \mathcal{I}(cl_2) \\
(v_1, v_2) \in \mathcal{I}(cl_1 \times cl_2) &\Leftrightarrow v_1 \in \mathcal{I}(cl_1) \wedge v_2 \in \mathcal{I}(cl_2) \\
v \in \mathcal{I}(ck \texttt{ on } c) &\Leftrightarrow \forall v' \in \mathcal{I}(ck),\ \forall i,\ ck(v')[i] \in ck(v) \text{ iff } val(c)[i] = true \\
v \in \mathcal{I}(ck \texttt{ on not } c) &\Leftrightarrow \forall v' \in \mathcal{I}(ck),\ \forall i,\ ck(v')[i] \in ck(v) \text{ iff } val(c)[i] = false \\
v \in \mathcal{I}(c : ck) &\Leftrightarrow v \in \mathcal{I}(ck) \wedge v = c \\
v \in \mathcal{I}(pck *_{.} k) &\Leftrightarrow \forall v' \in \mathcal{I}(pck), ck(v) = \\
&\quad \bigcup_{i \in \mathbb{N}} \{ck(v')[i], ck(v')[i] + \pi(v')/k, ck(v')[i] + 2\pi(v')/k, \ldots, \\
&\qquad ck(v')[i] + (k-1)\pi(v')/k\} \\
v \in \mathcal{I}(pck /_{.} k) &\Leftrightarrow \forall v' \in \mathcal{I}(pck),\ ck(v) = \{ck(v')[i] | (k | ck(v')[i])\} \\
v \in \mathcal{I}(pck \rightarrow_{.} q) &\Leftrightarrow \forall v' \in \mathcal{I}(pck),\ \forall i,\ ck(v)[i] = ck(v')[i] + q * \pi(v') \\
v \in \mathcal{I}(\forall \alpha_1 <: C_1, \ldots, \alpha_m <: C_m.cl) &\Leftrightarrow \forall ck_1, \ldots, ck_m, (\mathcal{I}(ck_1) \subseteq \mathcal{I}(C_1) \wedge \ldots \wedge \mathcal{I}(ck_m) \subseteq \mathcal{I}(C_m)) \\
&\quad \Rightarrow v \in \mathcal{I}(cl[ck_1/\alpha_1, \ldots ck_m/\alpha_m]) \\
v \in \mathcal{I}(\mathcal{P}_{k,q}) &\Leftrightarrow ck(v) \in \mathcal{P}_{k,q}
\end{aligned}
$$

In other words, $v \in \mathcal{I}(t)$ is a formal definition of the statement "v is a value of type t". The correctness of the clock calculus is then stated in the following theorem. The theorem says that if the clock calculus computes a clock cl for an expression e in an environment H, then if we take a variable assignment ρ that maps for every variable x in H a value v of the right clock type (ie $v \in \mathcal{I}(H(x))$), then we obtain a value the clock type of which is cl (ie we have $S_\rho(e) \in \mathcal{I}(cl)$, with $S_\rho(e)$ defined in Sect. 4.11).

Theorem 2. *If $[x_1 : \sigma_1, \ldots, x_m : \sigma_m] \vdash e : cl$, then for all $v_1, \ldots, v_m$, such that $v_1 \in \mathcal{I}(\sigma_1)$, ..., $v_m \in \mathcal{I}(\sigma_m)$, we have $S_{[v_1/x_1, \ldots, v_m/x_m]}(e) \in \mathcal{I}(cl)$.*

Proof. By induction on the structure of expressions e. For each inference rule of Fig. 6.3, we prove that if the premises of the rule hold then the property holds for the conclusion.

First, proving the base of the induction amounts to proving that the property holds for the expressions of the language that are not defined inductively. There are two cases:

- Case e is cst: The rule has no premise. We have $cst : ck$, so for any assignment ρ (that does actually not concern cst), $S_\rho(cst) \in \mathcal{I}(ck)$;
- Case e is x: If $x \in dom(H)$ (premise of the inference rule *(VAR)*), that is to say there is some H' and cl such that $H = H', x : cl$, then $H \vdash x : cl$. Now, if we consider a valuation that maps a value v to x such that $v \in \mathcal{I}(cl)$, then obviously $S_{[v/x]}(x) \in \mathcal{I}(cl)$.

So we have proved the base of the induction. Now we can proceed to the induction step for the remaining inference rules. For each inference rule, we assume that the property holds for the premises of the rule (induction hypothesis) and prove that in this case the property holds for the conclusion of the rule.

- Case e is (e_1, e_2): We assume that $S_\rho(e_1) \in \mathcal{I}(cl_1)$ and $S_\rho(e_2) \in \mathcal{I}(cl_2)$. Then by definition of $\mathcal{I}(cl_1 \times cl_2)$, we have $S_\rho((e_1, e_2)) \in \mathcal{I}(cl_1 \times cl_2)$;

- Case e is cst `fby` e': We assume that $S_\rho(cst) \in \mathcal{I}(cl)$ and $S_\rho(e') \in \mathcal{I}(cl)$. Then from the semantics of operator `fby`, the expression cst `fby` e' produces values exactly at the same tags as cst and e', so $S_\rho(cst\ \texttt{fby}\ e') \in \mathcal{I}(cl)$;

- Case e is e' `when` c: We assume that $S_\rho(e') \in \mathcal{I}(ck)$ and $S_\rho(c) \in \mathcal{I}(c : ck)$. From the semantics of operator `when`, for all i, $ck(S_\rho(e'))[i] \in ck(S_\rho(e))$ if and only if $val(c)[i] = true$. So by definition of $\mathcal{I}(ck\ \texttt{on}\ c)$, we have$S_\rho(e) \in \mathcal{I}(ck\ \texttt{on}\ c)$;

- Case e is e' `whennot` c: same as previous case;

- Case e is `merge` (c, e_1, e_2): We assume that $S_\rho(c) \in \mathcal{I}(c : ck)$, $S_\rho(e_1) \in \mathcal{I}(ck\ \texttt{on}\ c)$ and $S_\rho(e_2) \in \mathcal{I}(ck\ \texttt{on not}\ c)$. Then from the semantics of operator `merge`, the expression `merge` (c, e_1, e_2) produces values exactly at the same tags as c, so $S_\rho(\texttt{merge}\ (c, e_1, e_2)) \in \mathcal{I}(ck)$;

- Case e is $e' *\hat{}\, k$: We assume that $S_\rho(e') \in \mathcal{I}(pck)$. The semantics of operator $*\hat{}$ tells us that for all i, we have $pck[i] \in ck(S_\rho(e))$, $pck[i] + \pi(pck)/k \in ck(S_\rho(e)), \ldots, pck[i] + \pi(pck)/k * (k-1) \in ck(S_\rho(e))$. Furthermore, the other premises of the inference rule ensure that $k|\pi(pck)$. So by definition of $\mathcal{I}(pck *_. k)$, we have $S_\rho(e) \in \mathcal{I}(pck *_. k)$;

- Case e is $e' /\hat{}\, k$: We assume that $S_\rho(e') \in \mathcal{I}(pck)$. The semantics of operator $/\hat{}$ tells us that for all i, we have $pck[i] \in ck(S_\rho(e))$ if and only if $k|pck[i]$. So by definition of $\mathcal{I}(pck/_. k)$, we have $S_\rho(e) \in \mathcal{I}(pck/_. k)$;

- Case e is $e' \sim> q$: We assume that $S_\rho(e') \in \mathcal{I}(pck)$. The semantics of operator $\sim>$ tells us that for all i, we have $pck[i] + q * \pi(pck) \in ck(S_\rho(e))$. Furthermore, the right premise of the rule ensures that $pck[i] + q * \pi(pck)$ is in $\mathbb{N}$. So by definition of $\mathcal{I}(pck \rightarrow_. q)$, we have $S_\rho(e) \in \mathcal{I}(pck \rightarrow_. q)$;

- Case e is `tail` (e'): We assume that $S_\rho(e') \in \mathcal{I}(pck)$. The semantics of operator `tail` tells us that for all i, we have $pck[i] + \pi(pck) \in ck(S_\rho(e))$. So by definition of $\mathcal{I}(pck \rightarrow_. q)$, we have $S_\rho(e) \in \mathcal{I}(pck \rightarrow_. 1)$;

- Case e is cst `::` e': We assume that $S_\rho(e') \in \mathcal{I}(pck)$. The semantics of operator `::` tells us that for all i, we have $pck[i] - \pi(pck) \in ck(S_\rho(e))$. Furthermore, the right premise of the rule ensures that $pck[i] - q * \pi(pck)$ is positive. So by definition of $\mathcal{I}(pck \rightarrow_. q)$, we have $S_\rho(e) \in \mathcal{I}(pck \rightarrow_. -1)$;

- Case e is $N(e)$: Let $\sigma = \forall\alpha <: C.cl$. Let $cl_1 \rightarrow cl_2 = cl[(cl' \in C)/\alpha]$. Let $v \in \mathcal{I}(\sigma)$. According to the definition of $\mathcal{I}$, $v \in \mathcal{I}(cl_1 \rightarrow cl_2)$. Now, if we assume that $S_\rho(N) \in \mathcal{I}(\sigma)$, then $S_\rho(N) \in \mathcal{I}(cl_1 \rightarrow cl_2)$. We also assume that $S_\rho(e) \in \mathcal{I}(cl_1)$. According to the definition of $\mathcal{I}(cl_1 \rightarrow cl_2)$, for any $v \in \mathcal{I}(cl_1)$, $S_\rho(N(v)) \in \mathcal{I}(cl_2)$, so $S_\rho(N(S_\rho(e))) = S_\rho(N(e)) \in \mathcal{I}(cl_2)$;

- Case $x = e$: We assume that $S_\rho(x) \in \mathcal{I}(cl)$ and $S_\rho(e) \in \mathcal{I}(cl)$. Then the semantics of the equation is well-defined;

- Case $eq_1; eq_2$: Well-defined provided eq_1 and eq_2 are well-defined;

- Case `node` $N(in)$ `returns` (out) [`var` var;] `let` eqs `tel` : The induction hypothesis applied to the premises of the inference rule *(NODE)* says that if $S_\rho(in) \in \mathcal{I}(cl_1)$, $S_\rho(out) \in \mathcal{I}(cl_2)$ and $S_\rho(var) \in \mathcal{I}(cl_3)$, then the semantics of the equations eqs is well-defined. So for all $v \in \mathcal{I}(cl_1)$, $S_{[v/in]}(N(in)) \in \mathcal{I}(cl_2)$. Thus:
  $S_\rho($ `node` $N(in)$ `returns` (out) [`var` var;] `let` eqs `tel` $) \in \mathcal{I}(cl_1 \rightarrow cl_2)$;

- Case `imported node` $N(in)$ `returns` (out) `wcet` n: True by definition;

- Last, for the subsumption rule. We assume that $S_\rho(e) \in \mathcal{I}(ck)$. We also assume that $ck <: C$, so $\mathcal{I}(ck) \subseteq \mathcal{I}(C)$. As a consequence, we have $S_\rho(e) \in \mathcal{I}(C)$.

□

6.4 About Flows Initialisation

In LUSTRE, the use of the **pre** operator produces flows the values of which are undefined at the first instant. This requires an initialisation analysis [CP04] to verify that the program does not access to uninitialised flows. Indeed, the program would access to undefined values and its behaviour would not be deterministic. For instance the program consisting of the following simple node is rejected as its output is undefined at the first instant:

```
node non_init(i: int) returns (o: int)
let o=i+pre(i); tel
```

In our language, delays are handled using operator `fby` and are always initialised, so we do not need an initialisation analysis. Flows can be deactivated during initialisation steps using operator `tail` or $\sim>$ but this directly appears in the clock, which defines when the flow begins to be active. Thus, the clock calculus directly ensures that two flows can only be combined if they start being active at the same dates and we do not need an initialisation analysis.

6.5 Summary

In this chapter, we have defined:

- The typing of the language;
- The clock calculus of the language.

The causality analysis of the language is standard and was not detailed here. We have also shown that no initialisation analysis is required in our language because the properties it usually verifies are already verified by our clock calculus.

When all the static analyses succeed, the semantics of the program is well-defined and the compiler can generate the code corresponding to the program. This will be detailed in the next part of this dissertation. Before that, the next chapter concludes on the language definition.

Chapter 7

Conclusion on language definition

We proposed a language for programming embedded control systems with multiple real-time constraints. The language is defined as a real-time software architecture language with synchronous semantics. It defines real-time primitives based on a specific class of clocks called strictly periodic clocks, which are triggered at regular intervals of time (the period of the clock). Periodic clock transformations allow to produce a faster or a slower strictly periodic clock from an existing strictly periodic clock or to produce a strictly periodic clock with a different phase (ie to shift the initial activation date of a clock). Periodicity constraints can then be imposed in a program by specifying that the clock of a flow is strictly periodic. We defined rate transition operators based on periodic clock transformations, which can be used to combine flows of different rates. The programmer can implement various multi-rate communication patterns using rate transition operators and the semantics of the language ensures that the patterns are deterministic.

For the most part, strictly periodic clocks and their associated primitives could be programmed using Boolean clocks, as we did for the FAS in Fig. 1.6, using counter clocks. From this point of view, strictly periodic clocks and the associated clock transformations are a subclass of the usual Boolean clocks of synchronous languages. However, in the general case, the set of activations dates of a flow the clock of which is defined by Boolean activation conditions cannot be evaluated statically. The activation conditions can for instance depend on the value of an input of the program, which is known only at the execution of the program. On the contrary, strictly periodic clocks are only defined by their phase and by their period, thus we can easily determine the activation dates of a flow the clock of which is strictly periodic. The first consequence is that the problem of testing the synchronism of two flows is completely decidable and can be solved efficiently. The second consequence, as we will see in the next part of this dissertation, is that restricting to the specific class of strictly periodic clocks allows to compile the corresponding class of systems more efficiently, by reusing existing results of the scheduling theory of multi-periodic task sets.

Strictly periodic clocks can also be considered as a particular case of the N-synchronous approach [CDE$^+$06, CMPP08] or of the affine clock relations [SLG97]. On the contrary to the case of Boolean clocks, these approaches allow to evaluate statically the activation dates of a flow. The systems considered by these approaches remain more general than multi-periodic systems, so, again, by considering a smaller class of systems we are able to compile programs more efficiently. The same argument holds for more general GALS approaches [BS01, LGTLL03].

Rate transition primitives improve the previous real-time extensions of LUSTRE [Cur05] by providing more powerful operators. Compared to [ACGR09] however, our language does not support mode automata. It would be interesting to include them in our language.

Compared to other ADLs, we have seen that our language allows to specify more complex communication patterns.

Part II

Compiling into a Set of Real-Time Tasks

Chapter 8

Overview of the Compilation Process

In this part of the dissertation, we detail the compilation of the language, which translates a program into lower-level code. This code is then translated into executable code by an existing compiler. The complete compilation process is defined formally, in order to ensure that the generated code behaves according to the formal semantics of the language.

8.1 Limitation of the classic "single-loop" code generation

As for other synchronous languages, the compiler translates the input program into intermediate level code (C code). However, the translation differs from the classic "single-loop" sequential code generation, because we deal with multi-periodic systems. For instance, consider the example below:

```
imported node A(i: int) returns (o: int) wcet 1;
imported node B(i: int) returns (o: int) wcet 5;

node multi(i: rate(3, 0)) returns (o)
var v;
let
  v=A(i);
  o=B(v/^3);
tel
```

As shown in Fig. 8.1, a sequential execution of the program cannot respect all the deadline constraints of the program, the second repetition of operation `A` will miss its deadline (at date 6). This example cannot be implemented using sequential code generation, we cannot find a sequence of execution for operations A and B that will respect the periodicity constraints of the two operations.

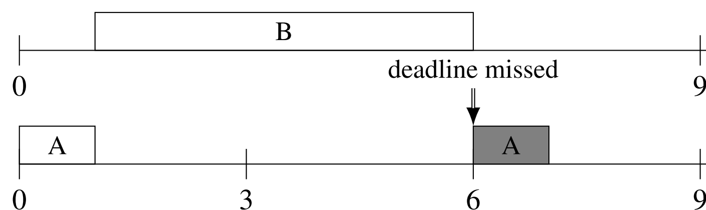


Figure 8.1: Sequential execution of node `multi`

8.2 Multi-task code generation

However, if we implement operations A and B as two separate tasks and execute them concurrently with a real-time operating system with multi-tasking capabilities (ie the RTOS is preemptive), we can find a schedule for the execution of the tasks that respects the periodicity constraints of the two operations.

Each operation (each imported node call) is translated into a task τ_i, which has a set of real-time attributes (T_i, C_i, r_i, d_i). τ_i is instantiated periodically with period T_i, $\tau_i[j]$ denotes the jth iteration of task τ_i. A task instance cannot start its execution before all its predecessors, defined by the precedence constraints due to data-dependencies between tasks, complete their execution. C_i is the (worst case) execution time of the task. r_i is the release date of the first instance of the task. The subsequent release dates are $r_i + T_i$, $r_i + 2T_i$, etc. d_i is the relative deadline of the task. The absolute deadline $D_i[j]$ of the instance j of a task τ_i is the release date $R_i[j]$ of this instance plus the relative deadline of the task: $D_i[j] = R_i[j] + d_i$. These definitions are illustrated in Fig. 8.2.

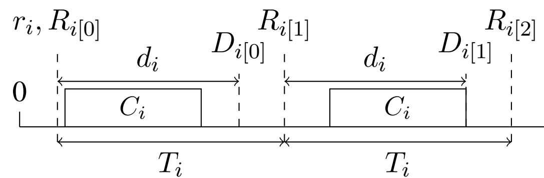


Figure 8.2: Real-time characteristics of task τ_i

Our example can be translated into two tasks τ_A of attributes $(3, 1, 0, 3)$ and τ_B of attributes $(9, 5, 0, 9)$, with a precedence from τ_A to τ_B. Using preemption, it is then possible to find a schedule that respects the deadline constraints of both tasks, as shown in Fig. 8.3. Marks on the time axis of B represent task preemptions/restorations: at date 3, B is suspended to execute the second repetition of A. After A completes, B is restored (at date 4) and resumes its execution where it stopped.

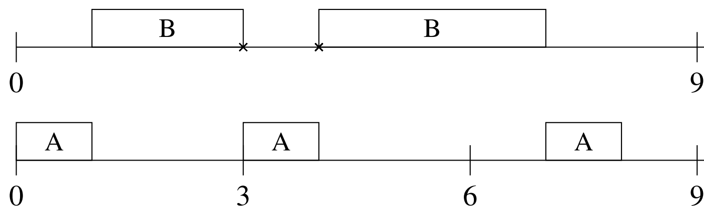


Figure 8.3: Preemptive scheduling of node `multi`

8.3 Compilation chain

The global structure of the compilation chain is described in Fig. 8.4. The Chapter 9 first details the translation of the program into a set of tasks related by precedence constraints called a *dependent task set*, where precedences are due to data-dependencies between tasks. The Chapter 10 describes the translation of this dependent task set into an equivalent *independent task set*, schedulable by an EDF scheduler. First, precedence constraints are encoded by adjusting the task real-time attributes of the tasks (by extending the work of [CSB90] to the case of multi-periodic dependent tasks). Then, a communication protocol is proposed to ensure the conservation of the synchronous semantics of the task set. Indeed, let us consider our previous example: data produced by the task instance $\tau_A[0]$ must be consumed by $\tau_B[0]$. Due to

preemptions, $\tau_A[1]$ executes before $\tau_B[0]$ completes its execution. To respect the synchronous semantics, data produced by $\tau_A[0]$ must remain available for $\tau_B[0]$ during its complete execution, thus after the execution of $\tau_A[1]$ has started. As a consequence, we need to define a communication protocol that makes sure that data produced by $\tau_A[0]$ is not overwritten by $\tau_A[1]$. The Chapter 11 then details the translation of the independent task set into a C program consisting of a set of communicating threads. Finally, Chapter 12 presents the implementation of a prototype of the compiler that generates code executable on the MARTE RTOS.

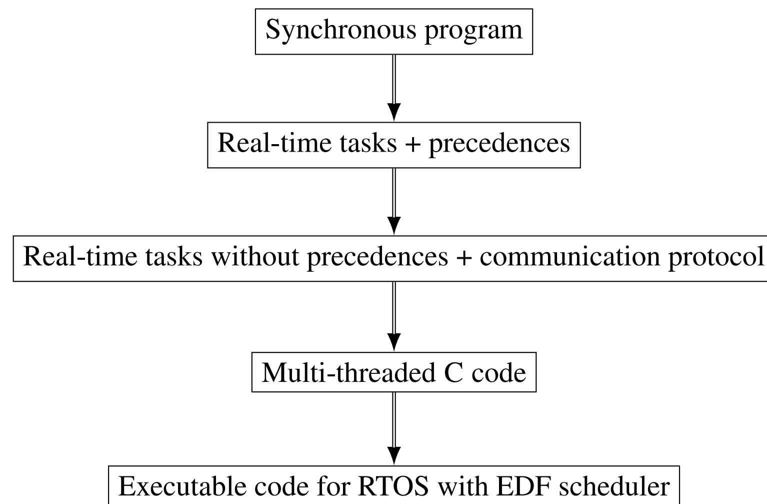


Figure 8.4: Compilation chain

Chapter 9

Extracting Dependent Real-Time Tasks

Once static analyses succeed, the compiler can translate the program into lower-level code. As we mentioned in Chap. 8, the program is translated into a set of communicating tasks, which will be scheduled concurrently by a multi-task RTOS. This chapter details the first step of this translation, the extraction of the task graph corresponding to a program.

9.1 Program Expansion

A program consists of a hierarchy of nodes, the leaves of which are predefined operators or imported nodes. The task graph generation process first inlines intermediate nodes appearing in the main node recursively, replacing each intermediate node call by its set of equations. For instance, the following node:

```
imported node A(i: int) returns (o: int) wcet 5;
imported node B(i: int) returns (o: int) wcet 10;

node sub (i, j) returns (o, p)
let
  o=A(i);
  p=B(j);
tel

node toexpand (i, j: rate (20, 0)) returns (o, p)
let
  (o, p)=sub(i, j);
tel
```

Is expanded into:

```
imported node A(i: int) returns (o: int) wcet 5;
imported node B(i: int) returns (o: int) wcet 10;

node expanded (i, j: rate (20, 0)) returns (o, p)
  var isub, jsub, osub, psub;
let
  isub=i; jsub=j;
  osub=A(isub);
```

```
  psub=B(jsub);
  (o, p)=(osub, psub);
tel
```

The algorithm performing the expansion of the content of a node is described in Alg. 7. This algorithm is applied recursively, starting from the main node of the program, to completely expand the program. The result of this expansion is a single node (the main node), in which node calls correspond to either predefined nodes or imported nodes (the leaves of the nodes hierarchy).

Algorithm 7 Expanding the content of a node N

1: **for** each node instance $M(args)$ in the body of N **do**
2:     **for** each argument arg of $args$ **do**
3:         add a local variable v to N
4:         add an equation $v = arg$ to N
5:     **end for**
6:     **for** each local or output variable of the node definition of M **do**
7:         add a local variable to N
8:     **end for**
9:     **for** each equation of the node definition M **do**
10:         add the equation to set of equations of N
11:     **end for**
12:     replace the node instance in N by a tuple made up of its outputs
13: **end for**

9.2 Task Graph

The expanded program is first translated into a *task graph*. A task graph consists of a set of tasks (the vertices of the graph) and a set of precedences between the tasks (the edges of the graph).

9.2.1 Definition

Let $g = (V, E)$ denote a task graph, where V is the set of tasks of the graph and E is the set of task precedences of the graph (a subset of $V \times V$). Precedences are annotated with the predefined operators of the language, which describe the communication pattern used for communicating from the source to the destination of the precedence. The union of two graphs $g = (V, E)$ and $g' = (V', E')$ is defined as $g \cup g' = (V \cup V', E \cup E')$. $last(g)$ denotes the set of tasks of g that have no successors, that is to say $last(V, E) = \{v \in V | \neg(\exists v' \in V, v \rightarrow v')\}$.

Fig. 9.1 gives a task graph example. Boxes represent tasks and edges represent precedences. Task `A` communicates with task `B` using the rate transition operator `/^3`, which means that only the first out of three successive instances of `A` send data to `B` and implies that `A` has a period three times shorter than that of `B`. The other precedences have no annotations, which means that they denote simple precedences that relate tasks of the same period.

9.2.2 Tasks

Each task τ_i has a set of functional attributes (in_i, out_i, f_i). Tasks are instantiated periodically and when a task instance executes, it applies its function f_i to its input variables in_i to compute its output variables

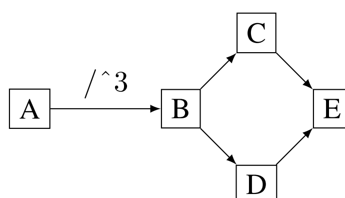


Figure 9.1: A task graph

out_i. The function f_i is an external function that executes sequentially and contains no synchronization point.

Extracting tasks from the expanded program is pretty straightforward: each imported node call appearing in the expanded main node is translated into a task. The inputs (in_i) and outputs (out_i) of the task are the inputs and outputs of the node call. The function of the task (f_i) is the function provided by the user to implement the imported node. Each variable of the expanded main node and predefined operator call appearing in it is also translated into a vertex but will later be reduced to simplify the graph (see Sect. 9.2.5).

9.2.3 Precedences

When a variable v is such that $v \in out_i$ and $v \in in_j$, there is a data-dependency from τ_i to τ_j. To respect the synchronous semantics, we must add a precedence constraint from τ_i to τ_j in the task graph, to force τ_i to execute before τ_j. A precedence from a task τ_i to a task τ_j will be denoted $\tau_i \to \tau_j$. Task precedences are deduced from data dependencies between expressions of the program.

We extend the notion of *syntactic* dependency of [HRR91] to our language. We write $e \prec e'$ when e' syntactically depends on e. For instance, the expression $N(e, e')$ syntactically depends on e and e'. The expression e `when` c syntactically depends on e and c, even though when c is false the expression does not *semantically* depend on e.

The syntactic dependency relation is overloaded for equations. The corresponding definition is given in Fig. 9.2. In the rule on equations, the x on the left denotes the definition of variable x while the x on the right denotes the expression x (as for instance the sub-expression of expression $x + 1$). Remember that calls to intermediate nodes are inlined before computing the task graph, therefore node applications appearing in the definition of the dependency relation are only applications of imported nodes. Notice that cst `fby` e syntactically depends on e. This implies that we allow syntactic dependency loops (though the causality analysis checks that a program does not contain semantic dependency loops). Such dependencies will only be discarded later in the compilation process (see in Sect. 10.2). Task precedences are deduced from this definition as described in the next section.

9.2.4 Intermediate graph

We first extract an intermediate graph structure from the program (ie not directly a task graph), where vertices can either be tasks or variables. The function $\mathcal{G}(e)$, which returns the intermediate graph corresponding to a program, is defined in Fig. 9.3.

For instance, let us consider the following program, previously presented in Sect. 5:

```
node sampling(i: rate (10, 0)) returns (o)
  var vf, vs;
let
  (o, vf)=F(i, (0 fby vs)*^3);
```

$$\begin{array}{lcl}
e & \prec & (e, e') \\
e' & \prec & (e, e') \\
e & \prec & (cst \texttt{ fby } e) \\
e, c & \prec & e \texttt{ when } c \\
e, c & \prec & e \texttt{ whennot } c \\
c, e_1, e_2 & \prec & \texttt{merge } (c, e_1, e_2) \\
e & \prec & (e\ op\ k) \text{ (with } op \in \{/\hat{}, *\hat{}, \sim>\} \\
e & \prec & (x = e) \\
(x = e) & \prec & x \\
e & \prec & (N(e))
\end{array}$$

Figure 9.2: Expression dependencies

$$\begin{array}{lcl}
\mathcal{G}(cst) & = & (\emptyset, \emptyset) \\
\mathcal{G}(x) & = & (\{x\}, \emptyset) \\
\mathcal{G}(N(e)) & = & \mathcal{G}(e) \cup (\{N\}, \{l \rightarrow N, \forall l \in last(\mathcal{G}(e)) \\
\mathcal{G}(op(e)) & = & \mathcal{G}(e) \cup (\{op\}, \{l \rightarrow op, \forall l \in last(\mathcal{G}(e)) \\
 & & \text{(with } op \in \{*\hat{}k,\ /\hat{}k,\ \sim> q, \texttt{tail}, cst\ \texttt{::}, cst \texttt{ fby}\} \\
\mathcal{G}(e \texttt{ when } c) & = & \mathcal{G}(e) \cup (\{c, \texttt{ when }\}, \{c \rightarrow \texttt{when}, l \rightarrow \texttt{when}, \forall l \in last(\mathcal{G}(e))\} \\
\mathcal{G}(e \texttt{ whennot } c) & = & \mathcal{G}(e) \cup \\
 & & (\{c, \texttt{ whennot }\}, \{c \rightarrow \texttt{whennot}, l \rightarrow \texttt{whennot}, \forall l \in last(\mathcal{G}(e))\} \\
\mathcal{G}(\texttt{merge } (c, e_1, e_2)) & = & \mathcal{G}(e_1) \cup \mathcal{G}(e_2) \cup \\
 & & (\{c, \texttt{merge}\}, c \rightarrow \texttt{merge}, l \rightarrow \texttt{merge}, \forall l \in last(\mathcal{G}(e_1) \cup \mathcal{G}(e_2)) \\
\mathcal{G}(x = e) & = & \mathcal{G}(e) \cup (\{x\}, \{l \rightarrow x, \forall l \in last(\mathcal{G}(e)\}) \\
\mathcal{G}(eq; eq') & = & \mathcal{G}(eq) \cup \mathcal{G}(eq')
\end{array}$$

Figure 9.3: Extraction of the intermediate task graph

```
  vs=S(vf/^3);
tel
```

Computing $\mathcal{G}$ for this node amounts to computing the union of $\mathcal{G}((o, vf) = F(i, (0 \texttt{ fby } vs) *\hat{}3);)$ and $\mathcal{G}(vs = S(vf/\hat{}3);)$. We have:

$$\mathcal{G}(vs = S(vf/\hat{}3);) = \mathcal{G}(S(vf/\hat{}3) \cup (\{vs\}, \{l \rightarrow vs, \forall l \in last(\mathcal{G}(S(vf/\hat{}3)))\})$$

With:

$$\mathcal{G}(S(vf/\hat{}3) = \mathcal{G}(vf/\hat{}3) \cup (\{S\}, \{l \rightarrow S, \forall l \in last(\mathcal{G}(vf/\hat{}3)))$$

And:

$$\mathcal{G}(vf/\hat{}3) = (\{vf\}, \{vf \rightarrow /\hat{}3\})$$

So:

$$\mathcal{G}(vs = S(vf/\hat{}3);) = (\{vs, vf, S\}, \{vf \rightarrow /\hat{}3, /\hat{}3 \rightarrow S, S \rightarrow vs\})$$

The precedence graph of the other equation is computed similarly:

$$\mathcal{G}((o,vf)=F(i,(0\ \texttt{fby}\ vs)*\hat{}3);) = (\{o,vf,F,i,\ \texttt{fby}\ ,vs,*\hat{}3\},\\ \{vs\rightarrow\ \texttt{fby}\ ,\ \texttt{fby}\ \rightarrow *\hat{}3, *\hat{}3\rightarrow F, i\rightarrow F, F\rightarrow o, F\rightarrow vf\})$$

To avoid cumbersome notations, we consider that precedences directly relate tasks, while they actually relate one output of a task to one input of another task. For instance, we simply write $i \rightarrow F$ and $*\hat{}3 \rightarrow F$, while In practice i precedes the first input of F and $*\hat{}3$ precedes the second input of F. We do not make this distinction in the rest of the dissertation as it has only minor impacts. It was however taken into account in the implementation of our compiler prototype.

Finally, the intermediate graph of the complete program is given in Fig. 9.4. Notice that even though the precedences form a loop, the computation of the graph terminates as we simply take the union of the graphs of the two equations.

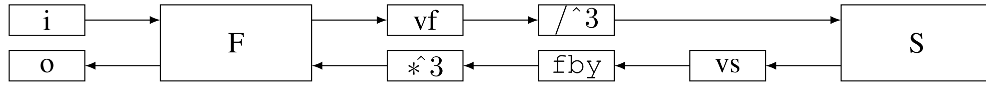


Figure 9.4: An intermediate graph

9.2.5 Graph reduction

Variables

We apply a series of simplifications to the intermediate graph structure. First, each input of the main node is translated into a (sensor) task and each output of the main node is translated into a (actuator) task. The inputs and outputs of the program are implemented as tasks so that they can directly be activated by the generated code.

Second, we replace recursively each pair of precedence $N \rightarrow x \rightarrow M$, where x is a local variable (not an input or output), by a single precedence $N \rightarrow M$. When all such pairs have been replaced, we remove variables from the graph. On our previous example, we obtain the graph of Fig. 9.5.

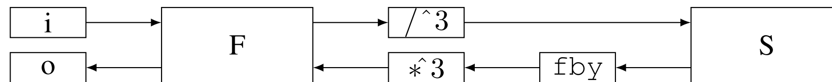


Figure 9.5: Graph after variable reduction

Predefined operators and extended precedences

We define a graph transformation system that removes predefined nodes from the graph. It translates predefined nodes into *precedence annotations* (labels on the graph edges). A precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$ represents an *extended precedence*, where the precedence annotation *ops* is a list of operators *op*, with $op \in \{/\hat{}k, *\hat{}k, \sim> q, \texttt{tail}, cst\ \texttt{::}, cst\ \texttt{fby}, \texttt{when}\ c, \texttt{whennot}\ c\}$. The annotation specifies that the precedence relates tasks of different rates or phases ($*\hat{}k, /\hat{}k, \sim> q$), or that the precedence is delayed ($cst\ \texttt{fby}$) or conditioned ($\texttt{when}\ c$). In the following, *op.ops* denotes the list whose head is *op* and whose tail is *ops* and ϵ denotes the empty list.

A survey of graph transformation systems and associated concepts can be found in [AEH+99]. A graph transformation system consists of a set of rules. In our context, a graph transformation rule $r =$

(L, R, K) simply consists of a *left-hand side* graph L, a *right-hand side* graph R and an *interface* graph K (actually only a set of vertices) occurring both in L and R. The application of rule r to a graph G consists of:

1. Finding a sub-graph S of G corresponding to graph L.

2. Removing S from G except for the interface K. This produces a context graph D.

3. Constructing the disjoint union of D and R.

This is illustrated in Fig. 9.6. The transformation rule to apply is given in Fig. 9.6(a). The vertices with thick borders are the interface K of the rule. In Fig. 9.6(b), the graph in the dashed rectangle is an occurrence of the right-hand side of the rule. This occurrence is then replaced by the right-hand side of the rule and we obtain the graph of Fig. 9.6(c).

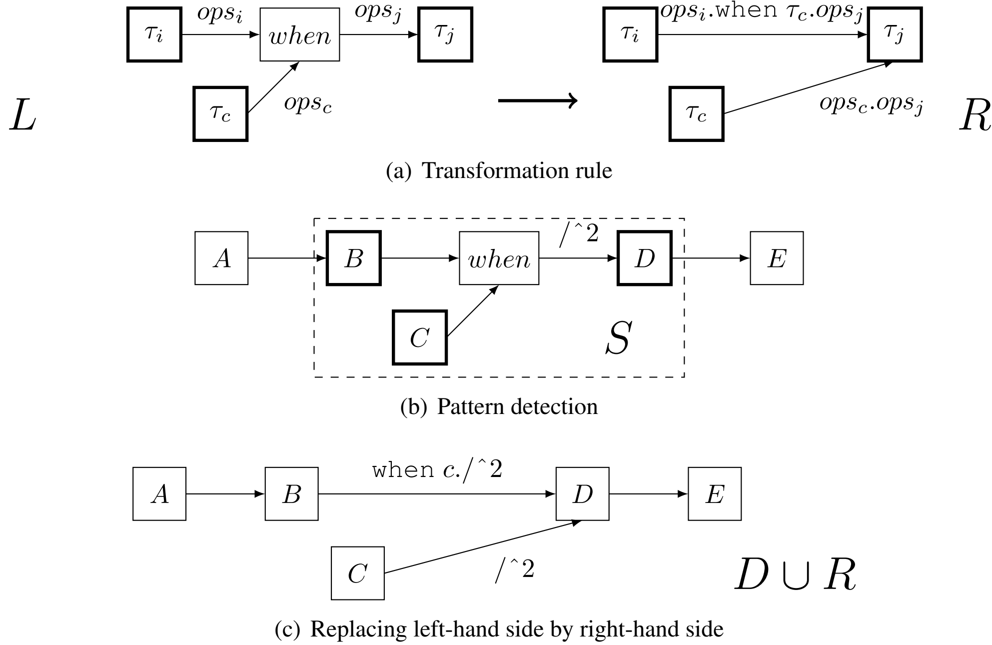
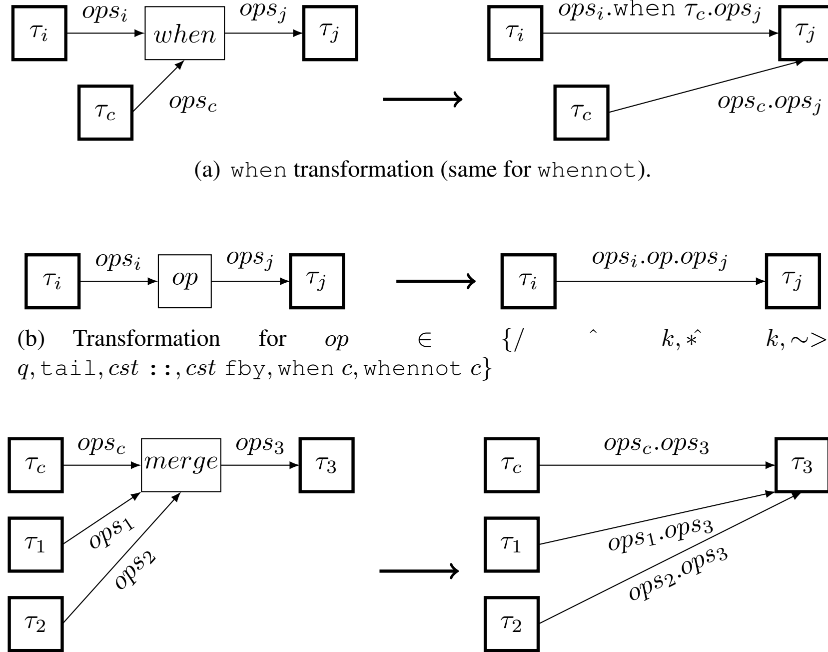


Figure 9.6: Application of a graph transformation rule

The rules of the graph transformation system $\mathcal{R}_\mathcal{G}$ that translates predefined operators to precedence annotations are defined in Fig. 9.7. Notice that the transformation of the **when** operator requires to add the condition of the **when** as an input of the task τ_j. We will indeed need the value of this condition at each repetition of the task to test if the task τ_j needs to be executed for the current repetition or not (see Sect. 9.4). This is handled transparently by the compiler and does not require the programmer to modify the code provided for the corresponding imported node.

- When a precedence relates τ_i to τ_j through operator `when` c, `when` c is transformed into an annotation on the precedence from τ_i to τ_j. A precedence is added from c to τ_j as τ_j will need to test if c is true before consuming the data produced by τ_i. The transformation rule for operator `whennot` c is similar;

- Rate transition and phase offset operators are similarly transformed into precedence annotations;



(a) `when` transformation (same for `whennot`).

(b) Transformation for $op \in \{/\hat{}\ k, *\hat{}\ k, \sim> q,$ `tail`$, cst$ `::`$, cst$ `fby`$,$ `when` $c,$ `whennot` $c\}$

Figure 9.7: The graph transformation system

- The `merge` is simply removed from the graph and its predecessors become predecessors of its successor. Even though the successor will as a result have several predecessors for the same input, the clock of those inputs tell us when to use one or the other.

Property 9.
- *The rewriting system $\mathcal{R}_\mathcal{G}$ is convergent;*
- *When the rewriting terminates, every vertex of the graph corresponds to either an imported node or a sensor or an actuator.*

Proof. The proof relies on the same principles as for term rewriting systems (see Sect. 6.3.3 and [BN98]). The termination is straightforward to prove: as every rule reduces the number of vertices of the graph, the rewriting system terminates.

The confluence is simple too. As the system terminates, we need to prove the local confluence of its critical pairs. The system does not contain critical pairs, so it is locally confluent and thus confluent. □

The reduced task graph of the previous example is given in Fig. 9.8.

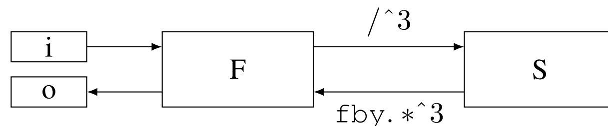


Figure 9.8: A reduced task graph

As the causality analysis rejects programs that contain cycles without delays, if we ignore delayed precedences the task graph corresponding to a program is a Directed Acyclic Graph (DAG).

In addition, we reject programs which contain a precedence $\tau_i \stackrel{ops}{\rightarrow} \tau_j$ where *ops* contains an operator $*\hat{}\ k$ that precedes the first operator `fby` appearing in *ops*. For instance, $B(0$ `fby` $(A(i) *\hat{}\ 2))$, which

gives $A \overset{*\hat{}2.\texttt{fby}}{\rightarrow} B$ should be replaced by $B((0 \texttt{ fby } A(i)) *\hat{}\, 2)$, which gives $A \overset{\texttt{fby}.*\hat{}2}{\rightarrow} B$ (though the semantics is not exactly the same). This constraint does not seem too restrictive and we will see that it allows important simplifications in the precedence encoding algorithm (Sect. 10.2). This verification is actually performed ealier in the compilation process, during the static analyses of the program. It is only detailled now for better readability.

9.2.6 Task clustering

The task extraction process basically translates each imported node call into a separate task. For complex examples, this might produce many tasks, which can be problematic for two reasons. First, the number of tasks supported by a RTOS is generally bounded and does not exceed a few hundreds at best. Second, scheduling a large set of tasks can result in an important run-time overhead, due to the time required to compute scheduling decisions.

Therefore, it would be interesting to group several node calls in the same task, which is referred to as *task clustering*. Task clustering is however not trivial. Reducing the number of tasks of the system indeed reduces the time required to compute scheduling decisions, but it may in return increase the number of preemptions, as longer tasks are more likely to require preemption, and also increase the cost of each preemption, as the size of the context of a task also increases. Clustering can also cause a system to become unschedulable. For instance, data produced by clustered tasks is only available at the end of the execution of the task cluster, while it would have been available earlier if the tasks had not been clustered.

Task clustering is not studied in the present dissertation, though it is clearly an important subject for future work. A first bibliographical study seems to suggest that this topic has not been studied much until now. Some interesting strategies have been studied in [Cur05] and guidelines have been proposed in [ACGR09]. We have not considered their integration in our work yet.

9.3 Real-time characteristics

For each task τ_i, we must extract its real-time characteristics (T_i, C_i, r_i, d_i) (see Chap. 8 for definitions). The clock calculus ensures that all the inputs and outputs of an imported node have the same strictly periodic clock parent pck_i. By extension, we will call pck_i the strictly periodic clock parent of τ_i. The real-time characteristics of a task can be extracted from a program as follows:

Periods The period of a task is obtained from its strictly periodic clock parent. We have $T_i = \pi(pck_i)$.

Deadlines By default, the deadline of a task is its period ($d_i = T_i$). Deadline constraints can also be specified on sensors or actuators (eg `o: due n`), then the deadline of the task is the minimum of the deadlines of its outputs.

Release dates The initial release date of a task is the phase of its strictly periodic clock parent: $r_i = \varphi(pck_i)$.

Execution times The execution time C_i of a task is specified by the wcet of the corresponding imported node, sensor or actuator declaration.

9.4 Activation conditions

If the clocks of the inputs/outputs of a task contain Boolean clock transformations, then the task may have an activation condition. At each task instantiation, the program must test the activation condition. If this condition is false, then the task instance completes immediately, otherwise it behaves normally.

Definition 10. *The condition activation $cond(ck)$ of a clock ck is defined as follows:*

$$cond(ck\ \texttt{on}\ c) = cond(ck)\ \texttt{and}\ c$$
$$cond(pck) = \texttt{true}$$

The activation condition of a variable x is the activation condition of its clock, $cond(x) = cond(ck(x))$. We now define the activation condition of a task.

Definition 11. *The activation condition $cond_i$ of a task τ_i is defined as:*

$$cond_i = \bigvee_{x \in in_i} cond(x)$$

In this definition, the input variables in_i correspond to the arguments provided for the corresponding imported node instance, not to the inputs specified in the definition of the imported node. Consider the following example:

```
imported node N(i, j: int) returns (o: int) wcet 10;

node inst(c, i, j) returns (o)
let
  o=N(i when c, j when c);
tel
```

Though the declaration of node `N` contains no activation condition, the activation condition of its node instance in node `inst` is $cond_N = c$, so it completes immediately when c is false.

Let us take another example:

```
imported node N(c: bool; i:  int when c; j: int whennot c)
returns (o: int) wcet 10;

node inst(c, i, j) returns (o)
let
  o=N(c, i when c, j whennot c);
tel
```

The activation condition of the node instance of `N` in `inst` is: $true \vee c \vee \neg c$. Thus, this node instance of `N` is executed normally at each task instantiation.

9.5 Summary

In this chapter we have detailed the extraction of a set of dependent real-time tasks from a program. The tasks are dependent because they are related by precedence constraints due to inter-task data-dependencies.

In the next chapter, we propose to transform the dependent task set into an equivalent independent task set.

Chapter 10

From Dependent to Independent Tasks

In the previous chapter, we detailed the translation of a program into a set of real-time tasks related by precedence constraints, represented by a task graph. Tasks are inter-dependent, due to precedence constraints, which makes the compilation problem more complex than in the case of independent tasks. First, we need a scheduling policy for periodic tasks with precedence constraints: a task instance cannot be scheduled until all its predecessors complete their execution. Second, we need to ensure data consistency for inter-task communications: when a task τ_j consumes data produced by a task τ_i, we need to ensure that the data produced by τ_i is available when τ_j starts and remains available until τ_j completes its execution.

10.1 Scheduling Dependent Tasks

We need a scheduling policy for a set of tasks with the following constraints:

1. Tasks are instantiated periodically.
2. Tasks have a relative deadline lower than their period.
3. Tasks can have an initial release date greater than 0.
4. Tasks are related by precedence constraints.

In the rest of this section, we review the possible scheduling policies for this problem.

10.1.1 With/without preemption

With non-preemptive scheduling, once the scheduler starts the execution of a task, the task executes until its completion. With preemptive scheduling, the execution of the currently running task can be suspended to execute another task and resumed later.

The scheduling problem we consider is NP-hard when preemption is not allowed [SSDNB95]. Allowing preemption yields polynomial complexity and enables the scheduling of a larger class of problems (see Fig. 8.3 for instance). Thus we choose a preemptive scheduling policy.

This choice does however have a noticeable drawback: each preemption has a cost in terms of execution time due to context switching. Preempting a task requires to save the context of the running task, to replace it by the context of the preempting task and to later restore the context of the preempted task when it resumes its execution. Usually, this overhead is either neglected or task wcets are over-approximated to include this overhead. We assume the latter. A possible extension would take advantage of the work of [MYS07], which allows to compute the exact cost of task preemptions.

10.1.2 Dynamic/static priority

Periodic tasks scheduling can be solved efficiently by using a priority-based scheduling approach. Each task is affected a priority and scheduling decisions are taken based on this priority. Task priorities can either be static or dynamic. By static priority, we mean that the priority of a task remains the same for the complete program execution. By dynamic priorities, we mean that the priority of a task can change during execution. Typically, with dynamic priorities, different task instances of the same task can have different priorities.

The fundamental work of [LL73] proposed the *rate-monotonic* (RM) static priority policy, where tasks with a shorter period are affected a higher priority and the *earliest-deadline-first* (EDF) dynamic priority policy, where task instances with a shorter absolute deadline are affected a higher priority. RM is optimal[1] in the class of static priority policies for scheduling a set of periodic tasks with deadlines equal to the period and release dates equal to 0. It can be extended to the *deadline-monotonic* policy (DM) [LW82], to schedule optimally a set of tasks with relative deadlines lower than the task period. [Aud91] defines an optimal algorithm based on DM for the case with release dates different from 0. EDF is optimal in the class of dynamic priority policies for scheduling a set of periodic tasks with relative deadlines lower than the task period and release dates different from 0.

RM tends to be favored instead of EDF by the developers of real-time applications. However, as emphasized in [But05], most of the claims that justify this choice are either false or do not hold in the general case. Though the author mentions DM only marginally, it seems that most remarks on RM also stand for DM. The results of the comparison between RM and EDF are the following:

- RM is easier to implement than EDF. With RM, task priorities can be computed before run-time, while with EDF they must be computed by the scheduler at run-time, each time a task is released;
- EDF does not have a higher run-time overhead than RM. EDF requires to compute task priorities at run-time, which introduces a run-time overhead that does not exist with RM. However, when context switches are taken into account, EDF introduces *less overhead* than RM because the number of preemptions that occur with EDF is usually much lower than with RM;
- EDF does not lead to higher task jitter than RM on average (the task jitter is the variation between the response times of different instances of the task, the response time being the time elapsed between task instance release and task instance completion). RM reduces the jitter of high priority tasks but increases that of low priority tasks, while EDF treats tasks more equally. As a result, when the utilization factor of the processor is high (ie when the processor has few idle times), the *average* task jitter of the task set is lower with EDF than with RM;
- RM is not more robust than EDF in the case of overloads (when the total execution time demand of tasks exceeds the processor capacity). This criterion is not of much interest in our context as the criticity of the systems requires that overloads never occur ;
- The class of complexity of the schedulability analysis is the same for EDF and RM, though in average EDF requires a little more execution steps;
- EDF enables better processor utilization than RM, thus allowing more complex computations to be performed.

In conclusion, the choice between DM and EDF should actually be made as follows:

[1]Here, optimality means that if there exists a priority affectation that schedules the task set correctly, then the algorithm finds one.

- If the programmer needs a policy easier to implement, he should choose DM;
- If the programmer needs better resource exploitation, he should choose EDF.

In this dissertation, we will only consider EDF scheduling, to enable better resource exploitation. We are currently working on a solution relying on DM.

10.1.3 Processing precedences before run-time/at run-time

EDF or DM were not originally designed for tasks with precedence constraints. There are two main different approaches to support precedence constraints. The first approach relies on semaphores to handle task synchronization. For instance, the Stack Resource Policy (SRP) [Bak91] was designed for scheduling tasks with shared resources. It can be used to handle binary semaphore and thus to handle task precedences: to each precedence $\tau_i \rightarrow \tau_j$ corresponds a binary semaphore and τ_j must wait for τ_i to unlock the semaphore before it can be scheduled. The SRP protocol ensures that access to shared resources does not lead to deadlocks and provides a tight bound for the duration of task priority inversions (the duration of blocking that a task may experience due to resources held by lower priority jobs). It handles either static or dynamic priorities policies.

In the second approach, the real-time characteristics of the tasks are adjusted before run-time so that the scheduler will respect task precedences at run-time, without needing explicit synchronizations (semaphores). Such a method is detailed in [CSB90], where the authors propose to encode precedences by adjusting the release dates and deadlines of the tasks and to schedule the adjusted task set with the standard EDF policy. This technique is optimal in the sense that a valid schedule can be found for the original task set if and only if a valid schedule can be found for the adjusted task set. The main advantage of this approach is that no synchronization primitive is required. This is particularly interesting in the context of critical systems as semaphores are usually not well accepted (due to the risk of deadlocks).

10.2 Encoding Task Precedences

This section first details the precedence encoding technique of [CSB90] and then describes how we adapt it to the extended precedences of the reduced task graphs of Sect. 9.2.5 and to the synchronous context.

10.2.1 Adjusting real-time characteristics

Simple precedences

The encoding technique was originally designed for aperiodic tasks related by precedence constraints. Let $succ(\tau_i)$ denote the successors of τ_i and $pred(\tau_i)$ denote its predecessors. Precedences are encoded by adjusting the release date and deadline of each task as follows:

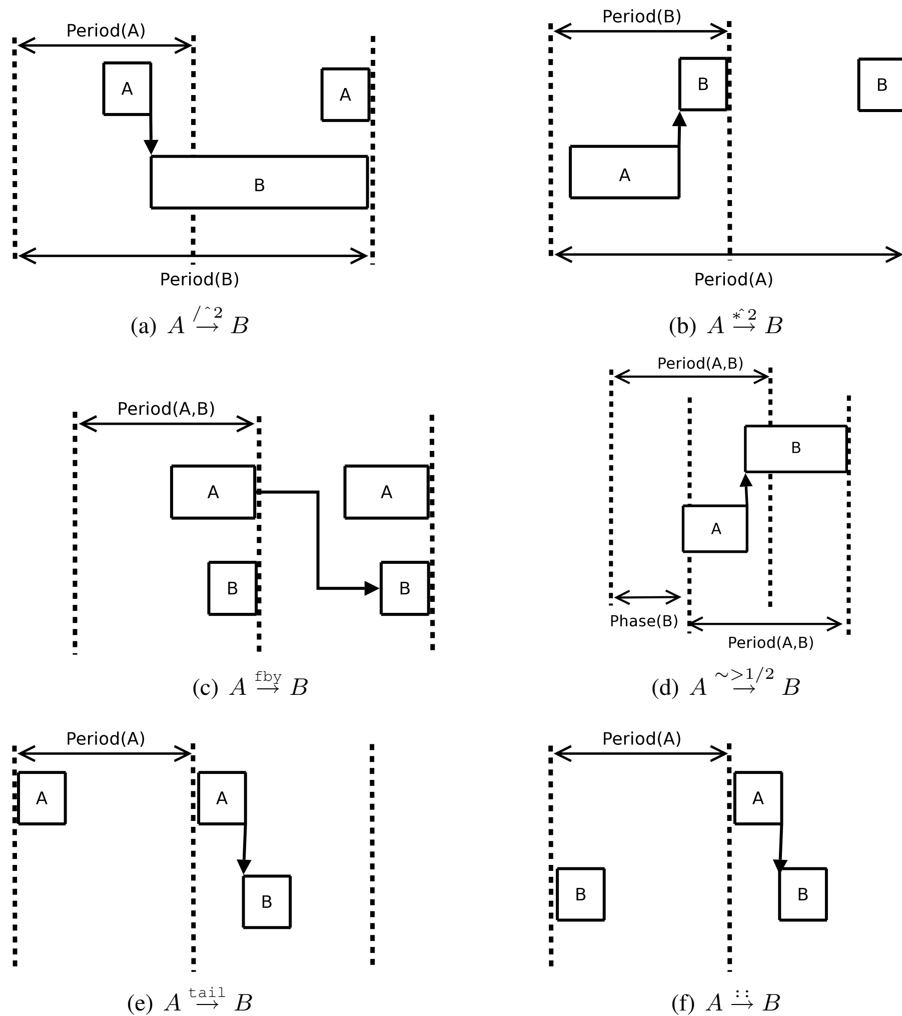
- A task τ_i must end early enough for its successors to complete before their own deadlines so the adjusted absolute deadline D_i^* is: $D_i^* = min(D_i, min(D_j^* - C_j))$ (for all $\tau_j \in succ(\tau_i)$);
- The release date adjustment proposed in [CSB90] is: $R_i^* = max(R_i, max(R_j^* + C_j))$ (for all $\tau_j \in pred(\tau_i)$). This formulae can be simplified in our context as: $R_i^{*'} = max(R_i, max(R_j^{*'}))$ (for all $\tau_j \in pred(\tau_i)$). Indeed, let A precede B, after deadline adjustment, we get $D_A^* < D_B^*$, so EDF will affect a higher priority to A. Consequently, to ensure that A is scheduled before B, we only need to make sure that B is not released before A, thus the second encoding. The first encoding enabled more efficient schedulability analysis in the context of [CSB90], but as we will

see in Sect. 10.3 this is of no use in our context as the schedulability analysis is a co-NP hard problem.

This technique can directly be extended to precedences relating tasks of the same period, which we will call *simple precedences*. Let $\tau_i[n] \rightarrow \tau_j[m]$ denote a precedence from the instance n of task τ_i to the instance m of task τ_j. A simple precedence $\tau_i \rightarrow \tau_j$ is such that for all n, $\tau_i[n] \rightarrow \tau_j[n]$. Thus, adjusting the relative deadline and initial release date of a task is equivalent to adjusting the absolute deadline and release date of all its instances.

Extended precedences

In the case of reduced task graphs as defined in Sect. 9.2.5, we need to encode *extended precedences*. We can see in Fig. 10.1 that extended precedences correspond to more complex task instance precedence patterns and that the release dates and deadlines adjustment to apply may vary from one task instance to the next. For instance in Fig. 10.1(a), the adjusted relative deadline of $A[0]$ is not the same as the one of $A[1]$.



(a) $A \overset{/\hat{}2}{\rightarrow} B$ (b) $A \overset{*\hat{}2}{\rightarrow} B$

(c) $A \overset{\texttt{fby}}{\rightarrow} B$ (d) $A \overset{\sim>1/2}{\rightarrow} B$

(e) $A \overset{\texttt{tail}}{\rightarrow} B$ (f) $A \overset{::}{\rightarrow} B$

Figure 10.1: Encoding extended precedences

A simple solution consists in unfolding the reduced task graph on the hyperperiod of the tasks and

performing the encoding on the unfolded graph. The *hyperperiod* of a task set, which we will denote HP, is defined as the least common multiple of the periods of the tasks. [LM80] proved that the schedule produced by EDF for a set of periodic tasks first goes through a transitory phase of length $max(r_i)+HP$ (for all τ_i) and then repeats a pattern of length HP.

Thus, a simple solution to the problem of scheduling tasks related by extended precedences is to unfold the extended precedence graph on the hyperperiod of the tasks (see [RCK01] for instance) and to apply the encoding on the unfolded graph, which only contains simple precedences.

This solution replaces each task τ_i by HP/T_i duplicates (if we do not consider the transitory phase) in the unfolded graph. This can lead to important computation overhead at execution as the scheduler needs to make its decisions according to a task set that will contain much more tasks than the original task set. In the following sections, we propose a different solution that does not duplicate any task, so that the scheduler takes less time to make its decisions. This also implies less memory consumption and smaller code size.

10.2.2 Definition of task instance precedences

We propose an extension of the encoding technique proposed by [CSB90] to the case of extended precedences, which does not require to unfold the reduced task graph. The intuition is that extended precedences correspond to regular patterns of simple precedences and thus that we should take advantage of this regularity to perform the encoding in a compact and factorized manner.

First, we need an exploitable definition of the set of task instance precedences corresponding to extended precedences. For precedences with a list of operators in the precedence annotation, we have $\tau_i \overset{ops}{\rightarrow} \tau_j \Rightarrow \forall n,\ \tau_i[n] \rightarrow \tau_j[g_{ops}(n)]$, with g_{ops} defined as follows:

$$\begin{aligned}
g_{*\hat{}k.ops}(n) =& g_{ops}(kn)\\
g_{/\hat{}k.ops}(n) =& g_{ops}(\lceil n/k \rceil)\\
g_{\sim>q.ops}(n) =& g_{ops}(n)\\
g_{\texttt{tail}.ops}(n) =& \begin{cases} g_{ops}(0) & \text{if n=0}\\ g_{ops}(n-1) & \text{otherwise}\end{cases}\\
g_{\texttt{::}.ops} =& g_{ops}(n+1)\\
g_{\texttt{fby}.ops}(n) =& g_{ops}(n+1)\\
g_{\texttt{when}.ops}(n) =& g_{ops}(n)\\
g_{\texttt{whennot}.ops}(n) =& g_{ops}(n)\\
g_{\epsilon}(n) =& n
\end{aligned}$$

The term $\lceil r \rceil$, with $r \in \mathbb{Q}^+$, denotes the smallest integer no less than r (for instance 3 for $\frac{5}{2}$). Implicitly, when we write $\lceil n/k \rceil$, here n/k denotes the rational $\frac{n}{k}$. In all other contexts, n/k denotes the division on integers, or more exactly the quotient of the division (for instance $4/3 = 1$). When required, we can use the following property to switch from $\lceil n/k \rceil$ to n/k:

Property 10.

$$\lceil n/k \rceil = (n-1)/k + 1 \qquad\qquad n/k = \lceil n/k \rceil + \frac{((n-1) \bmod k) + 1}{k} - 1$$

Proof. The property on the left is easily obtain from the definition of n/k and $\lceil n/k \rceil$.

To prove the property on the right, we have two case. First, if $k = 1$, then $n/k = n$ and:

$$\begin{aligned}\lceil n/k \rceil + \frac{((n-1) \bmod k) + 1}{k} - 1 &= n + (0+1)/1 - 1 \\ &= n = n/k\end{aligned}$$

So the property is true for $k = 1$.

Otherwise ($k > 1$), we first notice that:

$$(a+b)/k = a/k + b/k + ((a \bmod k) + (b \bmod k))/k$$

Indeed, let $a = q_1 k + r_1$ (with $0 \leq r_1 < k$) and $b = q_2 k + r_2$ (with $0 \leq r_2 < k$). We have $a + b = (q_1 + q_2)k + r_1 + r_2$. Let $r_1 + r_2 = q_3 k + r_3$ (with $0 \leq r_3 < k$). We have:

$$\begin{aligned}r_1 + r_2 = q_3 k + r_3 &\Leftrightarrow q_3 = (r_1 + r_2)/k \\ &\Leftrightarrow q_3 = ((a \bmod k) + (b \bmod k))/k\end{aligned}$$

We have $a + b = (q_1 + q_2 + q_3)k + r_3$, so:

$$\begin{aligned}(a+b)/k &= q_1 + q_2 + q_3 \\ &= a/k + b/k + ((a \bmod k) + (b \bmod k))/k\end{aligned}$$

Thus:

$$\begin{aligned}n/k &= (n-1)/k + 1/k + ((n-1) \bmod k + 1 \bmod k)/k \\ &= ((n-1)/k + 1) - 1 + 1/k + ((n-1) \bmod k + 1 \bmod k)/k \\ &= \lceil n/k \rceil + 1/k + ((n-1) \bmod k + 1 \bmod k)/k - 1\end{aligned}$$

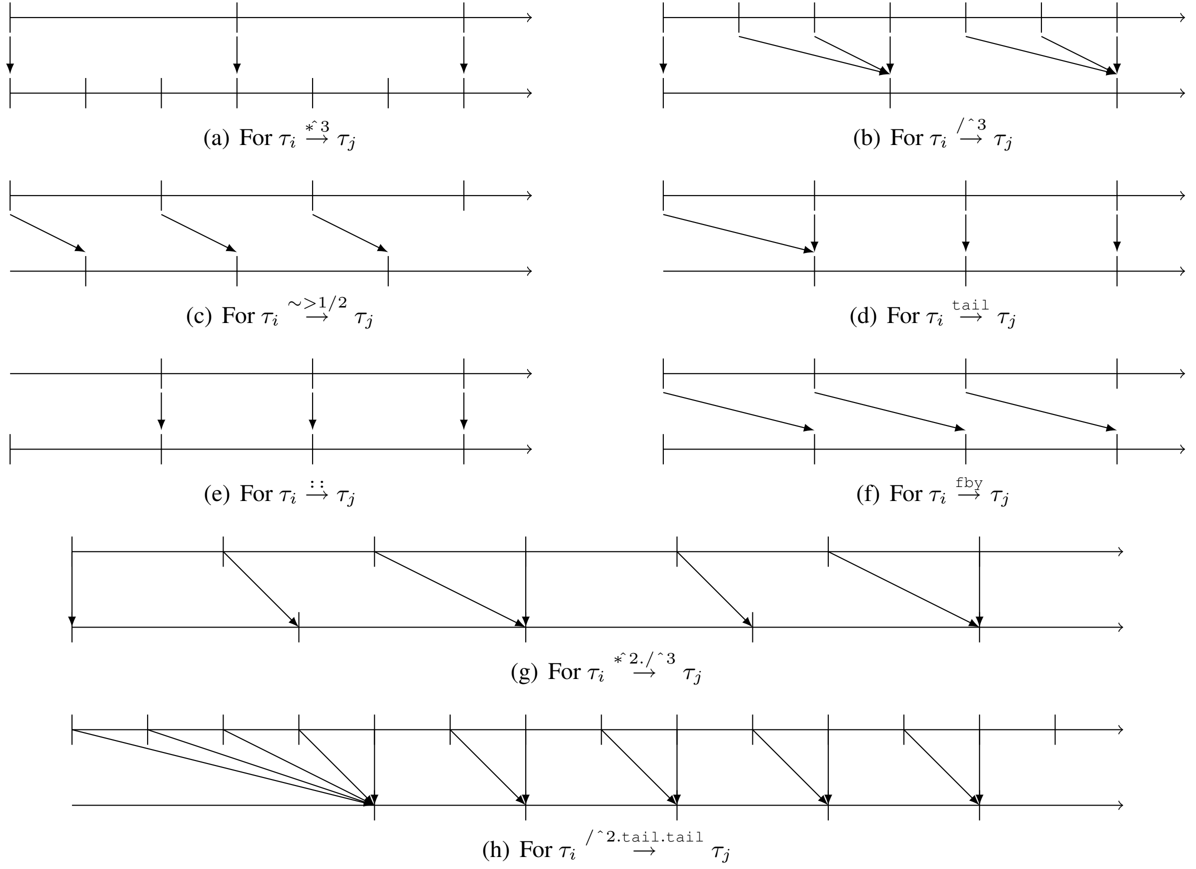
As $k > 1$, we have $1/k = 0$ and:

$$n/k = \lceil n/k \rceil + ((n-1) \bmod k + 1 \bmod k)/k - 1$$

□

The function g_{ops} is illustrated in Fig. 10.2. The two time axes of each subfigure show the release dates of two tasks τ_i and τ_j related by a precedence $\tau_i \xrightarrow{\tau}_j$. The arrows describe which instance of τ_i precedes which instance of τ_j, according to the definition of g_{ops}.

- For $\tau_i \overset{*\hat{}3}{\rightarrow} \tau_j$, $g_{ops}(n) = 0, 3, 6, 9, \ldots$, so $\tau_i[0] \rightarrow \tau_j[0]$, $\tau_i[1] \rightarrow \tau_j[3]$, $\tau_i[2] \rightarrow \tau_j[6]$, $\tau_i[3] \rightarrow \tau_j[9]$ and so on;
- For $\tau_i \overset{/\hat{}3}{\rightarrow} \tau_j$, $g_{ops}(n) = 0, 3, 3, 3, 6, 6, 6, \ldots$, so $\tau_i[0] \rightarrow \tau_j[0]$, $\tau_i[1] \rightarrow \tau_j[3]$, $\tau_i[2] \rightarrow \tau_j[3]$, $\tau_i[3] \rightarrow \tau_j[3]$, $\tau_i[4] \rightarrow \tau_j[6]$, $\tau_i[5] \rightarrow \tau_j[6]$, $\tau_i[6] \rightarrow \tau_j[6]$ and so on.
- For $\tau_i \overset{\sim>1/2}{\rightarrow} \tau_j$, $g_{ops}(n) = 0, 1, 2, 3, 4, \ldots$
- For $\tau_i \overset{\texttt{tail}}{\rightarrow} \tau_j$, $g_{ops}(n) = 0, 0, 1, 2, 3, 4, \ldots$
- For $\tau_i \overset{::}{\rightarrow} \tau_j$, $g_{ops}(n) = 1, 2, 3, 4, \ldots$
- For $\tau_i \overset{\texttt{fby}}{\rightarrow} \tau_j$, $g_{ops}(n) = 1, 2, 3, 4, \ldots$



(a) For $\tau_i \overset{*\hat{}3}{\rightarrow} \tau_j$

(b) For $\tau_i \overset{/\hat{}3}{\rightarrow} \tau_j$

(c) For $\tau_i \overset{\sim>1/2}{\rightarrow} \tau_j$

(d) For $\tau_i \overset{\texttt{tail}}{\rightarrow} \tau_j$

(e) For $\tau_i \overset{::}{\rightarrow} \tau_j$

(f) For $\tau_i \overset{\texttt{fby}}{\rightarrow} \tau_j$

(g) For $\tau_i \overset{*\hat{}2./\hat{}3}{\rightarrow} \tau_j$

(h) For $\tau_i \overset{/\hat{}2.\texttt{tail}.\texttt{tail}}{\rightarrow} \tau_j$

Figure 10.2: Task instance precedences, as defined by g_{ops}

- For $\tau_i \overset{*\hat{}2./\hat{}3}{\rightarrow} \tau_j$, $g_{ops}(n) = 0, 1, 2, 2, 3, 4, 4, \dots$
- For $\tau_i \overset{/\hat{}2.\texttt{tail}.\texttt{tail}}{\rightarrow} \tau_j$, $g_{ops}(n) = 0, 0, 0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4 \dots$

Notice that with this definition of extended precedences, $\tau_i[n] \rightarrow \tau_j[n']$ does not necessarily imply that $\tau_j[n']$ consumes data produced by $\tau_i[n]$. For instance, according to this definition, for the precedence $\tau_i \overset{*\hat{}2}{\rightarrow} \tau_j$, we have $\tau_i[1] \rightarrow \tau_j[1]$ and $\tau_i[2] \rightarrow \tau_j[1]$, while there is no data-dependency from $\tau_i[1]$ to $\tau_j[1]$. The precedence $\tau_i[1] \rightarrow \tau_j[1]$ is not really necessary, because implicitly $\tau_i[1] \rightarrow \tau_i[2]$ and so transitively due to precedence $\tau_i[2] \rightarrow \tau_j[1]$ we have $\tau_i[1] \rightarrow \tau_j[1]$. However, even though the precedence $\tau_i[1] \rightarrow \tau_j[1]$ is unnecessary, it is still correct to say it exists and this allows us to give a simple definition for function g_{ops}. Data-dependencies will be defined more precisely in Sect. 10.4.2. In the following sections, we use function g_{ops} to define a factorized encoding of extended precedences.

10.2.3 Release dates adjustment in the synchronous context

The synchronous semantics of the language has important consequences on the relationship between release dates of communicating tasks, which simplifies the encoding. Actually, we can prove that in our context we do not need to adjust task release dates.

Preliminary definitions

We first introduce some intermediate definitions to prove this property. As the proof will be done by induction, we start by defining for a precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$, T_i inductively from T_j and r_i inductively from r_j. Remember that the real-time attributes of a task $(T_i, C_i, r_i, d_i, R_i[n], D_i[n])$ have been defined in Sect. 9.3.

Definition 12. *For all $\tau_i \overset{ops}{\rightarrow} \tau_j$, we have for all n, $\tau_i[n] \rightarrow \tau_j[g_{ops}(n)]$ by definition.*

Let T_{ops} and r_{ops} be two functions such that for all precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$, $T_i = T_{ops}(T_j)$ and $r_i = r_{ops}(r_j, T_j)$. These functions can be defined inductively as follows:

$$
\begin{aligned}
r_{*\hat{}k.ops}(r,t) &= r_{ops}(r,t) & T_{*\hat{}k.ops}(t) &= kT_{ops}(t) \\
r_{/\hat{}k.ops}(r,t) &= r_{ops}(r,t) & T_{/\hat{}k.ops}(t) &= T_{ops}(t)/k \\
r_{\sim>q.ops}(r,t) &= r_{ops}(r,t) - q * T_{ops}(t) & T_{\sim>q.ops}(t) &= T_{ops}(t) \\
r_{\texttt{tail}.ops}(r,t) &= r_{ops}(r,t) - T_{ops}(t) & T_{\texttt{tail}.ops}(t) &= T_{ops}(t) \\
r_{\texttt{::}.ops} &= r_{ops}(r,t) + T_{ops}(t) & T_{\texttt{::}.ops}(t) &= T_{ops}(t) \\
r_{\texttt{fby}.ops}(r,t) &= r_{ops}(r,t) & T_{\texttt{fby}.ops}(t) &= T_{ops}(t) \\
r_{\texttt{when}.ops}(r,t) &= r_{ops}(r,t) & T_{\texttt{when}.ops}(t) &= T_{ops}(t) \\
r_{\texttt{whennot}.ops}(r,t) &= r_{ops}(r,t) & T_{\texttt{whennot}.ops}(t) &= T_{ops}(t) \\
r_{\epsilon}(r,t) &= r & T_{\epsilon}(t) &= t
\end{aligned}
$$

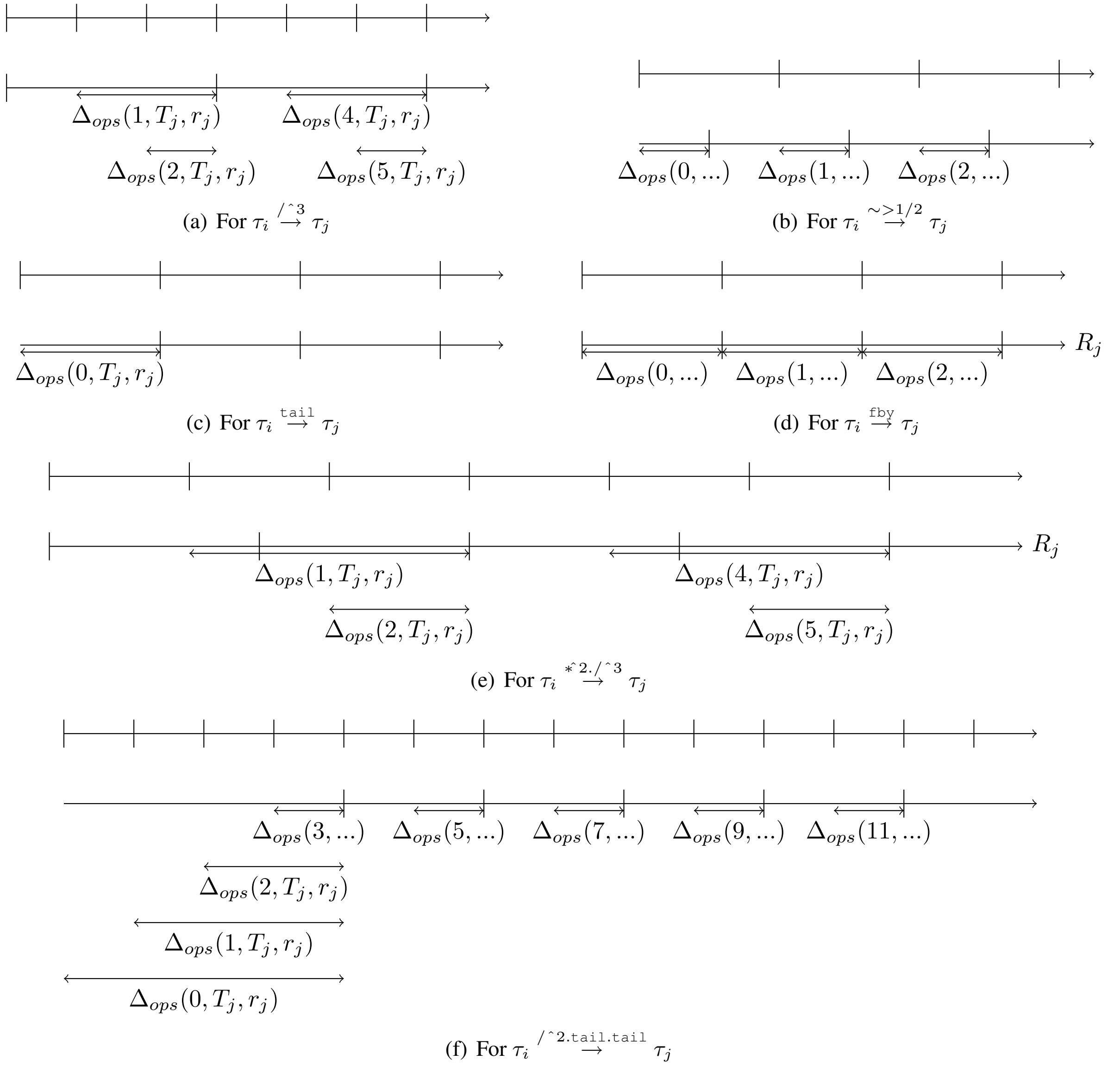
Then we define the function $\Delta_{ops}(n, T_j, r_j)$, which will play a key role in our precedence encoding algorithm:

Definition 13. *Let $\Delta_{ops}(n, T_j, r_j) = g_{ops}(n)T_j - nT_{ops}(T_j) + r_j - r_{ops}(r_j, T_j)$. From the definition of $R_i[n]$, we have:*

$$R_j[g_{ops}(n)] = R_i[n] + \Delta_{ops}(n, T_j, r_j)$$

Δ_{ops} represents the difference between the release dates of two task instances related by a precedence relation. This operator is illustrated in Fig.10.3, where we show the release dates of the successive instances of two tasks τ_i and τ_j related by a precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$. When $\Delta_{ops}(n, T_j, r_j) = 0$, it is not represented in the figure.

- If we take $T_i = 3$, $T_j = 1$, $r_j = 0$, $ops = *\hat{}3$, we have (for $n = 0, 1, 2, \ldots$): $\Delta_{ops}(n, T_j, r_j) = 0, 0, 0, 0, 0, 0, \ldots$
- If we take $T_i = 1$, $T_j = 3$, $r_j = 0$, $ops = /\hat{}3$, we have (for $n = 0, 1, 2, \ldots$): $\Delta_{ops}(n, T_j, r_j) = 0, 2, 1, 0, 2, 1, \ldots$
- If we take $T_i = T_j = 2$, $r_j = 1$, $ops = \sim> 1/2$, we have (for $n = 0, 1, 2, \ldots$): $\Delta_{ops}(n, T_j, r_j) = 1, 1, 1, 1, 1, 1, \ldots$
- If we take $T_i = T_j = 2$, $r_j = 2$, $ops = \texttt{tail}$, we have (for $n = 0, 1, 2, \ldots$): $\Delta_{ops}(n, T_j, r_j) = 2, 0, 0, 0, 0, 0, \ldots$
- If we take $T_i = T_j = 2$, $r_j = 0$, $ops = \texttt{::}$, we have (for $n = 0, 1, 2, \ldots$): $\Delta_{ops}(n, T_j, r_j) = 0, 0, 0, 0, 0, 0, \ldots$



(a) For $\tau_i \overset{/\hat{}3}{\rightarrow} \tau_j$

(b) For $\tau_i \overset{\sim>1/2}{\rightarrow} \tau_j$

(c) For $\tau_i \overset{\texttt{tail}}{\rightarrow} \tau_j$

(d) For $\tau_i \overset{\texttt{fby}}{\rightarrow} \tau_j$

(e) For $\tau_i \overset{*\hat{}2./\hat{}3}{\rightarrow} \tau_j$

(f) For $\tau_i \overset{/\hat{}2.\texttt{tail}.\texttt{tail}}{\rightarrow} \tau_j$

Figure 10.3: Illustration of Δ_{ops}

- If we take $T_i = T_j = 2$, $r_j = 0$, $ops = \texttt{fby}$, we have (for $n = 0, 1, 2, \ldots$):
  $\Delta_{ops}(n, T_j, r_j) = 2, 2, 2, 2, 2, 2, \ldots$
- If we take $T_i = 2$, $T_j = 3$, $r_j = 0$, $ops = *\hat{}2./\hat{}3$, we have (for $n = 0, 1, 2, \ldots$):
  $\Delta_{ops}(n, T_j, r_j) = 0, 4, 2, 0, 4, 2, 0, 4, 2, \ldots$
- If we take $T_i = 1$, $T_j = 2$, $r_j = 4$, $ops = /\hat{}2.\texttt{tail}.\texttt{tail}$, we have (for $n = 0, 1, 2, \ldots$):
  $\Delta_{ops}(n, T_j, r_j) = 4, 3, 2, 1, 0, 1, 0, 1, 0, \ldots$

Notice that the sequence of values of $\Delta_{ops}(n, T_j, r_j)$, for increasing values of n, is ultimately periodic (it follows a repetitive pattern after an initial prefix). Our deadlines adjustment technique relies on this key property (see next section), it allows us to represent the sequence of adjusted deadlines of a task as an ultimately periodic pattern (this will be proved in the next section).

As our proofs will heavily rely on function Δ_{ops}, we notice the following properties:

Property 11. *The function Δ_{ops} has the following properties:*

- $\Delta_{\epsilon}(n, T_j, r_j) = 0$;
- $\Delta_{*\hat{}k.ops}(n, T_j, r_j) = \Delta_{ops}(kn, T_j, r_j)$;
- $\Delta_{/\hat{}k.ops}(n, T_j, r_j) = \Delta_{ops}(\lceil n/k \rceil, T_j) + (1 - \frac{((n-1) \bmod k)+1}{k})T_{ops}(T_j, r_j)$;
- $\Delta_{\sim>q.ops}(n, T_j, r_j) = \Delta_{ops}(n, T_j, r_j) + q * T_{ops}(T_j, r_j)$;
- $\Delta_{\texttt{fby}.ops}(n, T_j, r_j) = \Delta_{ops}(n+1, T_j, r_j) + T_{ops}(T_j, r_j)$;
- $\Delta_{\texttt{tail}\ .ops}(n, T_j, r_j) = \begin{cases} \Delta_{ops}(0) + T_{ops}(n, T_j, r_j) & \textit{if } n = 0 \\ \Delta_{ops}(n-1, T_j, r_j) & \textit{otherwise} \end{cases}$
- $\Delta_{\texttt{::}.ops}(n, T_j) = \Delta_{ops}(n+1, T_j, r_j)$;
- $\Delta_{\texttt{when}.ops}(n, T_j, r_j) = \Delta_{\texttt{whennot}.ops}(n, T_j) = \Delta_{ops}(n, T_j, r_j)$

Proof. Case by case:

- Case ϵ: $\Delta_{\epsilon}(n, T_j, r_j) = nT_j - nT_j + r_j - r_j = 0$;
- Case $*\hat{}k$:

$$\begin{aligned} \Delta_{*\hat{}k.ops}(n, T_j, r_j) &= g_{ops}(kn)T_j - knT_{ops}(T_j) + r_j - r_{ops}(r_j, T_j) \\ &= \Delta_{ops}(kn, T_j, r_j) \end{aligned}$$

- Case $/\hat{}k$:

$$\begin{aligned} \Delta_{/\hat{}k.ops}(n, T_j, r_j) &= g_{ops}(\lceil n/k \rceil)T_j - n/kT_{ops}(T_j) + r_j - r_{ops}(r_j, T_j) \\ &= g_{ops}(\lceil n/k \rceil)T_j - \lceil n/k \rceil T_{ops}(T_j) + (1 - \frac{((n-1) \bmod k) + 1}{k})T_{ops}(T_j) + \\ &\quad r_j - r_{ops}(r_j, T_j) \\ &= \Delta_{ops}(\lceil n/k \rceil, T_j, r_j) + (1 - \frac{((n-1) \bmod k) + 1}{k})T_{ops}(T_j) \end{aligned}$$

- Case $\sim> q$:

$$\begin{aligned} \Delta_{\sim>q.ops}(n, T_j, r_j) &= g_{ops}(n, T_j) - nT_{ops}(T_j) + r_j - r_{ops}(r_j, T_j) + q * T_{ops}(T_j) \\ &= \Delta_{ops}(n, T_j, r_j) + q * T_{ops}(T_j) \end{aligned}$$

- Case `fby`:

$$\begin{aligned} \Delta_{\texttt{fby}.ops}(n, T_j, r_j) &= g_{ops}(n+1)T_j - nT_{ops}(T_j) + r_j - r_{ops}(r_j, T_j) \\ &= g_{ops}(n+1)T_j - (n+1)T_{ops}(T_j) + T_{ops}(T_j) + r_j - r_{ops}(r_j, T_j) \\ &= \Delta_{ops}(n+1, T_j, r_j) + T_{ops}(T_j) \end{aligned}$$

- Case `tail`: If $n = 0$:

$$\begin{aligned}\Delta_{\texttt{tail}.ops}(0, T_j, r_j) &= g_{ops}(0) + r_j - r_{ops}(r_j, T_j) + T_{ops}(T_j)\\ &= \Delta_{ops}(0, T_j, r_j) + T_{ops}(T_j)\end{aligned}$$

  Otherwise:

$$\begin{aligned}\Delta_{\texttt{tail}.ops}(n, T_j, r_j) &= g_{ops}(n-1)T_j - nT_{ops}(T_j) + r_j - r_{ops}(r_j, T_j) + T_{ops}(T_j)\\ &= g_{ops}(n-1)T_j - (n-1)T_{ops}(T_j) - T_{ops}(T_j) + r_j - r_{ops}(r_j, T_j) + T_{ops}(T_j)\\ &= \Delta_{ops}(n-1, T_j, r_j)\end{aligned}$$

- Case `::`:

$$\begin{aligned}\Delta_{::.ops}(n, T_j, r_j) &= g_{ops}(n+1) - nT_{ops}(T_j) + r_j - r_{ops}(r_j, T_j) - T_{ops}(T_j)\\ &= g_{ops}(n+1) - (n+1)T_{ops}(T_j) + T_{ops}(T_j) + r_j - r_{ops}(r_j, T_j) - T_{ops}(T_j)\\ &= \Delta_{ops}(n+1, T_j, r_j)\end{aligned}$$

- Cases `when` and `whennot` are trivial.

□

Main property

We are now ready to prove the main property:

Theorem 3. *For all precedence $\tau_i \stackrel{ops}{\to} \tau_j$, for all n:*

$$R_{\tau_j[g_{ops}(n)]} \geq R_i[n]$$

Proof. This is equivalent to proving that $\Delta_{ops}(n, T_j, r_j) \geq 0$ for all n. The proof is done by induction on ops. First, for the base of the induction. We have $\Delta_\epsilon(n, T_j, r_j) = 0$, so this case is proved.

Now, the induction hypothesis is $\Delta_{ops}(n, T_j, r_j) \geq 0$.

- We then prove the induction steps. For case $/\hat{}k$, we have:

$$\Delta_{/\hat{}k.ops}(n, T_j, r_j) = \Delta_{ops}(\lceil n/k \rceil, T_j, r_j) + (1 - \frac{((n-1) \bmod k) + 1}{k})T_{ops}(T_j)$$

  Notice that $(n-1) \bmod k \leq k-1$ and:

$$\begin{aligned}(n-1) \bmod k \leq k-1 &\Leftrightarrow (n-1) \bmod k + 1 \leq k\\ &\Leftrightarrow \frac{(n-1) \bmod k + 1}{k} \leq 1\\ &\Leftrightarrow 1 - \frac{(n-1) \bmod k + 1}{k} \geq 0\end{aligned}$$

  As $T_{ops}(T_j) \geq 0$, from the induction hypothesis this induction step is proved.

- For case $\sim>$, we have:

$$\Delta_{\sim>q.ops}(n, T_j, r_j) = \Delta_{ops}(n, T_j, r_j) + q * T_{ops}(T_j)$$

  which is positive from the induction hypothesis (as q can only be positive).

- For case `fby`, we have:

$$\Delta_{\texttt{fby}.ops}(n, T_j, r_j) = \Delta_{ops}(n+1, T_j, r_j) + T_{ops}(T_j)$$

which is positive from the induction hypothesis (as $T_{ops}(T_j)$ is positive).

- For case `tail`, in the case $n = 0$, we have:

$$\Delta_{\texttt{tail}.ops}(0, T_j, r_j) = \Delta_{ops}(0, T_j, r_j) + T_{ops}(T_j)$$

which is positive from the induction hypothesis (as $T_{ops}(T_j)$ is positive).

Otherwise ($n > 0$):

$$\Delta_{\texttt{tail}\ .ops}(n, T_j, r_j) = \Delta_{ops}(n-1, T_j, r_j)$$

which is positive from the induction hypothesis.

- The remaining cases are deduced trivially from the induction hypothesis and from Property. 11, so the property is proved.

□

As an immediate consequence, we have:

Corollary 1. *For all $\tau_j \in pred(\tau_i)$:*

$$R_i \geq max(R_i, max(R_j))$$

As a consequence, release dates adjustment is directly ensured by the synchronization constraints imposed by the clock calculus.

Particular case: delays

The following property allows us to ignore precedences for which *ops* contains a delay during deadlines adjustment:

Property 12. *For a precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$ such that ops contains a delay, $R_j[g_{ops}(n)] \geq R_i[n] + T_i$.*

Proof. This is equivalent to proving that for all, $\Delta_{ops}(n, T_j, r_j) \geq T_i$ for all n, if *ops* contains a delay. The proof is done by induction on *ops*.

- First, for the base of the induction:

$$\Delta_{\texttt{fby}.ops}(n, T_j, r_j) = \Delta_{ops}(n+1, T_j, r_j) + T_{ops}(T_j)$$

We just proved that $\Delta_{ops}(n+1, T_j, r_j) \geq 0$, so:

$$\Delta_{\texttt{fby}.ops}(n, T_j, r_j) \geq T_{ops}(T_j)$$

So the property holds in this case;

- Then, the induction hypothesis is that *ops* contains a `fby` and that the property holds for *ops*. Assuming this hypothesis, we prove that we can deduce that the property holds for *op.ops* for any *op*, except if *op* is an operation $*\hat{}k$ (hypothesis of Sect. 9.2.5). First for case $/\hat{}k$:

$$\Delta_{/\hat{}k.ops}(n, T_j, r_j) = \Delta_{ops}(\lceil n/k \rceil, T_j, r_j) + (1 - \frac{((n-1) \bmod k) + 1}{k})T_{ops}(T_j)$$

We proved that $(1 - \frac{((n-1) \bmod k)+1}{k})T_{ops}(T_j) \geq 0$, so from the induction hypothesis:

$$\Delta_{/\hat{}k.ops}(n, T_j, r_j) \geq T_{ops}(T_j)$$

- For the case $\sim>$:

$$\Delta_{\sim>q.ops}(n, T_j, r_j) = \Delta_{ops}(n, T_j, r_j) + q * T_{ops}(T_j)$$

We have $q * T_{ops}(T_j) \geq 0$, so from the induction hypothesis:

$$\Delta_{\sim>q.ops}(n, T_j, r_j) \geq T_{ops}(T_j)$$

- For the case `tail`, in the case $n = 0$:

$$\Delta_{\texttt{tail}.ops}(0, T_j, r_j) = \Delta_{ops}(0, T_j, r_j) + T_{ops}(T_j)$$

So, from the induction hypothesis we have:

$$\Delta_{\texttt{tail}.ops}(0, T_j, r_j) \geq T_{ops}(T_j)$$

Otherwise:

$$\Delta_{\texttt{tail}\ .ops}(n, T_j, r_j) = \Delta_{ops}(n-1, T_j, r_j)$$

So the property is true from the induction hypothesis;

- The remaining cases are deduced trivially from the induction hypothesis and from Property. 11, so the property is proved.

□

10.2.4 Deadline calculus

We have proved that release dates adjustment is not required in our context. We can now proceed to deadlines adjustment, which we call *deadline calculus*. The principle of the deadline calculus come from the observation that the constraint relating $d_i[n]$ to $d_j[g_{ops}(n)]$ is a periodic constraint. We will show that there always exists some integers b, p such that for all $n > b$, $d_i^*[n] = d_i^*[n+p]$. As a consequence, the sequence of the relative deadlines of the instances of a task is ultimately periodic. We call such patterns *deadline words*. The deadline calculus performs the encoding of task precedences by computing a deadline word for each task of the task graph.

Transposing the adjustment to relative deadlines

First, we need to transpose the absolute deadlines adjustment formula to relative deadlines adjustment to fit our task model. The function Δ_{ops} plays a key role in this transposition. Indeed:

Property 13. *For all precedence* $\tau_i \stackrel{ops}{\rightarrow} \tau_j$*:*

$$D_i[n] \leq D_j[g_{ops}(n)] - C_j \Leftrightarrow d_i[n] \leq d_j[g_{ops}(n)] + \Delta_{ops}(n, T_j, r_j) - C_j$$

Proof.

$$\begin{aligned} D_i[n] \leq D_j[g_{ops}(n)] - C_j &\Leftrightarrow R_i[n] + d_i[n] \leq R_j[g_{ops}(n)] + d_j[g_{ops}(n)] - C_j \\ &\Leftrightarrow d_i[n] \leq d_j[g_{ops}(n)] + R_j[g_{ops}(n)] - R_i[n] - C_j \\ &\Leftrightarrow d_i[n] \leq d_j[g_{ops}(n)] + \Delta_{ops}(n, T_j, r_j) - C_j \end{aligned}$$

□

The function Δ_{ops} thus represents the difference between the dates which d_i and d_j are relative to.

Thanks to Property 12, we notice that for a precedence $\tau_i \stackrel{ops}{\rightarrow} \tau_j$, if ops contains a delay we do not need to adjust $d_i[n]$ (for any n). Indeed:

$$D_i[n] \leq D_j[g_{ops}(n)] - C_j \Leftrightarrow d_i[n] \leq R_j[g_{ops}(n)] - R_i[n] + d_j[g_{ops}(n)] - C_j$$

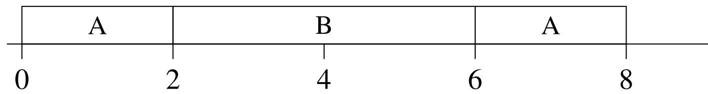
By default, we have $d_i[n] \leq T_i$, so from Property 12, by default $d_i[n] \leq R_j[g_{ops}(n)] - R_i[n]$. Furthermore, $d_j[g_{ops}(n)] - C_j \geq 0$ (otherwise the system is not schedulable), so by default $d_i[n] \leq R_j[g_{ops}(n)] - R_i[n] + d_j[g_{ops}(n)] - C_j$ and thus we do not need to encode precedences that contain a delay.

Task deadlines vs task instance deadlines

Figure 10.4 shows an example that is schedulable when we set different deadlines for different instances of a task, while it is not when we use the same deadline for every instance of the task. Indeed, as shown in Fig. 10.4(b), if we set $d_B[n] = 6$ for $n \in \mathbb{N}$, $d_A[n] = 2$ for $n \in 2\mathbb{N}$ and $d_A[n] = 4$ for $n \in 2\mathbb{N} + 1$, the task set $\{A,\ B\}$ is schedulable: $A[0]$ is scheduled during time interval $[0, 2[$, $B[0]$ during $[2, 6[$, $A[1]$ during $[6, 8[$ and this schedule repeats indefinitely. Now, if we set the same deadline for all instances of A, that is $d_A[n] = 2$ for $n \in \mathbb{N}$, then either $A[1]$ or $B[0]$ will miss their deadline (the absolute deadline 6). This motivates the introduction of task deadline words.

```
imported node A(i: int) returns (o: int) wcet 2;
imported node B(i: int) returns (o: int) wcet 4;
node M(i: rate(4,0)) returns (o: rate (8, 0) due 6)
let o=B(A(i)/^2); tel
```

(a) The program



(b) The corresponding schedule

Figure 10.4: Scheduling the program requires the instances of A to have different deadlines

Deadline Words

A *unitary deadline* specifies the relative deadline for the computation of a single instance of a task. A unitary deadline is simply an integer value d. An *ultimately periodic deadline word*, or simply a *deadline word* defines the sequence of unitary deadlines of a task. The set of deadline words $\mathcal{W}_{\mathrm{d}}$ is defined by the following grammar:

$$\begin{array}{rcl} w & ::= & u.(u)^\omega \\ u & ::= & d \mid d.u \end{array}$$

For a deadline word $w = u.(v)^\omega$, u is the prefix of w and $(v)^\omega$ denotes the infinite repetition of word v, called the repetitive pattern of w. For instance, the deadline word of A in Fig. 10.4 is $(2.4)^\omega$, so $d_A[0] = 2$, $d_A[1] = 4$, $d_A[2] = 2$, $d_A[3] = 4$ and so on.

We introduce some additional notations:

- $|u|$ denotes the length of the finite word u (the number of unitary deadlines in u);
- $w[n]$ denotes the nth unitary deadline of deadline word w ($n \in \mathbb{N}$). If $w = u.(v)^\omega$ then:

$$w[n] = \begin{cases} u[n] & \text{if } n < |u| \\ v[(n - |u|) \bmod |v|] & \text{otherwise} \end{cases}$$

- The ordering relation on deadline words is the lexicographic order: $\forall w_1, w_2 \in \mathcal{W}_{\mathrm{d}},\ w_1 \leq w_2 \Leftrightarrow \forall i \in \mathbb{N},\ w_1[i] \leq w_2[i]$;
- The minimum of two deadline words $w_m = min(w_i, w_j)$ is such that for all n, we have $w_m[n] = min(w_i[n], w_j[n])$. If $w_i = u_i.(v_i)^\omega$ and $w_j = u_j.(v_j)^\omega$, we can compute w_m by unfolding w_i into $w'_i = u'_i.(v'_i)^\omega$ and w_j into $w'_j = u'_j.(v'_j)^\omega$, such that $|u'_i| = |u'_j| = max(|u_i|, |u_j|)$, $|v'_i| = |v'_j| = lcm(|v_i|, |v_j|)$. We then simply compute the minimum of the two unfolded words point by point on $|w'_i|$;
- The sum of two deadline words $w = w_i + w_j$ is such that for all n, $w[n] = w_i[n] + w_j[n]$. It can be computed by unfolding w_i and w_j as we did to compute the minimum of two deadline words;
- The word $w = w_i + k$, with $w_i \in \mathcal{W}_{\mathrm{d}}$ and $k \in \mathbb{N}$ is such that for all n, $w[n] = w_i[n] + k$.

Computing Task Deadline Words

Let w_i denote the deadline word representing the sequence of unitary deadlines of task τ_i. From Sect. 10.2.2, we know that each task precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$ can be encoded as a constraint relating w_i to w_j of the form:

$$w_i \leq W_{ops}(w_j) + \Delta_{ops}(T_j, r_j) - C_j$$

where $W_{ops}(w_j)$ is a word such that $W_{ops}(w_j)[n] = w_j[g_{ops}(n)]$ for all n and $\Delta_{ops}(T_j, r_j)$ is a word such that $\Delta_{ops}(T_j, r_j)[n] = \Delta_{ops}(n, T_j, r_j)$ for all n. We must now prove that $W_{ops}(w_j)$ and $\Delta_{ops}(T_j, r_j)$ are both *ultimately periodic* words.

Property 14. *For all ops:*

$$\forall n \geq max(0, pref(ops)),\ \Delta_{ops}(n, T_j, r_j) = \Delta_{ops}(n + P(ops), T_j, r_j)$$

The length of the prefix of $\Delta_{ops}(T_j, r_j)$ is $max(0, pref(ops))$, while the length of its periodic pattern is $P(ops)$.

pref(ops) and P(ops) are defined as follows:

$$P(\epsilon) = 1 \qquad\qquad pref(\epsilon) = 0$$
$$P(\hat{*}k.ops) = P(ops)/gcd(P(ops), k) \qquad\qquad pref(\hat{*}k.ops) = \lceil pref(ops)/k \rceil$$
$$P(/\hat{}\,k.ops) = k * P(ops) \qquad\qquad pref(/\hat{}\,k.ops) = (pref(ops) - 1) * k + 1$$
$$P(op.ops) = P(ops) \text{ otherwise} \qquad\qquad pref(\mathtt{tail}.ops) = pref(ops) + 1$$
$$pref(\mathtt{::}.ops) = max(pref(ops) - 1, 0)$$
$$pref(op.ops) = pref(ops) \text{ otherwise}$$

Proof. By induction on the structure of ops and using the Property 11. Notice that we do not prove that $P(ops)$ is the smallest possible period of Δ_{ops} or that $pref(ops)$ is the smallest possible prefix of Δ_{ops}. We only find a bound so as to compute Δ_{ops} efficiently.

- First, for the base of the induction, $\Delta_\epsilon(n, T_j, r_j) = 0$, so $\Delta_\epsilon(n, T_j, r_j) = \Delta_\epsilon(n+1, T_j, r_j)$ for all $n \geq 0$;

- Then, the induction hypothesis is: for all $n \geq pref(ops)$, $\Delta_{ops}(n, T_j, r_j) = \Delta_{ops}(n+P(ops), T_j, r_j)$. We first prove the induction step for case $/\hat{}$. First, notice that:

$$\begin{aligned} n \geq pref(/\hat{}\,k.ops) &\Leftrightarrow n \geq (pref(ops) - 1) * k + 1 \\ &\Leftrightarrow (n-1)/k + 1 \geq pref(ops) \\ &\Leftrightarrow \lceil n/k \rceil \geq pref(ops) \end{aligned}$$

Then:

$$\begin{aligned} \Delta_{/\hat{}k.ops}(n + P(/\hat{}\,k.ops), T_j, r_j) &= \Delta_{/\hat{}k.ops}(n + k * P(ops), T_j, r_j) \\ &= \Delta_{ops}(\lceil (n + k * P(ops))/k \rceil, T_j, r_j) + \\ &\quad (1 - \frac{((n-1) \bmod k) + 1}{k}) T_{ops}(T_j) \end{aligned}$$

We have:

$$\begin{aligned} \lceil (n + k * P(ops))/k \rceil &= \lceil n/k + P(ops) + \frac{n \bmod k + (kP(ops) \bmod k)}{k} \rceil \\ &= \lceil n/k + P(ops) + (n \bmod k)/k \rceil \\ &= \lceil n/k + P(ops) \rceil \\ &= \lceil n/k \rceil + P(ops) \end{aligned}$$

So:

$$\begin{aligned} \Delta_{/\hat{}k.ops}(n + P(/\hat{}\,k.ops), T_j, r_j) &= \Delta_{ops}(\lceil n/k \rceil + P(ops), T_j, r_j) + \\ &\quad (1 - \frac{((n-1) \bmod k) + 1}{k}) T_{ops}(T_j) \end{aligned}$$

We proved that $\lceil n/k \rceil \geq pref(ops)$ so we can apply the induction hypothesis and:

$$\begin{aligned} \Delta_{/\hat{}k.ops}(n + P(/\hat{}\,k.ops), T_j, r_j) &= \Delta_{ops}(\lceil n/k \rceil, T_j, r_j) + \\ &\quad (1 - \frac{((n-1) \bmod k) + 1}{k}) T_{ops}(T_j) \\ &= \Delta_{/\hat{}k.ops}(n, T_j, r_j) \end{aligned}$$

- Then, for $\hat{*}k$, we have:

$$\begin{aligned}\Delta_{\hat{*}k.ops}(n+P(ops)/gcd(P(ops),k),T_j,r_j) &= \Delta_{ops}(kn+k*P(ops)/gcd(P(ops),k),T_j,r_j)\\ &= \Delta_{ops}(kn+dP(ops),T_j,r_j)\end{aligned}$$

with $d = k/gcd(P(ops),k)$ ($d \in \mathbb{N}^*$). Notice that:

$$n \geq \textit{pref}(\hat{*}k.ops) \Leftrightarrow n \geq \lceil \textit{pref}(ops)/k \rceil \Leftrightarrow kn \geq \textit{pref}(ops)$$

So we can apply the induction hypothesis and:

$$\Delta_{\hat{*}k.ops}(n+P(ops)/gcd(P(ops),k),T_j,r_j) = \Delta_{ops}(kn,T_j,r_j) = \Delta_{\hat{*}k.ops}(n,T_j,r_j)$$

- Then, for case `tail`, in the case $n = 0$:

$$\Delta_{\texttt{tail}.ops}(0+P(ops),T_j,r_j) = \Delta_{ops}(0+P(ops),T_j,r_j) + T_{ops}(T_j)$$

So from the induction hypothesis:

$$\begin{aligned}\Delta_{\texttt{tail}.ops}(0+P(ops),T_j,r_j) &= \Delta_{ops}(0,T_j,r_j) + T_{ops}(T_j)\\ &= \Delta_{\texttt{tail}.ops}(0,T_j,r_j)\end{aligned}$$

So the property is true for this case.

Otherwise:

$$\Delta_{\texttt{tail}\ .ops}(n+P(ops),T_j,r_j) = \Delta_{ops}(n-1+P(ops),T_j,r_j)$$

Notice that:

$$n \geq \textit{pref}(\texttt{tail}.ops) \Leftrightarrow n \geq \textit{pref}(ops)+1 \Leftrightarrow (n-1) \geq \textit{pref}(ops)$$

So we can apply the induction hypothesis and:

$$\Delta_{\texttt{tail}\ .ops}(n+P(ops),T_j,r_j) = \Delta_{ops}(n-1,T_j,r_j) = \Delta_{\texttt{tail}\ .ops}(n,T_j,r_j)$$

- Then, for case `::`:

$$\Delta_{::.ops}(n+P(ops),T_j,r_j) = \Delta_{ops}(n+1+P(ops),T_j,r_j)$$

Notice that:

$$n \geq \textit{pref}(::.ops) \Rightarrow n \geq max(\textit{pref}(ops)-1,0) \Rightarrow n+1 \geq \textit{pref}(ops)$$

So we can apply the induction hypothesis and:

$$\Delta_{::.ops}(n+P(ops),T_j,r_j) = \Delta_{ops}(n+1,T_j,r_j) = \Delta_{::.ops}(n,T_j,r_j)$$

- Then, for case `fby`:

$$\Delta_{\texttt{fby}.ops}(n+P(ops),T_j,r_j) = \Delta_{ops}(n+1+P(ops),T_j,r_j) + T_{ops}(T_j,r_j)$$

Notice that:

$$n \geq \textit{pref}(\texttt{fby}.ops) \Rightarrow n \geq max(\textit{pref}(ops)-1,0) \Rightarrow n+1 \geq \textit{pref}(ops)$$

So we can apply the induction hypothesis and:

$$\Delta_{\texttt{fby}.ops}(n+P(ops),T_j,r_j) = \Delta_{ops}(n+1,T_j,r_j) + T_{ops}(T_j) = \Delta_{\texttt{fby}.ops}(n,T_j,r_j)$$

- The remaining cases are deduced trivially from the induction hypothesis and from Property. 11, so the property is proved.

□

Property 15. *For all* $w_j = u_j.(v_j)^\omega$*, for all ops:*

$$\forall n \geq max(0, pref'(ops, w_j)), W_{ops}(w_j)[n] = W_{ops}(w_j)[n + P'(ops, w_j)]$$

The length of the prefix of $W_{ops}(w_j)$ *is* $max(0, pref'(ops, w_j))$*, while the length of its periodic pattern is* $P'(ops, w_j)$*.*

$pref'(ops, w_j)$ *and* $P'(ops, w_j)$ *are defined as follows:*

$$\begin{aligned} P'(\epsilon, w_j) &= |v_j| \\ P'(\hat{*}k.ops, w_j) &= P'(ops, w_j)/gcd(P'(ops, w_j), k) \\ P'(/\hat{}k.ops, w_j) &= k * P'(ops, w_j) \\ P'(op.ops, w_j) &= P'(op.ops, w_j) \text{ otherwise} \end{aligned}$$

$$\begin{aligned} pref'(\epsilon) &= |u_j| \\ pref'(\hat{*}k.ops) &= \lceil pref'(ops)/k \rceil \\ pref'(/\hat{}k.ops) &= (pref'(ops) - 1) * k + 1 \\ pref'(\texttt{tail}.ops) &= pref'(ops) + 1 \\ pref'(\texttt{::}.ops) &= max(pref'(ops) - 1, 0) \\ pref'(\texttt{fby}.ops) &= max(pref'(ops) - 1, 0) \\ pref'(op.ops) &= pref'(ops) \text{ otherwise} \end{aligned}$$

Proof. We prove that $W_{ops}(w_j)[n] = W_{ops}(w_j)[n + P'(ops, w_j)]$, by induction on the structure of ops.

The definitions of $P'(ops)$ and $pref'(ops)$ are exactly the same as those of $P(ops)$ and $pref(ops)$, except for the base of the induction (case $ops = \epsilon$). So we only prove the base of the induction. The rest of the proof is exactly that of Property. 14. For the base of the induction, for $n \geq |u_j|$, $W_\epsilon(w_j)[n] = w_j[g_\epsilon(n)] = w_j[n]$ and $W_\epsilon(w_j)[n + |w_j|] = w_j[n + |w_j|]$. So by definition of $|w_j|$, $W_\epsilon(w_j)[n] = W_\epsilon(w_j)[n + |w_j|]$. □

Thanks to these properties, $\Delta_{ops}(T_j, r_j)$ and $W_{ops}(w_j)$ are indeed deadline words and can simply be computed by computing each value of their prefix and of their repetitive pattern. The value of w_i can finally be computed as the combination of all the constraints on w_i:

$$w_i = min(w_i, min(\Delta_{ops}(T_j, r_j) + W_{ops}(w_j) - C_j))(\text{for all } \tau_j, \tau_i \xrightarrow{ops} \tau_j)$$

We give an algorithm for computing the task deadline words of a reduced task graph in Alg. 8. For this algorithm, delayed precedences are removed from the task graph. As a reduced task graph is a DAG (when we do not consider delayed precedences), the algorithm is basically a topological sort [Kah62] working backwards, starting from the end of the graph and computing deadline constraints at each step. This algorithm is similar to the one presented in [CSB90] but applied to extended precedences.

If we accept programs which contain a precedence $\tau_i \xrightarrow{ops} \tau_j$ where ops contains an operator $\hat{*}\, k$ that precedes the first operator `fby` appearing in *ops* (which we rejected in Sect. 9.2.5), then we can have a delayed precedence $\tau_i \xrightarrow{ops} \tau_j$ where w_i depends on w_j. As a consequence, we cannot compute task deadline words using a topological sort on tasks, but instead we must use a topological sort on task instances. This can be done by unfolding the task graph into a task instance graph. This will not be detailed here.

Algorithm 8 Computing task deadline words of a reduced task graph

1: **for** Every τ_i **do**
2:     **if** τ_i is asensor or an actuator with a deadline constraint (eg τ_i `due n`) **then**
3:         $w_i \leftarrow (n)^\omega$
4:     **else** $w_i \leftarrow (T_i)^\omega$
5:     **end if**
6: **end for**
7: $S \leftarrow$ List of tasks without successors
8: **while** S is not empty **do**
9:     Remove the head τ_j of S
10:     **for** Each τ_i such that $\tau_i \overset{ops}{\rightarrow} \tau_j$ and `fby` $\notin$ *ops* **do**
11:         $w_i \leftarrow min(w_i, \Delta_{ops}(T_j, r_j) + W_{ops}(w_j) - C_j)$
12:         Remove the precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$ from the graph
13:         **if** τ_i has no other successor **then**
14:             Insert τ_i in S
15:         **end if**
16:     **end for**
17: **end while**

Examples

We first consider the deadline calculus of the example of Fig. 10.4. To simplify, we consider that the durations of the main node (M) inputs and outputs are null. The calculus first computes $w_B = (6)^\omega$ due to the deadline constraint on o. Then it considers the precedence $A \overset{/\hat{}2}{\rightarrow} B$. We have *pref*$(/\,\hat{}\,2) = 0$, $P(/\,\hat{}\,2) = 2$ and $\Delta_{/\hat{}2}(T_B, r_B) = (0.4)^\omega$. We also have $W_{/\hat{}2}(w_B) = (6)^\omega$. So, this precedence imposes that $w_A \leq (0.4)^\omega + (6)^\omega - 4$ ie $w_A \leq (2.6)^\omega$. w_A was initialised to $(4)^\omega$ first, so we have $w_A = min((4)^\omega, (2.6)^\omega = (2.4)^\omega$.

For concision, let d^n denote the sequence consisting of n repetitions of d (for instance, $5^3 = 5.5.5$). On the example of the FAS of Fig. 5.7, we obtain the following deadlines:

$Gyro_Acq$	$(74.81^9)^\omega$
$gyro$	$(71.78^9)^\omega$
gps	$(71)^\omega$
gnc	$(300)^\omega$
Str_Acq	$(74)^\omega$
pde	$(100)^\omega$
GPS_Acq	$(74)^\omega$
TM_TC	$(9999)^\omega$
tc	$(8999)^\omega$
PWS	$(999)^\omega$
SGS	$(999)^\omega$
GNC_US	$(299)^\omega$
$FDIR$	$(89.96^9)^\omega$
sgs	$(1000)^\omega$
pws	$(1000)^\omega$

10.2.5 Optimality

By construction, our precedence encoding algorithm satisfies the following property:

Lemma 2. *For all $\tau_i \stackrel{ops}{\rightarrow} \tau_j$, the deadline words w_i and w_j computed by Alg. 8 satisfy the following property:*

$$\forall n, n', \tau_i[n] \rightarrow \tau_j[n'] \Rightarrow w_i[n] \leq W_{ops}(w_j)[n] + \Delta_{ops}(w_j, r_j)[n] - C_j \wedge n' = g_{ops}(n)$$

As a consequence, we have:

Corollary 2. *For all reduced task graph $g = (V, E)$, for all τ_i in V, let w_i denote the deadline word computed by Alg. 8 for τ_i, let $D_i^*[n] = R_i[n] + w_i[n]$. Then we have:*

$$D_i^*[n] = min(D_i[n], min(D_j^*[n'] - C_j)) \text{ (for all } \tau_j[n'] \in succ(\tau_i[n]))$$

This property states that for every task instance of a reduced task graph, our encoding algorithm respects the deadline adjustment formulae of [CSB90]. We have proved in Sect. 10.2.3 that the release date of every task instance of a reduced task graph also respects the release date adjustment formulae of [CSB90]. Therefore, the result established in [CSB90] hold for our encoding technique and so:

Theorem 4. *Let $g = (\{\tau_i(T_i, C_i, r_i, d_i), 1 \leq i \leq n\}, E)$ be a reduced task graph. Then, let $s = \{\tau_i'(T_i, C_i, r_i, w_i), 1 \leq i \leq n\}$ be a task set such that for all τ_i', w_i is the deadline word computed by Alg. 8 for τ_i. Then:*

g is schedulable if and only if s is schedulable.

EDF is optimal for scheduling the encoded task set, so as a corollary we have:

Corollary 3. *Let g be a reduced task graph and s be the task set computed by our encoding algorithm. Then:*

g is schedulable if and only if s is schedulable with EDF.

10.2.6 Complexity

Property 16. *For every $w_i = u_i.(v_i)^\omega$ computed by Alg. 8:*

$$|u_i| = max(\{0, \textit{pref}(ops), \textit{pref'}(ops), \text{ for all } \tau_i \stackrel{ops}{\rightarrow} \tau_j\}) \tag{10.1}$$

$$|v_i| = lcm(\{P(ops), P'(ops, w_j), \text{ for all } \tau_i \stackrel{ops}{\rightarrow} \tau_j\}) \tag{10.2}$$

$$|v_i| \leq lcm(\{T_k, \text{ for all } \tau_k\}) \tag{10.3}$$

Proof. Properties 10.1 and 10.2 directly come from the definition of the corresponding operations.

For part 10.3, let us first prove that for all $\tau_i \stackrel{ops}{\rightarrow} \tau_j$, $P(ops)|T_j$. Let h_{ops}^{-1} denote the inverse of function T_{ops}, ie $T_j = h_{ops}^{-1}(T_i)$. We prove by induction that $P(ops)|h_{ops}^{-1}(T_i)$ and so that $P(ops)|T_j$.

- The base of the induction is obviously true: $P(\epsilon) = 1$ and $1|T_j$;
- Then, the induction hypothesis is $P(ops)|h_{ops}^{-1}(T_i)$ for some ops. We first prove the induction step for case $\hat{*}\, k$. We have $h_{\hat{*} k.ops}^{-1}(T_j) = h_{ops}^{-1}(T_j)/k$, which implies $k|h_{ops}^{-1}(T_j)$. From the definition of gcd, there exists q, q', with $q \wedge q' = 1$ such that:
  - $q = P(ops)/gcd(P(ops), k)$;

- $q' = k/gcd(P(ops), k)$.

So:

- As $P(ops)|h^{-1}_{ops}(T_j)$ (from the induction hypothesis), then $q|(h^{-1}_{ops}(T_j)/gcd(P(ops), k))$;
- As $k|h^{-1}_{ops}(T_j)$, then $q'|(h^{-1}_{ops}(T_j)/gcd(P(ops), k))$.

And thus $qq'|(h^{-1}_{ops}(T_j)/gcd(P(ops), k))$, so $(qq' * gcd(P(ops), k))|h^{-1}_{ops}(T_j)$.

On the other hand:

$$
\begin{aligned}
P(*\hat{}k.ops) * k &= P(ops)/gcd(P(ops), k) * k \\
&= q * k \\
&= qq' * gcd(P(ops), k)
\end{aligned}
$$

So $(P(*\hat{}k.ops) * k)|h^{-1}_{ops}(T_j)$ and $(P(*\hat{}k.ops))|(h^{-1}_{ops}(T_j)/k)$.

- Then we prove the induction step for case $/\hat{}k$. We have: $P(/\hat{}k.ops) = kP(ops)$ and $h^{-1}_{/\hat{}k.ops} = kh^{-1}_{ops}(T_j)$. From the induction hypothesis $kP(ops)|kh^{-1}_{ops}(T_j)$, so the induction step is proved;
- The induction steps for cases $\sim>$, `when`, `tail`, `fby` and `::` are trivial, so the property is proved by induction.

Likewise, we can prove by induction that $P'_{ops}(w_j)|(|v_j|*T_j)$. Only the base of the induction changes: $P'_\epsilon(w_j) = |v_j|$ so obviously $P'_\epsilon(w_j)|(|v_j| * T_j)$.

From these two properties and from part 10.2, we have $|w_i| \leq lcm(|v_j| * T_j, T_j)$. So transitively $|w_i| \leq lcm(\{T_k, \text{ for all } \tau_k\})$. □

Property 17. *The complexity of Alg. 8 is $\mathcal{O}(|V| + |E| * |w_{max}|)$ where w_{max} denotes the deadline word which has the longest size in the task graph.*

Proof. The complexity of a topological sort for a graph $g = (V, E)$ is $\mathcal{O}(|V| + |E|)$. Our algorithm performs the same operations as a topological sort but in addition at each step (for each edge) it computes task deadline words. The operations it applies to compute deadline words are linear in $|w_{max}|$ so the complexity of the calculus is $\mathcal{O}(|V| + |E| * |w_{max}|)$. □

10.3 Schedulability analysis

As we consider critical systems, we would like to know before starting the system execution if the system is schedulable, that is to say if all real-time constraints will be met. From Corollary 3, we know that the system is schedulable if and only if the corresponding encoded task set is schedulable with EDF. We can therefore reuse existing results on the schedulability analysis of EDF.

We want to perform a schedulability analysis (also called a feasibility test) for a set of independent periodic tasks, with deadlines lower than task periods and with release dates different from 0, scheduled with EDF. A necessary and sufficient condition for such a system to be feasible was given in [LM80], which established that the feasibility of such a task set can be checked by examining the schedule computed by EDF on the time interval $[0, 2HP + max(r_i)]$ (where HP denotes the hyperperiod of the task set) and verifying that real-time constraints are met on this interval. The value HP can be exponential with respect to the periods of the system (the worst-case being when the periods of the system are co-prime) and the problem we consider was proved co-NP-hard in the strong sense in [BRH90]. In practice however, systems with co-prime periods do not occur that much and in many cases the test should not have such a dramatic complexity.

10.4 Inter-Task Communications

In this section we propose a communication scheme, which ensures that inter-task data communications respect the synchronous semantics of Sect. 4.6.

10.4.1 Ensuring communication consistency

A simple way to handle inter-task communications is to allocate a buffer in a global memory, the producer of the data writes to this buffer when it executes and the consumer reads from this buffer when it executes. To respect the synchronous semantics, we must ensure that, for each data-dependency:

1. The producer writes before the consumer reads.

2. As we deal with periodic tasks, we must also ensure that the current instance of the reader reads before the next instance of the writer overwrites the data produced by its previous instance.

The precedence encoding presented in the previous section ensures that for any data-dependency the producer of the data will be scheduled before the consumer of the data. So, we know that when a task is scheduled and starts its execution, all the data it consumes has already been produced and the requirement (1). is satisfied.

However, the precedence encoding does not ensure that the requirement (2). is satisfied. For a precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$, data produced by $\tau_i[n]$ may be consumed by $\tau_j[g_{ops}(n)]$ after $\tau_i[n+1]$ has started. This is illustrated in Fig. 10.5, which shows the schedule of two tasks related by extended precedences. $prio(\tau_i)$ denotes the priority affected to task τ_i. Vertical lines on the time axis represent task periods. Marks on the time axis represent task preemptions. An arrow from A at date t to B at date t' means that task B may read at time t' from the value produced by A at time t. In Fig. 10.5(a), 10.5(c) and 10.5(d), when $A[1]$ executes, it must not overwrite the data produced by $A[0]$ because it is consumed by $B[0]$ and $B[0]$ is not complete yet. Therefore, we need a buffer to keep the value of $A[n]$ after $A[n+1]$ has started. In Fig. 10.5(b), the same data is consumed several times but $A[1]$ can freely overwrite the data produced by $A[0]$, so no specific communication scheme is required. In Fig. 10.5(e) and Fig. 10.5(f), we only need to be careful with the initial value of the communication.

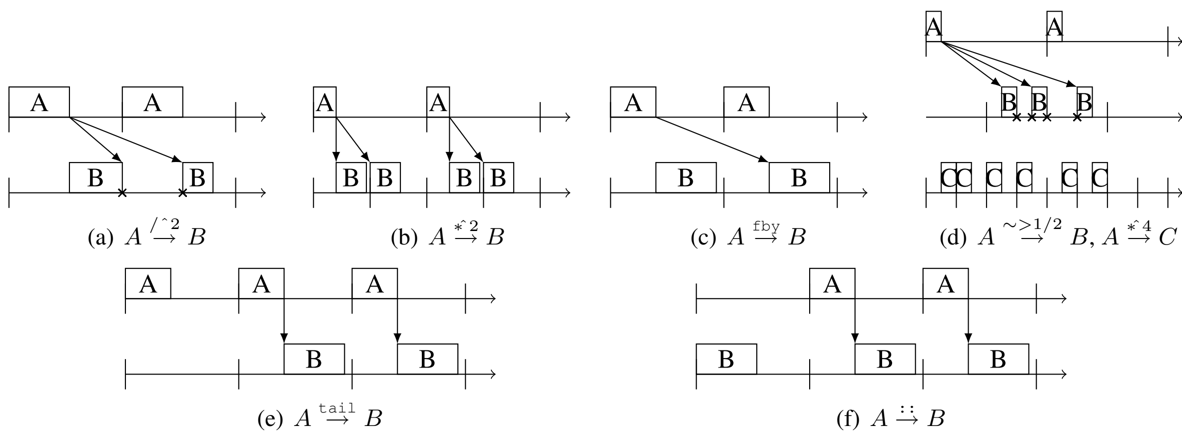


Figure 10.5: Communications for extended precedences, with $prio(A) > prio(C) > prio(B)$

10.4.2 Task instances communications

Definition

We mentioned that the definition of task instance precedences of Sect. 10.2.2 is an over-approximation of task instance data-dependencies. Task instance data-dependencies can be deduced from the definition of task instance precedences as follows:

Property 18. *Let $\tau_i[n] \rightarrow_d \tau_j[n']$ denote a data-dependency from $\tau_i[n]$ to $\tau_j[n']$. For any precedence $\tau_i \stackrel{ops}{\rightarrow} \tau_j$:*

$$\tau_i[n] \rightarrow_d \tau_j[n'] \Leftrightarrow n' \in [g_{ops}(n), g_{ops}(n+1)[$$

Proof. From the definition of g_{ops} and from the semantics of predefined operators. □

According to this definition, $\tau_i[n]$ may:

- Not be consumed at all: when $g_{ops}(n+1) = g_{ops}(n)$;
- Be consumed exactly once: when $g_{ops}(n+1) = g_{ops}(n) + 1$;
- Be consumed n times: when $g_{ops}(n+1) = g_{ops}(n) + n$.

This is illustrated in Fig. 10.6. The first line gives the values produced by the task τ_i and the following lines give the values consumed by the task τ_j for different extended precedences.

date	0	5	10	15	20	25	30	35	40	...
τ_i	x_0		x_1		x_2		x_3		x_4	...
$\tau_i \stackrel{*\hat{}2}{\rightarrow} \tau_j$	x_0	x_0	x_1	x_1	x_2	x_2	x_3	x_3	x_4	...
$\tau_i \stackrel{/\hat{}2}{\rightarrow} \tau_j$	x_0				x_2				x_4	...
$\tau_i \stackrel{\sim>1/2}{\rightarrow} \tau_j$		x_0		x_1		x_2		x_3		...
$\tau_i \stackrel{\texttt{tail}}{\rightarrow} \tau_j$			x_1		x_2		x_3		x_4	...
$\tau_i \stackrel{\texttt{tail}\ .0\ ::}{\rightarrow} \tau_j$	0		x_1		x_2		x_3		x_4	...

Figure 10.6: Data-dependencies between task instances

This implies the following data-dependencies:

- $\tau_i \stackrel{*\hat{}2}{\rightarrow} \tau_j$, we have (for $n = 0, 1, 2, \ldots$): $g_{*\hat{}2}(n) = 0, 2, 4, 6, \ldots$ Thus, $\tau_i[0] \rightarrow_d \tau_j[0]$, $\tau_i[0] \rightarrow_d \tau_j[1]$, $\tau_i[1] \rightarrow_d \tau_j[2]$, $\tau_i[1] \rightarrow_d \tau_j[3]$, ...
- $\tau_i \stackrel{/\hat{}2}{\rightarrow} \tau_j$, we have (for $n = 0, 1, 2, \ldots$): $g_{/\hat{}2}(n) = 0, 1, 1, 2, 2, 4, \ldots$ Thus, $\tau_i[0] \rightarrow_d \tau_j[0]$, $\tau_i[2] \rightarrow_d \tau_j[1]$, $\tau_i[2] \rightarrow_d \tau_j[4]$, ...
- $\tau_i \stackrel{\sim>1/2}{\rightarrow} \tau_j$, we have (for $n = 0, 1, 2, \ldots$): $g_{\sim>1/2}(n) = 0, 1, 2, 3, \ldots$. Thus, $\tau_i[0] \rightarrow_d \tau_j[0]$, $\tau_i[1] \rightarrow_d \tau_j[1]$, $\tau_i[2] \rightarrow_d \tau_j[2]$, ...
- $\tau_i \stackrel{\texttt{tail}}{\rightarrow} \tau_j$, we have (for $n = 0, 1, 2, \ldots$): $g_{\texttt{tail}\ 1/2}(n) = 0, 0, 1, 2, 3, \ldots$. Thus $\tau_i[1] \rightarrow_d \tau_j[0]$, $\tau_i[2] \rightarrow_d \tau_j[1]$, $\tau_i[3] \rightarrow_d \tau_j[2]$, ...
- $\tau_i \stackrel{\texttt{tail}\ .0\ ::}{\rightarrow} \tau_j$, we have (for $n = 0, 1, 2, \ldots$): $g_{\texttt{tail}\ .0\ ::}(n) = 1, 1, 2, 3, 4, \ldots$. Thus, $\tau_i[1] \rightarrow_d \tau_j[1]$, $\tau_i[2] \rightarrow_d \tau_j[2]$, $\tau_i[3] \rightarrow_d \tau_j[3]$, ...

Dependence words

As we did for deadline words, data-dependencies can be represented by finite repetitive patterns. Let $conso_{ops}(n) = g_{ops}(n+1) - g_{ops}(n)$ for all $n \in \mathbb{N}$. According to Property. 18, for any precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$, $conso_{ops}(n)$ denotes the number of instances of τ_j that consume data produced by $\tau_i[n]$ (possibly 0). We will prove that the sequence of values of $conso_{ops}$ can be represented by finite repetitive patterns, which we call *dependence words*, defined similarly to deadline words:

$$\begin{array}{rcl} w & ::= & u.(u)^\omega \\ u & ::= & i \mid i.u \end{array}$$

For a deadline word $w = u.(v)^\omega$, u is the prefix of w and $(v)^\omega$ denotes the infinite repetition of word v, called the repetitive pattern of w.

We now prove that the function $conso_{ops}$ is ultimately periodic and can thus be represented by a dependence word. The length of the prefix and the length of the repetitive pattern of $conso_{ops}$ are actually the same as those of Δ_{ops} (defined in Property. 14):

Property 19. *For all ops:*

$$\forall n \geq max(0, pref(ops)), conso_{ops}(n) = conso_{ops}(n + P(ops))$$

Proof. We proved previously that for all $n \geq pref(ops)$, $\Delta_{ops}(n, T_j, r_j) = \Delta_{ops}(n + Pops, T_j, r_j)$. Thus, for all $n \geq pref(ops)$:

$$\begin{aligned} & \Delta_{ops}(n, T_j, r_j) = \Delta_{ops}(n + P(ops), T_j, r_j) \\ \Leftrightarrow & g_{ops}(n)T_j - nT_{ops}(T_j) = g_{ops}(n + P(ops))T_j - (n + P(ops))T_{ops}(T_j) \\ \Leftrightarrow & T_j * (g_{ops}(n + P(ops)) - g_{ops}(n)) = P(ops)T_i \\ \Leftrightarrow & g_{ops}(n + P(ops)) - g_{ops}(n) = P(ops)T_i/T_j \end{aligned}$$

As $n \geq P(ops) \Rightarrow n + 1 \geq P(ops)$, this property is true for $n + 1$ also and we have:

$$\begin{aligned} & g_{ops}(n + P(ops)) - g_{ops}(n) = g_{ops}(n + 1 + P(ops)) - g_{ops}(n + 1) \\ \Leftrightarrow & g_{ops}(n + 1) - g_{ops}(n) = g_{ops}(n + 1 + P(ops)) - g_{ops}(n + P(ops)) \\ \Leftrightarrow & conso_{ops}(n) = conso_{ops}(n + P(ops)) \end{aligned}$$

□

Thanks to this property, we can compute $conso_{ops}$ as a dependence word, by computing each value of $conso_{ops}(n)$ on its prefix and on its period. For instance, for the example of Fig. 10.6:

- $\tau_i \overset{*\hat{}2}{\rightarrow} \tau_j$: we have $conso_{ops} = (2)^\omega$, meaning that each value produced by τ_i is consumed by two instances of τ_j;
- $\tau_i \overset{/\hat{}2}{\rightarrow} \tau_j$: we have $conso_{ops} = (1.0)^\omega$, meaning that only one out of two successive values produced by τ_i is consumed by τ_j;
- $\tau_i \overset{\sim>1/2}{\rightarrow} \tau_j$: we have $conso_{ops} = (1)^\omega$, meaning that each value produced by τ_i is consumed by an instance of τ_j;
- $\tau_i \overset{\texttt{tail}}{\rightarrow} \tau_j$: we have $conso_{ops} = 0.(1)^\omega$, meaning that the first value produced by τ_i is not consumed;
- $\tau_i \overset{\texttt{tail}.0::}{\rightarrow} \tau_j$: we also have $conso_{ops} = 0.(1)^\omega$.

Lifespans

We define the *lifespan* of a data-sample produced by a task as the interval of time from the date at which it is produced to the date at which it can last be consumed. More formally:

Definition 14. *For any precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$, for any n, the lifespan of $\tau_i[n]$ is:*

$$sp_{i,j}(n) = \begin{cases} [R_i[n], R_i[n][& \textit{if } conso_{ops}(n) = 0 \\ [R_i[n], R_j[g_{ops}(n)] + conso_{ops}(n) * T_j[& \textit{otherwise} \end{cases}$$

For an interval $I = [a, b[$, let $|I| = b - a$ denote the size of I.

Property 20. *For all precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$:*

$$\forall n \geq max(0, pref(ops)), |sp_{i,j}(n)| = |sp_{i,j}(n + P(ops))|$$

Proof. We have:

$$R_j[g_{ops}(n)] + conso_{ops}(n) * T_j = R_i[n] + \Delta_{ops}(n, T_j, r_j) + conso_{ops}(n) * T_j$$

So (the integer part of the division replaces the *if-then-else* of Def. 14):

$$|sp_{i,j}(n)| = \lceil \frac{conso_{ops}(n)}{conso_{ops}(n) + 1} \rceil * (\Delta_{ops}(n, T_j, r_j) + conso_{ops}(n) * T_j)$$

Thus our property. □

10.4.3 Communication protocol

We propose a communication protocol based on dependence words, which ensures that the inputs of a task remain available until its deadline and so which satisfies the requirement (2). of the previous section. An important feature of this protocol is that it requires no specific synchronization primitives (like semaphores), which allows to use a minimal OS. Still, the protocol strictly respects the formal semantics of the language and ensures that the functional behaviour of the program is exactly the one specified in Sect. 4.6 (it ensures the functional determinism of the system).

The protocol is based on dependence words and lifespans. A communication buffer is allocated for each precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$. Dependence words tell us which instance of τ_i must write in the buffer and which instances of τ_j must read the corresponding value. Lifespans tell us which instances of τ_i can write in the same cell of the buffer: if the lifespans of $\tau_i[n]$ and $\tau_i[n']$ have an intersection, then they cannot be stored in the same cell of the buffer.

Number of cells of the communication buffer

Let $cell_i[n]$ denote the buffer cell in which $\tau_i[n]$ will write. The number of cells of the buffer must be such that for all n, n' such that $sp_{i,j}(n)$ and $sp_{i,j}(n')$ intersect, $cell_i[n]$ and $cell_i[n']$ can be chosen so that $cell_i[n] \neq cell_i[n']$. Indeed, if $sp_{i,j}(n)$ and $sp_{i,j}(n')$ intersect, then $\tau_i[n']$ can start its execution while some instances of τ_j still need the value produced by $\tau_i[n]$, so $\tau_i[n]$ and $\tau_i[n']$ cannot write their values in the same cell. For a given lifespan, we can bound the number of lifespans that intersect it as follows:

Property 21. *For all $\tau_i \overset{ops}{\rightarrow} \tau_j$, for all n:*

$$|\{sp_{i,j}(n'), sp_{i,j}(n') \cap sp_{i,j}(n) \neq \emptyset\}| \leq \lceil sp_{i,j}(n)/T_i \rceil$$

Proof. Let $[a_n, b_n[$ denote $sp_{i,j}(n)$. Then:

$$\begin{aligned}[a_n, b_n[\cap[a_{n'}, b_{n'}[\neq \emptyset &\Leftrightarrow a_{n'} < b_n \\ &\Leftrightarrow R_i[n'] < b_n\end{aligned}$$

As τ_i is periodic, we have for all p, $R_i[p+1] = R_i[p] + T_i$. Let k be such that $R_i[n'] = R_i[n] + kT_i$. Then:

$$\begin{aligned}[a_n, b_n[\cap[a_{n'}, b_{n'}[\neq \emptyset &\Leftrightarrow R_i[n] + kT_i < b_n \\ &\Leftrightarrow kT_i < b_n - R_i[n] \\ &\Leftrightarrow k < \lceil (b_n - R_i[n])/T_i \rceil \\ &\Leftrightarrow k < \lceil |sp_{i,j}(n)|/T_i \rceil\end{aligned}$$

Thus our property. □

For any precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$, as the sequence of values of $|sp_{i,j}(n)|$ is ultimately periodic (Property 20), we can choose the size of the corresponding communication buffer as follows:

$$\max_{0 \leq n \leq pref(ops)) + P(ops)} (\lceil |sp_{i,j}(n)|/T_i \rceil)$$

The protocol

Now, we define the communication protocol for an extended precedence $\tau_i \overset{ops}{\rightarrow} \tau_j$. Let $dep_{i,j}$ denote the dependence word for this precedence. Let $buf_{i,j}$ be the communication buffer for this precedence (the size of which is computed as described above). First, we compute the sequence of instances of τ_i that must write to $buf_{i,j}$ as described in Alg. 9. This is represented by the sequence $write$: $\tau_i[n]$ writes to $buf_{i,j}$ only if $write[n]$ is true. The communication protocol for the writer is then described in Alg. 10. This protocol tests whether the current instance of the producer must write to the buffer or not. If so, it writes to the cell $buf_{i,j}[cell]$ and the next time it will write in $buf_{i,j}[cell + 1]$, modulo the size of the buffer.

Algorithm 9 Computing when the producer must write, for $\tau_i \overset{ops}{\rightarrow} \tau_j$

```
1: write ← ϵ;
2: for k = 0 to max(0, pref(ops)) + P(ops) − 1 do
3:     if dep_{i,j}[k] ≥ 1 then
4:         write ← write.true
5:     else
6:         write ← write.false
7:     end if
8: end for
```

We compute when the reader must change the cell of the buffer from which it reads as described in Alg. 11: $\tau_j[n]$ changes the cell of the buffer from which it reads only if $change[n]$ is true. The communication protocol for the reader is then described in Alg. 12. Each instance of the consumer reads a value from the current cell of the buffer. After each reading, the protocol tests whether the next instance must read from a different cell of the buffer and if so, it advances to the next cell modulo the size of the buffer.

Algorithm 10 Communication protocol for $\tau_i \stackrel{ops}{\to} \tau_j$, on the producer side

```
1: inst ← 0; cell ← 0;
2: while true do
3:     if write[inst] then
4:         Write to buf_{i,j}[cell]
5:         cell ← (cell + 1) mod |buf_{i,j}|
6:     end if
7:     inst ← inst + 1
8: end while
```

Algorithm 11 Computing reading patterns for $\tau_i \stackrel{ops}{\to} \tau_j$

```
1: change ← ϵ;
2: for k = 0 to max(0, pref(ops)) + P(ops) − 1 do
3:     if dep_{i,j}[k] ≥ 1 then
4:         for i = 0 to dep_{i,j}[k] − 2 do
5:             change ← change.false
6:         end for
7:         change ← change.true
8:     end if
9: end for
```

Algorithm 12 Communication protocol for $\tau_i \stackrel{ops}{\to} \tau_j$, on the consumer side

```
1: inst ← 0; cell ← 0
2: while true do
3:     Read from buf_{i,j}[cell]
4:     if change[inst] then
5:         cell ← (cell + 1) mod |buf_{i,j}|
6:     end if
7:     inst ← inst + 1
8: end while
```

10.4.4 Example

We illustrate the communication protocol on the examples of Fig. 10.7 (arrows denote data-dependencies). First, for the precedence $\tau_i \overset{*\hat{}3./\hat{}2}{\rightarrow} \tau_j$, with $T_i = 3, T_j = 2$, we have:

- $dep_{i,j}(n) = (2.1)^\omega$;
- $sp_{i,j}(n) = [0, 4], [3, 6], [6, 10], [9, 12], \ldots$
- $|buf_{i,j}| = 2$;
- $write = (true)^\omega$;
- $change = (false.true.true)^\omega$.

So:

- The sequence of cells τ_i writes to is: 0, 1, 0, 1, 0, 1, ...
- The sequence of cells τ_j reads from is: 0, 0, 1, 0, 0, 1, ...

Then, for precedence $\tau_i \overset{*\hat{}2./\hat{}3}{\rightarrow} \tau_j$, with $T_i = 2, T_j = 3$, we have:

- $dep_{i,j}(n) = (1.1.0)^\omega$;
- $sp_{i,j}(n) = [0, 3[, [2, 6[, [4, 4[, [6, 9[, [8, 12[, \ldots$
- $|buf_{i,j}| = 2$;
- $write = (true.true.false)^\omega$;
- $change = (true)^\omega$.

So:

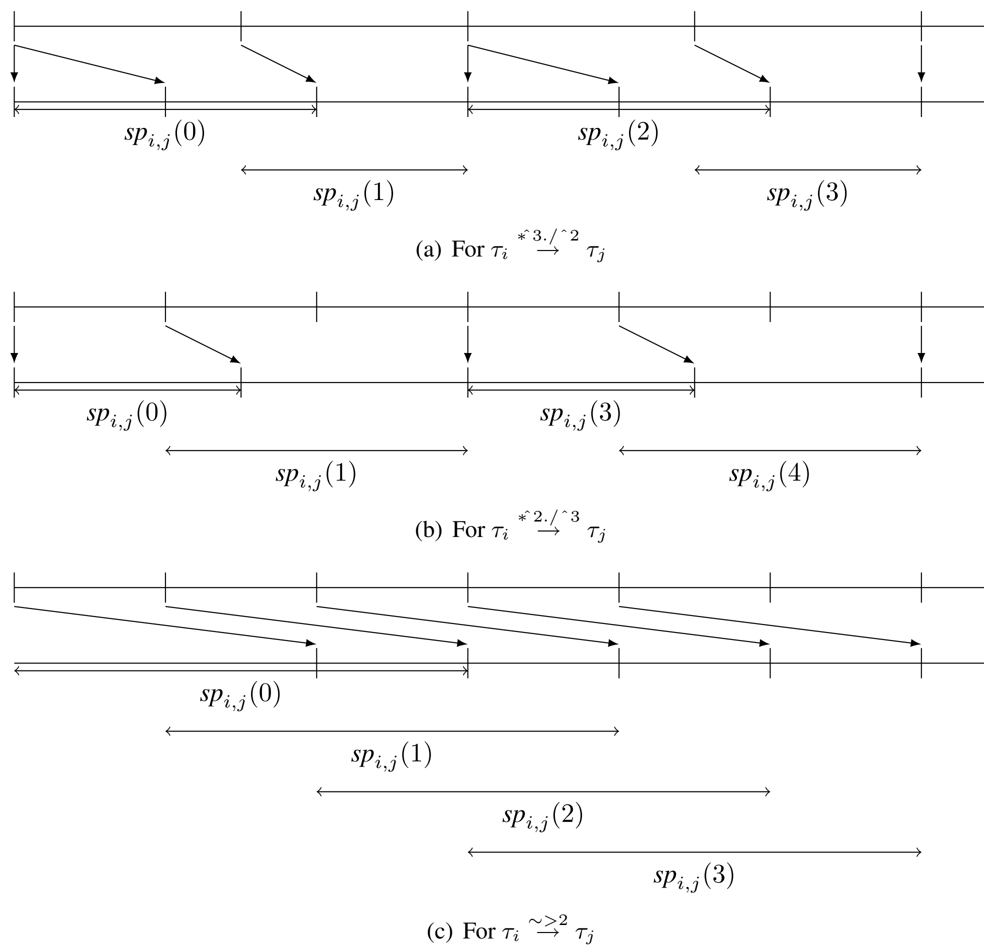
- The sequence of cells τ_i writes to is: 0, 1, none, 0, 1, none, ...
- The sequence of cells τ_j reads from are: 0, 1, 0, 1, 0, 1, ...

Finally, for precedence $\tau_i \overset{\sim>2}{\rightarrow} \tau_j$, with $T_i = T_j = 2$, we have:

- $dep_{i,j}(n) = (1)^\omega$;
- $sp_{i,j}(n) = [0, 6[, [2, 8[, [4, 10[, [6, 12[, [8, 14[, \ldots$
- $|buf_{i,j}| = 3$;
- $write = (true)^\omega$;
- $change = (true)^\omega$.

So:

- The sequence of cells τ_i writes to is: 0, 1, 2, 0, 1, 2, ...
- The sequence of cells τ_j reads from are: 0, 1, 2, 0, 1, 2, ...



(a) For $\tau_i \overset{*\hat{}3./\hat{}2}{\rightarrow} \tau_j$

(b) For $\tau_i \overset{*\hat{}2./\hat{}3}{\rightarrow} \tau_j$

(c) For $\tau_i \overset{\sim\geq 2}{\rightarrow} \tau_j$

Figure 10.7: Data-dependencies

10.5 Summary

In this chapter we have detailed how to translate the dependent task set extracted from a program into an equivalent independent task set.

- Precedences are encoded into task real-time attributes by adapting the technique proposed in [CSB90] to the case of extended precedences between tasks of different periods;
- The independent task set can then be scheduled optimally with EDF (scheduling the independent task set with EDF is equivalent to scheduling the dependent task set);
- Communication consistency is ensured by a non-blocking communication protocol based on data-buffering;
- Schedulability analysis is standard and relies on existing results.

In the next chapter, we show that the independent task set can easily be translated into C code.

Chapter 11

Code generation

The last step of the compilation process translates the independent task set obtained for a program into C code. The real-time aspects of the C program are handled by the real-time primitives of the underlying OS. We use the real-time extensions of the POSIX standard [POS98], to be as independent from the OS choice as possible. Data-communications are handled by the communication protocol we described in the previous chapter. The task set is implemented as a set of communicating threads, contained in a single file. The file contains:

1. The allocation of the inter-task communication buffers.
2. One function (or more precisely one function pointer) for each task, which describes the computations performed by the task. This corresponds to the "step" function of classic synchronous languages.
3. The `main` function, which creates one thread for each task, initializes the scheduler and attaches the threads to the scheduler.

11.1 Communication buffers

We allocate one buffer for each communication. The size of the buffer is computed as described in Sect. 10.4.3. The buffer is named after its producer and its consumer. As an example, we consider the simple multi-periodic communication loop with data-sampling communication patterns presented previously in Sect. 5.1:

```
imported node F(i, j: int) returns (o, p: int) wcet 20;
imported node S(i: int) returns (o: int) wcet 30;
sensor i wcet 1;
actuator o wcet 1;

node sampling(i: rate (100, 0)) returns (o)
  var vf, vs;
let
  (o, vf)=F(i, (0 fby vs)*^3);
  vs=S(vf/^3);
tel
```

As explained in Sect. 9.2.5, the reduced task graph for this example is that given in Fig. 11.1.

The buffers allocated for this program are given below:

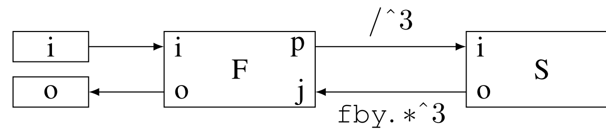


Figure 11.1: Reduced task graph

```
int S_o_F_j[2];
int i_F_i;
int F_o_o;
int F_p_S_i;
```

For instance, the buffer `i_F_i` corresponds to the communication from the sensor `i` to the input `i` of `F` (which we will denote `F.i`). For the communication from `S.o` to `F.j`, we have the declaration: `int S_o_F_j[2]`. The size of the buffer is 2 due to the delay between the two tasks.

11.2 Task functions

The function generated for a task mainly consists of an infinite loop, that wraps the function of the corresponding imported node with the code of the communication protocol. One step of the loop corresponds to the execution of one instance of the task. When it reaches the end of the current step of the loop, the function signals to the scheduler that the current task instance completed its execution so that it can schedule another task.

For instance, the code generated for the node `S` of our example is the following:

```
void *S_fun(void * arg)
{
  int instance=0;
  int cell_to_S_o_F_j=0;
  int S_o;

  while (1)
  {
    S_o=S(F_p_S_i);
    S_o_F_j[cell_S_o_F_j]=S_o;
    cell_to_S_o_F_j=(cell_to_S_o_F_j+1)%2
    instance++;
    invoke_scheduler (0);
  }
}
```

- The variable `cell_to_S_o_F_j` tells in which cell of the buffer the value of `S.o` must be copied for `F.j`;
- The first instruction of the loop calls the external function provided by the programmer for node `S` ;
- Then, `S` alternatively copies the value produced by this function call in the cell 0 or in the cell 1 of the communication buffer for `F` (`S_o_F_j[cell]=S_o`). This way, `S` can write as soon as it executes and `F` can still read the previous value produced by `S` when it executes;

- The function call `invoke_scheduler (0)` signals the end of the task instance execution to the scheduler.

On the other hand, the code generated for the node `F` is the following:

```
void *F_fun(void * arg)
{
  int cell_from_S_o_F_j=1;
  int change_S_o_F_j={0,0,1};
  int write_F_p_S_i[3]={1,0,0};
  int instance=0;
  struct F_outs_t F_fun_outs;
  S_o_F_j[1]=0;

  while (1)
  {
    F(i_F_i,S_o_F_j[cell_from_S_o_F_j],&F_fun_outs);
    if (change_S_o_F_j[instance%3])
      cell_S_o_F_j=(cell_from_S_o_F_j+1)%2
    F_o_o=F_fun_outs.o;
    if(write_F_p_S_i[instance%3])
      F_p_S_i=F_fun_outs.p;
    instance++;
    invoke_scheduler (0);
  }
}
```

- The variable `cell_from_S_o_F_j` tells from which cell of the buffer the value of `S.o` must be read. Notice that we start by reading the initialisation value of the delay contained by the cell 1 of the buffer;
- The variable `write_F_p_S_i` tells whether the buffer `F_p_S_i` must be updated or not ;
- The first instruction of the loop calls the external function provided for `F`;
- The second argument of the function is read from the communication buffer `S_o_F_j`. According to variable `change_S_o_F_j`, each 3 iterations, `F` must change the cell from which it reads;
- As `F` returns several values, the values are produced in a structure passed as an additional argument to the function (`&F_fun_outs`);
- The field `o` of the structure is copied at every instance of `F` for the actuator `o`;
- As described by variable `write_F_p_S_i`, the field `F_fun_outs.p` is copied in the buffer for `S` only once every 3 instances of `F`.

We produce a similar code for the sensor `i`:

```
void *i_fun(void * arg)
{
  int instance=0;
```

```
  int i;

  while (1)
  {
    i=input_i();
    i_F_i=i;
    instance++;
    invoke_scheduler (0);
  }
}
```

And also for the actuator o:

```
void *o_fun(void * arg)
{
  int instance=0;
  while (1)
  {
    output_o(F_o_o);
    instance++;
    invoke_scheduler (0);
  }
}
```

11.3 Task real-time attributes

We define data structures to describe the real-time attributes of a task. We first define a type structure to describe deadline words:

```
typedef struct dword {
  struct timespec *ddpref;
  int prefsize;
  struct timespec *ddpat;
  int patsize
} dword_t;
```

The field ddpref corresponds to the prefix of the deadline word and prefsize is the size of the prefix. The field ddpat correspond to the repetitive pattern of the deadline word and patsize is the size of the pattern.

The real-time attributes of a task are then defined as follows:

```
struct edf_sched_param {
  char* name;
  struct timespec period;
  struct timespec initial_release;
  dword_t dword;
};
```

11.4 Threads creation

The main function creates one thread for each task:

- The computations performed by a thread are those described in the function pointer previously generated for the corresponding task ;
- The real-time attributes of the task are specified as attributes of the task.

For instance, the task corresponding to the imported node call `F` is created as follows:

```
pthread_t tF;
pthread_attr_t attr;
struct edf_sched_param user_param;
pthread_attr_setappschedparam (&attr, &user_param); // system dependent
pthread_create (&tF, &attr, F_fun, NULL) );
```

The way the attributes of a task are attached to the task actually depends a lot on the target OS. Here, we give an example with MARTE OS [RH02].

The threads are then attached to the scheduler. This part also depends on the the chosen OS. For instance, in MARTE OS, the scheduler is itself implemented as a thread and a specific primitive allows to attach the threads to the scheduler thread. This will be detailled in the next chapter.

11.5 Summary

In this chapter, we have shown that the translation of the set of independent tasks generated for a program into C code is pretty straightforward. This translation is the last phase of the compilation process.

The next chapter describes a prototype implementation of the complete compilation scheme. In particular, it details the implementation of an EDF scheduler supporting deadline words.

Chapter 12

Prototype implementation

12.1 The compiler

A prototype of the compiler has been implemented in OCAML [Ler06]. It supports the language defined in Sec. 4.5 and implements the complete compilation chain, with proper error handling, in approximately 4000 lines of code. The current version of the prototype does not support array constructs and is restricted to constant parameters for rate transition operators (instead of static parameters like $3 * 2$ for instance). The compiler performs the following sequence of computations:

1. Syntax analysis: parser and lexer
2. Static analyses: typing and clock calculus
3. Translation into a reduced task graph: program expansion, task graph extraction, task graph reduction
4. Deadline calculus
5. Communication protocol computation
6. Code generation

The number of lines of code for each part of the compiler is given in Fig. 12.1.

Syntax analysis	350
Typing	400
Clock calculus and clock data types	1000
Graph extraction and reduction	600
Deadline calculus and deadline words data types	400
Communication protocol	200
C code generation	600
Base data types and utility functions	400
Command line processing and main file	100
Total	4050

Figure 12.1: The different parts of the prototype

12.2 Earliest-Deadline-First scheduler with deadline words

We chose to prototype the scheduler using MARTE Operating System [RH02], as this Operating System was designed to ease the implementation of application-specific schedulers while remaining close to the POSIX model. This OS also has the advantage of being executable on top of a standard LINUX station, which simplifies the debugging and test phases. The implementation can easily be adapted to another RTOS, as long as the RTOS allows to dynamically modify the priority of a task.

12.2.1 Application-defined schedulers in MARTE OS

MARTE OS supports application-defined scheduling policies implemented by a special kind of threads (scheduler threads), which can activate or suspend other threads. In this approach, scheduling is handled hierarchically. The root scheduler (the system scheduler) follows a classic fixed priority policy. Application-defined schedulers allow the definition of more complex, dynamic priority, application-dependent policies, and are themselves scheduled by the root scheduler. Threads can then either be scheduled by the root scheduler (called system-scheduled threads) or by application-defined schedulers (called application-scheduled thread). An application-scheduled thread always has lower priority than its scheduler.

MARTE OS proposes an API that extends the POSIX standard with a small set of extra functionalities, which allow an application to:

- Handle scheduling events;
- Execute scheduling actions;
- Create schedulers and application-scheduled threads.

An application scheduler is activated only when a *scheduling event* occurs, instead of actively testing the set of threads it must schedule at regular intervals of time to take scheduling decisions. Amongst other things, scheduling events will signal the scheduler that a thread has been created, has terminated, is blocked or is ready. Threads can also explicitly invoke the scheduler using a specific scheduling event.

The code of the scheduler is usually a loop where it waits for a scheduling event to be notified to it by the system, and then performs scheduling actions. Scheduling actions cause application-scheduled threads to be activated or suspended. They are prepared and stored in a list by the scheduler thread and then reported to the system. After reporting the scheduling actions, the scheduler becomes inactive until the next scheduling event.

12.2.2 EDF implementation

The MARTE distribution provides an implementation of the EDF scheduling policy that follows the principles described above. To summarize, each time a new task instance is released, the scheduler computes its absolute deadline and compares it with other ready task instances (the task instances that are released and unfinished), to sort ready task instances as a queue ordered by increasing deadlines. If the head of the queue (the most urgent task instance) is not the currently running task instance, the scheduler suspends the currently running task instance and activates the task instance at the head of the queue. Before extending this implementation to support deadline words, we had to make a few simplification to extend it so that it supports initial release dates greater than zero and relative deadlines lower than the task period.

12.2.3 Deadline words support

The support of deadline words requires few modifications to be made to the existing EDF scheduler. First, we define an auxiliary function that computes the value of the nth unitary deadline of a deadline word:

```
void get_nth_ud(struct timespec* dd, dword_t dw, int n)
{
  int pos;
  if (n<dw.prefsize)
    *dd=dw.ddpref[n];
  else {
    pos=(n-dw.prefsize)%dw.patsize;
    *dd=dw.ddpat[pos];
  }
}
```

Then, we only need to modify the function that programs the next instance of a task when the current instance completes. For a task τ_i, the attributes of which are described by the value `t_data`, the release date and the deadline of its new instance are computed as described below ($D_i[n] = R_i[n] + d_i[n]$):

```
dword_t dw = t_data->dword;
struct timespec dd;
t_data->instance++;
get_nth_ud(&dd,dw,instance);
incr_timespec (&t_data->next_release, &t_data->period);
add_timespec (&t_data->next_deadline, &t_data->next_release, &dd);
```

The function `incr_timespec(t1,t2)` increments `t1` by `t2` and the function `add_timespec(t1,t2,t3)` sets `t1` to `t2+t3`.

12.3 Summary

In this chapter, we described the implementation of a prototype supporting the compilation scheme described earlier:

- The compiler was implemented in about 4000 lines of code of OCAML;
- An EDF scheduler extended to support deadline words was implemented for the MARTE Operating System, which only required minor modifications to the existing EDF scheduler included in the OS distribution.

The next chapter concludes on the compilation scheme proposed in this part of the dissertation.

Chapter 13

Conclusion on language compilation

We detailed the compilation of the language defined in the previous part of this dissertation. A program is first translated into a set of dependent real-time tasks, related by precedence constraints. The dependent task set can then be translated into an equivalent set of independent tasks. To this intent, we showed that task precedences can be encoded in the real-time attributes of the tasks. Scheduling the encoded independent task set is equivalent to scheduling the original dependent task set. Therefore, EDF provides an optimal scheduling policy for our systems. Once precedences are encoded, we proposed a communication protocol based on data-buffering, which ensures that inter-task communications respect the synchronous semantics defined for the program. An important feature of the protocol is that it does not require synchronization primitives such as semaphores. The independent task set can then easily be translated into a C program and can be executed on an existing RTOS, on the condition that the RTOS supports dynamic priority scheduling.

Several approaches have been proposed to translate synchronous programs into a set of communicating tasks, prior to our work. Automatic distribution of LUSTRE programs into several "computation locations" has been studied in [GN03, GNP06], where different computation location can either correspond to different processors or different tasks. In this approach, the distribution is driven by the clocks of the program. The programmer classifies the clocks of the program into different partitions. A program is then generated for each partition, it contains the computations of all the expressions the clock of which belongs to this partition. Computation locations are synchronized by blocking communications (semaphores). A similar approach was proposed for SIGNAL [ALGM96]. A SIGNAL program is translated into a graph representing the clock hierarchy of the program, where each vertex of the graph corresponds to the computations that must be performed for a given clock of the hierarchy and edges correspond to dependencies between the vertices. This graph can then be partitioned into several subgraph, each subgraph is translated into sequential code and encapsulated into a task. Finally, the problem of preserving the synchronous semantics of a set of concurrent task scheduled with a preemptive policy has been studied in [STC06, Sof06]. The authors propose a buffering protocol called Dynamic Buffering Protocol similar to ours. The protocol allows to compute a sufficient number of cells[2] required for the communication buffer and determines for each task instance in which cell of the buffer the task instance must write (for the producing task) or read (for a consuming task). The computation of the priorities of the tasks generated for a program is out of the scope of all these approaches, while we describe how to encode task precedences such that task priorities can be affected automatically by an EDF scheduler.

A lot of work has been done on the problem of scheduling multi-rate data-flow programs in the context of SDF graphs, for instance [LM87a, BBHL93, ZUE00, ODH06]. In particular, [ZUE00] studies the implementation of SDF graphs with a dynamic scheduler using preemption, which may seem close

[2]If the tasks are all periodic, the number of cells computed by the protocol is proved minimal.

to our work. However, as we have seen in Chap. 1, SDF operations are *not* periodic. As a consequence, these results cannot be directly applied to our problem.

Part III

Case Study

Chapter 14

The Flight Application Software: Current Implementation

Before studying the implementation of the FAS with our language, we give an overview of the real implementation of the system provided by EADS Astrium Space Transportation.

The control system of the Automated Transfer Vehicle is to this date the most complex critical embedded software developed by EADS Astrium Space Transportation (it is also the most complex space software developed for ESA). The software consists of about 450.000 lines of ADA code, split into approximately 30 high level software functionalities (such as attitude control, navigation, rendez-vous with the International Space Station, ...).

The control system is made up of two parts: the Flight Application Software (FAS) and the Mission Safing Unit (MSU). The FAS is the main part of the system, it handles all the software functionalities of the system as long as no fault is detected. The MSU watches the FAS to detect if it is faulty. If a fault occurs, the MSU disables the FAS and starts a safety procedure, which stops the current ATV actions and moves it to a safe orbit, with respect to the station, waiting for further instructions from the ground station. In the rest of this chapter, we give an overview of the implementation of the FAS by EADS Astrium Space Transportation.

14.1 Hardware architecture

The FAS is a mission-critical system, which means that any failure in the system may cause the vehicle to be lost. In order to make the system more robust, the computations performed by the FAS are replicated on three identical processors called Data Processing Units, as shown in Fig. 14.1.

The DPUs are activated synchronously every 100ms by a clock signal transmitted by the 1553 communication bus. The clocks of the different processors are independent, which may cause some jitter between the production dates of the same output on different processors. As processors are resynchronized every 100ms, this jitter remains small (no more than 1ms).

The three DPUs acquire the same inputs, perform the same computations and produce the same outputs. All the inputs and all the critical outputs (except the telemetry) are voted. The voting mechanism compares the inputs received and the outputs produced by the different processors. Transfers required for the votes are performed by dedicated connections between the processors. If two processors agree on a value and the third one disagrees, the value produced by the third processor is considered erroneous. If three errors are detected on the same processor in less than 1s, the processor is declared faulty and is ignored until a correction is performed by the ground station. The MSU initiates the safety procedure when two processors are declared faulty.

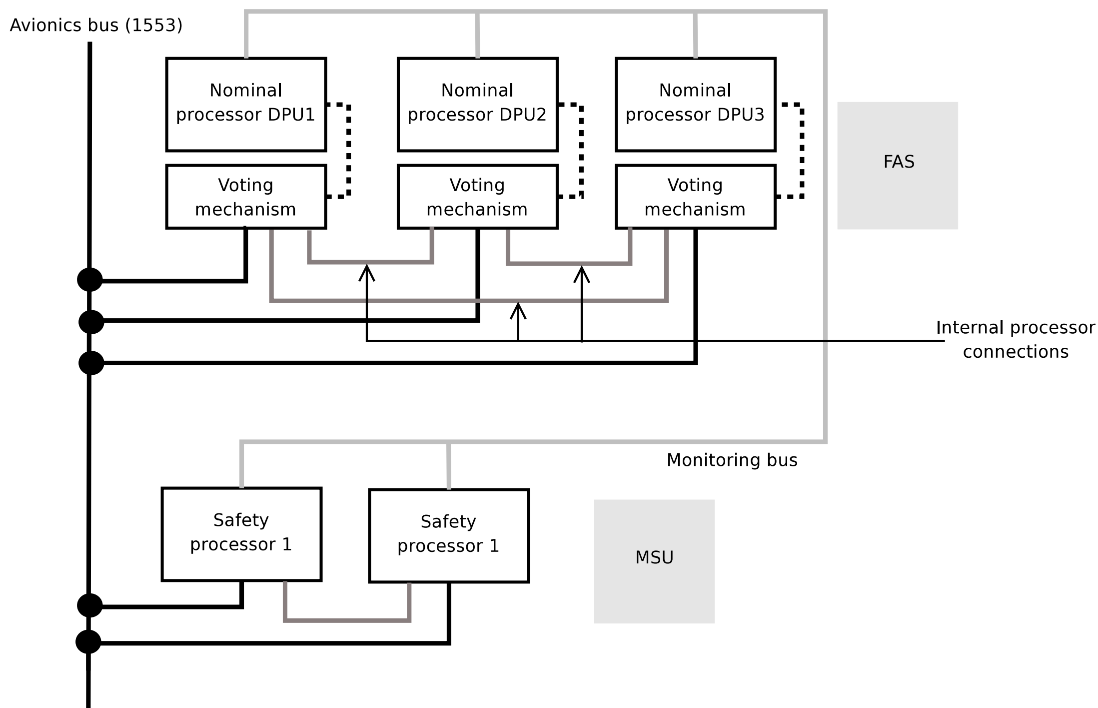


Figure 14.1: Hardware architecture of the ATV (FAS+MSU)

14.2 Software architecture

The 30 software functionalities of the FAS are grouped into three real-time tasks. The task frequencies are 0.1Hz, 1Hz and 10Hz. The tasks are scheduled concurrently by a Rate-Monotonic scheduler. The frequency of the faster task (10Hz) is defined by the clock signal occurring every 100ms on the 1553 bus.

The computations performed by each task consist of a sequence of operations called *services*. The sequence of services of each task is computed statically and *manually* by the FAS developers. The 10Hz task does not only contain services activated with a frequency of 10Hz, it also contains slower services, for instance services with a frequency of 1Hz, which are activated only one out of 10 successive instances of the 10Hz task. The wcet of each service is very stable, thanks to algorithmic simplifications. For instance loops always execute the same number of steps.

The partial static scheduling of the system leads to a very predictable behaviour and to minimal task jitters. The main drawback to this approach is that it is not very flexible. Small changes in the system, such as increasing the wcet of a service to include new functionalities for instance, can sometimes require to completely recompute the schedule of the tasks. The automation of this scheduling would clearly simplify the design process.

14.3 Bus communications

The communications on the 1553 bus follow a pattern called *major frame*, which is repeated with a period of 10s. Communications can have a frequency of 0.1Hz, 1Hz, 2Hz or 5Hz. A major frame is then split into 10 *minor frames* of 1s each. A minor frame is then split into 10 *processing frames* (PCF) of 100ms each. Finally, each PCF is split into 8 subframes of 12.5ms each.

Each subframe corresponds to 8 transfers on the 1553 bus (either read or write). All the PCFs have

the same format, for instance input acquisitions with a rate faster than 1Hz are performed in subframe 1, while input acquisitions with a rate slower than 1Hz are performed in subframe 3. Each input or output can only be transferred once in a PCF (ie not several times in different subframes). The major frame describes for a cycle of 10s which communications are performed in which slots.

14.4 Summary

In this chapter we described the current implementation of the FAS by EADS Astrium Space Transportation. The architecture of the system is simplified as much as possible to produce a predictable behaviour. We can see that the implementation of the system would greatly benefit from the automation of the scheduling of the system. The next chapter shows that this can be addressed with our language.

Chapter 15

Case study Implementation

In this chapter we study the implementation of a more detailed version of the Flight Application Software presented in the Introduction of this dissertation. We show that the language has sufficient expressiveness to specify the different aspects of the system and that the compiler supports easily such a larger example. Due to the complexity of the system, our objective is not to provide a complete implementation of the system but only to show that such an implementation is feasible. We emphasize the main difficulties for programming the system and detail how they can be solved with our language.

15.1 System specification

Our implementation is mainly based on two design documents provided by EADS Astrium Space Transportation. The first one is a chapter of the ATV-AS-ADD document, which specifies the real-time constraints of the different services of the FAS. The aim of this chapter of the document is to specify the (partial) order between the services of the system. For each service, it provides:

- The name of the service;
- The task that contains the service;
- The frequency of the service. The document also specifies acyclic services (activated only for a certain condition);
- The transfers between the service and the bus;
- The services before which it must be executed;
- The services after which it must be executed;
- Other information not relevant to our implementation.

From this document, we extract:

- The different services of the system;
- The frequency of each service: As we mentioned in the previous chapter, the frequency of the task that contains a service can be different from the real frequency of the service. In our case, we are only interested in the real frequency of the service. We discard asynchronous services as they are out of the scope of our language;

- Data-dependencies: To simplify, we consider that precedence constraints between services (when a service must be called before or after another) are always due to data-dependencies. Thus, for each precedence we have a data-dependency from the source service of the precedence to the destination service of the precedence. The document does not provide the complete data-flow between services of the system, which is very complex. Therefore, our implementation only considers part of the data-flow of the real system;

- System inputs/outputs: In the context of this case study, values read from and written to the bus are considered as inputs and outputs of the system. From the point of view of the program, the bus is part of the environment of the software.

The second document on which our implementation is based is the *ATV Data Bus Profile Definition*, which describes the bus traffic related to the DPUs. Among other things:

- It specifies for each bus transfer if the transfer is an input or an output of the DPUs;

- It details the structure of a major communication frame, specifying during which sub-frames transfers take place and at which frequency.

From this document, we extract:

- Inputs or outputs: We can determine whether the bus transfers specified in the previous documents correspond to inputs or outputs of the system;

- The period of inputs or outputs: Transfer frequencies directly provide the period of each input or output of the system;

- The phase of the inputs or outputs: For transfers that do not take place in every PCF (transfers of rate 1Hz or 0.1Hz), the PCF number determines the phase of the corresponding input/output. For instance, if an input has a rate of 1Hz and occurs in frame 2, its phase is 200ms.

15.2 Implementation overview

We write a program for a simplified version of the FAS based on the information detailed in the previous section. The code of the main node of the program is given in Appendix. A (the name of the services and of the program input and outputs have been changed for confidentiality reasons). To summarize:

- Each service is implemented as an imported node. To simplify, we only have one output for each imported node. For each imported node, we have one input for each service that precedes the service corresponding to the imported node. The exact wcet of the different services is unfortunately not available. We tried different values to check when the system is or is not schedulable. For instance, the service `S4`, which has four predecessors, is implemented as follows:

```
imported node S4 (i0, i1, i2, i3 : int)
returns (o: int) wcet 5;
```

- The main node assembles the different services together. We have one input in the main node for each transfer from the bus to the program. The strictly periodic clock of the input is determined by the frequency and the phase of the transfer. For instance, the following input corresponds to a transfer of frequency 1Hz that takes place in PCF 1:

```
i4 : int rate(1000,1/10)
```

- We have one output in the main node for each transfer from the program to the bus. The strictly periodic clock of an output is determined exactly as that of an input.
- We have one local variable in the main node for each output of an intermediate service, that is to say a service that does not directly produce an output of the program.
- Then we have one equation for each service call:
  - The left-hand side of the equation is either a local variable if the service is an intermediate service, or an output of the main node if the service produces data that must be sent to the bus.
  - The right-hand side of the equation is a node call of the imported node corresponding to the service. The arguments of the node call are either outputs produced by the services preceding this service or inputs of the main node if the service reads data from the bus.
  - We use rate transition operators when the arguments of a node call do not have the same rate as the corresponding service. For instance, the equation for the service `S4` is:
    `x4=S4( x1/^10, x3/^10, x9, (0 fby x10)*^10);`
    For data produced by faster operations (`x1` and `x3`) we use the under-sampling operator (`/^10`). For data produced by slower operations (`x10`) we use the over-sampling operator (`*^10`).

15.3 Specific requirements

In this section, we detail some specific requirements we encountered when programming the FAS case study or during our discussions with EADS Astrium Space Transportation.

15.3.1 Slow-to-fast communications

When data produced by a slow service is consumed by a fast service, we use the over-sampling operator $\hat{*}$. We have seen in Chap. 5 that it is usually preferable to insert a delay in the communication to avoid the reduction of the deadline of the slow service due to the over-sampling. For instance, if A is the slow service and B is 10 times faster, we write:

```
node slow-fast1(i: rate (1000,0)) returns (o)
let
  o=B((0 fby A(i))*^10);
tel
```

In this case the end-to-end latency, from the acquisition of a value of `i` to the production of the corresponding value of `o`, is one period of `A` plus one period of `B`, thus 1100ms.

We can achieve a better latency if `B` starts right after `A` completes. For instance, if `A` can complete in less than 600ms, we can write:

```
node slow-fast2(i: rate (1000,0)) returns (o)
let
  o=B((A(i)~>6/10)*^10);
tel
```

In this case, the clock of `B` is $(100, 6/10)$ and its deadline is $100 + 6 * 100 = 700ms$. The deadline for `A` is $700 - C_B$. Thus the end-to-end latency is 700ms.

15.3.2 Producing outputs at exact dates

Most outputs of the FAS must be sent to the bus communication driver (ie produced by the program) at precise dates, instead of being allowed to be sent anywhere during the output period. This can be programmed as follows:

```
node exact(i: int rate (1000, 0)) returns (o: rate (1000, 6/10) due 0)
var v: int;
let
  v=N(i);
  o=v~>6/10;
tel
```

The output o has clock $(1000, 6/10)$ and relative deadline 0. Thus, it must be produced exactly at dates $600, 1600, 2600, \dots$ The node N has clock $(1000, 0)$ and relative deadline 600. Thus, it must execute during time intervals $[0, 600[, [1000, 1600[, [2000, 2600[, \dots$

15.3.3 Output with a phase smaller than the related input

In some cases an output o is computed from an input i and has a smaller phase than i. This can be programmed as follows:

```
node phases(i: int rate (1000, 3/10))
returns (o: rate (1000, 2/10) due 0)
var v: int;
let
  v=N(i);
  o=0::(v~>9/10);
tel
```

First, concerning the clocks of the program, the variable v has clock $(1000, 3/10)$, so the expression v~>9/10 has clock $(1000, 12/10)$ and the expression 0::(v~>9/10) has clock $(1000, 2/10)$.

Concerning deadlines, o has relative deadline 0, so it must be produced at dates 200,1200,2200,... However, the first value of o, produced at date 200, is computed from the value 0, not from the variable v. Then, the second value of o produced at date 1200 is computed from the first value of v (denoted $v[0]$). Thus, the deadline for the production of $v[0]$ is 1200 and it can be computed during time interval $[300, 1200[$. Similarly, the third value of o, produced at date 2200, is computed from the second value of v, which is computed during time interval $[1300, 2200[$ and so on.

15.3.4 End-to-end latency longer than the period

Some programs require the definition of an end-to-end latency constraint, from one input i to an output o of the same period, where the latency is longer than this period. This can be programmed exactly as our previous example but deadlines calculus are slightly different:

```
node latency(i: rate (1000,2/10))
returns (o: rate (1000,4/10) due 0)
var v: int;
let
  v=N(i);
  o=0::(v~>12/10);
tel
```

Clocks are computed exactly as for our previous example. Concerning deadlines, the output o has relative deadline 0, so it must be produced at dates $400, 1400, 2400, \ldots$ The first value of o, produced at date 400, is computed from the value 0. The second value of o, produced at date 1400 is computed from the first value of v (denoted $v[0]$). This requires $v[0]$ to be produced before date 1400. However, v has clock $(1000, 2/10)$, so $v[0]$ cannot have a deadline bigger than 1200 (the end of its period). Thus v must be computed during time interval $[200, 1200[$ (instead of $[200, 1400[$). Similarly, the third value of o produced at date 2400 is computed from the second value of v, which is computed during time interval $[1200, 2200[$ and so on.

The reduction of the interval during which v must be computed does not seem too restrictive. Indeed, suppose we allow $v[0]$ to be computed during time interval $[200, 1400[$, $v[1]$ to be computed during time interval $[1200, 2400[$, and so on. As `v=N(i)`, this means that, for instance, during time interval $[1200, 1400[$, $N[0]$ and $N[1]$ can be scheduled concurrently by the processor. This requires tasks to be reentrant, which is not supported by many RTOS or scheduling policies. Furthermore, it is not obvious that this concurrency is likely to lead to a system that has a better chance of being schedulable.

15.4 Results

We kept 180 services out of the 240 periodic services specified in the ATV-AS-ADD document, discarding services for which neither bus transfers nor precedences with other services were specified. The main node has 70 inputs and 9 outputs. The complete program is about 500 lines of code long. On a modest PC (AMD Athlon XP 2000+ with 512Mb of RAM), the compilation of the program takes about 1s to complete. Most of the compilation time is spent writing the C file, the other compilation phases are almost immediate. The code generated for the program is about 10000 lines long. It consists of 400 lines of communication buffer allocations, 4000 lines for the definition of the task functions and 6000 lines for threads creation (in the main function).

We did a simulation of the system with MARTE OS to check its schedulability. We also added an option to the compiler that produces a `xml` file describing the task set computed for a program. This output can then be taken as input by existing schedulability analysis tools. For instance, we analyzed the FAS program using CHEDDAR [SLNM04]. We verified that with some wcet configurations the program produced by the compiler is schedulable, according to the simulation with MARTE OS, while if we take the same deadline for each instance of a task, that is to say we replace deadline words by simple deadlines, according to CHEDDAR the task set is not schedulable.

15.5 Summary

In this chapter we have shown that complex systems can be programmed with our language and that the compiler easily supports such systems. The next Chapter concludes this dissertation and details some possible perspectives for our work.

Conclusion

1 Summary

We proposed a language for the design of embedded control systems. It is built as a real-time software architecture language with synchronous semantics. It provides a high-level of abstraction with formal semantics, which allows to specify without ambiguities the integration of several locally mono-periodic systems into a globally multi-periodic system.

The language relies on a particular class of clocks, called strictly periodic clocks, which relate logical time (the sequence of instants of a program) to real-time. A strictly periodic clock is defined uniquely by its period and its phase and defines the real-time rate of a flow. Periodic clock transformations create new strictly periodic clocks from existing strictly periodic clocks, either by accelerating a clock, slowing it down or applying a phase offset. The language defines real-time primitives based on strictly periodic clocks and periodic clock transformations. These primitives enable the specification of multiple periodicity constraints, deadline constraints and rate transition operators, to precisely define communication patterns between operations of different rates. A program then consists of a set of imported nodes (implemented outside the program) assembled using rate transition operators. The semantics of the language is defined formally. Static analyses check that the semantics of a program is well-defined before translating it to lower-level code (mainly, they verify that the program is well-typed and well-synchronized).

If the static analyses succeed, the semantics of the program is well-defined. It is then translated into a set of communicating and concurrent real-time tasks, programmed in C. The translation first consists in extracting a set of real-time tasks from the program, related by precedence constraints due to data-dependencies. Second, the precedences are encoded in the real-time attributes of the tasks, by adjusting the deadlines of tasks related by a precedence constraint. When a precedence relates two tasks of different rates, it does not relate all the instances of the related tasks, thus we need to set different deadlines for different instances of the same task. We introduce deadline words for this purpose, which define the sequence of deadlines of the instances of a task as a finite repetitive pattern. The set of tasks obtained after the adjustment of real-time attributes is scheduled using the EDF policy. Encoding precedences and using the EDF policy yields an optimal scheduling policy, meaning that if a schedule exists that satisfies all the real-time constraints of the original task set (periods, deadlines and precedences), using a preemptive dynamic priority policy, then our scheduling technique finds it. An inter-task communication protocol is defined to ensure that the task set respects the formal semantics of the original program. The protocol is based on data-buffering mechanisms and requires no synchronization primitives (like semaphores). The complete compilation scheme was prototyped in OCAML. It generates C code that uses the real-time extensions of POSIX. A modified EDF scheduler supporting deadline words was implemented with MARTE OS. The expressiveness of the language and the efficiency of the compiler have been validated on a realistic Flight Application Software provided by EADS Astrium Space Transportation.

2 Limitations and perspectives

In this section, we give an overview of some improvements or extensions, from which our work would benefit.

2.1 Task clustering

For now, each imported node of the program is translated into a separate task. This can lead to an important number of tasks (about 180 for the FAS case study). Such a large set of tasks may not be supported by the target RTOS or may result in an important run-time overhead, due to the time required to compute scheduling decisions.

This could be improved by grouping several imported nodes in the same task, referred to as task clustering. However, task clustering is not trivial and its impact on the number of task preemptions is hard to predict. This is clearly a subject for future work, all the more so as few results seem to be available on the subject.

2.2 Static priority scheduling

Though EDF enables better processor utilization, developers of real-time critical application often do not trust the dynamic aspects of EDF and prefer the DM policy. We are currently studying a compilation scheme based on a static priority scheduling policy, where task priorities are affected at compile time, taking task precedences into account. It seems that we could reuse the communication protocol defined in this dissertation with this new scheduling policy. Indeed, the protocol only requires task priorities to be affected so that the source task of a precedence is scheduled before the destination task of the precedence, which should be ensured by this new scheduling policy.

2.3 Modes

In many embedded control systems, the behaviour of the system is made up of different modes, corresponding to different phases of the mission of the system. For instance, the ATV does not have the same behaviour when it is moving towards International Space Station as when it is performing the docking maneuver. The behaviour of the system can be represented as a an automaton: the computations performed by the program depend on the current state of the automaton and on certain conditions, the systems transits from one state to another.

LUSTRE has been extended with State Machines (in the style of SYNCCHARTS) in [CPP05] to express such mode automata. The central idea of this translation is to translate the new State Machine constructs into (almost) standard synchronous data-flow, by relying on the use of clocks to translate the different states and transitions of the State Machines. The main benefit of this approach is that it allows to reuse existing synchronous data-flow compilers.

This approach could be transposed to our language, using the same constructs and the same translation techniques. This does however require to extend periodic clock transformations to Boolean clocks. Let us consider a simple example:

```
node do_the_job(i) returns (o)
  var vf, vs;
let
  (o, vf)=F(i, (0 fby vs)*^3);
  vs=S(vf/^3, i/^3);
tel
```

```
node modes (i: (100, 0), c: (100, 0)) returns (o)
let
  automaton A
    initial state active
    let
      o=do_the_job(i);
    tel
    until c do resume idle;
    state idle
      o=i;
    until c do resume active;
tel
```

The automaton `A` has two modes, `active` and `idle`, `active` being the initial state of the automaton. When the automaton is in state `active` it computes the output `o` by calling the node `do_the_job`. When it is in state `idle`, `o` simply takes the value of `i`. The automaton transits from state `active` to state `idle` whenever `c` is true.

The translation proposed in [CPP05] creates a clock condition `vstate`, which describes the current state of the automaton (its value can either be true for `active` or false for `idle`). The value of the clock changes whenever `c` is true. The equation defining `o` is then translated into:

```
o=merge(vstate, do_the_job(i when vstate), i whennot vstate);
```

So, the clock of the input of `do_the_job` is $(100, 0)$ on $vstate$. In the definition of node `do_the_job`, the operation `/^3` is applied to `i`. The clock of flow `i/^3` must then be the result of the application of transformation $/_.3$ to clock $(100, 0)$ on c. To support this example, we thus need to extend periodic clock transformations so that they can be applied to Boolean clocks.

This can be done by defining the following transformations (α is a clock, c a clock condition):

- $(\alpha \text{ on } c)/_.k = \alpha/_.k \text{ on } c/\hat{}k,\ k \in \mathbb{N}^{+*}$;
- $(\alpha \text{ on } c) *_. k = \alpha *_. k \text{ on } c *\hat{}k,\ k \in \mathbb{N}^{+*}$;
- $(\alpha \text{ on } c) \rightarrow_. q = \alpha \rightarrow_. q \text{ on } c \rightarrow_. q,\ q \in \mathbb{Q}$.

This definition respects the constraint that in clock α on c, c must have clock α. In the example above, the clock of `i/^3` would be $((100, 0) \text{ on } c)/_.3$ or equivalently $(300, 0) \text{ on } c/\hat{}3$.

This extension does however make clocks unification significantly more complex as clock conditions are not just names anymore but can be flow expressions. Furthermore, the semantics of multi-periodic nodes activated inside an automaton can be tricky. For instance, in the example above, if c is false at date 200, `F` will not execute between dates 200 and 300 but the first instance of `S`, which was released at date 0, can still execute until date 300.

Bibliography

[ABLG95] T. P. Amagbegnon, Loïc Besnard, and Paul Le Guernic. Implementation of the data-flow synchronous language Signal. In *ACM Symposium on Programming Languages Design and Implementation (PLDI'95)*, La Jolla, California, June 1995.

[ACGR09] Mouaiad Alras, Paul Caspi, Alain Girault, and Pascal Raymond. Model-based design of embedded control systems by means of a synchronous intermediate model. In *International Conference on Embedded Software and Systems (ICESS'09)*, Hangzhou, China, May 2009.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2), 1994.

[AEH+99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Sci. Comput. Program.*, 34(1), 1999.

[ALGM96] Pascal Aubry, Paul Le Guernic, and Sylvain Machard. Synchronous distribution of Signal programs. In *29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, 1996.

[AM09] Charles André and Frédéric Mallet. Combining CCSL and Esterel to specify and verify time requirements. In *ACM SIGPLAN/SIGBED 2009 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'09)*, Dublin, Ireland, June 2009.

[Aud91] Neil C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS 164, Dept. Computer Science, University of York, December 1991.

[Bak91] Theodore P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3(1), 1991.

[BB01] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. In *Readings in hardware/software co-design*. Kluwer Academic Publishers, 2001.

[BBHL93] Shuvra S. Bhattacharyya, Joseph T. Buck, Soonhoi Ha, and Edward A. Lee. A scheduling framework for minimizing memory requirements of multirate DSP systems represented as dataflow graphs. In *VLSI Signal Processing, VI, [Workshop on]*, Veldhoven, Netherlands, October 1993.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proc. IEEE*, 91(1), 2003.

[BCFP09] Frédéric Boniol, Mikel Cordovilla, Julien Forget, and Claire Pagetti. Implantation multi-tâche de programmes synchrones multipériodiques. In *7ème colloque francophone sur la Modelisation des Systemes Réactifs (MSR'09)*, November 2009.

[BCP+01] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. TAXYS=Esterel+Kronos. A tool for verifying real-time properties of embedded systems. In *40th IEEE Conference on Decision and Control*, volume 3, 2001.

[BD91] Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Transactions on Software Engineering*, 17(3), March 1991.

[BdS91] Frédéric Boussinot and Robert de Simone. The Esterel language. *Proc. IEEE*, 79(9), 1991.

[BLGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the Signal language and its semantics. *Sci. of Compu. Prog.*, 16(2), 1991.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998.

[BRH90] Sanjoy K. Baruah, Louis E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Syst.*, 2(4), 1990.

[BS01] Gérard Berry and Ellen Sentovich. Multiclock esterel. In *11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, Livingston, Scotland, September 2001.

[But05] Giorgio C. Buttazzo. Rate Monotonic vs. EDF: Judgement Day. *Real-Time Systems*, 29(1), 2005.

[Car89] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts, IFIP State-of-the-Art Reports*, pages 431–507. Springer-Verlag, 1989.

[CDE+06] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-Synchronous Kahn Networks: a relaxed model of synchrony for real-time systems. In *ACM International Conference on Principles of Programming Languages (POPL'06)*, Charleston, USA, January 2006.

[CMP01] Paul Caspi, Christine Mazuet, and Natacha Reynaud Paligot. About the design of distributed control systems: The quasi-synchronous approach. In *20th International Conference on Computer Safety, Reliability and Security (SAFECOMP'01)*, Budapest, Hungary, September 2001.

[CMPP08] Albert Cohen, Louis Mandel, Florence Plateau, and Marc Pouzet. *Programming Languages and Systems*, volume 5356/2008 of *Lecture Notes in Computer Science*, chapter

Abstraction of Clocks in Synchronous Data-Flow Systems. Springer Berlin / Heidelberg, 2008.

[CMSV01] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.*, 20(9), September 2001.

[CP03] Jean-Louis Colaço and Marc Pouzet. Clocks as first class abstract types. In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, USA, October 2003.

[CP04] Jean-Louis Colaço and Marc Pouzet. Type-based initialization analysis of a synchronous dataflow language. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3), 2004.

[CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *5th ACM international conference on Embedded software (EMSOFT'05)*, 2005.

[CS02] Liliana Cucu and Yves Sorel. Real-time scheduling for systems with precedence, periodicity and latency constraints. In *10th International Conference on Real-Time Systems (RTS'02)*, Paris, France, April 2002.

[CSB90] Houssine Chetto, Marilyne Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2, 1990.

[Cuc04] Liliana Cucu. *Non-preemptive scheduling and schedulability condition for embedded systems with real-time constraints*. PhD thesis, University "Paris XI", May 2004.

[Cur05] Adrian Curic. *Implementing Lustre programs on distributed platforms with real-time constraints*. PhD thesis, Université Joseph Fourier, Grenoble, 2005.

[DOTY96] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The tool KRONOS. In *DIMACS/SYCON workshop on Hybrid systems III : verification and control*, 1996.

[Dur98] Emmanuel Durand. *Description et vérification d'architectures d'application temps réel : CLARA et les réseaux de Petri temporels*. PhD thesis, Ecole Centrale de Nantes, 1998.

[Est] Esterel Technologies, Inc. *SCADE Language - Reference Manual.*

[FBL+08] Julien Forget, Frédéric Boniol, David Lesens, Claire Pagetti, and Marc Pouzet. Programming languages for hard real-time embedded systems. In *4th European Congress on Embedded Real-Time Software (ERTS'08)*, Toulouse, France, January 2008.

[FBLP08] Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. A multi-periodic synchronous data-flow language. In *11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*, Nanjing, China, December 2008.

[FDT04] Sébastien Faucou, Anne-Marie Déplanche, and Yvon Trinquet. An ADL centric approach for the formal design of real-time systems. In *Architecture Description Language Workshop at IFIP World Computer Congress (WADL'04)*, August 2004.

[FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical report, Carnegie Mellon University, 2006.

[GAGB01] Paolo Gai, Luca Abeni, Massimiliano Giorgi, and Giorgio Buttazzo. A new kernel approach for modular real-time systems development. In *13th Euromicro Conference on Real-Time Systems (ECRTS'01)*, June 2001.

[GLA+00] Paolo Gai, Giuseppe Lipari, Luca Abeni, Marco di Natale, and Enrico Bini. Architecture for a portable open source real-time kernel environment. In *Second Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, November 2000.

[GLS99] Thierry Grandpierre, Christophe Lavarenne, and Yves Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *7th International Workshop on Hardware/Software Co-Design (CODES'99)*, Rome, Italy, May 1999.

[GN03] Alain Girault and Xavier Nicollin. Clock-driven automatic distribution of Lustre programs. In *3rd International Conference on Embedded Software, EMSOFT'03*, volume 2855 of *LNCS*, Philadelphia, USA, October 2003.

[GNP06] Alain Girault, Xavier Nicollin, and Marc Pouzet. Automatic rate desynchronization of embedded reactive programs. *ACM Trans. Embedd. Comput. Syst.*, 5(3), 2006.

[Hal93] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publisher, 1993.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proc. IEEE*, 79(9), 1991.

[HHK03] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proc. IEEE*, 91(1), 2003.

[Hil92] Dan Hildebrand. An architectural overview of qnx. In *Usenix Workshop on MicroKernels and Other Kernel Architectures*, Berkeley, USA, April 1992.

[HP85] David Harel and Amir Pnueli. *Logics and models of concurrent systems*, chapter On the development of reactive systems, pages 477–498. Springer-Verlag New York, Inc., 1985.

[HRR91] Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming (PLILP '91)*, Passau, Germany, 1991.

[Kah62] Arthur B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11), 1962.

[Kah74] Gilles Kahn. The semantics of simple language for parallel programming. In *International Federation for Information Processing (IFIP'74) Congress*, New York, USA, 1974.

[KB03] Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. *Proc. IEEE*, 91(1), 2003.

[KCS06] Omar Kermia, Liliana Cucu, and Yves Sorel. Non-preemptive multiprocessor static scheduling for systems with precedence and strict periodicity constraints. In *10th International Workshop On Project Management and Scheduling (PMS'06)*, Posnan, Poland, April 2006.

[Ker09] Omar Kermia. *Ordonnancement temps réel multiprocesseur de tâches non préemptives avec contraintes de précédence, de périodicité stricte et de latence*. PhD thesis, Université de Paris Sud, Spécialité Physique, 2009.

[Ler06] Xavier Leroy. *The Objective Caml system release 3.09, Documentation and user's manual*. INRIA, 2006.

[LGTLL03] Paul Le Guernic, Jean-Pierre. Talpin, and Jean-Christophe Le Lann. Polychrony for system design. Technical Report RR-4715, INRIA - Rennes, February 2003.

[LL73] Cheng L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

[LM80] J. Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Process. Lett.*, 11(3):115–118, 1980.

[LM87a] Edward Lee and David Messerschmitt. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Transaction on Computer*, C(36), 1987.

[LM87b] Edward Lee and David Messerschmitt. Synchronous Data Flow. *Proc. IEEE*, 75(9):1235–1245, 1987.

[LSV96] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli. Comparing models of computation. In *International Conference on Computer Aided Design (ICCAD'96)*, San Jose, USA, 1996.

[LW82] Joseph Y. T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4), 1982.

[Mat] The Mathworks. *Simulink: User's Guide*.

[MM04] Florence Maraninchi and Lionel Morel. Arrays and contracts for the specification and analysis of regular systems. In *Fourth International Conference on Application of Concurrency to System Design (ACSD'04)*, Hamilton, Canada, June 2004.

[MTGLG08] Hugo Metivier, Jean-Pierre Talpin, Thierry Gautier, and Paul Le Guernic. Analysis of periodic clock relations in polychronous systems. In *IFIP, Distributed Embedded Systems: Design, Middleware and Ressources (DIPES'08)*, September 2008.

[MYS07] Patrick Meumeu Yomsi and Yves Sorel. Extending Rate Monotonic analysis with exact cost of preemptions for hard real-time systems. In *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, Pisa, Italy, July 2007.

[Obj07a] Object Management Group. *UML Profile for MARTE, beta 1*, August 2007.

[Obj07b] Object Management Group. *Unified Modeling Language, Superstructure*, November 2007.

[ODH06] Hyunok Oh, Nikil Dutt, and Soonhoi Ha. Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In *Asia and South Pacific Design Automation Conference (ASP-DAC'06)*, Yokohama, Japan, January 2006.

[OSE03] OSEK. *OSEX/VDX Operating System Specification 2.2.1*. OSEK Group, 2003. www.osek-vdx.org.

[OSE04] OSE. *OSE Real-Time Operating System*. ENEA Embedded Technology, 2004. www.ose.com.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, USA, 2002.

[POS93a] POSIX.1b. *IEEE Std. 1003.1b-1993. POSIX, Real-time extensions*. The Institute of Electrical and Electronics Engineers, 1993.

[POS93b] POSIX.1c. *IEEE Std. 1003.1c-1993. POSIX, Threads extensions*. The Institute of Electrical and Electronics Engineers, 1993.

[POS98] POSIX.13. *IEEE Std. 1003.13-1998. POSIX Realtime Application Support (AEP)*. The Institute of Electrical and Electronics Engineers, 1998.

[Pou06] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, 2006.

[RCK01] Pascal Richard, Francis Cottet, and Claude Kaiser. Validation temporelle d'un logiciel temps réel : application à un laminoir industriel. *Journal Européen des Systèmes Automatisés*, 35(9), 2001.

[Rém92] Didier Rémy. Extending ML type system with a sorted equational theory. Technical Report 1766, INRIA, Rocquencourt, France, 1992.

[Rey98] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[RH91] Frédéric Rocheteau and Nicolas Halbwachs. Pollux, a Lustre-based hardware design environment. In P. Quinton and Y. Robert, editors, *Conference on Algorithms and Parallel VLSI Architectures II*, Chateau de Bonas, France, June 1991.

[RH02] Mario A. Rivas and Michael G. Harbour. POSIX-Compatible Application-Defined Scheduling in MaRTE OS. In *14th Euromicro Conference on Real-Time Systems (ECRTS'02)*, Washington, USA, 2002.

[SLG97] Irina Smarandache and Paul Le Guernic. A canonical form for affine relations in signal. Technical Report RR-3097, INRIA, 1997.

[SLNM04] Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. Cheddar: a flexible real time scheduling framework. *Ada Lett.*, XXIV(4), 2004.

[Sof06] Christos Sofronis. *Embedded Code Generation from High-level Heterogeneous Components*. PhD thesis, Université Joseph Fourier, Grenoble, November 2006.

[SSDNB95] John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6), 1995.

[STC06] C. Sofronis, S. Tripakis, and P. Caspi. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In *Sixth International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, October 2006.

[TSCC05] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time Simulink to Lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4), 2005.

[VxW95] VxWorks. *VxWorks Programmer's Guide: Algorithms for Real-Time Scheduling Problems*. Wind River, Inc., 1995.

[WBC+00] Daniel Weil, Valérie Bertin, Etienne Closse, Michel Poize, Patrick Venier, and Jacques Pulou. Efficient compilation of esterel for real-time embedded systems. In *2000 international conference on Compilers, architecture, and synthesis for embedded systems (CASES'00)*, 2000.

[ZUE00] Dirk Ziegenbein, Jan Uerpmann, and Rolf Ernst. Dynamic response time optimization for SDF graphs. In *IEEE/ACM international conference on Computer-aided design (ICCAD'00)*, San Jose, USA, November 2000.

Appendix A

Code for the FAS Case Study

This appendix details the code of the main node of our program for the FAS case study.

```
[...]
node fas (i1 : int rate(500,0); i2 : int rate(100,0); i3 : int rate(200,1/2);
          i4 : int rate(200,1/2); i5 : int rate(1000,1/10);
          i6 : int rate(1000,1/10); i7 : int rate(1000,0);
          i8 : int rate(1000,1/10); i9 : int rate(1000,1/10);
          i10 : int rate(1000,1/10); i11 : int rate(1000,1/10);
          i12 : int rate(1000,4/10); i13 : int rate(1000,4/10);
          i14 : int rate(1000,1/10); i23 : int rate(500,0);
          i15 : int rate(1000,0); i16 : int rate(1000,3/10); i17 : int rate(1000,7/10);
          i18 : int rate(1000,9/10); i19 : int rate(1000,1/10);
          i20 : int rate(1000,1/10); i21 : int rate(1000,1/10);
          i22 : int rate(1000,1/10); i24 : int rate(1000,10/10);
          i25 : int rate(1000,10/10); i26 : int rate(1000,10/10);
          i27 : int rate(1000,10/10); i28 : int rate(1000,10/10);
          i29 : int rate(1000,10/10); i30 : int rate(1000,10/10);
          i31 : int rate(1000,10/10); i32 : int rate(1000,10/10);
          i33 : int rate(1000,10/10); i34 : int rate(1000,10/10);
          i35 : int rate(1000,10/10); i36 : int rate(100,0);
          i37 : int rate(100,0); i38 : int rate(1000,5/10);
          i39 : int rate(10000,8/100); i40 : int rate(1000,5/10);
          i41 : int rate(1000,1/10); i42 : int rate(1000,9/10);
          i43 : int rate(1000,9/10); i44 : int rate(10000,8/100);
          i45 : int rate(10000,8/100); i46 : int rate(10000,8/100);
          i47 : int rate(10000,8/100); i48 : int rate(10000,8/100);
          i49 : int rate(1000,0); i50 : int rate(1000,0);
          i51 : int rate(1000,0); i52 : int rate(1000,0);
          i53 : int rate(10000,0); i54 : int rate(10000,0);
          i55 : int rate (100,0))

returns (o1 : int rate (1000,6/10) due 0;
        o2 : int rate (1000, 2/10) due 0;
        o3 : int rate (100,0);
        o4 : int rate (100,0);
        o5 : int rate (10000,6/10) due 0;
        o6 : int rate (1000,6/10) due 0;
        o7 : int rate (200,1/10) due 0;
        o8 : int rate (200,0);
        o9 : int rate (500,2/10) due 0)

var x0 ,x1 ,x2 ,x3 ,x4 ,x5 ,x6 , x7 ,x8 ,x9 , x10 ,x11 ,x12 ,x13 ,x14 ,x15 ,x16 ,x17 ,x18 ,
    x19 ,x20 ,x21 ,x22 ,x23 ,x24 ,x25 ,x26 ,x27 ,x28 ,x29 ,x30 ,x31 ,x32 ,x33 ,x34 ,x35 ,x36
    [...]
let

x0=S0(i1);
x1=S1(i2);
x2=S2(i3, i4);
x3=S3(i55);
```

```
x4=S4( x1/^10, x3/^10, x9, (0 fby x10)*^10);
x5=S5( (0 fby x4)*^10);
x6=S6( x1/^10);
x7=S7(i55/^10);
x8=S8(i55/^100);
x9=S9( x7);
x10=S10( x8);
x11=S11(i55);
x12=S12( x11, (0 fby x13)*^10);
x13=S13(i55/^10);
x14=S14(i55/^10);
x15=S15( x14, x17);
x16=S16( x14, x17);
x17=S17(i55/^10);
x18=S18(i5, i6, x15~>1/10);
x19=S19(i55);
x20=S20( x15~>1/10, x18);
x21=S21( (0 fby x15)*^10~>1/1, (0 fby x18)*^10, x19~>1/1);
x22=S22(i7~>1/10, i8, i9, i10, x15~>1/10);
x23=S23( x22, x27~>1/10);
x24=S24( x23, x27~>1/10);
x25=S25( x24, x27~>1/10);
x26=S26( x22);
x27=S27(i55/^10);
x28=S28(i11~>3/10, i12, i13, i10~>3/10,
    i9~>3/10, i5~>3/10, i6~>3/10, i14~>3/10);
x29=S29(i11~>3/10, i12, i13, i10~>3/10,
      i5~>3/10, i6~>3/10, i14~>3/10, x15~>4/10);
x30=S30( x29);
x31=S31(i55);
x32=S32(i23/^2);
x33=S33( x32);
x34=S34(i15);
x35=S35( x34);
x36=S36(i16);
x37=S37( x36);
x38=S38(i23/^2);
x39=S39( x38);
x40=S40(i17);
x41=S41( x39);
x42=S42(i18);
x43=S43( x42);
x44=S44(i8, i19, i9, i20, i10, i21, i14, i22, i5, i6, x15~>1/10);
x45=S45( x58);
x46=S46(i55/^10);
x47=S47(i55/^10);
x48=S48( x32~>9/10, x34~>9/10, x36~>6/10, x38~>9/10, x42);
x49=S49(i55/^10);
x50=S50(i55/^10);
x51=S51(i55);
x52=S52(i55/^10);
x53=S53(i55/^10);
x54=S54( x47);
x55=S55( x31);
x56=S56(i55/^10);
x57=S57( x34~>9/10, x36~>6/10, x42, x50~>9/10);
x58=S58( x47);2
o1= x58~>6/10;
x59=S59( x46);
o2=x59~> 2/10;
x60=S60( x61, x62, (0 fby x63)*^10~>1/10, (0 fby x64)*^10~>1/10);
x61=S61( x15~>1/10, x65, (0 fby x66)*^10);
x62=S62( x15~>1/10, x65, (0 fby x66)*^10);
x63=S63( x15/^10);
x64=S64( x15/^10);
x65=S65(i19, i9, i20);
x66=S66(i9/^10, i20/^10, i5/^10, i6/^10);
x67=S67(i24, i25, i26, i27, i28, i29, i30, i31, i32, i33, i34, i35);
```

```
x68=S68(i55/^10);
x69=S69( x68);
x70=S70( x69);
x71=S71(i36);
x72=S72( x72, x96);
x73=S73( x72, x73);
x74=S74( x73);
x75=S75( x72, x74, x75);
x76=S76(i37/^10~>1/10, i19, i9, i20, i5, i6, x15~>1/10);
x77=S77( x83/^10);
x78=S78(i37);
x79=S79(i19/^10, i9/^10, i20/^10, x81~>1/100);
x80=S80( x77);
x81=S81(i55/^100);
x82=S82(i37, (0 fby x80)*^10, x83);
x83=S83(i55);
x84=S84( x83/^100, x152/^10);
x85=S85( x83);
x86=S86( x156);
x87=S87( x83/^2);
x88=S88(i55);
o3=x88;
x89=S89(i55/^100);
x90=S90( x55, x83, (0 fby x84)*^100, x85, (0 fby x86)*^2, (0 fby x87)*^2, x88);
x91=S91( x90/^10);
x92=S92( x83/^100, x89, x91/^10);
x93=S93( x90);
o4=x93;
x94=S94(i13~>1/10, i12~>1/10, i38, i8~>4/10, i9~>4/10, x15~>5/10);
x95=S95(i39, i13/^10~>4/100, i38/^10~>3/100, i12/^10~>4/100, x122);
x96=S96( x94);
x97=S97( x95);
x98=S98( x95);
x99=S99( x94);
x100=S100( x95);
x101=S101( x96, x99);
x102=S102( x96, x99);
x103=S103(i40, i5~>4/10, i6~>4/10);
x104=S104(i41~>4/10, i40, i8~>4/10, i10~>4/10);
x105=S105(i55/^10);
x106=S106( x103, x105~>5/10);
x107=S107( x106);
o5=x107/^10~> 11/20;
x108=S108( x15~>5/10, x104, x106);
x109=S109(i5/^10, i6/^10);
x110=S110( x15/^10~>1/100, x109);
x111=S111( x108, (0 fby x110)*^10~>4/10);
x112=S112(i42, i43);
x113=S113( x114);
x114=S114( x112);
x115=S115( x113);
x116=S116( x120);
x117=S117( x131);
x118=S118( x117);
x119=S119( x118);
x120=S120( x119);
x121=S121( x126);
x122=S122( x124, x125~>3/100, x126);
x123=S123( x131);
x124=S124(i44, i45, i46, i39, i47, x120~>8/100, x123~>8/100);
x125=S125(i13/^10~>1/100, i12/^10~>1/100, i38/^10, x120~>5/100, x123~>5/100);
x126=S126(i5/^10~>7/100, i6/^10~>7/100, i8/^10~>7/100, i9/^10~>7/100, i10/^10~>7/100,
      i14/^10~>7/100, i48, x120~>8/100, x123~>8/100);
x127=S127( x129);
x128=S128(i55/^100);
x129=S129( x122);
x130=S130( x128);
x131=S131( x130);
```

```
o6=(0 fby x131)*^10~>6/10;
x132=S132(i49, i50, i51);
x133=S133(i55/^10);
x134=S134( x133);
x135=S135( x134);
x136=S136( x135);
x137=S137(i52, (0 fby i53)*^10, (0 fby i54)*^10);
x138=S138( x137);
x139=S139( x138);
x140=S140( x139);
x141=S141( x140);
x142=S142( x15/^10);
x143=S143( x15/^10);
x144=S144(i5/^10, i5/^10);
x145=S145( x15/^10);
x146=S146( x151);
x147=S147(i55/^10);
x148=S148( x153);
x149=S149( x155);
x150=S150( x149/^10);
x151=S151(i55/^10);
x152=S152(i55/^10);
x153=S153( x147);
x154=S154( x156/^5);
x155=S155( x147);
x156=S156(i55/^2);
o7=x156~> 1/10;
x157=S157( (0 fby x49)*^5, x51/^2, (0 fby x52)*^5, (0 fby x53)*^5, (0 fby x56)*^5, x87);
x158=S158(i55);
x159=S159(i55/^10);
x160=S160(i55/^100);
x161=S161( x171);
x162=S162(i55);
x163=S163( x162);
x164=S164( x163);
x165=S165( x164);
x166=S166( x165);
o8=x166/^2;
x167=S167(i55);
x168=S168( x166, x167);
x169=S169( x168);
o9=x169/^5~> 1/5;
x170=S170( x169);
x171=S171(i55/^10);
x172=S172(i55/^10);
tel
```

Un langage synchrone pour les systèmes embarqués critiques soumis à des contraintes temps réel multiples

Ce travail porte sur la programmation de systèmes de Contrôle-Commande. Ces systèmes sont constitués d'une boucle de contrôle qui acquiert l'état actuel du système par des capteurs, exécute des algorithmes de contrôle à partir de ces données et calcule les commandes à appliquer sur les actionneurs du système dans le but de réguler son état et d'accomplir une mission donnée. Ces systèmes sont critiques, leur implantation doit donc être déterministe, sur le plan fonctionnel (produire les bonnes sorties en réponse aux entrées) mais aussi sur le plan temporel (produire les données aux bonnes dates).

Nous définissons un langage de description d'architecture logicielle temps réel et son compilateur pour programmer de tels systèmes. Il permet d'assembler les composants fonctionnels d'un système multi-rythme avec une sémantique formelle synchrone. Un programme consiste en un ensemble d'opérations importées, reliées par des dépendances de données. Des contraintes de périodicité ou d'échéance peuvent être spécifiées sur les opérations et un ensemble réduit d'opérateurs de transition de rythme permet de décrire de manière précise et non ambiguë des schémas de communication entre opérations de périodes différentes.

Des analyses statiques assurent que la sémantique d'un programme est bien définie. Un programme correct est ensuite compilé en un ensemble de tâches temps réel concurrentes implantées sous forme de threads C communicants. Le code généré préserve la sémantique du programme original et s'exécute à l'aide d'un Système d'Exploitation Temps Réel standard disposant de la politique d'ordonnancement EDF.

Mots-clés : Systèmes embarqués, Systèmes critiques, Temps réel, Langages formels, Compilation

A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints

This work deals with the programming of Embedded Control Systems. Such systems consist of a control loop, which acquires the current state of the system through sensors, computes control algorithm from this data and produces commands to apply to the actuators of the system, in order to control its state and to accomplish a given mission. These systems are critical, thus their implementation must be deterministic, functionally (producing the right outputs for given inputs) as well as temporally (producing outputs at the right time).

We define a real-time software architecture description language and the associated compiler to program such systems. It enables to assemble the functional components of a multi-rate system with a formal synchronous semantics. A program consists of a set of imported operations related by data dependencies. Periods and deadline constraints can be specified on operations and a reduced set of rate transition operators enables the accurate description of non-ambiguous communication schemes between operations of different periods.

Static analyses ensure that the semantics of a program is well-defined. A correct program is then compiled into a set of concurrent real-time tasks implemented as communicating C threads. The generated code preserves the semantics of the original program and can be executed with a standard Real-Time Operating System that provides the EDF scheduling policy.

Keywords : Embedded systems, Critical systems, Real-Time, Formal Languages, Compiling