Compilation efficace de spécifications de contrôle embarqué avec prise en compte de propriétés fonctionnelles et non-fonctionnelles complexes

Thomas Carle

THÈSE DE DOCTORAT DE
L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité Informatique
(École Doctorale Informatique, Télécommunication et Électronique)

Présentée par THOMAS CARLE

Pour obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

COMPILATION EFFICACE DE SPÉCIFICATIONS DE CONTRÔLE EMBARQUÉ AVEC PRISE EN COMPTE DE PROPRIÉTÉS FONCTIONNELLES ET NON-FONCTIONNELLES COMPLEXES

Soutenue le 31 Octobre 2014, devant le jury composé de

Prof.	ISABELLE PUAUT	IRISA	Rapportrice
Prof.	FRANÇOIS IRIGOIN	Mines ParisTech	Rapporteur
Prof.	ALIX MUNIER	UPMC	Examinatrice
Dr.	DAVID LESENS	Airbus DS	Examinateur
Prof.	LAURENT GEORGE	ESIEE	Examinateur
Dr.	DUMITRU POTOP-BUTUCARU	INRIA	Directeur de thèse

PH.D. THESIS OF THE UNIVERSITY PIERRE AND MARIE CURIE

Department : COMPUTER SCIENCE AND MICRO-ELECTRONICS

Presented by: THOMAS CARLE

Thesis submitted to obtain the degree of
DOCTOR OF THE UNIVERSITY PIERRE AND MARIE CURIE

EFFICIENT COMPILATION OF EMBEDDED CONTROL SPECIFICATIONS WITH COMPLEX FUNCTIONAL AND NON-FUNCTIONAL PROPERTIES

Defence on October 31st, 2014, Committee:

Prof.	ISABELLE PUAUT	IRISA	Reviewer
Prof.	FRANÇOIS IRIGOIN	Mines ParisTech	Reviewer
Prof.	ALIX MUNIER	UPMC	Examiner
Dr.	DAVID LESENS	Airbus DS	Examiner
Prof.	LAURENT GEORGE	ESIEE	Examiner
Dr.	DUMITRU POTOP-BUTUCARU	INRIA	Advisor

Remerciements

Je tiens à remercier ici Dumitru Potop-Butucaru, mon directeur de thèse, pour la confiance qu'il m'a accordée, pour tous les conseils qu'il m'a prodigués, pour sa disponibilité et sa patience tout au long de ma thèse, pour son ouverture d'esprit et pour son implication sans faille à mes côtés tant durant cette thèse que pour la préparation de mon avenir.

J'aimerais également remercier Isabelle Puaut et François Irigoin, qui ont accepté d'être les rapporteurs de cette thèse, pour leurs remarques constructives qui m'ont permis de prendre du recul sur mon travail et de l'améliorer. Je remercie de même Alix Munier, David Lesens et Laurent George de m'avoir fait l'honneur d'accepter de participer à mon jury.

Je voudrais remercier les membres passés et présents de mon équipe pour leur bienveillance et leur sympathie, ainsi que pour leurs conseils et leur soutien dans les moments difficiles, notamment de la rédaction. Mes pensées vont notamment vers Manel Djemal, Raul Gorcitz, Abderraouf Benyahia, Walid Talaboulma, Falou N'doye, Daniel de Rauglaudre, Cécile Stentzel, Yves Sorel, Liliana Cucu-Grosjean, Adriana Gogonel et Codé Lo.

Enfin, je remercie ma famille et mes amis, et plus particulièrement mes parents pour m'avoir soutenu tout au long de mes (longues) études.

"Un jour viendra où ces avaleurs de feu que nous sommes se mettront à le cracher, et notre plus belle création sera l'œuvre des incendies que nous aurons ainsi allumés."

- Romain Gary, *Les enchanteurs*.

Contents

Résumé

Conception de systèmes embarqués

Les systèmes embarqués sont des systèmes informatiques ayant une fonction dédiée, au sein d'un système physique ou informatique plus large. Dans cette thèse, nous nous intéressons en particulier aux systèmes de contrôle embarqué [Lee and Seshia, 2011], dont le rôle est de réguler l'évolution de processus (physiques) afin d'assurer la correction de leur fonctionnement. Ici, les termes contrôle et régulation sont compris dans le sens de la théorie du contrôle [Doyle et al., 1990], c'est à dire qu'un système de contrôle embarqué est vu comme le moyen d'effectuer le *contrôle automatique* d'un *équipement* comme un avion, un train, un réacteur nucléaire, un téléphone mobile, *etc.* Comme on le voit dans la figure 1, un système de contrôle embarqué intéragit avec l'équipement contrôlé par le biais de *capteurs* et d'*actuateurs*. Les capteurs mesurent des caractéristiques (analogiques) spécifiques de l'équipement, et présentent l'information correspondante sous une forme discrétisée, permettant un traitement digital par le contrôleur embarqué. Les valeurs de retour discrètes du contrôleur sont utilisées par les actuateurs pour exercer un contrôle (analogique) sur l'équipement.

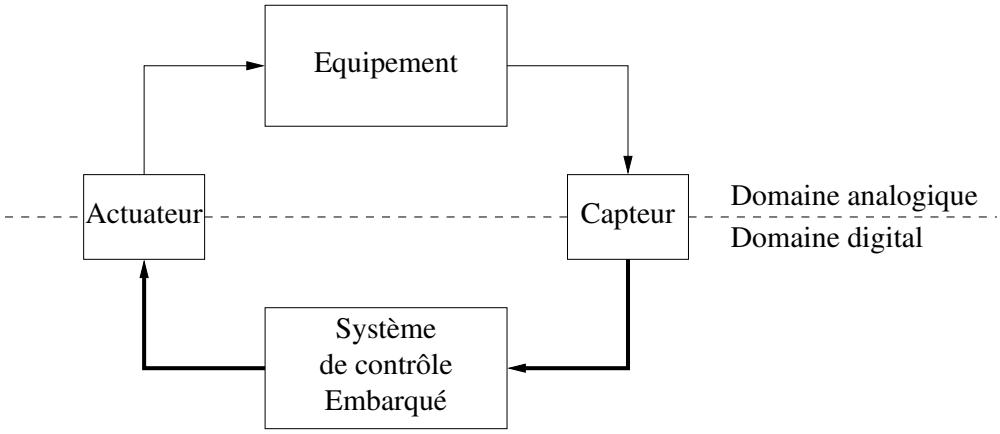


Figure 1: Interaction entre un équipement et un système de contrôle embarqué

Un exemple de ces interactions est donné par les contrôleurs embarqués de guidage, navigation et contrôle (GNC) des avions. Ceux-cis utilisent différents capteurs (de pression, de température, GPS, accéléromètres, gyroscopes, *etc.*) pour déterminer l'emplacement, la vitesse, le roulis, le tangage et le lacet de l'appareil. Ces valeurs sont utilisées de manière cyclique par le contrôleur GNC pour manipuler la position des ailerons et la puissance des réacteurs (par le biais d'actuateurs), afin de maintenir la trajectoire désirée. La conception de systèmes de contrôle embarqué est inter-disciplinaire. A haut niveau, elle est dirigée par des ingénieurs contrôle commande et par des ingénieurs système. Les

ingénieurs contrôle commande fournissent une spécification du comportement attendu du contrôleur, incluant ses fonctionnalités et ses besoins en termes temporels. Définir les fonctionnalités d'un système consiste à spécifier les opérations qui doivent s'exécuter ainsi que leur ordre d'exécution, pour calculer les commandes des actuateurs à partir des valeurs produites par les capteurs[1]. Cependant, la spécification fonctionnelle seule n'est pas suffisante pour traduire le comportement d'un système de contrôle. Pour assurer un contrôle effectif, les changements dans les processus physiques de l'équipement considéré doivent être suivis *en temps voulu* par des changements dans les actuateurs. Par exemple, le contrôleur embarqué GNC évoqué plus haut doit réagir rapidement aux coups de vent latéraux afin de maintenir la stabilité de l'avion, particulièrement lors des phases d'atterrissage. C'est pour cette raison que la partie correspondant au contrôle d'attitude du contrôleur GNC est exécutée une fois toutes les 50ms [Kent and Williams, 2002].

La spécification de ces contraintes temporelles s'effectue sous la forme de *contraintes temps-réel* telles que des *périodes*, des *latences*, des *dates de déclenchement* et des *échéances* qui sont attachées aux divers éléments de la spécification fonctionnelle. Les systèmes de contrôle embarqué appartiennent donc à la catégorie des *systèmes temps-réel* [Butazzo, 2002], qui forment eux-même une sous-classe des *systèmes réactifs* [Harel, 1987].

Alors que les ingénieurs contrôle commande spécifient le comportement des contrôleurs embarqués, les ingénieurs système étudient le problème de leur implantation, et fournissent des outils permettant de les gérer durant toute leur durée de vie. Cette durée de vie s'élève à plus de 20 ans pour un avion commercial, plus de 30 ans pour un réacteur nucléaire, et plus de 150000km en moyenne pour une voiture. A haut niveau, l'ingénierie des systèmes considère des propriétés macroscopiques telles que le coût (de développement, de maintenance, et parfois de mise hors service), la fiabilité (pour des raisons de cout et de sécurité), le risque, *etc.* Ces objectifs macroscopiques peuvent être traduits en termes de *besoins non-fonctionnels*, et l'ingénierie des systèmes fournit des méthodes pour *raffiner* de tels besoins de haut niveau et définir des besoins de plus bas niveau tels que ceux que nous considérons dans cette thèse. Par exemple, le chapitre 7 considère les besoins d'ingénierie système suivants : la *plateforme d'exécution* est fixée[2], chaque opération de la spécification fonctionnelle s'est vue assignée un *niveau de criticité*, certaines de ces opérations peuvent être *préemptées* (les autres sont dites *non préemptables*), et une partie de la distribution des opérations de la spécification fonctionnelle sur les ressources de la plateforme est également déjà fixée. D'autres propriétés non-fonctionnelles qu'il est possible de considérer à ce niveau sont, par exemple, la tolérance aux pannes, la consommation d'énergie et l'évolutivité du système. C'est le niveau auquel mon travail se place : dans cette thèse je considère que les phases préliminaires d'ingénierie ont pro-

[1] L'état interne du système peut aussi être utilisé.

[2] Cela peut également imposer des contraintes temps-réel supplémentaires, en plus des celles provenant de la spécification du contrôle.

duit une spécification fonctionnelle, ont fixé l'architecture d'exécution cible (qui peut être une architecture multiprocesseur/distribuée), et ont généré un ensemble de besoins non-fonctionnels qui inclut notamment des contraintes temps-réel. A partir de ces données, mon objectif est de construire automatiquement un système de contrôle embarqué correct. La correction implique ici deux aspects : la correction fonctionnelle et le respect des besoins non-fonctionnels. La correction fonctionnelle se traduit par le fait que le système se comporte tel que la spécification fonctionnelle le prescrit. Assurer la correction fonctionnelle d'un tel système implique plusieurs disciplines telles que la *compilation* (pour assurer que chaque opération de la spécification est correctement implantée par du code séquentiel), le *calcul parallèle/distribué* (pour la construction d'implantations multiprocesseurs correctes), *etc.* Un domaine revêtant une importance particulière est celui de *la conception et de l'implantation de langages synchrones* qui regroupent toutes les disciplines nécessaires pour assurer la correction fonctionnelle. Cela est dû au fait que les langages et formalismes synchrones, qui seront décrits dans la thèse, ont été conçus spécifiquement pour permettre la spécification de systèmes de contrôle embarqué. Cela explique leur omniprésence dans la conception des systèmes embarqués et leur utilisation dans cette thèse.

Assurer le respect de besoins non-fonctionnels tels que ceux listés plus haut (temps-réel, criticité, préemptabilité, consommation d'énergie, *etc.*) est principalement étudié dans le domaine de *l'ordonnancement temps-réel*, qui lui même tire avantage de résultats venant de la conception de matériel et de systèmes d'exploitation, du calcul parallèle/distribué, de la tolérance aux pannes, *etc.*

Plaidoyer pour une intégration renforcée. La conception d'un système embarqué complexe rassemble en général des experts venant des domaines cités précédemment. Cependant, dans les travaux scientifiques et dans la pratique industrielle actuels, ces domaines restent largement séparés. Mon opinion est qu'une telle ségrégation a pour résultat d'augmenter les coûts de conception et de dégrader l'efficacité de l'implantation. Cela est dû à deux principales causes :

- Les interfaces entre les différents domaines, souvent matérialisées par des transformations de modèles, ont tendance à abstraire des détails significatifs présents dans les modèles de départ, ce qui entraine une perte d'efficacité. Cette perte d'efficacité est souvent accompagnée de l'utilisation d'abstractions peu sûres, qui imposent que la validation du système passe par une phase de tests intensive, ce qui augmente les coûts de développement[3],

[3]Par exemple, les modèles de tâches utilisés dans le domaine de l'ordonnancement temps-réel font souvent l'abstraction de tout le contrôle conditionnel attaché aux tâches, ce qui ajoute du pessimisme à l'analyse. De plus, beaucoup de modèles de tâches utilisés dans le cadre d'ordonnancement multipro-

- Le manque de communication entre les communautés scientifiques et industrielles implique que des solutions développées dans un domaine ne seront pas appliquées dans un autre, même lorsque cela serait approprié.

Le travail réalisé durant cette thèse s'attaque à ces deux points pour des classes spécifiques d'architectures cibles. Afin de réduire la perte d'efficacité nous utilisons des modèles précis des applications (où le contrôle conditionnel est pris en compte), des architectures (où les coûts de communication et les mécanismes d'exécution précis sont considérés), et des besoins non-fonctionnels, qui sont adaptés à nos plateformes d'exécution cibles (des systèmes partitionnés dirigés par le temps). La modélisation précise de la plateforme d'exécution réduit également le recours à des abstractions non sûres. Néanmoins la contribution principale de ma thèse s'articule autour du second axe. Plus précisément, je vais montrer que **par l'intégration de concepts provenant de l'ordonnancement temps-réel, de la compilation et de la programmation synchrone, nous sommes en mesure de produire automatiquement des implantations temps-réel dirigées par le temps à la fois correctes par construction (et qui ont donc besoin de moins de travail de validation) et plus efficaces que celles produites par les approches existantes jusqu'alors.** Dans notre cas, cela revient à appliquer une approche de type compilation lors de la synthèse d'implantations temps-réel à partir de spécifications qui sont complexes à la fois dans leurs aspects fonctionnels et non-fonctionnels. Réciproquement, on peut également voir cela comme une extension de l'état de l'art de la compilation, afin de prendre en compte des besoins non-fonctionnels et de permettre une prise en compte précise des temps d'exécution au pire cas dans le processus de compilation.

Comparaison entre compilation et ordonnancement temps-réel

Une séparation existe de longue date entre les domaines de la construction de compilateurs et de l'ordonnancement temps-réel. Si ces deux domaines ont le même objectif - la construction d'implantations correctes - la séparation se justifie historiquement par des différences significatives entre les modèles et les méthodes utilisés. En effet, les travaux initiaux dans la communauté de la compilation se sont concentrés sur la génération de code machine *séquentiel* et *ordonnancé statiquement*. L'objectif des optimisations était d'améliorer *la vitesse d'exécution du cas moyen*, mais sans fournir de *garantie formelle d'optimalité ou de performance*. L'optimisation était basée sur l'exploitation

cesseur font l'hypothèse que les communications ne prennent pas de temps. Pour que cette abstraction reste sûre, les durées au pire cas des communications doivent être prises en compte en ajoutant des marges de sécurité dans les durées caractérisant les tâches. Cependant, dans la plupart des cas, ces marges ne sont pas calculées en utilisant une analyse formelle du matériel et des protocoles de communication.

d'informations détaillées concernant la microarchitecture. Etant donné la nature du problème à résoudre, un compilateur devait *toujours retourner un résultat* pour un programme syntactiquement correct.

Dans les premiers systèmes temps-réel multi-tâches, les compilateurs fournissaient les *tâches séquentielles* que le système d'exploitation executait. La conception et l'analyse au niveau système constituaient l'objet de travail de la communauté de l'ordonnancement temps-réel. Les travaux sur ce sujet, tel que l'article fondateur de Liu et Layland [Liu and Layland, 1973], tentaient de fournir *des garanties d'ordonnançabilité formelles* pour des tâches exécutées par un *ordonnanceur dynamique/en ligne (et souvent preemptif)*. Au contraire de la compilation, où l'objectif est la synthèse complète d'un morceau de code, l'ordonnancement temps-réel *ne synthétise pas ou peu de paramètres de l'implantation* (par exemple, les priorités des tâches, ou l'allocation des tâches sur les processeurs dans un système distribué). L'analyse d'ordonnancement était basée sur *des abstractions fortes des tâches et de la plateforme d'exécution*, appelées *modèles de tâches*. Assurer qu'une telle abstraction forte est sûre est un sujet rarement couvert dans les articles d'ordonnancement temps-réel. Cela représentait un problème significatif lors de la mise en pratique industrielle, et particulièrement parce que l'implantation du système était elle même produite pour la plupart à la main (contrairement à une compilation complètement automatique, où tous les aspects de l'ordonnancement et de la génération de code seraient contrôlés par le compilateur). Enfin, l'analyse d'ordonnançabilité pouvait échouer. Les différences entre les modèles et les méthodes mentionnés plus haut sont d'une importance certaine, et peuvent expliquer la séparation initiale entre les communautés de recherche et d'ingénierie. Cependant, ces deux domaines ont significativement évolué depuis. Lors de la conception de techniques de compilation pour les architectures superscalaires et VLIW, la communauté de la compilation a introduit une prise en compte précise des durées d'exécution ainsi que l'utilisation de multiples resources indépendantes. Plus récemment, elle a considéré des modèles d'architecture moins précis et des modèles d'exécution plus dynamiques pour permettre la compilation vers des cibles GPU. D'un autre côté, les approches temps-réel basées sur *l'ordonnancement statique/hors ligne*, plutôt que sur l'ordonnancement dynamique/en ligne, ont gagné en importance lorsque des besoins industriels ont mené à la définition de standards tels que IMA/ARINC 653 (pour les systèmes avioniques), AUTOSAR (pour les systèmes automobiles), ou TTA, FlexRay, TTEthernet (pour les bus de communication). Qui plus est, les systèmes basés sur ces standards nécessitent d'être configurés en détail (ce qui peut nécessiter de synthétiser l'information de configuration), ce qui a pour conséquence la nécessité d'avoir recours à des approches de type compilation pour traiter tous les aspects de l'implantation, et pas seulement l'ordonnancement.

Dès lors, sans pour autant se confondre entièrement, les objets d'étude de ces deux

communautés se recouvrent aujourd'hui largement, ce qui justifie une mise en commun des efforts de recherche et d'ingénierie. De plus, dans ce processus de mutualisation, un troisième domaine peut jouer un rôle important : la conception et l'implantation de langages synchrones.

Comme il est écrit au chapitre 2, les premiers langages synchrones ont été introduits par la communauté du temps réel dans le but de permettre la spécification (formellement complète) de systèmes de contrôle embarqué complexes. Ces systèmes sont de taille importante, et nécessitent donc le recours à une modélisation hiérarchique, et incluent des structures de contrôle complexes, qui impliquent de la concurrence, du contrôle conditionnel, de la préemption, et des comportements à états qui ne peuvent pas être bien représentés par les modèles de tâches très abstraits. L'étude des langages et formalismes synchrones sont devenus un domaine de recherche indépendant qui a des applications dans la spécification, l'analyse formelle, et l'implantation à la fois des systèmes embarqués et de la conception matérielle [Benveniste et al., 2003]. Deux aspects de ce domaine sont particulièrement intéressants dans le contexte de cette thèse. Tout d'abord, les formalismes synchrones peuvent être vus comme des extensions naturelles des formalismes utilisés pour l'ordonnancement temps réel (les graphes de tâches dépendantes), mais aussi pour la compilation (les représentations *single static assignment* et les graphes de dépendences de données). Les formalismes synchrones peuvent donc être utilisés comme un terrain d'entente entre ces domaines pour la modélisation formelle. Mais au-delà d'être un terrain d'entente, des travaux existants sur les langages synchrones nous procurent également des techniques efficaces pour la manipulation de structures de contrôle complexes par le biais de ce que l'on appelle les *horloges* et les *retards*.

Contribution

Comme on le voit dans la figure 2, ma thèse s'articule à l'intersection des trois domaines présentés précédemment. Son objectif général est de montrer qu'une meilleure intégration entre ces domaines résulte dans la simplification de la construction de systèmes de contrôle embarqué, et dans une amélioration de leur qualité. Plus précisément, nous montrons que cela est possible pour la classe des systèmes basés sur l'utilisation d'un paradigme d'ordonnancement temps-réel hors-ligne (basé sur des tables). Pour de tels systèmes, nous prenons en compte une technique de compilation avancée (le *software pipelining*) pour améliorer les résultats du processus d'ordonnancement hors-ligne. Notre approche est similaire aux travaux classiques de compilation selon deux autres aspects :

- Contrairement aux travaux classiques sur le temps-réel, nous ne nous limitons pas à l'ordonnancement des systèmes, mais permettons la génération complètement automatique de leurs implantations,

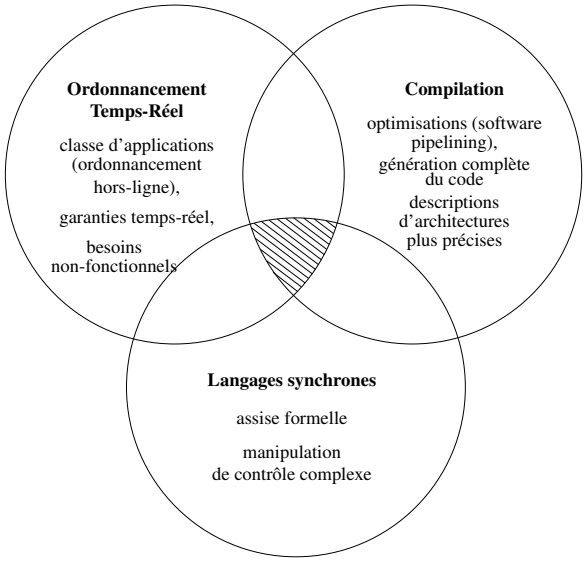


Figure 2: Cette thèse se positionne au carrefour entre trois domaines de recherche

- Pour préserver l'efficacité et permettre la génération de code notre approche utilise une plateforme d'exécution et des modèles d'applications qui sont plus précis que ceux utilisés dans le domaine de l'ordonnancement temps-réel. En particulier, nous utilisons des formalismes synchrones pour représenter des fonctionnalités complexes et nous permettons la prise en compte de divers types de besoins non-fonctionnels, ce qui permet la modélisation naturelle de tous les besoins d'un cas d'étude avionique.

Cependant, notre travail ne se résume pas simplement à l'application de techniques de compilation à une nouvelle classe de systèmes. En effet, contrairement aux travaux de compilation classiques, notre approche :

- se situe au niveau système, ce qui nécessite une abstraction significative de la plateforme d'exécution,
- fournit des garanties temps-réel dûres, ce qui nécessite une prise en compte sûre du temps,
- considère l'exécution conditionnelle des systèmes,
- considère des besoins non-fonctionnels de différents types, ce qui modifie fondamentalement la nature du problème d'optimisation à résoudre.

La définition de notre approche est basée sur l'adaptation non triviale de concepts, de modèles et d'algorithmes venant des domaines de l'ordonnancement temps-réel, de la compilation et des langages synchrones. Cette adaptation a été facilitée par l'utilisation

de formalismes synchrones qui nous ont fourni un terrain d'entente pour la modélisation et l'analyse des systèmes considérés. Notre travail concerne tous les aspects du problème d'implantation : la conception de formalismes de modélisation, la définition d'algorithmes d'ordonnancement hors-ligne basés sur l'utilisation de tables, et la génération de code exécutable.

Une description détaillée des contributions et de l'originalité de notre travail par rapport aux travaux existants dans les trois domaines de recherche considérés sera fournie lorsque les différents concepts relatifs aux trois domaines auront été introduits (en particulier aux sections 6.1 et 7.1).

Organisation de la thèse. Etant donné que l'objectif de mon travail a été la conception d'une technique de compilation (permettant l'ordonnancement et la génération de code de façon complètement automatisée), celui-ci n'aurait pas été complet sans une évaluation des approches de modélisation, d'ordonnancement et de génération de code que nous fournissons. Pour permettre cette évaluation, les modèles et techniques décrits dans cette thèse ont été accompagnés par leur implantation dans un outil décrit dans la figure 3. Nous appelons cet outil *le compilateur temps-réel LoPhT*[4].

L'organisation de ma thèse suit de près celle de LoPhT, qui est présentée dans la figure 3. C'est pour cela que cette figure est utilisée pour représenter à la fois ma thèse et l'outil LoPhT. Afin d'évaluer la généralité du travail effectué, le compilateur LoPhT a deux cibles : les systèmes partitionés dirigés par le temps qui se conforment au standard IMA/ARINC 653, et les systèmes multiprocesseurs sur puce (MPSoCs) dans lesquels les communications inter-processeurs sont réalisées par le biais d'un réseau sur puce (NoC). Le travail sur les cibles MPSoC a été réalisé en collaboration avec Manel Djemal dans le cadre de sa thèse de doctorat [Carle et al., 2013, 2014] et sera seulement mentionné rapidement dans cette thèse. Dans la figure 3, les éléments en noir sont ceux qui existaient déjà avant le début de ma thèse. Les éléments en rouge (les boites remplies de jaune, ainsi que les éléments à gauche dans la boite "Ordonanceur") sont ceux que j'ai développés lors de ma thèse et qui sont décrits dans ce manuscrit, et les éléments verts (les boites à droite de la figure, contenant la mention M. Djemal/T. Carle) correspondent aux extensions spécifiques aux cibles MPSoC.

Mon manuscrit ne comporte pas un chapitre dédié à l'état de l'art. Cela est dû au fait que ma thèse recouvre trois domaines de recherche ainsi que les relations existant entre eux. Présenter toute l'information correspondante en une fois aurait dégradé la clarté de la présentation. Au contraire, nous fournissons dans le chapitre 2 une introduction aux formalismes synchrones (l'assise formelle de notre approche), puis chaque chapitre technique fournit un état de l'art partiel, souvent regroupé dans une section dédiée (par

[4] de l'anglais **Lo**gical to **Ph**ysical **T**ime Compiler : compilateur du temps logique vers le temps physique

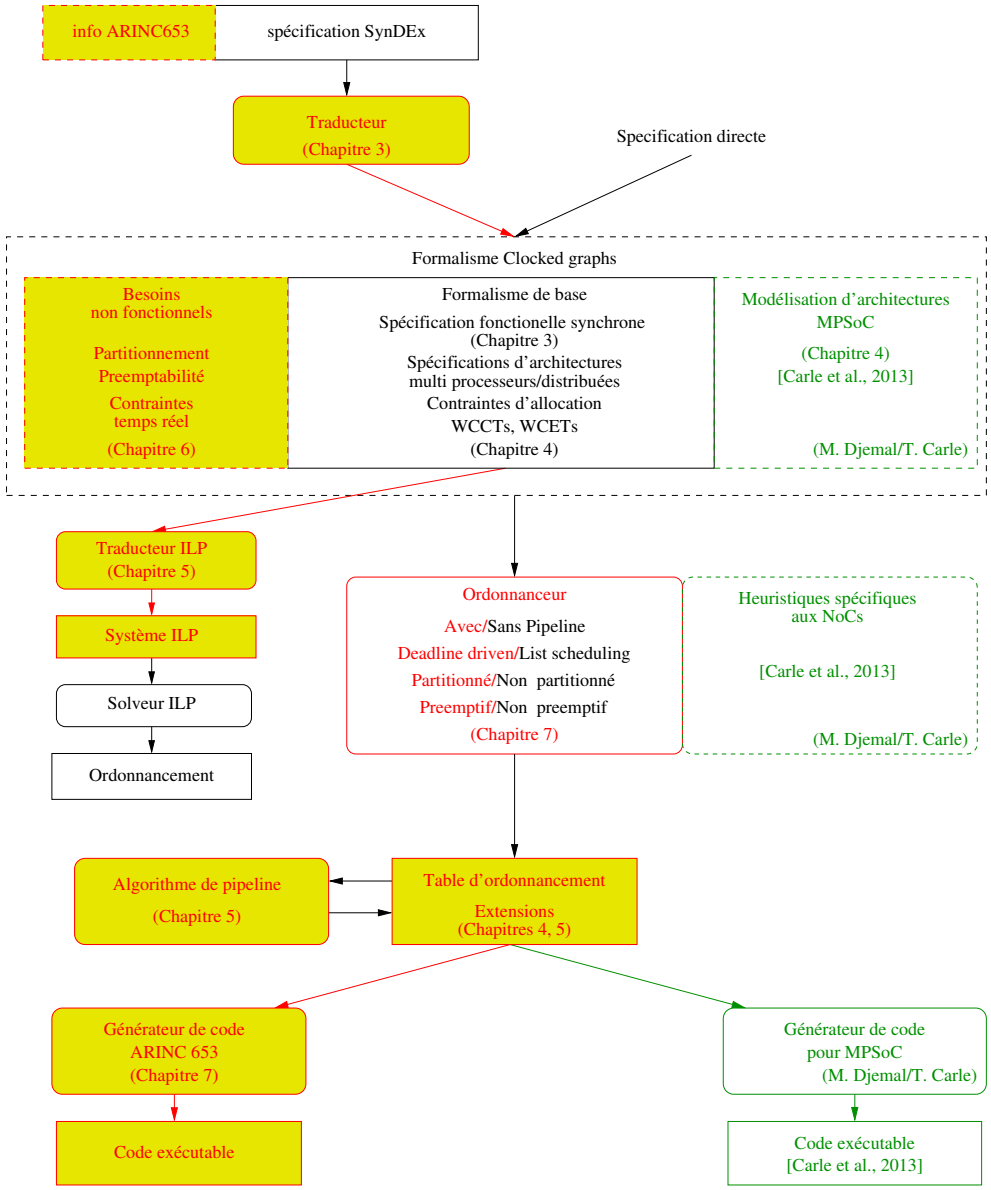


Figure 3: Description du compilateur temps-réel LoPhT

exemple les sections 6.1 et 7.1). De cette façon, l'exploration des relations complexes existant entre les modèles et les algorithmes utilisés dans les trois domaines de recherche considérés ne se fait qu'après que ces modèles et algorithmes aient été définis.

Le formalisme utilisé par les algorithmes d'ordonnancement de LoPhT s'appelle Clocked Graphs. Une spécification Clocked Graphs comprend une spécification fonctionnelle et une spécification non-fonctionnelle incluant la définition de l'architecture et des besoins non-fonctionnels de l'application. La spécification fonctionnelle est donnée sous la forme d'un programme flot de données synchrone de bas niveau, écrit dans un langage qui s'appelle également Clocked Graphs. Ce langage synchrone, qui a été défini pour la première fois avant le début de ma thèse, est présenté au chapitre 3. Ce chapitre explore également les relations existant entre ce langage et :

- des formalismes de spécification fonctionnelle de haut niveau. J'y explique comment des programmes synchrones de haut niveau écrits en SynDEx peuvent être transformés en programmes Clocked Graphs,

- les modèles de tâches utilisés pour l'ordonnancement temps-réel. J'y explique comment des programmes synchrones peuvent être abstraits dans des représentations basées sur des systèmes de tâches dépendentes.

Le chapitre 4 présente notre approche de modélisation des ressources de la plateforme d'exécution, ainsi que notre approche de modélisation de l'allocation des ressources, basée sur l'utilisation de tables d'ordonnancement. Ce chapitre définit tout d'abord le formalisme utilisé pour la description des architectures et celui utilisé pour la description des tables d'ordonnancement, tels qu'ils existaient dans LoPhT avant le début de ma thèse. Il inclut également une comparaison rapide entre le modèle de réservation de ressources que nous avons choisi (l'ordonnancement hors-ligne basé sur les tables d'ordonnancement) et d'autres techniques de réservation de ressources, telles que la réservation de bande passante ou l'ordonnancement en-ligne basé sur les priorités.

Le chapitre 5 introduit les extensions que j'ai apportées à ces modèles pour permettre l'application du software pipelining, et pour permettre la modélisation des plateformes MPSoC.

Le chapitre 6 présente la première de nos deux principales contributions techniques : l'utilisation de techniques de software pipelining pour améliorer la qualité des tables d'ordonnancement temps-réel. Ce chapitre inclut un état de l'art approfondi et une section d'évaluation de la méthode.

Le chapitre 7 définit les extensions que j'ai apportées au formalisme Clocked Graphs pour permettre la modélisation de systèmes partitionnés dirigés par le temps incluant des besoins non-fonctionnels complexes. Il inclut une introduction aux systèmes dirigés par le temps (ainsi qu'un état de l'art sur ce sujet).

Le chapitre 8 présente les techniques d'ordonnancement hors ligne et de génération de code capables d'opérer sur les spécifications décrites au chapitre précédent. Finalement, le chapitre 9 conclut ce manuscrit.

Abstract

There is a long standing separation between the fields of compiler construction and real-time scheduling. While both fields have the same objective - the construction of correct implementations - the separation was historically justified by significant differences in the models and methods that were used. Nevertheless, with the ongoing complexification of applications and of the hardware of the execution platforms, the objects and problems studied in these two fields are now largely overlapping. In this thesis, we focus on the automatic code generation for embedded control systems with complex constraints, including hard real-time requirements. To this purpose, we advocate the need for a reconciled research effort between the communities of compilation and real-time systems. By adapting a technique usually used in compilers (software pipelining) to the system-level problem of multiprocessor scheduling of hard real-time applications, we shed light on the difficulties of this unified research effort, but also show how it can lead to real advances. Indeed we explain how adapting techniques for the optimization of new objectives, in a different context, allows us to develop more easily systems of better quality than what was done until now. In this adaptation process, we propose to use synchronous formalisms and languages as a common formal ground. These can be naturally seen as extensions of classical models coming from both real-time scheduling (dependent task graphs) and compilation (single static assignment and data dependency graphs), but also provide powerful techniques for manipulating complex control structures. We implemented our results in the LoPhT compiler.

Chapter 1

Introduction

1.1 Embedded systems design

Embedded systems are computing systems with a dedicated function inside a larger physical or computing system. In this thesis, we are particularly interested in *embedded control systems* [Lee and Seshia, 2011] whose task is to regulate the evolution of (physical) processes in order to ensure their correct functioning. Control and regulation must be understood here in the sense of control theory [Doyle et al., 1990], meaning that an embedded control system is the means of performing the *automatic control* of a *plant* such as a plane, a train, a nuclear reactor, a mobile phone, *etc.*

As pictured in Fig. 1.1, an embedded control system interacts with the controlled plant by means of *sensors* and *actuators*. The sensors measure specific (analog) characteristics of the plant and present the resulting information in a discrete form allowing digital treatment by the embedded controller. The discrete output values of the controller are used by the actuators to exert (analog) control over the plant.

For instance, the guidance, navigation, and control (GNC) embedded controller of a

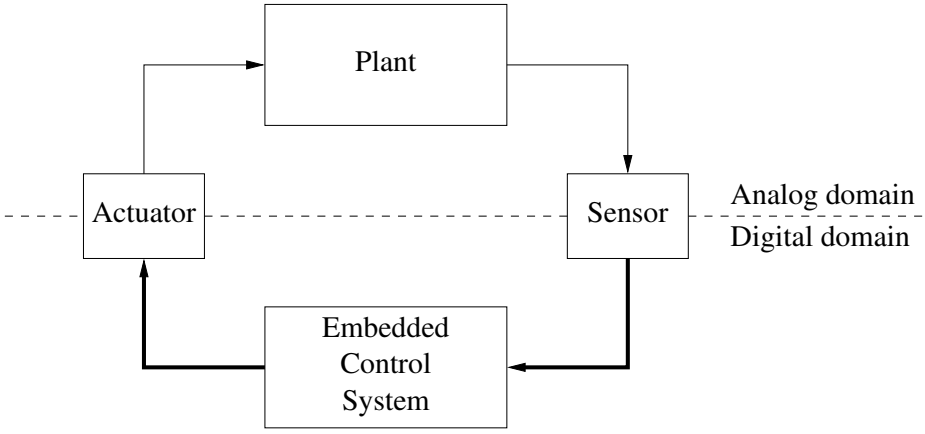


Figure 1.1: Interaction between plant and embedded control system

plane uses a variety of sensors (pressure, temperature, GPS, accelerometers, gyroscopes, *etc.*) to determine its location, speed, and attitude. These values are cyclically used by the GNC controller to manipulate (through actuators) the position of steering controls and the power of thrusters in order to maintain the desired trajectory.

The design of embedded control systems is highly inter-disciplinary. At the top level, it is driven by control engineers and systems engineers. Control engineers provide a specification of the expected behavior of the controller covering functionality and temporal requirements. Defining the functionality of a system consists is specifying which operations must be executed, and in which order, to compute the commands of the actuators starting from the values produced by sensors[1]. But the functional specification alone is not enough to represent the behavior of a control system. To perform effective control, changes in the physical processes of the plant must be followed *in a timely fashion* by changes in actuators. For instance, the GNC embedded controller mentioned above must rapidly react to cross-wind gusts to maintain the stability of the plane, especially during landing phases. For this reason, the attitude control part of the GNC controller is executed once every 50ms [Kent and Williams, 2002].

The specification of these timeliness constraints is realized under the form of *real-time requirements* such as *periods*, *latencies*, *release dates*, *deadlines* attached to the elements of the functional specification. Embedded control systems are therefore *real-time systems* [Butazzo, 2002], which are a sub-class of *reactive systems* [Harel, 1987].

While control engineers are concerned with specifying the behavior of the embedded controller, systems engineers consider the problems of implementing the embedded controller and then providing tools for managing it during its entire lifetime, which is typically of more than 20 years for a commercial aircraft, more than 30 years for a nuclear reactor, more than 150000km on average for a car, *etc.*. At top level, systems engineering considers macroscopic properties such as cost (in development, maintenance, and possibly disposal), reliability (for both cost and safety reasons), risk, *etc.* Such macroscopic objectives can be set under the form of *non-functional requirements*, and systems engineering provides methods for *refining* such high-level requirements into lower-level requirements such as the ones we consider in this thesis. For instance, Chapter 7 will consider the following requirements coming from systems engineering: the *execution platform* is already fixed[2], each operation of the functional specification has been assigned a *criticality*, some operations are *preemptable* and the others are *non-preemptable*, and part of the mapping of the functional specification onto the platform is already fixed. Other non-functional properties that can be considered at this lower level are fault tolerance, energy consumption, evolutivity, *etc.*

[1] The internal state of the system may also be used.

[2] Which may also impose some supplementary *real-time* requirements in addition to the ones coming from the control specification.

This is the level where my work takes place: I assume in this thesis that previous engineering phases have produced a functional specification, have fixed the target execution architecture (which can be multiprocessor/distributed), and have provided a set of non-functional requirements including real-time ones. Starting from this input, my objective is to automatically build a correct embedded control system.

Correctness involves here two aspects: functional correctness and the respect of the non-functional requirements. Functional correctness means that the system behaves as prescribed by the functional specification. Ensuring the functional correctness of such a system involves multiple disciplines such as *compilation* (to ensure that each operation of the specification is correctly implemented by a sequential piece of code), *parallel/distributed computing* (for the construction of correct multiprocessor implementations), *etc.* Of particular interest here is the domain of *synchronous language design and implementation* which puts together all the disciplines needed to ensure functional correctness. This is due to the fact that synchronous languages and formalisms, described later in this thesis, were designed specifically to allow the functional specification of embedded control systems. This explains their pervasiveness in embedded systems design and their use in this thesis.

Ensuring the respect of non-functional requirements such as those mentioned above (real-time, criticality, preemptability, power consumption, *etc.*) is mainly studied in the field of *real-time scheduling*, which itself takes advantage of results from hardware design, operating systems design, parallel/distributed computing, fault tolerance, *etc.*

The case for stronger integration. The design of a complex embedded system usually brings together people from all the aforementioned fields. However, in current scientific work and industrial practice the fields remain largely segregated. My opinion is that such a strong segregation results in increased design costs and decreased implementation efficiency, mainly due to 2 causes:

- The interfaces between the various fields, which are usually materialized under the form of model transformations, often abstract away significant details of the source models, which results in efficiency loss. This efficiency loss is often accompanied by the use of unsafe abstractions, which requires validation through extensive tests, and thus increases development costs[3].

- The lack of communication between the scientific and industrial communities means

[3]For instance, tasks models used in real-time scheduling often abstract away all conditional control associated to the tasks, thus adding pessimism to the analysis. Furthermore, many task models used in multiprocessor scheduling assume that communications take no time. For this abstraction to be safe, the worst-case duration of communications should be taken into account by means of overheads added to the durations of tasks. But in most cases, these overheads are not derived from a formal analysis of the communication hardware and protocols.

that some solutions developed in one field will not be applied in another, even when appropriate.

The work of this thesis addresses both points for specific classes of target architectures. To reduce efficiency loss, we use precise models of the application (where conditional control is taken into account), of the platform (where communication costs and the precise execution mechanisms are considered), and of the non-functional requirements, which are adapted to our target execution platforms (time-triggered partitioned systems). Having a precise modeling of the execution platform also reduces the need for unsafe abstractions.

But the main contribution of my thesis is along the second axis. More precisely, I will show that **by closely integrating concepts from real-time scheduling, compilation, and synchronous programming, we are able to automatically produce time-triggered real-time implementations that are both correct by construction (thus requiring less validation) and more efficient than those produced by existing approaches.** This amounts to applying a compilaton-like approach in the synthesis of real-time implementations from specifications that are complex in both their functional and non-functional aspects. Conversely, this can also be seen as extending the compilation state of the art to take into account non-functional requirements and to allow for a careful accouting of worst-case execution time in compilation.

1.2 Compilation *vs.* Real-time scheduling

There is a long standing separation between the fields of compiler construction and real-time scheduling. While both fields have the same objective - the construction of correct implementations - the separation was historically justified by significant differences in the models and methods that were used. Indeed, initial work in the compilation community was focused on generating *statically scheduled sequential* machine code. The optimization objective was to improve *average-case speed*, but with *no formal optimality or performance guarantees*. Optimization was based on the *use of detailed microarchitectural information*. Given the nature of the problem it solved, a compiler was expected to *always provide a result* for a syntactically-correct program.

In early real-time multi-tasking systems, compilers were meant to provide the *sequential tasks* that were run by the OS. The system-level design and analysis were the object of the real-time scheduling community. Work on this subject, such as the seminal paper of Liu and Layland [Liu and Layland, 1973], focused on providing *formal schedulability guarantees* for tasks run by a *dynamic/on-line (usually priority-preemptive) scheduler*. Unlike in compilation, where the objective is the complete synthesis of a piece of code, in real-time scheduling *few or no parameters of the implementation would be synthesized* (*e.g.* the priorities of the tasks, or the allocation of tasks to processors in a distributed

system). Schedulability analysis was based on *coarse abstractions of the tasks and the execution platform*, known as *task models*. Ensuring that such a coarse abstraction is safe is a subject seldom covered in real-time scheduling papers. This was a significant problem in industrial practice, especially given that the system implementation itself remained largely manual (as opposed to the fully automatic compilation, where all aspects of scheduling and code generation were controlled by the compiler). Finally, schedulability analysis could fail.

The differences between the models and methods mentioned above are indeed important and may explain the initial separation of the research and engineering communities. However, both fields have significantly evolved since. When designing compilation techniques for superscalar and VLIW architectures, the compilation community introduced precise timing accounting and the use of multiple, independent execution resources. More recently, it has considered less precise hardware models and more dynamic execution models in order to allow compilation for GPU targets. On the other hand, the real-time scheduling approaches based on *static/off-line*, rather than dynamic/on-line scheduling, gained more importance when industrial needs resulted in the definition of standards such as IMA/ARINC 653 (for avionics systems), AUTOSAR (for automotive systems), or TTA, FlexRay, TTEthernet (for communication buses). Furthermore, systems based on these standards require significant configuration (synthesis), which results in the need for compilation-like approaches handling all aspects of an implementation, and not just the scheduling.

Thus, without becoming indistinct, the types of objects studied by the two communities are today largely overlapping, which should justify a mutualization of the research and engineering effort. Furthermore, in this mutualization process a third research field should play a significant part: the design and implementation of synchronous languages.

As we shall see in the next chapter, the first synchronous languages have been introduced in the real-time community in order to allow the formally sound specification of complex embedded control system. Such systems have large sizes, thus requiring hierarchical modeling, and feature complex control structures, involving concurrency, conditional control, preemption, and stateful behaviors that are not well represented using the very abstract task models. The study of synchronous languages and formalisms became a stand-alone research field, with applications in the specification, formal analysis, and implementation of both embedded systems design and hardware design [Benveniste et al., 2003]. Two particular aspects of this field are appealing from the point of view of this thesis. First of all, synchronous formalisms can be seen as natural extensions of formalisms of both real-time scheduling (the dependent task graphs) and compilation (static single assignment representations and the data dependency graphs). Synchronous formalisms can therefore be used as a common ground for formal modeling. But beyond being a formal

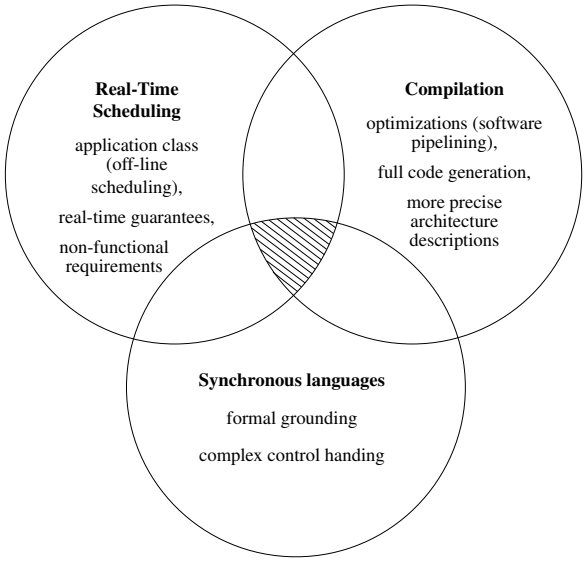


Figure 1.2: This thesis is positioned at the intersection of 3 research fields

ground, previous work on synchronous languages also provides powerful techniques for manipulating complex control structures by means of so-called *clocks* and *delays*.

1.3 Contribution

As pictured in Fig. 1.2, my thesis is developed at the intersection of the 3 research fields presented above. Its most general objective is to show that better integration between these fields allows the easier construction of better embedded control systems.

More precisely, we show that this is possible for the class of systems relying on an off-line (table-based) real-time scheduling paradigm. For such systems, we take into account an advanced compilation technique (software pipelining) in order to improve offline scheduling results. Our approach is similar to classical compilation work in two other aspects:

- Unlike in classical real-time work, we not only perform scheduling, but allow fully automatic generation of the implementation.

- To preserve efficiency and allow code generation our approach uses execution platform and application models that are more precise than those used in real-time scheduling. Particular points here are the use of synchronous formalisms to represent complex functionalities and the ability to consider multiple types of non-functional requirements, which allows the natural modeling of all the requirements of an avionics case study.

But our work does not consist in simply applying compilation techniques to a new class of systems. Indeed, unlike in classical compilation work, our approach:

- works at system level, requiring a significant abstraction of the execution platform,
- provides hard real-time guarantees, which in turn requires a safe accouting of time,
- considers conditional (predicated) execution, and
- considers non-functional requirements of various types, which fundamentally changes the nature of the optimization problem that must be solved.

The definition of our approach is based on a non-trivial adaptation of concepts, models, and algorithms of the real-time scheduling, compilation, and synchronous languages fields, which were facilitated by the use of synchronous formalisms to provide a common formal ground for modeling and analysis. Our work covered all aspects of the implementation problem: the design of modeling formalisms, the definition of table-based off-line scheduling algorithms, and the generation of executable code.

A detailed description of the contributions and originality of our work with respect to existing work in the 3 research fields will be provided once the various concepts of the 3 fields will be introduced (and in particular in Sections 6.1 and 7.1).

Organization of the thesis. Given that the object of my work has been the design of what is essentially a compiling technique (allowing fully automatic scheduling and code generation), my work would not have been complete without an evaluation of the modeling, scheduling, and code generation approaches we provide. To allow evaluation, all the developments described in this thesis were accompanied by a significant prototyping activity that resulted in the tool flow of Fig. 1.3. We call this tool flow the *LoPhT*[4] *real-time compiler.*

The organization of my thesis closely follows that of the LoPhT tool, presented in Fig. 1.3. This is why we use this figure to detail both. To evaluate the generality of the work, the LoPhT compiler has two targets: partitioned time-triggered systems compliant with the IMA/ARINC653 standard and multiprocessor systems-on-chips (MPSoCs) where interprocessor communication is realized through a network-on-chip (NoC). The work specific to the MPSoC targets was realized in collaboration with Manel Djemal as part of her PhD thesis [Carle et al., 2013, 2014] and will only briefly be mentioned in this thesis. In Fig. 1.3, black elements are those that existed before my thesis started. Red elements (the boxes filled in yellow, as well as the left part of the "Scheduling toolbox" box) are those developed as part of and described in my thesis, and green elements (the boxes

[4]For **Lo**gical to **Ph**ysical **T**ime Compiler

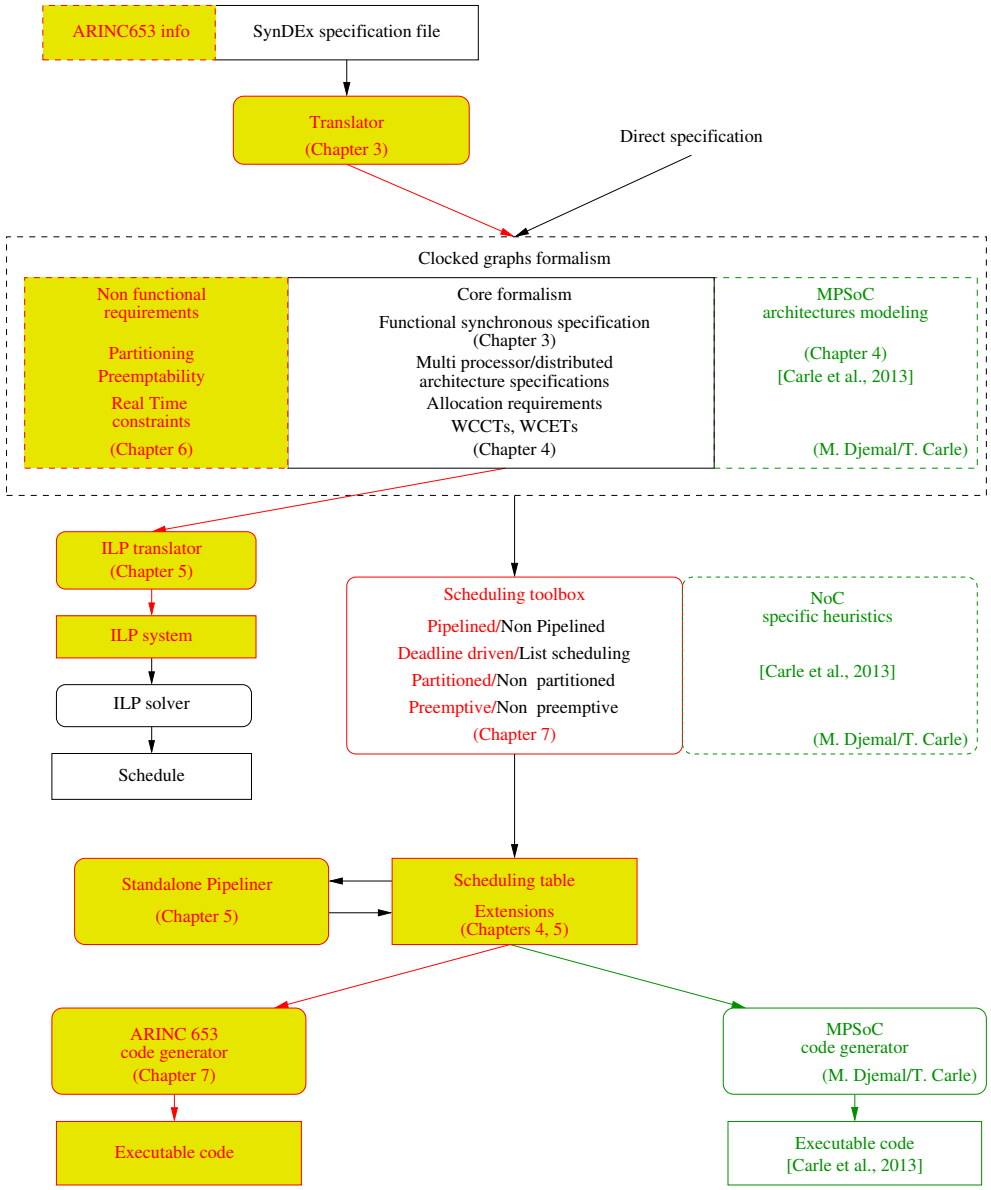


Figure 1.3: Tool flow of the LoPhT real-time compiler

in the right part of the figure, labelled with (M. Djemal/T. Carle)) are those corresponding to the MPSoC-specific extensions.

My thesis does not have a single, dedicated state of the art chapter. This is due to the fact that my thesis covers 3 research fields and the relations between them, and presenting this amount of information at once would have compromised the clarity of the presentation. Instead, we provide in Chapter 2 an introduction to synchronous formalisms (the formal basis of our approach) and then, each technical chapter will provide partial state of the art information, often grouped in a dedicated section (such as Sections 6.1 and 7.1). In this way, exploring the complex relationships existing between particular models and algorithms used in the 3 research fields is only done after the models and algorithms are properly defined.

The formalism taken as input by the scheduling algorithms of LoPhT is called Clocked Graphs. A Clocked Graphs specification comprises a functional specification and a non-functional specification including the definition of the architecture and that of the non-functional requirements. The functional specification part consists in a low-level dataflow synchronous program written in a language that is also called Clocked Graphs. This synchronous language, which was first defined before the beginning of my thesis, is presented in Chapter 3. The chapter also explores the relations between this language and:

- Higher-level functional specification formalisms. I explain here how high-level SynDEx synchronous programs can be translated into Clocked Graphs.
- Task models used in real-time scheduling. I explain here how synchronous programs can be abstracted away as dependent task systems.

Chapter 4 presents our approach to modeling the resources of the execution platform and then our approach to modeling resource allocation (by means of scheduling tables). The chapter first defines the basic architecture description formalism and the basic scheduling table description formalism that existed in LoPhT before the start of my thesis. The chapter also includes a brief comparison between the resource allocation model we chose (table-based off-line scheduling) and other resource allocation techniques, such as bandwidth reservation or on-line priority-based scheduling.

Chapter 5 introduces the extensions I brought to these models to allow the application of software pipelining, and to allow the modeling of MPSoC platforms.

Chapter 6 presents the first of our two main technical contributions, namely our use of software pipelining techniques to improve the quality of real-time scheduling tables. It includes a thorough state of the art and an extensive evaluation section.

Chapter 7 defines the extensions I added to the Clocked Graphs formalism to allow the modeling of partitioned time-triggered systems with complex non-functional requirements. It includes an introduction to time-triggered systems (and a related work on the

subject).

Chapter 8 presents the off-line scheduling and code generation technique capable of handling the specifications of the previous chapter. Finally, Chapter 9 concludes.

Part I

Offline scheduling: fundamental notions and contributions on the specification models for embedded systems

Chapter 2

Introduction to synchronous formalisms

Contents

2.1 The synchronous model of computation

Synchronous languages emerged from the real-time systems community as a way to describe precisely the functional aspects of complex embedded control systems with hard timing constraints [Potop-Butucaru and Sorel, 2014]. This family of languages relies on the *synchronous model of computation* which provides sound formal foundations for describing applications with a cyclic execution model. The synchronous model allows the non-ambiguous representation of behaviors at a high level of abstraction, and then supports formally sound analysis and implementation techniques allowing the synthesis of correct and efficient implementations.

In the synchronous model, the execution of a program is divided into an infinite sequence of execution steps called *reactions* [Benveniste et al., 2003]. The key point of the synchronous model is the *synchrony hypothesis* which states that each reaction happens atomically, as if its computations take zero time [Halbwachs, 1993]. Under this hypothesis, time flows as a sequence of discrete, ordered *instants*, each corresponding to an instantaneous reaction of the system [Potop-butucaru et al., 2005]. Computation instants do not need to be linked to physical time. They just define a sequence of abstract events,

partially or totally ordered by a precedence relation. This is why we say that they define a *logical time* scale.

In the synchronous model, the synchrony between computations of a given instant has a very strong meaning:

- Two reads of the same variable/signal performed during the same instant must always provide the same result, because they happen at the same time.
- If a variable/signal is written during an instant, then all reads will produce this value. Furthermore, no variable/signal should be written twice during an instant, or otherwise it must be specified which of the writes gives the signal/variable its value for the instant.

This property applies to all variables/signals, be them inputs, state variables, or internal variables/signals. Thus, at each instant all the computations of a synchronous model have access to a single, coherent state. No computations occur outside of the time scale defined by the logical instants of the system, the system and its environment being invariant: all changes in the inputs, outputs and in the system state occur only during the logical instants.

The second main tenant of the synchronous model is *causality*. Inside an execution instant, computations take place in zero time, but their execution are performed *causally*, without speculation. Causality amounts to requiring that inside each execution instant the computations can be (partially) ordered so that signal/variable productions precede their reads. In other terms, this amounts to enforcing the respect of the data and control dependencies inside each reaction [Berry, 1996].

Enforcing causality has two important practical consequences: first of all it allows the compilers to automatically detect and reject specifications that cannot be scheduled in a causal way. The second consequence is *functional determinism*: a causal synchronous system will always produce the same outputs when given the same set of inputs. This property is verified by the system regardless of the schedule of the operations, as long as it respects all the dependencies between operations inside each instant.

2.1.1 Abstraction issues

Of course, computations performed by real hardware do take time, which means that synchronous models provide an abstract view of embedded applications. However, current industrial adoption shows that this abstraction level is a good one. We provide here several points justifying (*a posteriori*) this adoption.

As explained above, the synchronous model and the synchronous languages natively include characteristics of embedded control systems: a cycle-based execution model, concurrency, causality.

In conjunction with causality, the synchrony hypothesis provides a strong formal basis for defining well-formed properties that have emerged as good design practices in all fields dealing with the specification and implementation of concurrent behaviors: the absence of races or hazards in software and hardware design, the use of atomic interactions such as transactions, the enforcement of functional determinism. For instance, the development of internal compiler representations such as static single assignment (SSA) had similar objectives. All synchronous formalisms allow simple interpretation in universally recognized mathematical models such as the Mealy machines and the digital circuits. In turn, this allows the use of established formal verification and optimization techniques.

Finally, abstracting away physical time under the form of logical time provides a powerful mechanism of enforcing the separation of concerns between functional aspects of the system on one side, and the real-time non-functional requirements on the other. Indeed, the instantaneity assumption of the *synchronous hypothesis* is interpreted in practice as the capacity of the system to react quickly enough to changes in the physical processes of the controlled plant. In this case, it is not necessary to take into account physical changes occurring between the sensor input and the actuator output.

Furthermore, since computations (conceptually) happen instantaneously, the modeling of an application is independent of the platform on which it will be executed. This means that the application developper can focus on the specification of the functional behaviour of the application, and leave the mapping (distribution) and scheduling of the computations to the compiler or online scheduler.

Mature synchronous languages have been developped and used in both academic and industrial contexts. In the next sections, we will focus on the definition of logical time in synchronous specifications, and how its representation has evolved from a simple model to more complex and expressive forms, as new languages were developped. Our last section briefly presents how real-time constraints can be applied to a synchronous specification, and how this specification can be implemented in a particular case.

2.1.2 Clocks and Single-clock synchronous languages

A key notion in synchronous languages is that of *logical clock*. We saw in the previous section that the synchronous model defines a notion of logical time, where the execution of the system is organized as a set of instants that are partially or totally ordered. At each instant, a reaction of the system is performed. Logical clocks are *totally ordered sets of execution instants*. Logical clocks are used to identify the execution instants where a certain event takes place. In particular, logical clocks are used to determine when the various operations of the system are executed, which means that they are used as *activation conditions* to represent the conditional control of the application. As sets of instants, clocks are partially ordered by the inclusion relation. We say that a clock $c1$ is a sub-clock of

clock $c2$, denoted $c1 \leq c2$ if the set of instants of $c1$ is included in that of $c2$.

The first synchronous languages [Halbwachs et al., 1991, Berry, 1996] only considered systems where execution instants are totally ordered. This set, by itself, is a logical clock, known as the *base clock* of the system. Any synchronous language where each specification has a base clock is classified as a single-clock synchronous language. In single-clock languages, any clock of a program is defined starting from the base clock (by means of predicates, affine relations, *etc.*) and is a sub-clock of the base clock. The synchronous language Clocked Graphs, used in this thesis, is a single-clock synchronous language, and it will be presented in Section 3.2.2.1.

2.1.3 Polychronous languages

For certain embedded control systems, having a single logical time base may not allow a natural description of the system behavior. This happens in systems where various parts need to synchronize on different physical quantities. For example, the ignition control system of a car requires both time-triggered computations for handling the input from the sound sensors, and computations triggered by the engine crankshaft (*i.e.* triggered at certain angles during the rotation of the engine). Defining a base clock from which both the time-triggered and the rotation-triggered clocks are derived can be done for specification purposes, but it imposes artificial relations between the two time bases which must be removed at implementation time. To allow a more natural specification and analysis, languages such as Signal [Guernic et al., 1986, Benveniste and Guernic, 1990, Guernic et al., 2003] and ΨC [Chabrol et al., 2009] allow the definition of multiple *independent* time bases on which the various parts of the applications can synchronize. The independent time bases allow different parts of an application to evolve in total temporal independence from one another. These languages are called *polychronous*.

2.1.4 Languages with affine clocks

Some languages such as Giotto [Henzinger et al., 2000] or PRELUDE [Forget et al., 2010] have an intermediate way of representing time bases. These formalisms define multiple time bases, but then relate them in a deterministic way by means of *affine relations* defining frequency (rate, period) and phase relations between them. We call such clocks *affine clocks*. Execution instants of these systems are totally ordered, which would allow their faithful representation using single-clock formalisms. However, using multiple base clocks related in phase and offset allows a more natural description, analysis and code generation of *multi-periodic task systems*.

One particular problem in the synchronous modeling of such systems is the representation of tasks with low period and long duration. Such tasks are not fast enough to be seen as instantaneous in the time base provided by the clocks of tasks of higher period.

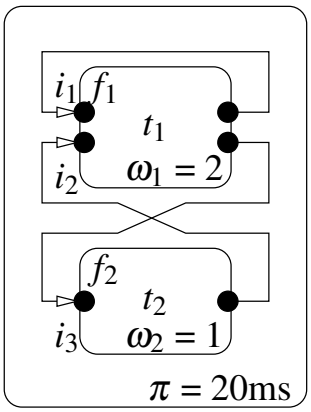


Figure 2.1: Small specification example in the Giotto formalism

In languages based on affine clocks, both time bases (slow and fast) are represented and each task satisfies the synchrony hypothesis in its time base.

Communication and synchronization between tasks belonging to different time bases is deterministic, performed according to rules that take into account the affine relations between clocks. For instance, the Giotto formalism considers that for any given given task t, its outputs are made available in the instant i that follows the one in which t was started, in the time base associated to the clock of t. This means that any other task that starts before i in this model only has access to the previous value of the outputs of t, regardless of its own finishing date. PRELUDE allows a finer control of synchronization by dividing each logical instant into smaller instants and by refining the clocks of the operations that need a finer synchronization. Then the communication semantics are defined using these new, more precise time bases.

2.1.5 From logical time to real time

In the synchronous model, clocks are logical time bases, and the synchronous model does not relate clocks to the flow of some physical quantity such as time, length, *etc.* However, when modeling the real-time aspects of an embedded system the clocks offer the natural places where the non-functional information is attached. Indeed, transforming a logical clock into a "physical" one can be as simple as attaching it a period and an offset specified in milliseconds.

We shall explain how this is done in the Giotto language using the example of Fig. 2.1. Our example specification consists of two Giotto *tasks* t_1 and t_2 grouped in a Giotto *module*, and which are responsible for the computation of functions f_1 and f_2 respectively. Task t_1 has two outputs, one of which is connected to its own input port i_1, and the other which is connected to the input port i_3 of task t_2. Task t_2 has only one output port, which is connected to the second input port of t_1, named i_2. The ratio between the activation

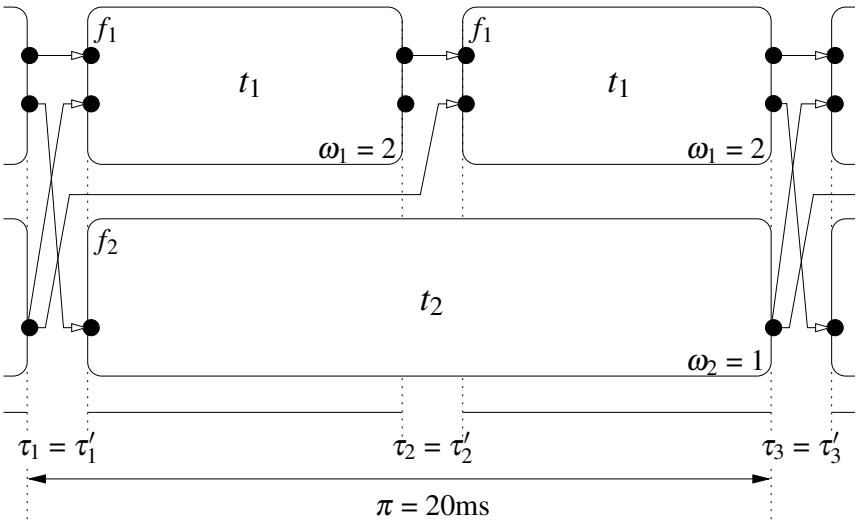


Figure 2.2: Time line for a round of execution of the Giotto specification of Fig. 2.1

periods of both tasks imposes that t_1 activates twice as fast as t_2 (as specified by the ω_1 and ω_2 parameters).

The system designer imposed a real-time constraint on the module: its execution period must be 20 ms. This constraint is immediately translated into execution periods for the tasks: t_2 will inherit the module period (20 ms) because $\omega_2 = 1$, and t_1 will have a period of 10 ms because $\omega_1 = 2$. The first release date for each task is by definition 0. The output of a task is made available not at its activation date, but at the next activation date. In a synchronous interpretation, this means that the output of each task is delayed by one instant. Thus, the output of t_2 will be available at dates 20, 40, 60, ..., and the outputs of t_1 at dates 10, 20, 30, ...

An intuitive view of this execution model is depicted in Fig. 2.2. This figure displays the temporal behaviour of one period of the example application. In this figure, execution instants of task t_1 are started at dates 0, 10, and 20, and execution instants of task t_2 at dates 0 and 20. The outputs produced by the instance of t_1 started at date 0 become available at date 10. The outputs of the first instance of t_2 only become available at date 20. Thus the second instance of t_1 cannot use them even if the actual execution of t_2 completes before date 10.

The traditional code generation scheme followed by Giotto then consists in mapping each task of the specification to a thread (for example a POSIX thread), that is activated periodically, using the period computed from the specification. In our example, the implementation would comprise two threads, one of period 10 ms for t_1 and the other of period 20 ms (for t_2). Then, Giotto relies on a preemptive rate-monotonic (RM) on-line scheduler to execute the system. This allows to analyse the system using well-known schedulability results concerning the RM policy [Liu and Layland, 1973].

Chapter 3

Functional specification

Contents

3.1 Introduction

This chapter presents our approach to formally modeling the functional aspects of an embedded control system. The first part of this chapter is dedicated to the definition of the formalism taken as input by our tool LoPhT, and to justifying our choice of an input formalism. This formalism, called Clocked Graphs (CG), is a simple data-flow synchronous language that was designed to be an intermediate compilation format. Unlike other data-flow synchronous languages, it is non-hierarchic, and imposes a full separation between control (represented with explicitly-defined *clocks*) and the data-flow itself (represented with *variables* and *blocks*).

The CG language has been designed to serve as an intermediate compilation representation. As such, it lacks many features meant to facilitate programming: hierarchy, the use

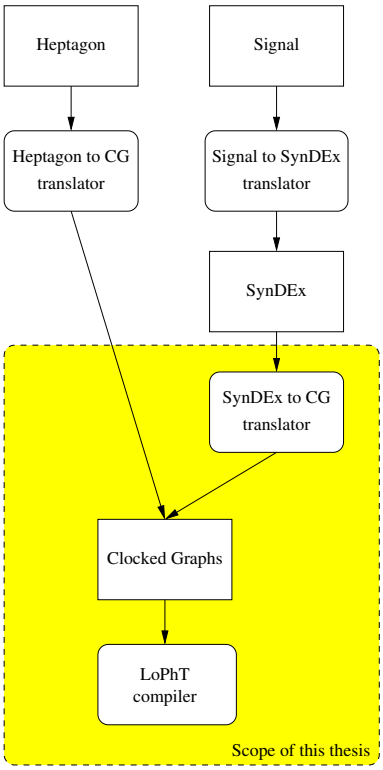


Figure 3.1: The two current compiling chains using CG as an intermediate representation language

of complex expressions, etc. Instead, all objects are identified and declared inside tables, and all clocks are identified (so that no complex clock inference is needed). Transforming high-level specifications written in languages such as Signal, SynDEx, or Lustre into CG programs requires a non-trivial transformation, detailed in Section 3.3. Fig. 3.1 pictures the two compilation chains that exist up to this day: one starts with programs written in the Heptagon reactive language (a language similar to Lustre) which can be translated into CG programs using a translator developed by Valentin Perrelle at IRT SystemX. The other starts with Signal programs which can be translated into SynDEx programs using a translator. These SynDEx programs can then be translated into CG programs using the translator that I developped during my thesis and which is presented in Section 3.3.

While simpler than other synchronous languages, CG is a full-fledged programming language which contains all the elements needed to support scheduling and code generation. However, for concision reasons, scheduling algorithms in both the real-time scheduling and the compilation communities are described on more abstract models, such as the various real-time *task models* or the *data dependency graphs (DDGs)*. The definition of our algorithms will follow the same approach. Thus, in order to reason about the correctness of our implementation flows we define in this chapter (in Section 3.4) the abstract models used for scheduling in subsequent chapters and their formal relation with the CG

programs.

Following the classification of Chapter 2, the CG language is a single-clock synchronous language, which means that there exists a clock (logical time base), called the *primitive* clock, from which all other clocks of the program are derived through sub-sampling. Moreover, in its current version, the language does not provide specific constructs for the definition of periodic sub-sampling, nor support for the description of long tasks whose execution lasts for more than one synchronous execution cycle. This poses a problem when the CG language is used to represent the functionality of systems involving multiple execution periods and long tasks. Encoding such a system in the CG language requires the use of a *hyperperiod expansion* which transforms a multi-period specification into a single period one. This transformation is presented in Section 3.5.

Overall, this chapter mostly reviews and systematizes existing work on the CG language (first defined in [Potop-Butucaru et al., 2009]). However, it contains two important elements of originality:

- The explanation of how a single-clock synchronous language can be used to model our complex avionics case study which involves real-time tasks with multiple periods.
- The extension of the CG language with logical invariants relating the inputs and outputs of a function. These invariants are needed in the definition of our pipelining technique in Chapter 6.

Less original, yet novel pieces of information provided in this chapter include the first full presentation of the CG language (its concrete syntax), and the description of the translation from a high-level data-flow formalism to CG.

3.2 The Clocked Graph synchronous language

This section defines the syntax and semantics of the CG language. As explained above, this language was designed to serve as an intermediate compilation representation. As such it does not offer *syntactic sugar* constructs meant to facilitate the work of human programmers. Instead, it clearly identifies and tabulates all objects it defines in a way that facilitates automatic treatment.

In a CG program, applications are modeled using a set of *tables*. Each table collects all definitions of a given kind: types, functions, constants, clocks, variables or data-flow blocks. Each element of a CG specification belongs to one of these tables, and can be uniquely referenced using a *typed index*, which points to a location in one of the tables. This allows the non-ambiguous referencing of any element of a CG program. The declaration of the indices follows the syntax given in Fig. 3.2: each index is composed of a

keyword which designates the table, and an integer which points to the element location inside this table. The 6 tables of a CG program are divided into two *sections*, defined next.

Like in other synchronous languages such as Lustre, Esterel, and Signal, the CG language relies for its definition of types, constants, and computation functions on a sequential, imperative programming language such as C, Ada, etc. This language, known as the *host language*, is in our case (for the scope of this thesis) the C language. The first section of a CG program, named the *global definitions* section, defines the interface between the synchronous data-flow described in CG and the host language. This section contains three tables that declare the types, constants, and functions that are defined in the host language and used in the synchronous program. One important remark here is that typing inside the CG language is strict, and no notion of sub-type exists.

The second section of a CG program defines the data-flow. The CG language largely follows a data-flow paradigm by organizing data computations as a graph formed of blocks connected through dependencies. However, unlike classical data-flow synchronous languages such as Signal, Lustre, or SynDEx, it does not represent control using subsampling and oversampling data-flow operators. Instead, it fully separates the description of control, which is represented with *clocks*, from that of the data flow. In particular, this separation means that no clock inference is needed during program analysis. Instead, a simple correctness check is needed, realized through the verification of simple well-formedness properties defined in Section 3.2.3.

A second difference with respect to existing languages concerns causality analysis. We shall assume throughout this thesis that the CG programs we manipulate are causally correct, in the sense of the definitions of Chapter 2. Moreover, we will assume that these programs satisfy a certain number of well-formed properties (defined below) facilitating analysis and code generation. These properties include the data-flow acyclicity required in Lustre or SynDEx. But as the CG language represents clocks separately, our causality analysis must include clocks. To allow this, the data dependencies related to clock computations are explicitly represented using so-called *supports*, defined next. Then, the notion of *endochrony*, which characterizes causally-correct single-clock programs that can be statically scheduled, is enforced on CG specifications using well-formed properties that are easily checked at scheduling and/or code generation time.

3.2.1 Host language and global definitions

As previously explained, all data types, computation functions, and most constants that are used in a CG program must be:

- Defined in a so-called host language, which in our case is C.
- Declared in the global definitions section of the CG program.

```
type_idx   := Type:  <Int>
fun_idx    := Function:  <Int>
cst_idx    := Const:  <Int>
var_idx    := Variable:  <Int>
clk_idx    := Clock:  <Int>
block_idx  := Block:  <Int>
```

Figure 3.2: CG indexes syntax

```
type_table        := Type Table type_list
type_list         :=
                     |type type_list
type              := type_idx <Identifier> type_description
type_description  := Predefined
                     | Simple
```

Figure 3.3: CG types declaration syntax

The CG declarations, grouped in 3 tables (type table, function table, and constant table) are the interface allowing scheduling and code generation algorithms to check the correctness of the specification and then generate C code. This code, together with the external C definitions provided in the host language, can be compiled to generate the executable code of the implementation.

This section details the form of the 3 CG tables that compose the host language interface (the global definitions section), and also details the form that must be respected by the external C definitions to allow interfacing at code generation time.

3.2.1.1 Types

Data types are declared in the type table following the rules of Fig. 3.3. Each type is referenced using a table index, an identifier (*i.e.* a string that defines the name of the type), and a description. The description defines the kind of the type. There are two kinds of types:

- Predefined types, identified with the **Predefined** keyword, have built-in support within LoPhT.
- All other types are identified with the keyword **Simple**.

The predefined types are bool, int, float and string. Literals of these types can be directly used as constants, and specific built-in operators can be used on Booleans, as we shall see in Section 3.2.2.

The following example of a type table defines three types.

```
function_table  :=  Function Table function_list
function_list   :=
                    |function function_list
function        :=  fun_idx <Identifier> (type_list)->(type_list)
```

Figure 3.4: CG function declarations syntax

```
Type Table

Type:0 int Predefined
Type:1 point Simple
Type:2 bool Predefined
```

The table begins with the keywords Type Table, and is then composed of the successive declarations of the types, following the order of their indices. In our table, the first one is the int type that is predefined in C (and thus needs no user-defined external definition), and the second one is a user defined type used to represent points in a two-dimensional space. The third line interfaces with the classical C `bool` type.

The C definition of the non-predefined type point is given in the external types.h library:

```
\\types.h

typedef struct {int x; int y;} point;
```

3.2.1.2 Functions (basic syntax)

The functions table collects abstract descriptions - prototypes - of all the functions used by the application. Along with the name of the function, which is needed to allow code generation, the interface includes the types of the input and output variables. This allows type-checking prior to the code generation phase. To allow code generation, the interface of the C function must comply with the CG definition. Compliance consists in having the same name, having a void return type, and having a list of parameters with the following structure:

- arguments that correspond to the inputs declared in the prototype are of the same type as in the prototype,
- arguments that correspond to the outputs declared in the prototype are pointers to elements of the types declared in the prototype: the outputs are passed by reference.

Moreover, it is required that the order in which the input and output types are declared in the prototype be respected in the implementation.

When modeling an application using the CG language, we assume that the implementations of the functions do not have an internal state and that they have no side effects. The language provides means of explicitely defining and managing the state of the application by the means of delay constructs, as we will see in Section 3.2.2.3.

The syntax for function definitions in CG is given in Fig. 3.4. An example of function table is provided below. In this example the types reference the type definitions given in Section 3.2.1.1.

```
Function Table

Function:0 highest_y (Type:1 Type:1) -> (Type:0)
Function:1 shift_right (Type:1 Type:0) -> (Type:1)
```

The keyword Function Table announces the definitions of the functions. The first function, called *highest_y* takes two points (as defined in the previous section) as inputs, and returns the value of the highest y-coordinate. The library file, called functions.h comprises the source code for the two functions:

```
//functions.h

void highest_y (point p1, point p2, int *return_val) {
     if (p1.y > p2.y){
        *return_val = p1.y;
     }
     else {
        *return_val = p2.y;
     }
};

void shift_right (point p1, int shift, point *return_point) {
        return_point->x = p1.x + shift;
        return_point->y = p1.y ;
};
```

Note the parameter order in the C definitions. The first function has three parameters, the return value being passed by reference. The second function takes a point and returns another point which has the same y coordinate as the first one and its x coordinate that has been shifted by a length given by an integer.

```
function        :=  fun_idx <Identifier> (var_list)->(var_list)
                    ensures_clause
var_list        :=
                    | fun_var var_list
fun_var         :=  <Identifier> :  type_idx
ensures_clause  :=
                    |Ensures p_test_e
p_test_e        :=  And(p_test_e p_test_e)
                    |Or(p_test_e p_test_e)
                    |Not p_test_e
                    |<Identifier>
```

Figure 3.5: CG complete function declaration syntax

3.2.1.3 Functions (extended syntax with invariants)

The function declaration syntax defined above is the one used in previous versions of the LoPhT tool. However, it does not provide enough information to allow the application of advanced scheduling techniques such as software pipelining (presented in Chapter 6).

To support such advanced scheduling algorithms, I have extended the function description language as follows:

- Instead of declaring only the types of the inputs and outputs of each function, we associate a name to each input and output. These names do not correspond to any other element of the specification graphs: they are local to each function definition, so the same names can be used in the definition of two different functions. However, these names must be unique inside each function definition, to avoid any ambiguity.
- An *ensures* clause can be defined for each function. It consists in a Boolean formula constructed from the names of the inputs and outputs of the function. The clause of a function defines a logical invariant that must be true at all execution cycles. The definition of the predicate can use the logical connectors **And**, **Or** and **Not**, along with identifiers corresponding to inputs and outputs of Boolean type.

The revised syntax for the functions specification is given in Fig. 3.5. We illustrate the declaration of an *ensures* clause in the following example:

```
Function:0  bool_neg  (i:Type:2)->(o:Type:2)
            Ensures Or(And(i Not o) And(Not i o))
```

In this example, we specify a function called bool_neg (for boolean negate), which takes an input *i* of type Type:2 (Boolean), and returns an output of the same type. The *ensures* clause states that at each instant, the output is the negation of the input. The implementation code corresponding to this function is the following:

```
constant_table   :=  Constant Table constant_list
constant_list    :=
                     |constant constant_list
constant         :=  cst_idx <Identifier> type_idx cst_kind
cst_kind         :=  External
                     |fun_idx (cst_or_lit_list)
cst_or_lit_list  :=
                     |cst_or_lit cst_or_lit_list
cst_or_lit       :=  cst_idx
                     | <Int>
                     | True
                     | False
                     | <Float>
                     | <String>
```

Figure 3.6: CG constants declaration syntax

```
\\Functions.h

void bool_neg (bool i, bool *o){
  *o = !i
};
```

3.2.1.4 Constants

As explained above, literals of the predefined types can be directly used in CG specifications. However, all constants of non-predefined types must be explicitly declared in the constant table. It is also possible to define constants of predefined types, whenever having named constants is needed.

There are two types of constant declarations: the first ones are identified by the **External** keyword. They are defined in an external C file. The second type of constants are initialized with a simple constant expression formed by applying functions to other constants (literals or tabulated constants). Note that recursive definitions are not allowed, in the sense that a constant cannot depend on itself.

The syntax is given in Fig. 3.6, and illustrated in this example, which uses the type and function tables used as examples in the previous sections:

```
Constant Table

Const:0 origin Type:1 External
Const:1 shifted_origin Type:1 Function:1 (Const:0 3)
```

The external constants must be defined in C. In our case, only one constant must be defined:

```
//constants.h

static point origin = { 0,0 } ;
```

The second constant is defined in the CG specification. It can be implemented in one of two ways: through a constant definition synthesized by our tool, or by inlining the function call at each point of use. In our example, constant shifted_origin is defined by the application of function `shift_right` to the `origin` constant and to the literal `3`.

3.2.2 Dataflow definition

The second section of a CG program contains the tables defining the control and dataflow of the application.

Control is described under the form of *logical clocks* which define the activation conditions (triggers) of the various computations and communications of the application. One particularity of the CG clocks is that apart from defining an activation condition (a boolean predicate), a clock also carries the necessary information for the computation of that condition: this allows to easily check that the specification is causal, and lets the application designer specify the way he wants the clocks to be computed (which in turn may allow a minimization of the amount of data sent through the communication media).

The data flow of the application is specified using nodes, called blocks in the CG formalism, and variables, which represent the flows of data between blocks. The blocks represent the computations and internal state of the application. Another particularity of the language is that the dataflow specification does not rely on communication arcs. Instead the specification file contains variables that are unambiguously amalgamated to the output ports of the dataflow blocks. The description of the application thus involves objects that are close to the ones used in the implementation code.

3.2.2.1 Clocks

In our formalism, application control is specified by the means of clocks. The clocks are the activation conditions governing the execution of all computations and communications. In CG, a clock defines a Boolean predicate and a way of computing this predicate.

The Boolean predicate is the actual activation condition. At each execution cycle, the predicate associated with a clock is evaluated. A computation or communication whose clock is c is performed in execution cycle n whenever the value of c in cycle n, denoted $c[n]$, is 1 (**true**, active) in the given cycle. If $c[n] = 0$, then the associated computation or communication is not performed.

```
clock_table    :=  Clock Table clock_list
clock_list     :=  clock
                   |clock clock_list
clock          :=  clk_idx clk_name clk_def clk_support
clk_name       :=
                   |<Identifier>
clk_def        :=  Primitive
                   |clk_predicate
clk_predicate  :=  clk_idx
                   |Or(clk_predicate clk_predicate)
                   |And(clk_predicate clk_predicate)
                   |Diff(clk_predicate clk_predicate)
                   |Test(clk_predicate test_expr)
test_expr      :=  And(test_expr test_expr)
                   |Or(test_expr test_expr)
                   |Diff(test_expr test_expr)
                   |Not(test_expr)
                   |rval
rval           :=  var_idx
                   |cst_or_lit
clk_support    :=
                   |clkd_var clk_support
clkd_var       :=  var_idx On clk_idx
```

Figure 3.7: CG clocks declaration syntax

Clocks are partially ordered by the $\leq$ relation defined as follows: $c_1 \leq c_2$ *iff* $c_1[n]$ implies $c_2[n]$ for all execution instant n. In the classification of Chapter 2, the CG language is a single-clock synchronous language, which means that in each CG specification there exists a unique maximal clock which is true on each execution instant. We call it the *primitive clock*, because all the other clocks of the specification are obtained by successively subsampling it.

The predicate associated with the primitive clock is predefined (always true), but the definition of all other clocks must include a predicate built from the other clocks and from the variables of the data flow using a small set of operators defined below.

Clocks are following the syntax given in Fig. 3.7. The *primitive clock* must be declared in first position in each specification (this is why the syntax does not allow the `clock_list` to be empty). This is done by using the **Primitive** keyword. Other clocks are specified by directly defining their predicate. A *clock predicate* is defined by:

- a reference to another declared clock,
- the union, intersection or difference of two other clock predicates,

		Time instants										
		0	1	2	3	4	5	6	7	8	9	10
Variables												
	x	1	1	0	1	0	0	1	0	0	0	0
	y	1	1	1	0	1	0	1	0	0	0	1
	z	1	5		2			3				
Clocks												
Clock:0	$Primitive$	1	1	1	1	1	1	1	1	1	1	1
Clock:1	$Test(Clock:0 \quad x)$	1	1	0	1	0	0	1	0	0	0	0
Clock:2	$Test(Clock:0 \quad y)$	1	1	1	0	1	0	1	0	0	0	1
Clock:3	$Test(Clock:0 \quad And(x \quad y))$	1	1	0	0	0	0	1	0	0	0	0
Clock:4	$And(Clock:2 \quad Clock:3)$	1	1	0	0	0	0	1	0	0	0	0

Figure 3.8: Predicates associated to clocks

- a *test expression* which is evaluated on the instants when a specified clock evaluates to true.

A test expression is the definition of a logical formula over the boolean variables and constants of the data flow:

- the conjunction, disjunction or difference of two other test expressions,
- the negation of a test expression,
- a *rvalue*.

Rvalues are defined as references to either boolean variables or constants of the data flow, the application of a function to other rvalues (the output of the function must be boolean), or boolean constant literals[1]. As you can see, there are two different levels of specification: one works at clock level (`clk_predicate`) and allows the direct combination of clocks. The other works at the variables and constants level by allowing the definition of logical formulas using the variables and constants of the data flow. This particularity of the syntax allows the definition of equivalent formulas under different forms, as shown in the next example. The expressiveness of this language can be used when automatically transforming a high-level specification to a CG specification, in order to capture to a certain extent the structure of the initial (hierarchical) program inside the clocks. Keeping this information can be very useful, especially when deciding how to evaluate the clocks values.

[1]The syntax can be extended to include the definition of regular clocks and predicates defined over non-boolean variables. Nevertheless the support for such clocks in our scheduling algorithms would require the use of a SMT solver instead of the currently used SAT solver.

In the example of Fig. 3.8, we consider two boolean variables: x and y, which are defined at any instant of the system. Five clocks are also depicted in this example. The first one is the *primitive clock*: it is true at each instant and thus lays the basis for the definition of the other clocks. Clock:1 is defined by the value of variable x at any instant defined by the primitive clock: it is true at instants 0, 1, 3 and 6, and false at any other instant. Clock:2 is defined in the same fashion, but using variable y instead. Finally the value of Clock:3 at any instant is given by the result of the logical conjunction of variables x and y. Since both of these variables are defined on any instant given by the primitive clock, Clock:3 will also be defined on all instants of the system. Clock:4 is equivalent to Clock:3 (they are synchronous). Both are present at all instants, and at each instant they evaluate to the same value. Nevertheless, one (Clock:4) is defined at the clock predicates level, as a conjunction between two clocks, while the other is defined at the data flow variables level as a conjunction between two variables. You may have noticed that the example also includes one integer variable called z. It is only defined in the instants when Clock:1 is true. In all the instants when Clock:1 is false, z has no value (we say that it is absent). For this reason, we say that z is a *clocked variable*: a clocked variable $cv = (var, clk)$ is a variable *var* whose value is considered only in the instants defined by a clock *clk*. Clocked variables are involved in the definition of the clock supports described below, and in the definition of the input ports of the blocks of the data flow, as we will see later.

Clock support As said before, the definition of a clock is not limited to the predicate that defines it. It also includes the elements needed to evaluate this predicate: the *clock support*. This information is exploited to ensure causality, and allows optimizations to occur during the offline scheduling phase, when reserving time for the communication of the corresponding data. The support of a clock *c* is the set of all variables needed to compute *c*, along with the clocks defining the instants when these variables are needed. As we saw before, the association of a variable *var* with a clock *clk* is called a *clocked_variable* $cv = (var, clk)$. In order to ensure causality when declaring clocks that involve the recursive computations of other clocks, the support of each clock must be *endochronous*, that is to say for each clocked variable *cv* of a support *s*:

- there must exist a subset $sub(s)$ of *s* such that the clock of *cv* can be computed using only the clocked variables present in $sub(s)$,
- there are no cyclic dependencies in *s*.

This means, for example, that for any two clocked variables $cv_1 = (var_1, clk_1)$ and $cv_2 = (var_2, clk_2)$ in *s*, if clk_1 is defined using var_2, then clk_2 cannot itself make a reference to $var1$ or be defined using any clock making a reference to var_1. In practice, we impose

that the clocked variables defining the support of a clock be declared in a list following a certain partial order: for each clocked variable in the list, it must be possible to compute its clock using only the clocked variables declared earlier in the list. The primitive clock can always be computed, since it is the basis of all clocks. Intuitively, this means that the first elements of the list (at least the first one) are always defined over the primitive clock.

The example of Fig. 3.9 illustrates how clocks are declared in CG, and how supports work. First of all, the primitive clock has to be uniquely defined. This clock has no dependency, since it is always present and always returns true. Clock *var0* is defined as being the value of the boolean variable *Variable:0*, evaluated when the primitive clock is true (that is to say always). The support of this clock thus states that its computation requires the value of *Variable:0* at any instant when *Clock:0* is true. Clock *not_var1* returns the negated value of a boolean variable *Variable:1* at any instant defined by $Clock : 0$. The next five clocks are synchronous in the sense that for any given instant (and thus any given configuration of *Variable:0* and *Variable:1*), they all return the same value: $(Variable : 0 \wedge \neg Variable : 1)$. Nevertheless, we have declared them in order to illustrate the importance of clock supports in the definition of clocks. Indeed, although they all return the same value, the amount of data required to compute it is different, and varies according to the values of *Variable:0* and *Variable:1*. Clock *version1* makes the conjunction of clocks *var0* and *not_var1*, and its support states that the values of *Variable:0* and *Variable:1* are needed at any time (on *Clock:0*) to compute this formula. In other terms, any time this clock has to be evaluated, the scheduler must ensure that the values of both variables are present on the processor that computes the clock. Clock *version2* makes exactly the same conjunction, except that *Variable:1* is only required when *var0* is true. Indeed, if *var0* is false, then the whole conjunction will be false, and there is no need to evaluate *not_var1*. In clock *version3*, the conjunction is still the same, but *not_var1* is evaluated before *var0*, and thus the clock support must be changed accordingly. Using *version1*, *version2* or *version3* has an impact on the result of the scheduling phase, since the communications will be optimized depending on the clock dependencies. Moreover, when running the application, having made the good choice (given there is one) can speed up the execution: if we consider an application where both *Variable:0* and *Variable:1* are frequently false, using *version2* will cause the sending of *Variable:0* at every execution cycle, but *Variable:1* will rarely be sent, since *var0* will often evaluate to false. On the other hand, using *version3* will cause the frequent sending of both variables, since *Variable:1* will often be false (and thus, *not_var1* will be true).

Clocks *version4* and *version5* are equivalent to *version2* and *version3* respectively, from the point of view of their return value and their clock support. They illustrate the fact, already intuitively pointed out in the example of Fig. 3.8, that there are two different levels of clock predicate definition, which may be used to declare equivalent clocks.

```
Clock:0 tick Primitive
Clock:1 var0 Test(Clock:0 Variable:0) Variable:0 On Clock:0
Clock:2 not_var1 Test(Clock:0 Not(Variable:1)) Variable:1 On Clock:0
Clock:3 version1 And(Clock:1 Clock:2) Variable:0 On Clock:0 Variable:1 On Clock:0
Clock:4 version2 And(Clock:1 Clock:2) Variable:0 On Clock:0 Variable:1 On Clock:1
Clock:5 version3 And(Clock:2 Clock:1) Variable:1 On Clock:0 Variable:0 On Clock:2
Clock:6 version4 Test(Clock:1 Not(Variable:1)) Variable:0 On Clock:0 Variable:1 On Clock:1
Clock:7 version5 Test(Clock:2 Variable:0) Variable:1 On Clock:0 Variable:0 On Clock:2
```

Figure 3.9: Example of clock definitions in CG

```
variable_table  :=  Variable Table variable_list
variable_list   :=
                    |variable variable_list
variable        :=  var_idx type_idx Single Assignment port_ref
port_ref        :=  <Identifier> @ block_idx
```

Figure 3.10: CG variables declaration syntax

3.2.2.2 Variables

As said before, we chose not to use arcs to depict the data dependencies in the CG language, but rather to use specification variables, that is to say objects that are very close to the variables that will be used in the implementation. We made this choice because we believe that using the same object, which seemlessly evolves by successive operations, all along the scheduling and code generation flow is much more convenient in an intermediate representation than using different objects at different stages of the flow, and having to convert them.

In CG, variables are produced by the output ports of the dataflow blocks: each output port corresponds to exactly one variable, and each variable corresponds to exactly one output port. This way, there is no ambiguity on the definitions of the variables, and as LoPhT schedules the dataflow blocks, the variables produced by these blocks are directly mapped on the resources. The specification syntax of the variables of the system is given in Fig. 3.10. Each variable type is defined by referencing an entry of the types interface. Then the variable is linked to the output port that produces it: a reference is made to this port, by declaring its identifier and the block it belongs to. The *Single assignment* keyword states that for now, LoPhT only supports single assignment of variables within an execution cycle, but this model may be extended in the future to variables that can be reassigned multiple times within a cycle. The example in Fig. 3.11 illustrates the declaration of the two boolean variables used to define the clocks of Fig. 3.9.

```
Variable:0 Type:2 Single Assignment is_ok@Block:8
Variable:1 Type:2 Single Assignment broken_booster@Block:5
```

Figure 3.11: Example of variable definitions in CG

```
block_table       :=  Block Table block_list
block_list        :=  block
                      |block block_list
block             :=  block_idx clk_idx block_definition
block_definition  :=  (iport_list)->(oport_list) block_fun
iport_list        :=
                      | input_port iport_list
input_port        :=  <Identifier> Is clkd_var_list
clkd_var_list     :=  clkd_var
                      | clkd_var clkd_var_list
oport_list        :=
                      | output_port oport_list
output_port       :=  <Identifier> Is var_idx
block_fun         :=  fun_idx
                      | Delay delay_info
delay_info        :=  type_idx Depth <Int> Init cst_or_lit_list
```

Figure 3.12: CG blocks syntax

3.2.2.3 Dataflow blocks

The dataflow blocks ($\mathcal{N}$) are separated in two kinds: computation blocks ($\mathcal{N}^{\mathcal{C}}$) and delay blocks ($\mathcal{N}^{\Delta}$). The computation blocks represent the stateless computations that happen during a cycle of execution. The supported model for computations does not allow hidden side effect using unspecified variables between different blocks, or between successive computations of a computation block, because it would imply that a part of the system state be stored in these unspecified, hidden variables. Nevertheless, it allows some non-determinism, for example for the implementation of sensors, which do not have a specified input port, and instead acquire their data via side effect. In order to cope with that limitation of the computation blocks, the state of the system is completely stored in the delay blocks, which carry explicitly all the necessary data from one execution cycle to the next one.

Syntax The syntax for the declaration of the dataflow blocks is given in Fig. 3.12. Each block specification includes a reference to its *activation clock*, and the definition of the blocks. The definition of the blocks contains:

- The list of *output ports* which are simply defined by a name and a reference to the

unique variable they produce. For the variables specification to remain unambiguous, two output ports of the same block should never have the same name,

- The list of *input ports*. Since our representation relies on the use of variables, the usual dataflow arcs are replaced by the definition of the input ports of the blocks. Each of these ports is characterized by a name, and contains a list of *clocked variables* that designate the variables that give their value to the input port, and under which condition they shall give it,

- The nature of the block.

The *nature* of the block is defined either by directly refering to the function that the block computes, in the case of a computation block, or by using the **Delay** keyword, followed by the relevant information about the delay block: the type of the delayed variable, the depth of the delay - the number of successive execution cycles it spans - and the non-empty list of constants or constant literals that are used to initialize the value of its output port. An important point is that the current implementation of LoPhT only supports delay blocks with one single input and one single output port. This is coherent with the usual delay construct in traditional synchronous languages and should not be subject to change in the future.

We illustrate the declaration of blocks in the following example, where three blocks are declared:

```
Block Table

Block:0 Clock:0 (point Is Variable:3 On Clock:0 shift_int Is Variable:4 On Clock:0)->(shifted_point Is Variable:2) Function:1
Block:1 Clock:0 ()->(polled_int Is Variable:4) Function:2
Block:2 Clock:0 (i_point Is Variable:2 On Clock:0)->(o_point Is Variable:3) Delay Type:1 Depth 1 Init Const:0
```

The first one is a functional block which takes as input a point and an integer variable, and computes function shift_right on these inputs. The second block models an input interface which returns the integer used to shift the point in Block:0. The last block is a delay which takes as input the output of the first block, and gives it back to its next occurence (in the next execution cycle). It is initialized with the value of the constant Const:0.

3.2.3 Well-formed properties

Correctness properties We summarize here the correctness properties which ensure that the specification respects the synchronous hypothesis. Thus, they must be respected by any CG specification, and in the remainder of the thesis, we will always assume that they are.

The formalization of the two first properties uses the following notations: for any block n, we denote by $\mathcal{I}(n)$ the set of its input ports and by $clk(n)$ its activation clock. For

any clocked variable $cv = (var, clk)$ of the specification, we denote by $src(cv)$ the block that produces *var* and by $clk(cv)$ its clock *clk*. Finally, for any input port *i*, we denote by $inputs(i)$ the set of clocked variables that are declared in this input port.

- **Property 1:** At any time instant when a block *n* is active (i.e. its activation clock is true), all its inputs must be computed and transmitted. Formally:

$$\boxed{\forall n \in \mathcal{N}, \forall i \in \mathcal{I}(n), clk(n) \leq \bigvee_{cv \in inputs(i),} clk(cv)}$$

  meaning that for each block *n*, each of its input ports *i* receives clocked variables whose clocks reunion covers at least all the time instants defined in $clk(n)$. In other terms, each times *n* is active, its inputs are defined.

  and

$$\boxed{\forall cv, clk(cv) \leq clk(src(cv))}$$

  meaning that for any clocked variable *cv* in the specification, the block producing its variable activates on a clock that contains at least all the activation instants of $clk(cv)$.

- **Property 2:** For any block *n*, each of its inputs can come from at most one source at each instant. Therefore, we forbid write conflicts and the non-determinism that could result from multiple assignments of the same input port in the same cycle. In a formal way:

$$\boxed{\forall n \in \mathcal{N}, \forall i \in \mathcal{I}(n), \forall cv_1, cv_2 \in inputs(i), cv_1 \neq cv_2 \implies clk(cv_1) \wedge clk(cv_2) = false}$$

- **Property 3:** It is forbidden to specify a dataflow graph comprising a cycle that does not include a delay block. This condition imposes that there are no *causality cycles* in the spec, and in conjunction with the synchronous hypothesis, that the execution of each computation cycle performs in bounded time.

Endochrony of a Clocked Graph specification Apart from these correctness properties, we define a preorder $\preceq$ as follows (this definition is adapted from [Potop-Butucaru et al., 2009]):

- $o \preceq cv$ for each clocked variable *cv* created by an output port *o*,
- $cv \preceq i$ for each clocked variable *cv* declared in an input port *i*,

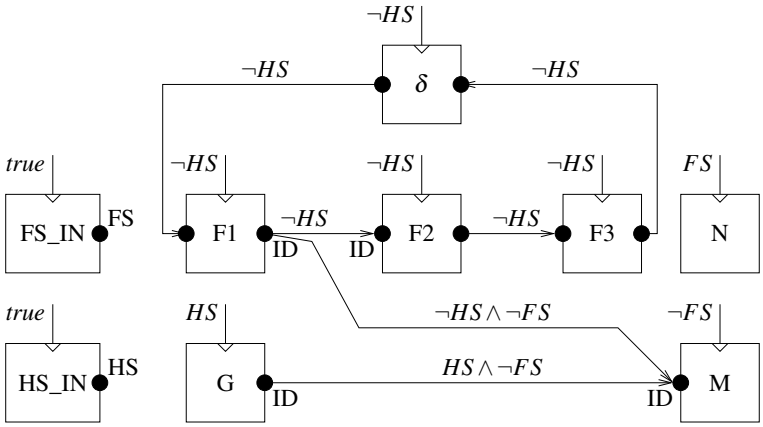


Figure 3.13: Example of specification in the clocked graph formalism

- $o \preceq x$ for each output port o and each block or clocked variable x such that the variable created by o is used in the support of $clk(x)$,
- $i \preceq o$ for each input port i and output port o of a block that is not a delay.

If $\preceq$ is a partial order relation and for each block n, and each output port of o of n, any clocked variable cv created by o has its clock $clk(cv) \leq clk(n)$, then we say the functional specification is endochronous. This ensures causality, even for the computations of the clocks, as well as the existence of a static schedule for all the operations, including the computations of the clocks (which are thus endochronous).

3.2.4 Example

Let us illustrate how to program a specification in the CG language with a graphical example. The example of Fig. 3.13 is a graphical representation of a clocked graph specification. It is a simplified model of an application controlling the ignition cycle in an explosion engine. Between two successive ignitions, the computations represented in this specification must be executed at most once, depending on their execution condition. Since the engine can run at variable speed, the time between two successive ignitions is also variable. When the engine runs at a low pace, a long time interval is available to perform accurate computations. On the other hand, the faster the engine is running, the smaller the time interval available for computations. In order to tackle this problem, two operational modes are defined for the system. A sensor acquires the speed of the engine, and outputs a boolean stating if this speed exceeds a certain stage. In the specification, block *HS_IN* is responsible for reading this boolean value at the begining of each cycle of computation, and to make the value of HS available to the system. Then, in each cycle where HS evaluates to false (i.e. when the engine is not running at high speed), operations $F1$, $F2$ and $F3$ are computed. The delay block δ saves the last output value generated

by $F3$ (in the last execution cycle where HS was false), and feeds it to $F1$. This way we specify a feedback loop, while keeping the specification valid from the point of view of causality. In the cycles where HS is true, since there is less time between two ignitions, only one coarser grain, quicker computation is performed: function G. Following the same scheme, a failsafe mode is defined using a sensor and the input block FS_IN. If an abnormal behaviour is detected in the system, then FS_IN returns the value true, to signal the application that something is going wrong. When it is the case, graceful degradation of the system is achieved by computing one generic function, N, that does not depend on any other computation (it has no input). On the other hand, in nominal mode a function M is called and takes as input the value computed either by G or by $F1$, depending on the speed of the engine.

Blocks that have no dependency, or dependency chain, between them can be scheduled in parallel. For example, in the case when FS is true at a given instant, N can be scheduled in parallel to $F1$, $F2$, $F3$, or G. On the contrary, on instants when clock $\neg HS$ is true, $F2$ has to wait for the production of the output variable of $F1$, and $F3$ of the output variable of $F2$.

Example in the clocked graph syntax We will now show the CG specification corresponding to this example. To do so, we need to specify three data types: one will be used to type the boolean variables of the system (HS and FS), and the others are two user defined data types. Then we must declare the functions table containing the eight functions of the system. In this example, we also declare one external constant that is used to initialize the delay block δ. Seven variables must be declared: the two boolean variables HS and FS, the variables produced by the output ports of functional blocks $F1$, $F2$, $F3$ and G, and the output of the delay block δ. We must then declare seven clocks. First, the **PrimitiveClock** which contains all the time instants considered in the system. By convention, we name it Tick, since it contains all the clock ticks of the system. Then the clocks corresponding to the value of variable HS being true or false (one clock for each) on the time instants defined by the base clock, and the corresponding clocks for the value or variable FS. For each of those four clocks, the support consists in the considered variable, sampled on the primitive clock. Finally we declare two clocks that are the conjunctions of previous clocks, and that are needed to activate or inhibit the communications to the block computing function M. The support of those clocks is defined in the following way: for each of them, we need the variable corresponding to the first part of the conjunction, sampled on the primitive clock, and the variable of the second part of the conjunction, only when the first part of the conjunction evaluates to true. Finally, we must declare the 9 dataflow blocks corresponding to the two input functions, the 6 computation functions and the delayed variable. The full functional specification of this example, in

the CG syntax, is given in Fig. 3.14.

Checking correctness properties in our example In order to illustrate the three correctness properties we defined, we will now check that they are indeed enforced in our example.

- **Property 1:** This rule applies to the three blocks $Block : 3$, $Block : 4$ and $Block : 6$, since they are the only blocks with functional inputs. Clearly, for blocks 3 and 4, their only input arc has the same activation clock than the blocks ($\neg HS$), so the property is verified. For block 6, we must check the reunion of the clocks of its two arcs against its own activation clock. We have :

  $$(\neg HS \wedge \neg FS) \vee (HS \wedge \neg FS) \Leftrightarrow \neg FS$$

  and since $\neg FS$ is precisely the activation clock of block 6, property 1 is verified.

- **Property 2:** This rule applies only to the communications of variables 2 and 4 from block 2 and 5 to block 6. We can compute the logical conjunction of the two communication clocks:

  $$(\neg HS \wedge \neg FS) \wedge (HS \wedge \neg FS) \Leftrightarrow false$$

  so the second property is verified in our example.

- **Property 3:** The third property is easily verified in the example since the only dependency cycle which exists between $F1$, $F2$ and $F3$ includes a delay block δ.

We have just presented the core language of the CG formalism, along with its semantics and correctness properties. Amongst the particular features of the language dedicated to improving scheduling, we introduced the possibility to define contracts on the functions of the specifications: the application developer can declare logical invariants between the inputs and outputs of the boolean functions of the application. This additional information can then be used in order to improve the results of our scheduling algorithms.

3.3 Translation from higher-level specifications

We have already explained that the CG language was not defined as a specification language, but as an intermediate representation to be used during compilation and/or real-time scheduling. This means that actual specification must be done in higher-level synchronous languages such as Signal, SynDEx, or Lustre, and then the specification must be translated into a lower-level CG program.

```
ClockedGraph

Global Definitions

Type Table

 Type:0  boolean  Predefined
 Type:1  ID_type  Simple
 Type:2  V_type   Simple

Function Table

 Function:0  FS_IN  ()->(o:Type:0)
 Function:1  HS_IN  ()->(o:Type:0)
 Function:2  F1     (i:Type:2)->(o:Type:1)
 Function:3  F2     (i:Type:1)->(o:Type:2)
 Function:4  F3     (i:Type:2)->(o:Type:2)
 Function:5  G      ()->(o:Type:1)
 Function:6  M      (i:Type:1)->()
 Function:7  N      ()->()

Constant Table

 Const:0  init_del  Type:2  External

Functional Specification

Variable Table

 Variable:0  Type:0  Single Assignment  FS@Block:0
 Variable:1  Type:0  Single Assignment  HS@Block:1
 Variable:2  Type:1  Single Assignment  ID@Block:2
 Variable:3  Type:2  Single Assignment  V1@Block:3
 Variable:4  Type:1  Single Assignment  ID@Block:5
 Variable:5  Type:2  Single Assignment  V2@Block:4
 Variable:6  Type:2  Single Assignment  V2_del@Block:8

Clock Table

 Clock:0  Tick  Primitive
 Clock:1        TestClock(Clock:0 Variable:1) Variable:1 On Clock:0
 Clock:2        TestClock(Clock:0 Not(Variable:1)) Variable:1 On Clock:0
 Clock:3        TestClock(Clock:0 Variable:0) Variable:0 On Clock:0
 Clock:4        TestClock(Clock:0 Not(Variable:0)) Variable:0 On Clock:0
 Clock:5        And(Clock:2 Clock:4) Variable:1 On Clock:0 Variable:0 On Clock:2
 Clock:6        And(Clock:1 Clock:4) Variable:1 On Clock:0 Variable:0 On Clock:1

Block Table

 Block:0  Clock:0  ()->(FS Is Variable:0)                                        Function:0
 Block:1  Clock:0  ()->(HS Is Variable:1)                                        Function:1
 Block:2  Clock:2  (i Is Variable:6 On Clock:2)->(ID Is Variable:2)              Function:2
 Block:3  Clock:2  (ID Is Variable:2 On Clock:2)->(V1 Is Variable:3)             Function:3
 Block:4  Clock:2  (V1 Is Variable:3 On Clock:2)->(V2 Is Variable:5)             Function:4
 Block:5  Clock:1  ()->(ID Is Variable:4)                                        Function:5
 Block:6  Clock:4  (ID Is Variable:2 On Clock:5 Variable:4 On Clock:6)->()       Function:6
 Block:7  Clock:3  ()->()                                                        Function:7
 Block:8  Clock:2  (del_i Is Variable:5 On Clock:2) -> (V2_del Is Variable:6)    Delay Type:2
                                                                                 Depth 1 Init Const:0
```

Figure 3.14: CG functional specification of the previous example

To allow interfacing with high-level specification languages, I have developed a translator that generates CG programs from input specifications of the SynDEx [Sorel, 2005] tool. We briefly present here this translator. In doing so, we present the principles that must be respected when translating into CG any high-level data-flow formalism (*e.g.* Lustre/Scade). Furthermore, given that a Signal-to-SynDEx translator already exists [Pan et al., 2003], our translator allows the translation of Signal/Polychrony specifications into CG.

In defining our translation scheme we make a number of important hypotheses that are used during translation. First of all, we assume that the input SynDEx specification is correct (in the sense that it can be scheduled by SynDEx). Important aspects of this correctness are type correctness, clock consistency, and causal correctness. In particular, we assume that the input specification represents a single-clock synchronous program, where all clocks are derived from a single base clock.

Our transformations preserve these correctness properties. We do not check whether types are consistent, assuming that it was checked by the SynDEx graphical specification environment. The causal correctness is checked by the translation process itself, as the translation will fail for a causally incorrect specification.

In defining our translation, we start by introducing in Section 3.3.1 the SynDEx specification formalism. Then in Section 3.3.2 we intuitively define the translation algorithm, using the engine iginition control example of Section 3.2.4.

3.3.1 Functional specifications in SynDEx

3.3.1.1 Hierarchy

The synchronous language SynDEx allows the description of embedded applications under the form of hierarchical dataflow graphs. A SynDEx specification consists in a dataflow definition which can hierarchically use other definitions. Inside each definition, the dataflow is extremely simple. It consists in a set of dataflow nodes, and a set of arcs connecting the input and output ports of the nodes. Each time the dataflow description is executed, all its nodes are executed in the order prescribed by the dataflow arcs[2].

The nodes of a SynDEx dataflow definition have one of 3 types:

- Computation nodes are direct references to external C functions, much like the computation blocks of CG. They are subject to the same constraints as the computation blocks of CG. For instance, they cannot have an internal state or make side effects.

- Delay nodes are the state holders of SynDEx. They are similar to CG's delays, defined above.

[2]In the terminology of Dennis [Dennis, 1974], SynDEx uses no control actors (nodes).

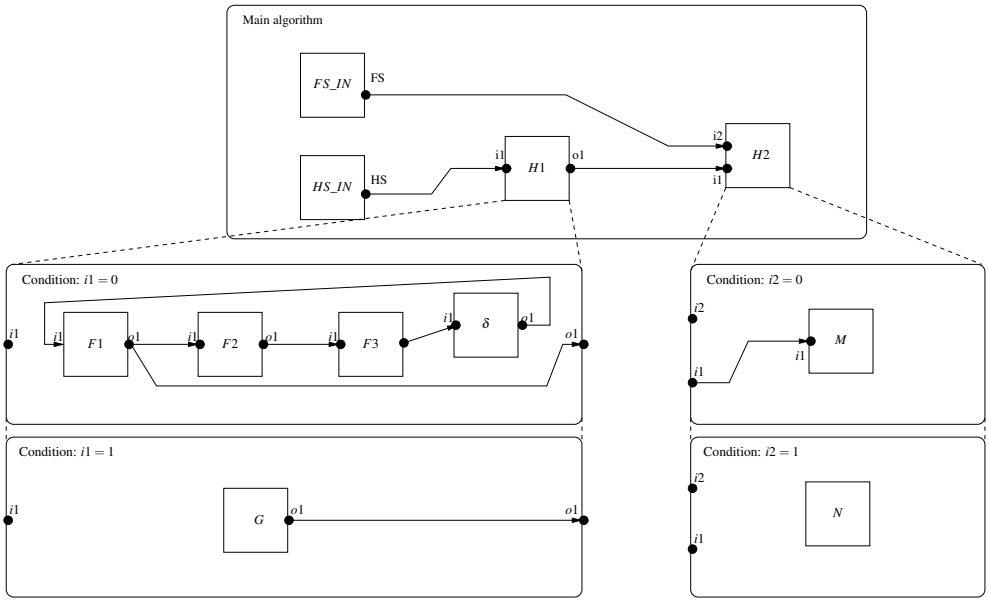


Figure 3.15: Example of specification in SynDEx

- Hierarchical nodes are references to other dataflow definitions.

The semantics of hierarchical nodes is that execution must take place as if the referenced dataflow replaces the hierarchical node. The input and output ports of a hierarchical node allow to interface between the nodes inside the referenced description and the other nodes of the top-level definition.

The top-level SynDEx specification of the engine ignition control application described earlier is given in Fig. 3.15. At this description level, the two input functions HS_IN and FS_IN are present, as well as two hierarchical blocks $H1$ and $H2$. HS_IN is linked to the input port $i1$ of node $H1$ and FS_IN to the input port $i2$ of node $H2$. Moreover, the output $o1$ of $H1$ is provided to the input port $i1$ of $H2$.

3.3.1.2 Execution conditions

Hierarchy is often used in programming to allow the structuring of a specification into descriptions of reasonable size. This is also true in SynDEx, but not only. Indeed, in SynDEx hierarchy is also the only mechanism allowing the description of *conditional control*.

To describe conditional control, a hierarchical node in a dataflow definition can have not one, but several references to other dataflow definitions. Each of these references corresponds to one *execution mode*, identified through a predicate over the input ports of the hierarchical node. This predicate determines which reference is used as the replacement of the hierarchical node at each execution instant of the dataflow description. This means that the predicates of any two references must be exclusive, and that one of them must

be true at each execution cycle of the top-level dataflow definition. The input ports used in the definition of these predicates are called the conditioning ports of the hierarchical node. We shall assume, as SynDEx requires, that only integer ports can be conditioning ports, and that each hierarchical node has only one conditioning port.

In the engine ignition control example, the input ports $i1$ and $i2$ (of $H1$, resp. $H2$) are of integer type, and can take values 0 or 1[3]. Input port $i1$ (resp. $i2$) is the conditioning port of node $H1$ (resp. $H2$), which means that for each of these two hierarchical nodes, two descriptions must be provided, as detailed below.

3.3.2 Translation technique

From this input specification, the translator produces the clocked graph program displayed in Fig. 3.14. To do so, a first step consists in building an intermediate representation of the specification where the conditional control is no longer hidden: each mode inside a hierarchical node is expanded, and execution conditions are explicited. The relationship between the ports of hierarchical nodes and the nodes inside their referenced descriptions are also made explicit. Fig. 3.16 shows the intermediate representation of the engine ignition example. We see for example that the two modes inside the $H1$ hierarchical node are replaced by one single mode including two explicitely conditioned nodes. Each corresponds to one of the execution modes that were initially in $H1$. These new nodes are still hierarchical, but each describes only one mode. Note that at this stage $H1$ has not yet disappeared: it still plays the role of intermediate to link the arcs between its inner new nodes and the top-level nodes. The same holds for node $H2$. From this representation, the translator performs two important transformations:

- Flatten the hierarchy so that the final clocked graph specification will not be hierarchical. The hierarchical blocks disappear, and only the computation nodes remain. The translator performs disambiguation of the names of such nodes, so that no two remaining nodes will have the same name.

- Compute the execution conditions associated to all dataflow nodes and communications, and transform them into clocks. Since SynDEx uses integer variables for its conditioning, and CG uses boolean clocks, a conversion must be done at this stage. For input specifications where the values of the conditioning port can be different than 0 or 1, we have designed a module that rewrites the considered node into nested hierarchical nodes with binary (0/1) conditions.

The most difficult part of the translation is the handling of the arcs: the translator must create corresponding clocked variables labelled with the good conditions and that globally

[3]But no range check is performed in SynDEx or by our tools.

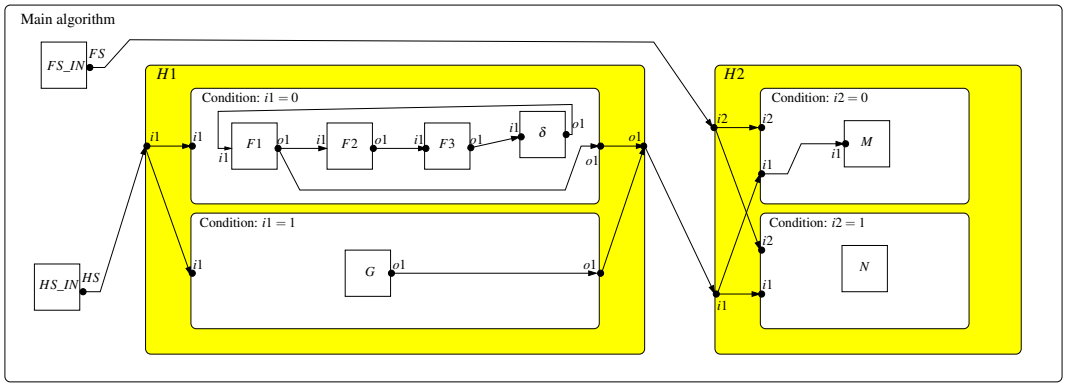


Figure 3.16: Intermediate translation of the SynDEx specification

reproduce the semantics of the initial specification. In the hierarchical description, arcs representing the same variable may be split or merged at various places, go through different hierarchical and conditioned blocks, and the translator must replace them by variables present in the correct input ports under the good condition in the final specification.

From the intermediate representation of Fig. 3.16, the flattening of the hierarchy will preserve only the computation blocks HS_IN, FS_IN, $F1$, $F2$, $F3$, G, M, and N. The arcs corresponding to the conditioning ports are hidden, since the control will be handled using clocks. The arcs that lead from $F1$ and from G to M are represented with two clocked variables. The arcs that do not come from an output port or lead to an input port of a computation node are erased. Finally, the control is translated into clocks that are explicitely applied to the computation nodes: condition $i1 = 0$ is translated into $\neg HS$, $i1 = 1$ into HS, $i2 = 0$ into $\neg FS$ and $i2 = 1$ into FS. This process leads to the graphical representation already given in Fig. 3.13.

3.4 Abstraction as single-period task systems

We have now completed the definition of the formalisms we use for functional specification: the low-level CG we use as an internal compiler representation, and the high-level synchronous languages that can be translated into CG. But both CG and the high-level synchronous languages are full-fledged synchronous languages allowing the description of complex behaviors involving data-dependent control.

However, in both compilation and in the real-time systems community it is common to define mapping (allocation and scheduling) algorithms on simpler models that abstract away much of the complexity of a full-fledged synchronous specification. For instance, task models in real-time scheduling either have no way of representing conditional execution, or represent them using exclusion relations, instead of providing the execution conditions themselves. Another common simplification used in both real-time task mod-

els and the data dependency graphs (DDGs) used in compilation is that data flows between various computations are abstracted away as simple dependencies. Furthermore, no information is provided on the interface with computation functions provided in external C libraries.

These abstractions facilitate the definition of mapping algorithms, by placing focus on the properties (*e.g.* exclusion properties) that are directly exploited by the mapping algorithms. However, it is important that the abstraction is clearly defined, to ensure that once mapping is done in the abstract model the result is implementable when the lower-level detail is taken into account.

We define in this section the basic version of the abstract task model we use throughout this thesis (possibly with extensions defined later). This task model is very similar to formalisms existing in both real-time scheduling (the task models of, among others, Blazewicz and Chetto [Blazewicz, 1977, Chetto et al., 1990]) and in the compilation world (the data dependency graphs [Allan et al., 1995]). As such, the model will allow the direct application of techniques coming from both the real-time scheduling and compilation communities, which are detailed in the following chapters. The model is also close to the CG language. The algorithms allowing the replacement of CG's clocks with an exclusion relation are provided in Chapter 6, and the other elements of the abstraction are straightforward[4]. We also provide, in the following section, an explanation of how CG and the abstract task model cover our modeling needs.

Our scheduling technique works on *deterministic* functional specifications of dataflow synchronous type, such as those written in SCADE/Lustre [Caspi et al., 2003]. These formalisms, which are common in the design of safety-critical embedded control systems, allow the representation of *dependent task systems* featuring *multiple execution modes*, *conditional execution*, and *multiple relative periods*.

It is important to recall at this point that our formalization is based on a dataflow synchronous paradigm, and not on Lee and Messerschmitt's Synchronous Data Flow (SDF) [Lee and Messerschmitt, 1987b] and derived models. Confusion sometimes occurs, as both classes of models aim at describing deterministic, tightly synchronized systems, and have a data-flow form. However, there is a fundamental difference concerning synchronization. While in SDF-like formalisms synchronization is local, driven by the flow of messages, the synchronous paradigm relies on a notion of global synchronization which is driven by one or more global time bases, known as *(logical) clocks*. Through their use of global time bases, synchronous formalisms are very close to formalisms used in control theory, which explains the widespread use of synchronous formalisms in the design of embedded control systems in fields such as avionics or automotive.

[4] An important remark here is that our thesis emphasizes, through its definition of this abstract model, the close relations existing between formalisms used in 3 communities: synchronous programming, real-time scheduling, and compilation.

The abstract model we use is also synchronous, in the sense that it divides execution in a sequence of instants. We define our task model in two steps. The first one covers systems with a single execution mode:

Definition 1 (Non-conditioned dependent task set) *A non-conditioned dependent task system is a directed graph defined as a triple*

$$D = \{T(D), A(D), \Delta(D)\}.$$

Here, $T(D)$ is the finite set of tasks. The finite set $A(D)$ contains typed dependencies of the form $a = (src(a), dst(a), type(a))$, where $src(a), dst(a) \in T(D)$ are the source, respectively the destination task of a, and $type(a)$ is its type (identified by a name). The directed graph determined by $A(D)$ must be acyclic. The finite set $\Delta(D)$ contains delayed *dependencies of the form $\delta = (src(\delta), dst(\delta), type(\delta), depth(\delta))$, where $src(\delta)$, $dst(\delta)$, $type(\delta)$ have the same meaning as for regular dependencies and $depth(\delta)$ is a strictly positive integer called the* depth *of the dependency.*

Like synchronous (CG) programs, non-conditioned dependent task sets have a cyclic execution model. At each execution cycle of the task set, each of the tasks is executed exactly once. We denote with t^n the instance of task $t \in T(D)$ executed in cycle n. The execution of the tasks inside a cycle is partially ordered by the dependencies of $A(D)$. If $a \in A(D)$ then the execution of $src(a)^n$ must be finished before the start of $dst(a)^n$, for all n. Note that dependency types are explicitly defined, allowing us to manipulate communication mapping.

The dependencies of $\Delta(D)$ impose an order between tasks of successive execution cycles. If $\delta \in \Delta(D)$ then the execution of $src(\delta)^n$ must complete before the start of $dst(\delta)^{n+depth(\delta)}$, for all n. We make the assumption that a task has no state unless it is explicitly modeled through a delayed arc. Thus, the code of each task is a simple sequential function with no internal state. Note that for tasks with no state, it is possible that two instances are executed in parallel if the architecture has enough resources.

This first definition is similar to classical definitions of dependent task sets in the real-time scheduling field [Chetto et al., 1990], and to definitions of data dependency graphs used in software pipelining [Allan et al., 1995, Chiu et al., 2011].

But we need to extend this definition to allow the efficient manipulation of specifications with multiple execution modes. The extension is based on the introduction of a new *exclusion relation* between tasks, as follows:

Definition 2 (Dependent task set) *A dependent task set is a tuple*

$$D = \{T(D), A(D), \Delta(D), EX(D)\}$$

where $\{T(D), A(D), \Delta(D)\}$ is a non-conditioned dependent task set and $EX(D)$ is an exclusion relation $EX(D) \subseteq T(D) \times T(D) \times \mathbb{N}$.

The introduction of the exclusion relation modifies the execution model defined above as follows: if $(\tau_1, \tau_2, k) \in EX(D)$ then ${\tau_1}^n$ and ${\tau_2}^{n+k}$ are never both executed, for any execution of the modeled system and any cycle index n. For instance, if the activations of τ_1 and τ_2 are on the two branches of a test we will have $(\tau_1, \tau_2, 0) \in EX(D)$.

The relation $EX(D)$ is obtained by analysis of the execution conditions (logical clocks) in the data-flow synchronous specification.

The relation $EX(D)$ needs not be computed exactly. Any sub-set of the exact exclusion relation between tasks can safely be used during scheduling (even the void sub-set). However, the more exclusions we take into account, the better results the scheduling algorithms will give because tasks in an exclusion relation can be allocated the same resources at the same dates.

3.5 Modeling multi-period task systems

The task model defined above is defined over a single-clock synchronous model, which most naturally models systems where tasks are repeated at the same pace (modulo some data-dependent control). However, most real-life scheduling problems involve multi-period task systems where tasks run at different paces. In such cases, period information is usually classified as non-functional. However, it also impacts functional specification: without knowing the *ratio* between the periods of the tasks it is impossible to precisely define how the tasks exchange data, and therefore it is impossible to ensure the determinism of the functional specification. In this section we explain how our model allows the representation of multi-periodic task systems through the use of an operation called *hyperperiod expansion*.

To illustrate our approach, we consider a task system composed of five tasks $\{F, R, G, R', G'\}$ whose periods are respectively 20ms, 20ms, 40ms, 20ms, and 40 ms. Fig. 3.17 provides the functional specification of this system under the form of a multi-clock synchronous program written in the Signal/Polychrony [Guernic et al., 2003] synchronous language. It deterministically defines the communication scheme between the tasks, taking into account the execution modes they belong to. It starts by the declaration of an affine constraint which defines the relationship between the activation rates of the two main clocks of the application: Gclock activates once every two activations of Fclock, each time with the second activation of Fclock. The boolean variable *mode* is used to define in which execution mode the program is running at each activation instant. If *mode* evaluates to true, then the program runs in mode1, in which tasks R and G are executed, but not Rprime and Gprime. Conversely, if variable *mode* evaluates to false, the program is running in mode2, where tasks Rprime and Gprime are executed, but not R and G. Task F is executed in both modes. The execution mode is initialized as mode1 when the execution starts, and

```
process MultiRate()
(| AffineConstraint(Fclock,2,1,Gclock,1,0)
 | mode := (Fmode when Gclock) default
             (mode $1 init true)                    %initial mode=mode1%
 | gtofoversampled := gtof default
             (gtofoversampled $1 init K2)
 | (Fs,Fmode,ftor,ftog) := F(Fs $1 init K0,
             rtof $1 init K1, gtofoversampled)          %F%
 | rtof := R(ftor when mode) default
             Rprime(ftor when not mode)                 %R and Rprime%
 | gtof := G(f2g when (mode when Gclock)) default
             Gprime(f2g when ((not mode) when Gclock))  %G and Gprime%
 |)
 where Fclock,Gclock:event ;                            %rates (clocks)%
       mode:boolean ; Fs:FStatetype ;                   %state variables%
       ftor:FtoRtype ; rtof:RtoFtype ;
       ftog:FtoGtype ; gtof:GtoFtype ;
 end;
```

Figure 3.17: Synchronous specification of the multi-rate dependent task system of Section 3.5

is assigned a new value (given by the output Fmode of F) at each execution instant where clock Gclock is active. In other execution instants the value of the mode is preserved.

At each execution of task F, it reads the last value produced by R or Rprime and the last value produced by G or Gprime, depending on the execution mode. These variables are named *rtof* and *gtof* in the program, and their value explicitely depends on the execution mode: in the activation instants when *mode* is set to true (mode1), task R is activated, and its return value is assigned to *rtof*. Otherwise, task Rprime is activated and gives its output to *rtof*. The same happens with *gtof* and tasks G and Gprime, except that G and Gprime are only activated on the subsampled clock Gclock. Since task F needs an input coming from either G or Gprime at each of its activations, the value *gtof* has to be oversampled, in order to be present at each activation of task F. Variable *gtofoversampled* performs this by replicating its own value on the activation instants when *gtof* is not present (when Gclock is not activated), and by updating this value every time *gtof* gets a new value. At each activation, task F also reads the value of Fs. This value was produced by F during its previous activation. The value Fs explicitly models the state of task F. Note that transferring data from one execution instance to the next requires the use of a special construct, named a *delay*, which is represented here with $1.

This example shows how Signal/Polychrony allows the faithful functional specification representation of multi-period systems. However, the definition of a real-time implementation problem requires the representation of non-functional information, most importantly the real-time characterization of the task system. This characterization includes global characteristics of the system (*e.g.* is it strictly periodic, periodic, sporadic, *etc.*)

and individual characteristrics of the tasks, such as their periods, release dates, deadlines, *etc.* The main synchronous languages, such as Signal/Polychrony or Scade do not specify a real-time execution model, nor do they provide the language contructs needed to specify task characteristics. In our case, the program specifies that one task is executed twice as often as another, but it cannot specify the real-time period of either task.

Several extensions have been proposed to existing synchronous languages to allow the specification of real-time characteristics. One of the most advanced is the one of Pagetti *et al.* [Pagetti et al., 2011], which draws influences from Chetto *et al.* [Chetto et al., 1990] and Cohen *et al.* [Cohen et al., 2006], among others. The focus of Pagetti *et al.* is on the *compact* representation of multi-period specifications, and on its *efficient* manipulation for scheduling purposes. They target implementations where all tasks are periodic and use specification-level task periods to drive the on-line scheduler of the implementation.

In this thesis we make a very different choice, detailed in Chapter 7. Instead of requiring that all tasks are periodic, we only require periodicity for the input acquisitions and for the outputs (actuations) of the system. Computation tasks can follow any schedule, even if it is not periodic in the sense of real-time scheduling. For instance, multiple instances of a task can be executed in a burst, if data dependencies and real-time constraints allow it. Such bursty execution often improves efficiency by reducing cache-related traffic and context changes.

This implementation model explains why we did not invest time during this PhD thesis in defining constructs for the modeling of multi-periodic specifications, or algorithms for their manipulation. Instead, we prefered to focus on scheduling problems for single-period task systems. For dealing with multi-periodic systems we rely on *hyperperiod expansion*. Hyperperiod expansion is a classical operation of scheduling theory [Munier, 1996, Richard et al., 2001, Zheng et al., 2005], consisting in replacing a multi-period task system with an equivalent one in which tasks have all the same period. The period of the resulting tasks is equal to the hyper-period of the initial task system, which is the least common multiple of the periods of its tasks/operations. For example, the hyperperiod of the task system we consider in this section is 40 ms.

Hyperperiod expansion works by determining how many instances of the tasks in the initial system are needed to cover the hyperperiod. In the worst case, this operation may lead to an exponential increase in the number of tasks to consider (as a function of the number of initial tasks). This happens when the periods of the input tasks are relatively prime. However, this exponential explosion does not happen in our case, because realistic task systems have harmonic periods (as in our industrial example) or follow simple period ratios. Furthermore, the increase in terms of generated code length is negligible, as only calls to the task functions are replicated (the code of the functions is not inlined in the generated application). More fundamentally, hyperperiod expansion is one solution to the

trade-off between code size and schedulability. Existing solutions often privilege code size by requiring tasks to be strictly periodic or by imposing implicit deadlines (equal to the periods). We do not impose such constraints, which are not part of the initial industrial requirements, which gives more freedom to the off-line scheduling algorithms.

Note that hyperperiod expansion does not imply in our case an increase in the size of the initial phase of the scheduling (often called prologue in repetitive scheduling contexts). There are two causes to this: first of all, scheduling is fully static, so there is no dynamic pipeline filling phase. Second, we do not aim at optimizing the overall duration of a finite loop computation (including its prologue). Therefore, we do not try to optimize the duration of the prologue. As we shall see in Section 8.2.1 (cf. Page 156), we schedule its operations following the pattern of the steady state.

In our example, F has period 20ms, so it must be replaced by 2 tasks F_1 and F_2, both of period 40 ms. Similarly, R is replaced by R_1 and R_2 and R' is replaced by R'_1 and R'_2. Tasks G and G' do not require replication because their periods are already equal to the hyperperiod.

Replication of tasks is accompanied by the replication of dependency arcs [Munier, 1996], which follows the same rules, but is more complicated due to the fact that an arc can connect tasks of different periods, and thus may involve under- or over-sampling (as it can be specified in languages such as Prelude or Giotto). The period-driven replication must infer which instances of the source and destination task must be connected through arcs.

The full hyperperiod expansion of our example is pictured in Fig. 3.18. Regular dependencies are represented here using solid arcs. Delayed dependencies are represented using dashed arcs. The label of a delayed dependency gives its depth. For instance, a regular dependency connects F_1 and F_2 to signify that inside a hyperperiod F_1 must be executed before F_2. We did not graphically represent here the type information specified by our formal model, which determines the kind (and amount) of data that is passed from F_1 to F_2. The delayed dependency of depth 2 between G and F_1 means that the instance of G started in the execution cycle of index n must be completed before F_1 is started in the execution cycle of index $n+2$, for all n. Note that task F has an internal state, which is expanded into arcs connecting its instances F_1 and F_2, whereas the other tasks are stateless.

Fig. 3.18 also emphasizes exclusion relations: an exclusion relation of depth 0 relates each task of *mode1* to each task of *mode 2*. We have graphically represented these relations with a relation between the two sets of tasks that are activated in only one of the two modes. Tasks F_1 and F_2 belong to both modes.

As explained above, dependent task systems are the abstract model containing just the formal elements needed to define our scheduling algorithms. But our tools work on full-

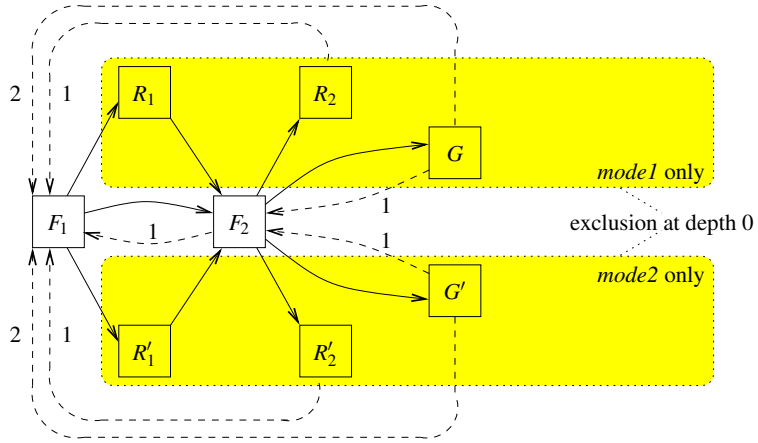


Figure 3.18: Example of dependent task system

```
process MTFfunction()
(| mode := Fmode2 $1 init true
 | (Fs1,Fmode1,ftor1,ftog1) :=
       F(Fs2 $1 init K0, rtof1 $1 init K1, gtof $2 init K2)        %F1%
 | (Fs2,Fmode2,ftor2,ftog2) :=
       F(Fs1, rtof2, gtof $1 init K2)                              %F2%
 | rtof1 := R(ftor1 when mode) default
              Rprime(ftor1 when not mode)              %R1 and R1prime%
 | rtof2 := R(ftor2 when mode) default
              Rprime(ftor2 when not mode)              %R2 and R2prime%
 | gtof  := G(ftog when mode) default
              Gprime(ftog when not mode)                 %G and Gprime%
 |)
 where mode,Fmode1,Fmode2:boolean ;                    %state variables%
       F1s,F2s:FStatetype ;   ftor1,ftor2:FtoRtype ;
       rtof2,rtof1:RtoFtype ; ftog1,ftog2:FtoGtype ;
       gtof:GtoFtype ;
 end;
```

Figure 3.19: Synchronous program corresponding to the task system of Fig. 3.18

fledged dataflow synchronous programs defining all details needed to allow executable code generation. Fig. 3.19 provides a data-flow synchronous program corresponding to the dependent task system in Fig. 3.18. This program is also written in the Signal/Polychrony language, and since it defines single-rate tasks, our tool can take it as input. Translating this program into CG is straightforward: calls to R, R', F, and G become the blocks of the data-flow graph, data-flow dependencies become the variables, and the delays of depth 1 and 2, identified with `$1` and `$2`, become delay blocks.

3.6 Conclusion

This chapter defined the CG language, which is the functional specification language taken as input by our tools. CG is a synchronous language, and more precisely a single-clock synchronous language. CG programs can be abstracted as mono-periodic task systems that only exhibit the elements used by our scheduling algorithms. To allow the modeling of multi-periodic systems in our single-clock language, we use a hyperperiod expansion technique, which is presented using a comprehensive example. The next two chapters will show how the non-functional aspects of an embedded application are modeled in CG. In the next chapter, we will describe how the execution platform and the constraints it imposes on the system are specified. Then in Chapter 7, we will focus on the non-functional extensions made to the CG language in order to model complex non-functional constraints (real-time, preemptability, and partitionning).

Chapter 4

Modeling resources and resource allocation

Contents

This chapter starts the presentation of the non-functional aspects that we consider in our implementation work. It is focused on the modeling of the execution platform and of the way resources of the execution platform are allocated to the computations and communications of the application.

Modeling of the execution platform in a way that reconciles predictability (safety) and performance is a central problem in real-time scheduling. Indeed, some form of platform model is required by all mapping (allocation and scheduling) algorithms. But when hard real-time guarantees are needed, a platform model must include, in one way or the other, several types of information:

1. The resources of the platform. These include computation resources, such as processors or accelerators, and communication resources, such as buses, FIFO buffers, or shared RAM banks. For multi-processor platforms such as the ones we consider, this platform model is often provided under the form of a graph representing the way these resources are linked to each other, and the capabilities of each resource (the operations it can execute and the cost of executing each operation) [Sorel, 2005].

2. An execution model. There is more to the platform model than a mere interconnection graph. Indeed, timing analysis and schedulability analysis must rely on an

analytical model of the way operations are executed on these resources at runtime. This model must include implicitly or explicitly:

- A description of the way applications are run on the execution platform. Such a description may include the task scheduling paradigm, the network routing and switching algorithms, memory size and organization hypotheses, *etc.*.
- A formal definition of what it means for an implementation to be correct with respect to its functional and non-functional specification. This definition must include properties such as functional correctness (*e.g.* specification-level data dependencies are respected in the implementation), real-time correctness (*e.g.* each operation is allocated enough time), and possibly correctness with respect to other non-functional requirements (partitioning, preemptability, *etc.*).

The execution model can be specified operationally, under the form of allocation and scheduling algorithms (as is often the case in WCET analysis), under the form of a set of constraints to be respected by the mapping (like in scheduling approaches based on constraint solving), or a combination of both.

3. An implementation model. This is the output of the mapping algorithms. It consists of all the configuration information required to configure the various resources of the platform. It usually includes:

   - Allocation information: distribution of computations to computation resources, distribution of data and code to memory banks, routing of communications onto the interconnect, *etc.*
   - Configuration of all arbiters/sequencers/schedulers that allow/require configuration. Depending on the platform, this may include priorities, TDM tables, assigned bandwidths, scheduling tables, *etc.* in both computation and communication resources.

   An implementation model should allow the synthesis of a running implementation on the actual execution platform by means of simple code generation transformations.

Defining a good platform model (resources, implementation model, execution model) is difficult. First of all, a platform model must be a conservative abstraction of the actual execution platform. Once a correct implementation model is built, the implementation built from it on the actual execution platform should be correct itself[1].

[1] Keep in mind that correctness here includes both functional aspects (causality) and non-functional ones (timing predictability, reliability, *etc.*).

A platform model must also provide a good compromise between *simplicity* and *precision*. Simplicity, that is a higher abstraction level, is needed to ensure the tractability of the mapping and analysis algorithms. Timing precision in the allocation of resources is needed to ensure that resources are not wasted.

To reconcile performance, predictability, and tractability, multiple abstraction levels of the same architecture are usually needed in the analysis of a single system. Two commonly-used platform abstraction levels are those used in Worst Case Execution Time (WCET) analysis and in schedulability analysis, respectively:

- In WCET analysis, the platform model includes very detailed micro-architectural features such as caches, CPU pipelines, *etc.* Such a complex resource model of the hardware is often used in conjunction with rather simple models for the software that is executed (sequential code with no or limited interrupts) to provide very precise execution time estimates for the system *tasks*.

- In schedulability analysis, the whole system is modeled, but with a much coarser grain. For instance, in a distributed system each single-board computer (SBC) may be represented using a single execution resource.

Great care must be taken to ensure that the various abstraction levels are consistent with one another. For instance, it must be ensured that the hypotheses made during WCET analysis (*e.g.* absence of interrupts) are respected in the system through the use of a non-preemptive scheduling model, or that schedulability analysis uses safety margins that are proven safe by taking into account the worst-case overhead due to interrupts.

In our experience, the consistency between various abstraction levels and the consistency between abstractions and the platform itself is a critical point in current industrial practice of real-time systems design. There are two main reasons to this:

- Building formal abstractions that are correct, simple, and precise for complex systems formed of both hardware and software (OS, drivers, etc.) is difficult in itself.

- In most cases, the platform descriptions that must be used to develop the abstract platform models are incomplete. Typical missing details are the exact cache algorithms, the exact arbitration policies, OS internals, *etc.*

This is why, in current design practice, the margins used during schedulability analysis are often derived from experience with little or no formal justification. This practice is even less acceptable today, given that the platform hardware and software undergo significant changes such as the move to multi-/many-core platforms or major changes in the OS type through the inclusion of mechanisms such as time/space partitioning.

All these arguments explain why off-line scheduling approaches [ARINC, AUTOSAR, Kopetz and Bauer, 2003] are gaining acceptance in various industrial settings.[2] Indeed, in off-line scheduling approaches conformance between the actual system and the formal models of the platform and of the implementation can be validated at a smaller cost in both time and money. When implementing hard real-time systems, off-line scheduling also has a second major advantage: it can be done with high temporal precision, resulting in very efficient implementations. For both reasons, my work in this thesis will also follow an off-line scheduling paradigm.

In this chapter we introduce the formal platform model used during my PhD work. As we shall see, we did not use here a single formalism, but a family of closely related formalisms based on a unique scheduling paradigm, namely *table-based off-line scheduling*. Table-based off-line resource allocation provides the unifying formal basis of my work. Then, variations in the syntax and semantics of the actual platform models allow us to take into account a variety of platforms ranging from the bus-based time-triggered platforms considered in Chapter 8 to many-core platforms such as the ones used in [Carle et al., 2014].

Format variations are needed not only to take into account different types of platforms, but also to accomodate various abstraction levels used by different mapping algorithms. As we shall see, some of our allocation and scheduling algorithms [Carle et al., 2012] take a coarse-grain approach to the modeling of memory, whereas others require a more precise representation [Carle and Potop-Butucaru, 2014].

Chapter outline To facilitate presentation, this chapter is organized as follows: we start by presenting the simplest version of the resource description formalism. Based on this formalism we explain how table-based off-line mapping is performed. we also provide the formalism used to represent the resulting implementation models (the scheduling tables), define its semantics, and explain how actual implementation code is generated from it.

We will then discuss in Chapter 5 variations of the resource description formalism, used to represent different architecture types and different abstraction levels.

Note that this chapter only considers non-functional properties related to topology and real time. Other non-functional properties, such as *preemptability*, *partitioning*, *deadlines* and *release dates* shall be covered later, in Chapter 7.

4.1 Resource description formalism

We define here the simplest version of our resource description formalism. Its constructs allow the description of distributed architectures formed of a set of sequential processors

[2] Often in conjunction with a time-triggered execution model and strong space/time isolation mechanisms.

$\mathcal{P} = \{P_i | i = 1, ..., n\}$ interconnected by a unique broadcast bus $\mathcal{B}$. Timing information relates the processors and buses with elements of the functional specification defined in the previous chapter:

- For each processor P and for each function f of the functional specification that can be executed on P we provide the duration of executing f on P, denoted $d_P(f)$. This value is obtained through a conservative WCET analysis of f on P. For uniformity, we shall extend the notation $d_P(f)$ to all processors P and functions f by assigning a value of $d_P(f) = \infty$ whenever the function f cannot be scheduled on processor P.

- For each processor P and for each data type t (of the functional specification) that can be stored locally on P we provide the duration of executing on P the bookkeeping operation associated with a delay block of type t. This value is denoted with $d_P(t)$, and it should also be a safe WCET estimation.

- For each data type that can be transferred over the bus $\mathcal{B}$, we provide the worst case duration of transferring one value of type t over the bus, assuming this bus is free from other traffic. This value is denoted $d_{\mathcal{B}}(t)$.

Durations are provided as integers, the unit not being specified.

A graphical example of such an architecture, with timing information corresponding to the example application of Section 3.2.4 is given in Fig. 4.1. This architecture has three processors and:

- Processor P_1 can execute functions HS_IN, FS_IN, $F1$ and $F2$, with execution durations: $d_{P_1}(HS\_IN) = 1$, $d_{P_1}(FS\_IN) = 1$, $d_{P_1}(F1) = 3$ and $d_{P_1}(F2) = 8$ time units. P_1 is the only processor capable of executing HS_IN and FS_IN: this models the fact that P_1 is the only processor connected to the I/O interface device of the system.

- Processor P_2 can execute G with a duration $d_{P_2}(G)$ of 3 time units, and $F3$ also in 3 time units.

- Processor P_3 can execute M and N in 3 time units each, and has a hardware accelerator that allows it to execute $F3$ in only 1 time unit.

Since P_2 and P_3 cannot compute HS_IN, we have $d_{P_2}(HS\_IN) = d_{P_3}(HS\_IN) = \infty$ (not represented in the figure). Each processor can store values of type V_type, such as the one produced by $F3$, and the bookkeeping operation for a delay of type V_type takes one time unit on any processor. The bus can transmit variables of types boolean and V_type in 2 time units, and variables of type ID_type in 5 time units: $d_{\mathcal{B}}(V\_type) = d_{\mathcal{B}}(bool) = 2$ and $d_{\mathcal{B}}(ID\_type) = 5$.

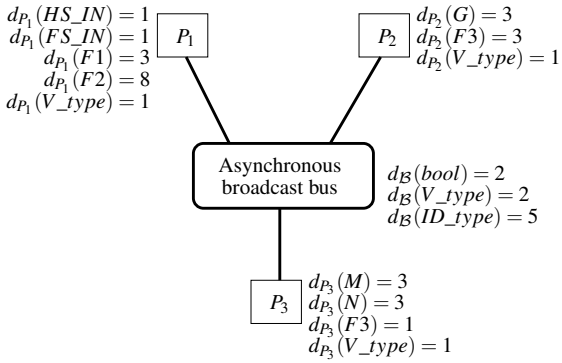


Figure 4.1: Hardware architecture example

Syntax The general syntax for modeling hardware architectures in CG, including the timing information necessary for the scheduling process is given in Fig. 4.2. As for the functional specification part of CG, the information declared to describe the architecture is tabulated. Thus, we need once again to define indices to denote without ambiguity any element of the specification inside its table. The rest of Fig. 4.2 defines one table for the bus declaration, and one table for the processors. Each of them can be left empty: if both are left empty, then the CG program is just a functional specification, which cannot be scheduled. If the bus table is empty, then the processors cannot communicate with one another (this is for example the case in mono-processor architectures). If no processor is declared, then declaring a bus is useless, since scheduling cannot be done without at least one processor. Each processor has an index, a name, and a list of WCET values. This list is composed of the functions that can be executed on the processor (in finite time) along with the corresponding WCETs, as well as the types of the variables that can be stored on the processor (for delayed variables), along with a safe WCET for the storing operation. The same syntax is used for the bus table (which in our case will contain only one record).

The formal, textual representation of the architecture of Fig. 4.1 is provided in Fig. 4.3.

Now that we have presented the syntax for specifying the architectures on which applications are executed, we will explain in more details the principles of the table based off-line scheduling approach that we advocate and which is implemented in LoPhT.

4.2 Scheduling tables

The simplest way of using the LoPhT compiler is by providing it an input file containing a functional specification following the syntax of Chapter 3 and a matching resource description as defined above. Starting from this input, LoPhT will perform the mapping of the functional specification on the architecture model. This consists in performing the

```
archi            := Architecture bus_table proc_table
bus_table        := Bus Table bus
bus              :=
                    |bus_idx Broadcast Bus comm_durations
comm_durations   := comm_duration
                    |comm_duration comm_durations
comm_duration    := Duration(type_idx)= <Int>
proc_table       := Processor Table proc_list
proc_list        :=
                    |proc proc_list
proc             := proc_idx <Identifier> comp_durations
comp_durations   := comp_duration
                    |comp_duration comp_durations
comp_duration    := Duration(fun_idx)= <Int>
                    |Duration(type_idx)= <Int>
bus_idx          := Bus:  <Int>
proc_idx         := Processor:  <Int>
```

Figure 4.2: Clocked graphs architecture syntax

```
Architecture

Bus Table

 Bus:0   BroadcastBus   Duration(Type:0)=2
                        Duration(Type:1)=5
                        Duration(Type:2)=2

Processor Table

 Processor:0   P_1   Duration(Function:0)=1
                     Duration(Function:1)=1
                     Duration(Function:2)=3
                     Duration(Function:3)=8
                     Duration(Type:2)=1

 Processor:1   P_2   Duration(Function:4)=3
                     Duration(Function:5)=3
                     Duration(Type:2)=1

 Processor:2   P_3   Duration(Function:4)=1
                     Duration(Function:6)=3
                     Duration(Function:7)=3
                     Duration(Type:2)=1
```

Figure 4.3: Example architecture syntax

(spatial) allocation and (temporal) scheduling of the blocks of the functional specification onto the processing elements of the architecture model, as well as the scheduling of the bus communications needed to transmit data produced by a block on one processor to blocks that are executed on other processors. Once the mapping is performed, LoPhT can produce executable code implementing the result of the mapping phase.

The output of the mapping phase, taken as input by the code generation phase, is an implementation model consisting in a scheduling table (known in some fields as a reservation table). This section will define our scheduling table formalism, its semantics, intuitively explain how these tables are synthesized, and how code can be generated from a scheduling table. We shall also provide a brief comparison with other implementation models used in real-time scheduling.

4.2.1 Table-based off-line scheduling (the principle)

In this thesis we only consider execution platforms whose computation and communication resources are sequential. This means that each moment in time, a processor can only be computing one function, and a bus can only be transmitting one given value/message. On such platforms, the role we assign to scheduling is not only to determine the order in which the computations and communications happen, but to reserve *time intervals* for them on the various resources. A time interval is characterized by a starting date and a duration, and defines when a resource is completely dedicated to a certain computation or communication. A *scheduling table* contains the collection of all the time intervals that must be reserved on each resource in order to perform the various operations (function execution, communication, data storage) of the application.

A scheduling table describes an allocation pattern of fixed *length*. The scheduling of the system is performed by periodically applying this pattern, with a periodicity equal to the length of the table. The length of the table depends on the application that is being mapped. For the control applications that we target, the length of the scheduling table is often closely related to the periods of the application tasks (equal, a multiple, or a divisor). For the single-clock functional specifications we defined in Chapter 3, a natural assumption that we shall make for the scope of this thesis is that the scheduling table describes the scheduling of one generic execution cycle of the functional specification. Consequently, the objective of our algorithms is to produce a table describing the schedule of one execution cycle of the application.

4.2.1.1 Semantics of scheduling tables

In this section we intuitively describe the execution model associated with our conditional scheduling tables. Fig. 4.4 displays a graphical representation of a scheduling table obtained by automatically mapping the engine ignition functional specification of Sec-

tion 3.2.4 (cf. Page 53) on the architecture of Fig. 4.1. In this representation, time flows from top to bottom, and each column represents the schedule of one resource of the architecture. In each column, the time intervals reserved for computations and communications are represented by colored boxes: for example, a time interval is reserved from date 0 to date 1 to compute *HS_IN* on processor P_1, and the time interval starting at date 1 and lasting 2 time units on the bus is dedicated to sending variable HS from P_1 to P_2 and P_3.

The complete schedule of the system is obtained by iteratively traversing this table from top to bottom: once the bottom is reached (after date 18), the execution is continued by restarting from the top.

As explained before, a time interval completely reserves a resource for a certain duration, starting at a particular date in order to perform an operation. Thus, in the general case, given a resource there should never be overlaps between two or more reserved time intervals. For example, if a time interval T_1 starts at date 0 and lasts 5 time units on processor P_0, then no other time interval can be reserved before date 5 on P_0. Nevertheless, we make an exception to this rule in order to take advantage of the explicit data-dependent control that can be specified using the CG language. To do so, we attach a condition to each reserved time interval. This condition corresponds to the clock of the block (or communication) for which the interval is reserved. In our graphical formalism, the condition attached to the reservation of a time interval is denoted using the operator @. For example, on processor P_1, the two first reservations are made under condition $true$, meaning that they are effective in any execution cycle. The third reservation (the interval starting at date 2), is considered only under the condition that $HS = false$: this reservation is only effective (and the corresponding function will only be executed) during execution cycles where HS evaluates to false. This adds a new dimension to the mapping problem: two time intervals T_1 and T_2 reserved on the same processor can overlap in time, but only if the conditions c_1 and c_2 that are attached to them are exclusive, that is to say if for any given configuration of the variables of the system, $c_1 \wedge c_2 = false$. In the example of Fig. 4.4, the two time intervals reserved on the bus starting respectively at dates 5 and 6 are overlapped from date 6 to date 10. This *double reservation* is correct because their conditions are exclusive: one reservation is made for the transfer of the output variable of $F1$ from processor P_1 under the condition that $(HS = false) \wedge (FS = false)$, and the other for the transfer of the output variable of G from processor P_2 under the condition that $(HS = true) \wedge (FS = false)$. These two conditions are clearly exclusive which allows this reservation optimization to occur. Since in any execution cycle of the application, at most one of the two conditions can be true, we have the insurance that at most one reservation can effectively be active at a time, and that there will be no contention for the use of the bus. In the abstract model defined in Section 3.4, this information is directly encoded in the exclusion relation $EX(D)$. In this formalism, when scheduling two tasks τ_1 and τ_2

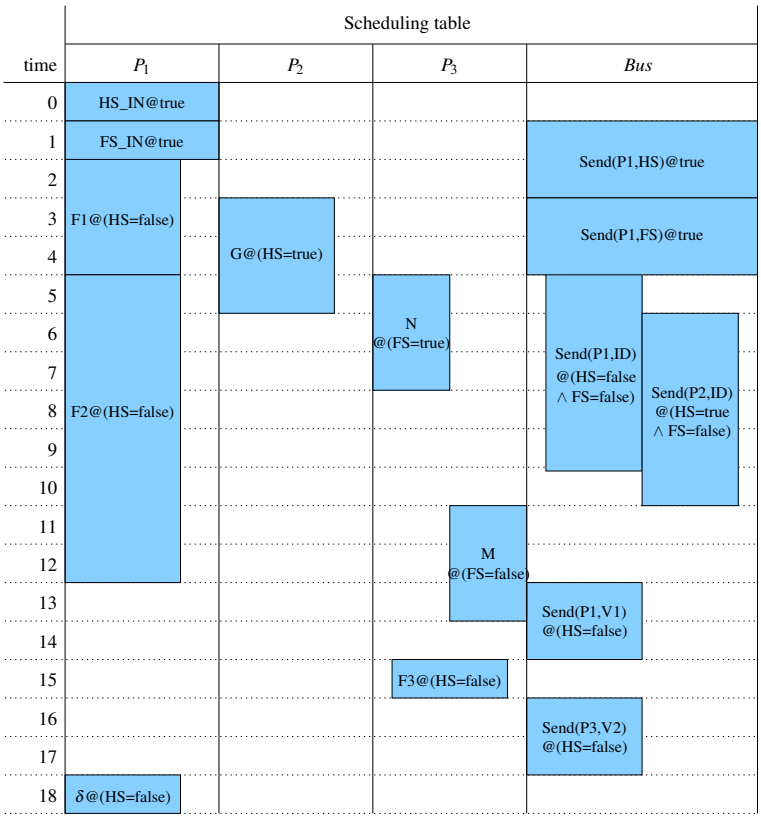


Figure 4.4: Example of a scheduling table for the result of the mapping of the engine ignition application

on the same processor, their corresponding reserved time intervals can overlap in time if $(\tau_1, \tau_2, 0) \in EX(D)$.

4.2.1.2 Implementing scheduling tables

In this section, we briefly discuss the effective implementation of embedded systems using scheduling tables. In particular we provide hints on how scheduling tables can be used to produce code for two particular execution paradigms.

The code generation scheme in SynDEx produces one infinite loop for each processor of the architecture. On any given processor, the code inside the loop is a sequence of function calls whose order is defined by the scheduling table. Moreover, communication primitives are inserted between the function calls to send or receive data on the bus whenever it is needed. The receiving primitive is blocking, meaning that it forces the considered processor to stall until the data has been received. Once the sending primitive has been executed, it is not allowed to execute again until the data that it sent during its last execution has been received. Such implementations allow the execution to advance at its own pace on the various resources, and pause it only to synchronize processors when they have to communicate, while keeping the number of buffers used for communications

low.

The code generated by LoPhT relies instead on the time-triggered paradigm, which we define in detail in Section 7.2.1. It considers that a globally synchronized timer is shared by all resources. Computations and communications are started when this timer reaches the date at which their time intervals start in the table. If safe WCET/WCCT bounds were used to determine the length of the time intervals, then the operations necessarily finish their execution (or transmission) before, or at the end of their reserved interval[3].

These two implementation schemes are illustrated in Fig. 4.5. This figure depicts the code generated for processor P_1 of the last example in both the SynDEx and LoPhT fashions. As said before, the code generated by SynDEx is an infinite loop, in which the function calls follow the order defined in the scheduling table. First, the processor makes the acquisition of variable HS, sends it on the bus and releases the bus. Then it does the same with variable FS. Afterwards, it evaluates the value of HS and computes $F1$ if it is false. The same is then done for $F2$. Finally, if HS is false, P_1 waits until $V2$ is received from the bus, and stores it in memory for the next cycle. The loop then starts again from the beginning. In the cycles where HS is true, the last four instructions are not executed, and thus no further synchronization occur for the cycle, since the blocking RECEIVE primitive is not called. Thus in this case, the next acquisition of HS is performed right after sending FS and releasing the bus.

On the other hand, in the time-triggered code generated by LoPhT, a global timer advances regularly and its value is read modulo 19 time units, since it is the table length. Then on processor P_1, HS is acquired at date 0 and FS at date 1 (in this example the bus is also considered to be time-triggered and statically programmed so that it knows which variable to send at which date, and where to read it in memory). Then at date 2, HS is evaluated. If it is false, $F1$ is started. At date 5, HS is evaluated again. If it is still false, $F2$ is started. At date 18, HS is evaluated one last time, and the code storing $V2$ is executed if HS is false. Note that no blocking RECEIVE primitive is used: we consider instead that at date 18 variable $V2$ will have been entirely received on P_1, since the bus is also statically scheduled. Another difference with the previous implementation is that if HS is true, although none of the three last function calls occur, the next cycle will only start when the global timer reaches 0 mod 19, which means a global synchronization for the cycle is preserved in any case.

4.2.1.3 Comparison with other resource allocation paradigms

Some of the advantages of table-based off-line scheduling have already been discussed in this document. Most important, this is the scheduling paradigm where the distance between implementation model and the implementation itself is the smallest possible, which

[3] We will show in Chapter 7 that more complex execution mechanisms can be applied to this model.

```
initialize;
loop
 HS_IN(HS);
 SEND(HS);
 unlock_bus;
 FS_IN(FS);
 SEND(FS);
 unlock_bus;
 if(!HS)F1(V2_del);
 if(!HS)F2(ID);
 if(!HS) RECEIVE(V2);
 if(!HS)exec_delay(V2);
end loop
          (a)
```

```
initialize;
every 19 ms
 0:  HS_IN(HS);

 1:  FS_IN(FS);

 2:  if(!HS)F1(V2_del);
 5:  if(!HS)F2(ID);

 18:  if(!HS)exec_delay(V2);
                    (b)
```

Figure 4.5: Illustration of the 2 main implementation schemes: (a) infinite loop in SynDEx, (b) Fully time-triggered in LoPhT

largely facilitates system *validation*. In particular, scheduler-related overheads are small and predictable. The second major advantage stems from the fact that off-line algorithms *theoretically* allow the computation of scheduling tables specifying an *optimal* allocation and real-time scheduling of the various computations and communications onto the resources of an architecture. These advantages are counterbalanced by three major disadvantages:

- Theoretical optimality may be unattainable due to a combination of 3 factors:
  - The application may exhibit a high degree of dynamicity due to either environment constraints (*e.g.* unknown task arrival dates) or to execution time variability resulting from data-dependent conditional control. Implementing an optimal scheduling for such applications may require more resources than the application itself.
  - The execution platform may not allow the implementation of scheduling tables. For instance, most multi- and many-core architectures provide little or no control over the scheduling of communications on the on-chip interconnect.
  - The mapping problems we consider are NP-hard. In practice, this means that optimality cannot be attained, and that efficient heuristics are needed.
- When applications exhibit significant dynamicity, it may be impossible in a static scheduling paradigm to ensure short response times for urgent tasks.
- Ease of validation, which is a major argument for the recent adoption of time-triggered industrial standards, is counterbalanced by the more complex develop-

ment process, which must produce a relatively complex implementation model (the scheduling table).

These arguments explain why off-line table-based methods are mainly used for two classes of applications: safety-critical embedded control applications[Cordovilla et al., 2011] where reducing validation costs is a major issue, and signal/image processing applications with largely regular control structure [Lee and Messerschmitt, 1987a].

Aside from the table-based paradigm, two other major resource allocation paradigms are used in real-time scheduling, which provide different trade-offs between ease of implementation, ease of validation, performance, *etc.*:

- Bandwidth reservation approaches, such as those based on real-time calculus [Thiele et al., 2000].
- Priority-based approaches.

Like table-based techniques, bandwidth reservation approaches also rely on a form of resource reservation. However, while in table-based techniques a reservation is a time window with fixed start and end date, in bandwidth reservation techniques a reservation involves a sliding window. A typical bandwidth reservation specification is that a task t will be able to execute on processor P for 10ms every 30ms (thus being able to use 30% of the computing bandwidth).

The reservations manipulated in table-based and bandwidth reservation approaches under the form of scheduling tables, real-time calculus specifications, *etc.* can be seen as *contracts* governing the real-time behavior of tasks and applications. These contracts can be manipulated in a *modular* fashion using *low-complexity* algorithms. Modularity in this context means the ability to infer the real-time properties of a system from the contracts associated with its direct sub-systems, without the need of considering internal aspects of these sub-systems. Modularity facilitates the incremental design and analysis of large systems.

By comparison, the priority-based resource allocation paradigm does not manipulate real-time contracts on system components (*e.g.* tasks). Instead, it focuses on the operational definition of the way resources are allocated to tasks at execution time using *on-line* priority-based algorithms. Real-time aspects are taken into account indirectly through assignment of *priorities* to the various tasks, which can be done using off-line (*e.g.* FP, RM, DM) or on-line (*e.g.* EDF, LLF) algorithms. Determining whether tasks meet their real-time requirements is done through a *schedulability analysis* phase that is usually holistic (not modular).

The three paradigms of resource allocation are not exclusive. They can be used for different parts of a system or at different levels of the same system. For instance, time

division multiplexing (TDM) scheduling, which is a table-based resource allocation technique, is often used to provide bandwidth reservation guarantees in both communication networks[Goossens et al., 2005] and for processor time [ARINC].

From the point of view of its ability to handle application dynamicity, the bandwidth reservation paradigm is intermediate between table-based off-line scheduling and priority-based on-line techniques. It can use on-line scheduling algorithms, yet it allows for a modular analysis (unlike priority-based approaches). The timing characterization is less precise in bandwidth reservation approaches than in table-based off-line scheduling. This may result in a waste of computing resources when the objective is to model systems with little dynamic execution. Maybe more important, dealing with execution modes requires non-trivial extensions to the resource reservation models, which in turn complicates analysis [Phan et al., 2008].

Priority-based scheduling approaches facilitate system design. Indeed, such systems usually rely on the use of proven real-time operating systems (RTOSs) whose real-time configuration consists in choosing task priorities (or the priority update policy). Implementation is followed by the application of schedulability analyses [Liu and Layland, 1973] meant to ensure that the resulting system satisfies its real-time requirements. A major advantage of priority-based techniques, when compared to table-based and bandwidth reservation techniques, is their ability of ensuring short response time for high-priority tasks. The main drawback associated with priority-based scheduling is the difficulty of precisely accounting for RTOS-related costs. Not only RTOS internals are often unknown, but schedulability analyses work best on simple task models where must of the complexity of the execution platform is hidden by means of heavy abstraction resulting in significant over-approximations and resource waste. Furthermore, in current industrial practice these abstractions are not formally defined (due to cost reasons) which requires expensive test-based validation phases.

4.2.2 Scheduling tables in LoPhT

We now present in detail the scheduling tables used in LoPhT. We start by describing the various elements composing the scheduling tables, as well as the syntax used in LoPhT. Then we define the properties that must be checked on the tables to ensure the correctness of the implementation. Finally, we illustrate the creation and use of a scheduling table in an example.

4.2.2.1 Syntax

The scheduling tables used in LoPhT to represent the result of the mapping process are divided in three parts: the table length, the list of implementation variables and the schedule of operations. The full syntax is displayed in Fig. 4.6.

Table length The table length is an integer value giving the duration of one execution cycle of the application, in the implementation model defined by our scheduling tables.

Implementation variables The list of implementation variables is used when generating the code: it defines the list of all variables that must be statically allocated by the implementation. This list may differ from the list of specification variables present in the initial CG program. Indeed, in our model, each time a variable v is produced on one processor and needed on another, memory must be allocated on both processors to hold the value of v. To account for this we must declare two implementation variables corresponding to v, one allocated on the producer processor, and the other on the consumer. In the list, each implementation variable is designated by a unique index, the type of the variable, the processor on which it is allocated, and the index of the specification variable it implements.

Schedules Each resource is then assigned its own schedule. For each processor, the schedule comes in the form of a table of *scheduled operations* which correspond to the blocks of the functional specification. A block b scheduled on a processor is characterized by an index which allows its non-ambiguous designation from any other scheduled operation in the system (and not just on the same processor). It also comprises the execution condition of the computation, the function (or delay) to compute, the starting date (refered to as $start(b)$ in the remainder of the chapter) and duration of the time interval reserved for the computation, and the input and output variables of the function. On the bus, the syntax is equivalent, except there is no reference to a function or delay. Instead we give the source variable of the communication, and its destination variables (since we consider a broadcast bus, each communication may have several destinations).

4.2.2.2 Correctness of a table

Verifying the correctness of a schedule consists in checking that some properties are verified in the scheduling table. For example, we want to ensure that the operations will never take more time than what has been allocated for them. To do so, as we suggested before, we impose that the time interval reserved for any function or delay (resp. communication) have a length computed using a WCET (resp. WCCT) analysis of the function/storing operation (resp. transmission) execution. We also want to enforce the exclusive use of resources by the scheduled operations, and the causal correctness of the scheduled application. The exclusive use of resources is guaranteed as long as:

- for each processor P, if two blocks b_1 and b_2 are scheduled on P, and $clk(b_1) \wedge clk(b_2) \neq false$ then $start(b_1) \geq (start(b_2) + d_P(b_2))$ or $start(b_2) \geq (start(b_1) + d_P(b_1))$,

```
scheduling table   :=  Scheduling Table length variables schedule
length             :=  ScheduleLength= <Int>
variables          :=  Implementation Variables var_table
var_table          :=
                       |implem_var var_table
implem_var         :=  var_idx type_idx Allocated On proc_idx Implements var_idx
schedule           :=  proc_schedules bus_schedule
proc_schedules     :=  proc_sched
                       |proc_sched proc_schedules
proc_sched         :=  ProcSchedule( proc_idx ) delay_list comp_list
delay_list         :=
                       |delay_op delay_list
delay_op           :=  sched_op_idx Condition clk_expr Delay StartDate <Int>
                                 Duration <Int> block_idx ( iport_list)->( oport_list)
comp_list          :=
                       |comp_op comp_list
comp_op            :=  sched_op_idx Condition clk_expr StartDate <Int> Duration <Int>
                                                    fun_idx ( iport_list)->( oport_list)
bus_schedule       :=  Bus Schedule( bus_idx ) comm_list
comm_list          :=
                       |comm comm_list
comm               :=  sched_op_idx Condition clk_expr StartDate <Int> Duration <Int>
                                               Source var_idx Destinations var_idx_list
var_idx_list       :=  var_idx
                       |var_idx var_idx_list
sched_op_idx       :=  SO: <Int>
```

Figure 4.6: Scheduling table syntax

- on the bus $\mathcal{B}$, if two different communications $comm(v_1)$ and $comm(v_2)$ are scheduled with conditions that are not exclusive, then $start(comm(v_1)) \geq (start(comm(v_2)) + d_{\mathcal{B}}(type(v_2)))$ or $start(comm(v_2)) \geq (start(comm(v_1)) + d_{\mathcal{B}}(type(v_1)))$ (where $type(v_2)$ and $type(v_1)$ designate the types of the two variables).

The causal correctness is a bit harder to define formally, and necessitates many notations that are defined in [Potop-Butucaru et al., 2009], which we will not copy here. Instead we will intuitively show how causality is checked in the scheduling tables. It amounts to statically verifying that any variable needed to start an operation (computation or communication) will be available at the starting date of the operation on the relevant processor, and that the computation of the corresponding supports is endochronous. By this we mean:

- **Causality for blocks.** Considering a computation corresponding to a block b, scheduled on processor P: for any clocked variable cv in the input ports of b and in the support $supp(clk(b))$, cv has been assigned a value, and this value is available on P, at least under the condition $clk(cv)$, at date $start(b)$. In other terms, either $src(cv)$ has been scheduled on P before $start(b)$ and with an execution condition that implies $clk(cv)$, or it has been scheduled on another processor, and a communication of the value of cv has been scheduled to finish before $start(b)$ and under a condition that implies $clk(cv)$.

- **Causality for communications.** Considering the communication on the bus of a

variable v under the condition clk initiated by processor P: for any cv in the support $supp(clk)$, cv has been assigned a value which has been made available to the whole system, at least under the condition $clk(cv)$, at date $start(comm(v))$. This ensures that any potential recipient of the communication can compute the condition under which it must receive the variable, and that P can compute the condition under which it must send it. Moreover, $src(v)$ must have been scheduled on P in such a way that $start(src(v)) + d_P(src(v)) \leq start(comm(v))$.

- **Endochronous schedule.** For any block b or clocked variable x in an input port of a block, we consider all clocked variables $cv = (var, clk)$ in the support of $clk(x)$, of $clk(b)$ or in the support of the clocked variables of the input ports of b. If a communication $comm(var)$ has been scheduled at date t under a condition $c \neq false$, then c can be computed using only the clocked variables that have already been sent on the bus before date t. Also, for all clocked variable $cv_1 = (var_1, clk_1)$ in the endochronous support of clk, var_1 has been sent on the bus at date t under a condition that implies clk_1.

Provided the scheduling table verifies these conditions, it is consistent and causal, and thus provides a correct intermediate representation of the final program, to be used for code generation. We will now see on an example how correct tables can be built using a very simple heuristic that ensures the correction of the produced table by construction.

4.2.2.3 Example

We consider once again the functional specification of Section 3.2.4 (cf. Page 53), and the simple bus-based architecture of Fig. 4.1. Our mapping heuristic works in two phases:

- Define a total order over the *functional* blocks of the specification. This total order must be compatible with the explicit data-flow dependencies, but must also take into account data dependencies due to clock computation. To ensure this, we impose that for any functional blocks b_1 and b_2, b_1 cannot be a predecessor of b_2 in the chosen total order if one of the outputs of b_2 is either:
  - in an input port of b_1,
  - in the support of $clk(b_1)$,
  - in the support of the clock of any clocked variable declared in the input ports of b_1.

  Variables that are outputs of delay blocks are considered available on any processor at any time. Thus, there is no need to consider delay blocks, or their variables when defining the total order. Multiple compatible total orders may exist for a given

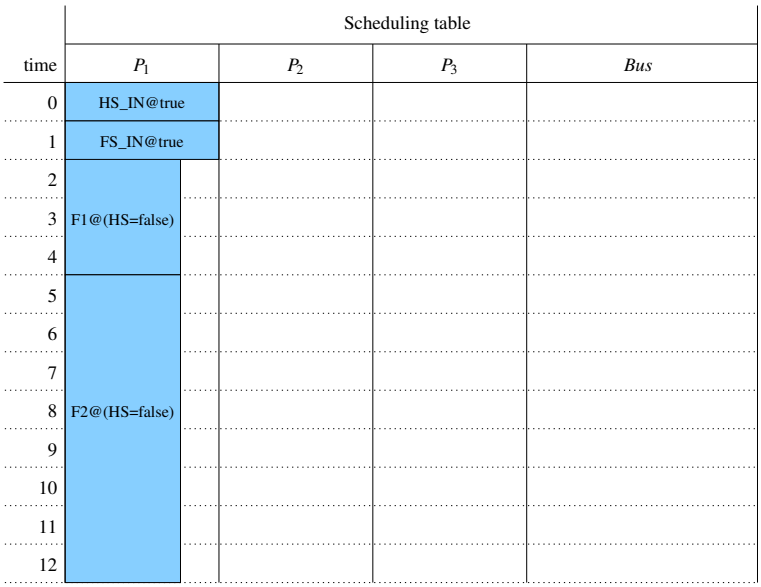


Figure 4.7: Temporary scheduling table after having scheduled HS_IN, FS_IN, $F1$ and $F2$ on P_1

specification, and all may not result in the same final schedule. In this case, we need to arbitrarily choose one. In our example, we will use the following mapping order: HS_IN, FS_IN, $F1$, $F2$, $F3$, G, N and M. Delays are separately mapped after all other blocks.

- Iteratively map all blocks of the application, following the order that we just defined. To do so, we start with an empty scheduling table, and map all the block one by one. At each step, we map a new block in the scheduling table that results from the previous step. The mapping decision is based on a simple cost function. For any processor that can execute the function of the considered block, we build a scheduling table in which the block is mapped on this processor. In each of these tables, and considering the data dependencies, we automatically schedule data communications when necessary, and schedule the block as soon as possible on the considered processor. Once this is done, we keep the scheduling table in which the end date of the reservation for the block is the smallest, and discard the other tables. Note that in order to keep this heuristic very fast, no backtracking is performed.

In our example, the first block to be scheduled is HS_IN, followed by FS_IN. In this particular architecture, we specified the fact that these operations could only be executed on P_1. Consequently we schedule HS_IN as soon as possible on P_1: as the initial scheduling table is empty, and block HS_IN has no input (and its clock support is empty), we can schedule it at date 0. The same happens for FS_IN, except that P_1 is already occupied from date 0 to date 1. FS_IN is thus scheduled from date 1 to date 2. Then, following the

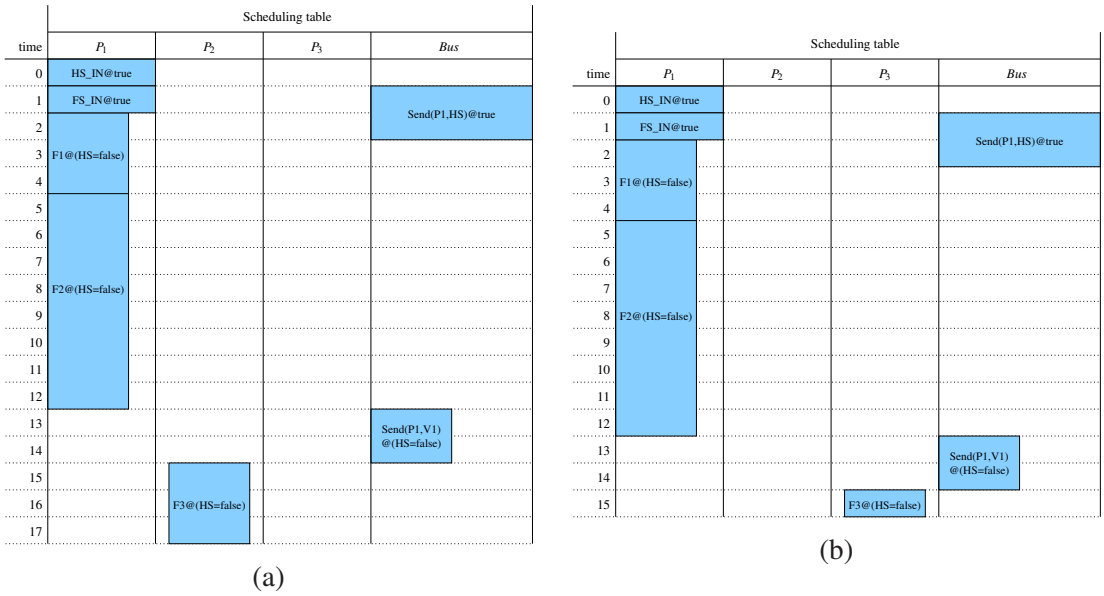


Figure 4.8: The two possibilities for mapping $F3$. Variant (b) is retained because it minimizes the end date of $F3$

order that we defined, we must schedule $F1$. Its input is created by a delay, so we consider that it is available at any time on any processor. $F1$ can only be executed on P_1 (the specification imposes it), and its clock depends only on HS which is also acquired on P_1 and thus made available at date 1. As a consequence, $F1$ can be scheduled from date 2 to date 5. Following the order, the algorithm will then schedule $F2$: the only input variable of this block is produced by $F1$ which is scheduled on P_1. Once again, the specification only allows $F2$ to be executed on this processor, so using the same reasoning as before, we schedule it on P_1 between dates 5 and 13. These decisions lead to the temporary scheduling table of Fig. 4.7. For the mapping of $F3$, the process is a bit harder since it involves a choice. $F3$ can be scheduled on P_2 or P_3, so we will have to try both variants, and select the one that minimizes its end date. Since the clock corresponding to the computation of this block is $HS = false$, we must make sure that HS is present on the processor that will compute $F3$. HS is acquired on P_1 from date 0 to 1, so its value must be sent on the bus to the corresponding processor. This communication is scheduled in both variants from date 1 to date 3. Moreover, $F3$ has one input: variable $V1$, which is produced by $F2$ on prcessor P_1. Once again it must be sent on the bus, under the condition that $HS = false$. For causality reasons, $V1$ cannot be sent on the bus before HS is available system-wide, that is to say at date 3. We can thus send it as soon as it has been produced, that is to say between dates 13 and 15 in both variants. Fig. 4.8 shows the scheduling tables corresponding to the two mapping variants. On the left, $F3$ has been mapped on P_2, where its computation takes 3 time units in the worst case. On the right, it has been mapped on P_3, where it is computed in 1 time unit in the worst case. The variant that will be retained

by the scheduler is thus the second one, because $F3$ finishes its computation at date 16, whereas in the first variant it finishes at date 18.

The algorithm then continues to map the remaining blocks in order, until they have all been scheduled, and we obtain the scheduling table of Fig. 4.4[4]. The corresponding textual table, described in the syntax that we provided before is displayed in Fig. 4.9. It comprises the schedule length, the implementation variables, and the schedules for each resource. An important thing to point out is that in the processor schedules, all references to variables are references to the variables of the specification file, whereas in the bus schedules, the conditions are also expressed with references to the specification variables, but the source and destinations of the communications are implementation variables. On the processors, a straightforward translation can be done, using the implementation variables table. On the bus, we needed to express the communication conditions in one way that could be easily translated for each processor, so we kept the conditions of the specification. At the same time, for code generation purposes, we needed to describe precisely which implementation variables are sent and what are their destinations, so we used direct references to the implementation variables.

4.3 Conclusion

In this chapter, we have presented the features of the CG language for modeling embedded systems execution platforms. Our models usually rely on a distributed architecture composed of several sequential processors linked together by a broadcast bus. The complexity of the processors and bus is abstracted by only characterizing the WCET or WCCT of the operations on the various resources where they are allowed to execute. From these architecture models, and the corresponding CG functional specifications, LoPhT is able to perform the mapping of the operations of the functional specification onto the resources of the architecture model. The result of this phase is described in a data structure called scheduling table, which contains all the necessary elements for code generation. From this intermediate representation, it is possible to target multiple execution models, depending on the support offered by the execution platform and by its operating system (or real-time kernel). In the next chapter, we will define a more general representation of these scheduling tables, that is general and precise enough to be used by the software pipelining algorithms of Chapter 6.

[4]We do not give here a formal definition of the scheduling algorithm nor do we discuss its complexity. This is due to the fact that this algorithm is not a contribution of this thesis (it already existed before I started working), and also to the fact that it is close to the high-level algorithm routine described and discussed in Section 8.2.2 (cf. Page 157)

```
Scheduling Table
ScheduleLength= 19

Implementation Variables
Variable:0 Type:0 Allocated On Processor:0 Implements Variable:0
Variable:1 Type:0 Allocated On Processor:2 Implements Variable:0
Variable:2 Type:0 Allocated On Processor:0 Implements Variable:1
Variable:3 Type:0 Allocated On Processor:1 Implements Variable:1
Variable:4 Type:0 Allocated On Processor:2 Implements Variable:1
Variable:5 Type:1 Allocated On Processor:0 Implements Variable:2
Variable:6 Type:1 Allocated On Processor:2 Implements Variable:2
Variable:7 Type:2 Allocated On Processor:0 Implements Variable:3
Variable:8 Type:2 Allocated On Processor:2 Implements Variable:3
Variable:9 Type:1 Allocated On Processor:1 Implements Variable:4
Variable:10 Type:1 Allocated On Processor:2 Implements Variable:4
Variable:11 Type:2 Allocated On Processor:2 Implements Variable:5
Variable:12 Type:2 Allocated On Processor:0 Implements Variable:5
Variable:13 Type:2 Allocated On Processor:0 Implements Variable:6

ProcSchedule(Processor:0)
SO:0 Condition Test(True) Delay StartDate 18 Duration 1 Block:8
     (del_i Is Variable:5 On Clock:2) -> (V2_del Is Variable:6)
SO:1 Condition Test(True) StartDate 0 Duration 1 Function:1
     () -> (HS Is Variable:1)
SO:2 Condition Test(True) StartDate 1 Duration 1 Function:0
     () -> (FS Is Variable:0)
SO:3 Condition Test(Not(Variable:1)) StartDate 2 Duration 3 Function:2
     (i Is Variable:6 On Clock:2) -> (ID Is Variable:2)
SO:4 Condition Test(Not(Variable:1)) StartDate 5 Duration 8 Function:3
     (ID Is Variable:2 On Clock:2) -> (V1 Is Variable:3)

ProcSchedule(Processor:1)

SO:5 Condition Test(Variable:1) StartDate 3 Duration 3 Function:5
     () -> (ID Is Variable:4)

ProcSchedule(Processor:2)
SO:6 Condition Test(Variable:0) StartDate 5 Duration 3 Function:7
     () -> ()
SO:7 Condition Test(Not(Variable:0)) StartDate 11 Duration 3 Function:6
     (ID Is Variable:2 On Clock:5 Variable:4 On Clock:6) -> ()
SO:8 Condition Test(Not(Variable:1)) StartDate 15 Duration 1 Function:4
     (V1 Is Variable:3 On Clock:2) -> (V2 Is Variable:5)

Bus Schedule(Bus:1)
SO:9 Condition Test(True) StartDate 1 Duration 2
     Source Variable:2 Destinations Variable:3 Variable:4
SO:10 Condition Test(True) StartDate 3 Duration 2
      Source Variable:0 Destinations Variable:1
SO:11 Condition Test(And(Not(Variable:1) Not(Variable:0))) StartDate 5 Duration 5
      Source Variable:5 Destinations Variable:6
SO:12 Condition Test(And(Variable:1 Not(Variable:0))) StartDate 6 Duration 5
      Source Variable:9 Destinations Variable:10
SO:13 Condition Test(Not(Variable:1)) StartDate 13 Duration 2
      Source Variable:7 Destinations Variable:8
SO:13 Condition Test(Not(Variable:1)) StartDate 16 Duration 2
      Source Variable:11 Destinations Variable:12
```

Figure 4.9: Textual description of the scheduling table of Fig. 4.4

Part II

Software pipelining of scheduling tables

Chapter 5

Extensions of the basic formalism

Contents

We have already explained in the introduction of Chapter 4 that defining a good platform model is a complex task that must take into account many parameters, such as the structure of the hardware platform, the performance and predictability objectives, *etc.* While in Chapter 4 we have defined the basic resource description formalism of LoPhT, we are now at the point where we will explore how changes in the platform modeling objectives require changes in the platform model itself to allow correct and efficient mapping.

More precisely, we shall define in this section here two variants of the previously-defined platform model. The first one explicitly represents memory allocation. This is required to allow the modeling of memory usage during software pipelining (in Chapter 6). This model can be seen as a refinement of the previously-defined platform model, because it allows the representation of bus-based systems (but with greater detail). This is the platform model that will be used in the remainder of the thesis.

The second variant, which is only sketched in Appendix A, targets quite different execution platforms: many-core with a network-on-chip interconnect. Interesting aspects of this platform model are the abstraction of multiple processors as single execution resources, the modeling of the network-on-chip, and the structure of the generated executable code.

This section does not include a significant extension of the resource reservation model, which is needed to represent preemptive time-triggered partitioned systems, and which

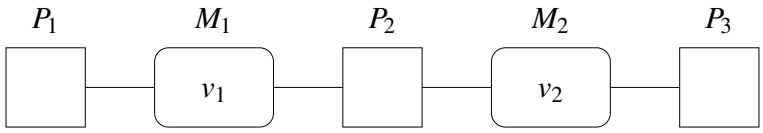


Figure 5.1: Simple architecture

will be defined in Chapter 8.

While our work on platform modeling has put in evidence general platform modeling principles, my PhD work did not produce a general model for platform modeling for off-line scheduling. This will be the subject of future work.

5.1 Memory representation for pipelining

5.1.1 Architecture model

For our software pipelining algorithms, we model execution architectures using a very simple language defining *sequential execution resources*, *memory blocks*, and their *interconnections*. Formally, an architecture model is a bipartite undirected graph $\mathcal{A} =< \mathcal{P}, \mathcal{M}, \mathcal{C} >$, with $\mathcal{C} \subseteq \mathcal{P} \times \mathcal{M}$. The elements of $\mathcal{P}$ are called *processors*, but they model all the computation and communication devices capable of independent execution (CPU cores, accelerators, DMA and bus controllers, etc.). We assume that each processor can execute only one operation at a time. We also assume that each processor has its own sequential or time-triggered program. This last assumption is natural on actual CPU cores. On devices such as DMAs and accelerators, it models the assumption that the cost of control by some other processor is negligible.

The elements of $\mathcal{M}$ are RAM blocks. We assume each RAM block is structured as a set of disjoint *cells*. We denote with *Cells* the set of all memory cells in the system, and with $Cells_M$ the set of cells on RAM block M. Our model does not specify memory size limits. To limit memory usage in the pipelined code we rely instead on the mechanism detailed in Section 6.4.1.

The elements of $\mathcal{C}$ are the interconnections. Processor P has direct access to memory block M whenever $(P,M) \in \mathcal{C}$. All processors directly connected to a memory block M can access M at the same time. Therefore, care must be taken to prohibit concurrent read-write or write-write access by two or more processors to a single memory cell, in order to preserve functional determinism (we will assume this is ensured by the input system model, and will be preserved by the pipelined one).

The simple architecture of Fig. 5.1 has 3 processors (P_1, P_2, and P_3) and 2 memory blocks (M_1 and M_2). Each of the M_i blocks has only one memory cell v_i.

5.1.2 Implementation model

On such architectures, we execute time-triggered implementations of embedded control applications with a *periodic non-preemptive* execution model. Once again, we represent such an application with a scheduling/reservation table.This pattern defines the computation of one period (also called an *execution cycle*). The infinite execution of the embedded system is the infinite succession of periodically-triggered execution cycles.

Formally, a reservation/scheduling table is a triple $\mathcal{S} =< p, \mathcal{O}, Init >$, where p is the *activation period* of execution cycles, $\mathcal{O}$ is the set of *scheduled operations*, and *Init* is the *initial state* of the memory.

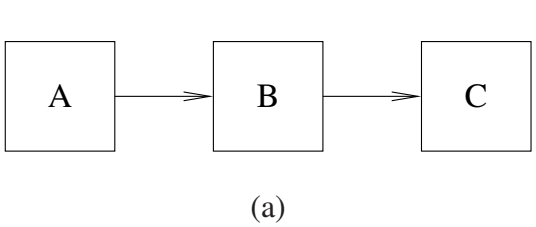
The activation period gives the (fixed) duration of the execution cycles. All the operations of one execution cycle must be completed before the following execution cycle starts. The activation period thus sets the *length* of the scheduling/reservation table, and is denoted by $len(\mathcal{S})$.

The set $\mathcal{O}$ defines the operations of the scheduling table. Each scheduled operation $o \in \mathcal{O}$ is a tuple defining:

- $In(o) \subseteq Cells$ is the set of memory cells read by o.
- $Out(o) \subseteq Cells$ is the set of cells written by o.
- $Guard(o)$ is the execution condition of o, defined as a predicate over the values of memory cells. We denote with $GuardIn(o)$ the set of memory cells used in the computation of $Guard(o)$.
- $Res(o) \subseteq \mathcal{P}$ is the set of processors used during the execution of o.
- $t(o)$ is the start date of o.
- $d(o)$ is the duration of o. The duration is viewed here as a time budget the operation must not exceed. As in the original model, this can be statically ensured through a worst-case execution time analysis.

All the resources of $Res(o)$ are exclusively used by o after $t(o)$ and for a duration of $d(o)$ in cycles where $Guard(o)$ is true. The sets $In(o)$ and $Out(o)$ are not necessarily disjoint, to model variables that are both read and updated by an operation. For lifetime analysis purposes, we assume that input and output cells are used for all the duration of the operation. The cells of $GuardIn(o)$ are all read at the beginning of the operation, but we assume the duration of the computation of the guard is negligible (zero time)[1].

[1]The memory access model where an operation reads its inputs at start time, writes its outputs upon completion, and where guard computations take time can be represented on top of our model.



(a)

	Processor		
time	P1	P2	P3
0	A@true		
1		B@true	
2			C@true

(b)

Figure 5.2: Simple specification in the abstract model Section 3.4 (a) and (non-pipelined) scheduling table for this specification (b)

To cover cases where a memory cell is used by one operation before being updated by another, each memory cell can have an *initial value*. For a memory cell m, $Init(m)$ is either *nil*, or some constant.

5.1.3 A simple example

To exemplify, we consider the simple data-flow synchronous specification of Fig. 5.2(a), which we map onto the architecture of Fig. 5.1. This example is described in the graphical formalism corresponding to the abstract model of Section 3.4. Depending on the non-functional requirements given as input to the scheduling tool of Phase 1 (allocation constraints, WCETs, *etc.*) one possible result is the scheduling table pictured in Fig. 5.2. We assumed here that A is must be mapped onto P_1 (*e.g.* because it uses a sensor peripheral connected to P_1), that B must be mapped onto P_2, that C must be mapped onto P_3 and that A, B, and C have all duration 1.

This table has a length of 3 and contains the 3 operations of the data-flow specification (A, B, and C). Operation A reads no memory cell, but writes v_1, so that $In(A) = \emptyset$ and $Out(A) = \{v_1\}$. Similarly, $In(B) = \{v_1\}$, $Out(B) = In(C) = \{v_2\}$, and $Out(C) = \emptyset$. All 3 operations are executed at every cycle, so their guard is *true* (guards are graphically represented with "@true"). The 3 operations are each allocated on one processor: $Res(A) = \{P_1\}$, $Res(B) = \{P_2\}$, $Res(C) = \{P_3\}$. Finally, $t(A) = 0$, $t(B) = 1$, $t(C) = 2$, and $d(A) = d(B) = d(C) = 1$. No initialization of the memory cells is needed (the initial values are all *nil*).

5.1.4 Well-formed properties

This abstract formalism provides the syntax of the implementation models that we use as input to our pipelining algorithms, and allows the definition of operational semantics. However as we saw in Section 4.2.2, not all syntactically correct specifications model correct implementations. Some of them are non-deterministic due to data races or due to operations exceeding their time budgets. Others are simply un-implementable, for instance because an operation is scheduled on processor P, but accesses memory cells on

a RAM block not connected to P.

We already provided in Section 4.2.2.2 the set of properties defining the correctness and implementability of the non-pipelined scheduling tables. These properties must still be satisfied by our extended tables. In addition, explicitly representing memory allocation means that we have to define the correctness properties concerning data locality and data races. These properties are the following:

- **Data locality.** The memory cells needed for the evaluation of guards, or because they correspond to inputs or outputs of an operation must be located in RAM blocks that are directly connected to the processor that will perform the operation,
- **No data races.** If some memory cell m is written by o_1 ($m \in Out(o_1)$) and is used by o_2 ($m \in In(o_2) \cup Out(o_2)$), then $Guard(o_1) \wedge Guard(o_2) \neq false \implies t(o_1) + d(o_1) \geq t(o_2) \vee t(o_2) + d(o_2) \geq t(o_1)$. In other words, if the operations guards are not exclusive, then the operations executions cannot overlap in time.

5.2 Conclusion

In this chapter, we defined an abstract and more general representation of the scheduling tables that we introduced in Chapter 4. This model is well-suited for the description of the software pipelining algorithms that we will define in Chapter 6.

Chapter 6

Throughput optimization by software pipelining of conditional reservation tables

Contents

One of the main contribution of my thesis is to take inspiration from classical compilation techniques, such as software pipelining, in order to improve the system-level task scheduling of specific classes of embedded systems. In this chapter we show how software pipelining can be used to improve the throughput of existing scheduling tables. Applying software pipelining as a stand-alone optimization phase has two main advantages: it can be used in conjunction with existing, state-of-the-art real-time scheduling techniques (used to build the initial scheduling table), and it allows us to focus on the differences between the ways software pipelining is used in compilation and in system-level real-time

scheduling. In particular, it allowed us to define the extensions needed to the scheduling table representations defined in the previous chapter to allow pipelining. Therefore, this chapter is important by itself, through its optimization results, but also as a basis for the work of Chapter 8, where pipelining is performed at the same time as the scheduling.

Software pipelining Compilers are expected to improve code speed by taking advantage of micro-architectural instruction level parallelism [Hennessy and Patterson, 2007]. *Pipelining* compilers usually rely on *reservation tables* to represent an efficient (possibly optimal) static allocation of the computing resources (execution units and/or registers) with a timing precision equal to that of the hardware clock. Executable code is then generated that enforces this allocation, possibly with some timing flexibility. But on *VLIW architectures*, where each instruction word may start several operations, this flexibility is very limited, and generated code is virtually identical to the reservation table. The scheduling burden is mostly supported here by the compilers, which include *software pipelining* techniques [Rau and Glaeser, 1981, Allan et al., 1995] designed to increase the throughput of loops by allowing one loop cycle to start before the completion of the previous one.

Static (offline) real-time scheduling A very similar picture can be seen in the system-level design of safety-critical real-time embedded control systems with *distributed (parallel, multi-core)* hardware platforms. The timing precision is here coarser, both for starting dates, which are typically given by timers, and for durations, which are characterized with WCETs. However, safety and efficiency arguments mentioned in the previous chapters [Fohler et al., 2008] lead to the increasing use of tightly synchronized *time-triggered* architectures and execution mechanisms, defined in well-established standards such as TTA, FlexRay[Rushby, 2001], ARINC653[ARINC], or AUTOSAR[AUTOSAR]. Systems based on these platforms typically have hard real-time constraints, and their correct functioning must be guaranteed by a schedulability analysis. As we saw in the previous chapters, we are interested here in statically scheduled systems where resource allocation is described with scheduling tables whose correctness ensures schedulability. Such systems include:

- Periodic time-triggered systems [Caspi et al., 2003, Zheng et al., 2005, Monot et al., 2010, Eles et al., 2000, Potop-Butucaru et al., 2010] that are naturally mapped over ARINC653, AUTOSAR, TTA, or FlexRay.

- Systems where the scheduling table describes the reaction to some sporadic input event (meaning that the table must fit inside the period of the sporadic event). Such systems can be specified in AUTOSAR, allowing, for instance, the modeling of computations synchronized with engine rotation events [André et al., 2007].

- Some systems with a mixed event-driven/time-driven execution model, such as those synthesized by SynDEx[Grandpierre and Sorel, 2003].

Synthesis of such systems starts from specifications written in domain-specific synchronous formalisms such as Simulink or SCADE[Caspi et al., 2003] which can be translated into the CG formalism of Chapter 3.

The problem The optimal scheduling of such specifications onto platforms with multiple, heterogenous execution and communication resources (distributed, parallel, multicore) is NP-hard regardless of the optimality criterion (throughput, makespan, etc.) [Garey and Johnson, 1979]. Existing scheduling techniques and tools [Caspi et al., 2003, Zheng et al., 2005, Grandpierre and Sorel, 2003, Potop-Butucaru et al., 2010, Eles et al., 2000] *heuristically* solve the simpler problem of synthesizing a scheduling table of *minimal length* which implements one generic cycle of the embedded control algorithm. The algorithm briefly presented in Section 4.2.2.3 follows this trend: it maps a CG specification to an architecture while trying to minimize the length of the obtained scheduling table. In a hard real-time context, minimizing table length (*i.e.* the makespan, as defined in the glossary of Fig. 6.2) is often a good idea, because in many applications it bounds the response time after a stimulus.

But optimizing makespan alone relies on an execution model where execution cycles cannot overlap in time (no pipelining is possible), even if resource availability allows it. At the same time, most real-time applications have both makespan and throughput requirements, and in some cases achieving the required throughput is only possible if a new execution cycle is started before the previous one has completed.

This is the case in the electronic control units (ECU) of combustion engines. Starting from the acquisition of data for a cylinder in one engine cycle, an ECU must compute the ignition parameters before the ignition point of the same cylinder in the next engine cycle (a makespan constraint). It must also initiate one such computation for each cylinder in each engine cycle (a throughput constraint). On modern multiprocessor ECUs, meeting both constraints requires the use of pipelining[André et al., 2007]. Another example is that of systems where a faster rate of sensor data acquisition results in better precision and improved control, but optimizing this rate must not lead to the non-respect of requirements on the latency between sensor acquisition and actuation. *To allow the scheduling of such systems we consider in this chapter the static scheduling problem of optimizing both makespan and throughput, with makespan being prioritary.*

Contribution To (heuristically) solve this optimization problem, we use a two-phase scheduling flow that can be seen as a form of *decomposed software pipelining* [Wang and Eisenbeis, 1993, Gasperoni and Schwiegelshohn, 1994, Calland et al., 1998]. As pictured

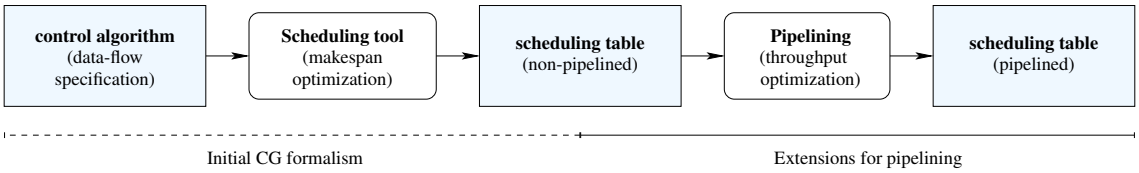


Figure 6.1: Proposed pipelined scheduling flow

in Fig. 6.1, the first phase of this flow consists in applying one of the previously-mentioned makespan-optimizing tools. The result is a scheduling table describing the execution of one generic execution cycle of the embedded control algorithm with no pipelining. In our evaluations, we will consider the output of the algorithm intuitively described in Chapter 4, which is implemented in LoPhT. Nevertheless, using our table-based representation, the results described in this chapter apply to any scheduling table obtained regardless of the tool or method used to obtain it.

The second phase uses a novel software pipelining algorithm, introduced in this chapter, to significantly improve the throughput without changing the makespan and while preserving the *periodic* nature of the system. The approach has the advantage of simplicity and generality, allowing the use of existing makespan-optimizing tools.

The proposed software pipelining algorithm is a very specific and constrained form of *modulo scheduling* [Rau, 1996]. Like all modulo scheduling algorithms, it determines a shorter *initiation interval* for the execution cycles (iterations) of the control algorithm, subject to resource and inter-cycle data dependency constraints. Unlike previous modulo scheduling algorithms, however, it starts from an already scheduled code (the non-pipelined scheduling table), and preserves all the intra-cycle scheduling decisions made at phase 1, in order to preserve the makespan unchanged. In other words, our algorithm computes the best *initiation interval* for the non-pipelined scheduling table and re-organizes resource reservations into a *pipelined scheduling table*, whose length is equal to the new initiation interval, and which accounts for the changes in memory allocation.

We have implemented our algorithm into a pipelining tool that is used, as we desired, in conjunction with LoPhT[1]. The resulting two-phase flow gives good results on architectures without temporal partitioning [ARINC], like the previously-mentioned AUTOSAR or SynDEx-generated applications and, to a certain extent, applications using the FlexRay dynamic segment.

For applications mapped onto partitioned architectures (ARINC 653, TTA, or FlexRay, the static segment) or where the non-functional specification includes multiple release date, end-to-end latency, or periodicity constraints, separating the implementation process in two phases (scheduling followed by pipelining) is not a good idea. We therefore

[1]Here we refer to the simple version of LoPhT corresponding to the algorithm of Chapter 4, that is to say without all the extensions of Chapter 7 and 8.

developed a single-phase pipelined scheduling technique documented elsewhere [Carle et al., 2012], but which uses (with good results) the same internal representation based on scheduling tables to allow a simple mapping of applications with execution modes onto heterogenous architectures with multiple processors and buses. This contribution is detailed in the following chapters.

Chapter outline The remainder of this chapter is organized as follows: an emphasis on related work is given, including the comparison of our originality points with the classical methods. Then we give an intuitive presentation of our technique using examples, followed by a precise description of our pipelining algorithms. We then introduce the necessary elements for the support of pipelining concerning the code generation process. Finally, we evaluate our method and conclude the chapter.

6.1 Related work and originality

This section reviews existing work and details the originality points related to this precise contribution. Performing this comparison required us to relate concepts and techniques belonging to two fields: software pipelining and real-time scheduling. To avoid ambiguities when the same notion has different names depending on the field, we define in Fig. 6.2 a glossary of terms that will be used throughout the paper.

6.1.1 Decomposed software pipelining

Closest to our work are previous results on *decomposed software pipelining* [Wang and Eisenbeis, 1993, Gasperoni and Schwiegelshohn, 1994, Calland et al., 1998]. In these papers, the software pipelining of a sequential loop is realized using two-phase heuristic approaches with good practical results. Two main approaches are proposed in these papers.

In the first approach, used in all 3 cited papers, the first phase consists in solving the loop scheduling problem while ignoring resource constraints. As noted in [Calland et al., 1998], existing decomposed software pipelining approaches solve this loop scheduling problem by using *retiming* algorithms. Retiming [Leiserson and Saxe, 1991] can therefore be seen as a very specialized form of pipelining targeted at cyclic (synchronous) systems where each operation has its own execution unit. Retiming has significant restrictions when compared with full-fledged software pipelining:

- It is oblivious of resource allocation. As a side-effect, it cannot take into account execution conditions to improve allocation, being defined in a purely data-flow context.

Concept	Description
scheduling table	These are essentially modulo reservation tables [Lam, 1988], extended to allow the representation of conditional/-multiple reservations. The scheduling tables of this chapter are described in the abstract model defined in Section 5.1.2.
initiation interval	The length of a schedule table is also called its initiation interval (II).
execution cycle	One iteration of either a scheduling table or the control algorithm before scheduling.
non-pipelined vs. pipelined	In classical software pipelining, reservation tables are used to represent the pipelined schedule. In our case, a reservation table is also used to represent the input of the pipelining algorithm. To avoid ambiguity, uses of "scheduling table", "initiation interval", "throughput", *etc.* will be qualified with "non-pipelined" or "pipelined" whenever necessary. For instance, the pipelined scheduling table is usually known in software pipelining as the *kernel*.
makespan	Worst-case duration of one execution cycle of the control algorithm, from the start of its first operation, to the end of its last operation. In our approach, by construction, it is the same in both the non-pipelined and pipelined scheduling tables. It is equal to the initiation interval of the non-pipelined scheduling table.
throughput	The number of execution cycles of the control algorithm executed per time unit. It is defined as the inverse of the initiation interval.
inter-cycle dependency	Data dependencies between operations of different execution cycles of the non-pipelined scheduling table.
scheduled operation	Complex data structure defining a resource reservation in a scheduling table (a time interval on one or several resources), but providing at the same time information on the operation (real-time task) to be executed inside this reservation (input and output variables, execution condition).

Figure 6.2: Glossary of terms used in this chapter. All notions are formally defined later

- It requires that the execution cycles of the system do not overlap in time, so that one operation must be completely executed inside the cycle where it was started.

Retiming can only change the execution order of the operations inside an execution cycle. A typical retiming transformation is to move one operation from the end to the beginning of the execution cycle in order to shorten the duration (critical path) of the execution cycle, and thus improve system throughput. The transformation cannot decrease the makespan but may increase it.

Once retiming is done, the second transformation phase takes into account resource constraints. To do so, it considers the acyclic code of one generic execution cycle (after retiming). A list scheduling technique ignoring inter-cycle dependences is used to map this acyclic code (which is actually represented with a *directed acyclic graph*, or *DAG*) over the available resources.

The second technique for decomposed software pipelining, presented in [Wang and Eisenbeis, 1993], basically switches the two phases presented above. Resource constraints are considered here in the first phase, through the same technique used above: list scheduling of DAGs. The DAG used as input is obtained from the cyclic loop specification by preserving only some of the data dependences. This scheduling phase decides the resource allocation and the operation order inside an execution cycle. The second phase takes into account the data dependences that were discarded in the first phase. It basically determines the fastest way a specification-level execution cycle can be executed by several successive pipelined execution cycles without changing the operation scheduling determined in phase 1 (preserving the throughput unchanged). Minimizing the makespan is important here because it results in a minimization of the memory/register use.

6.1.2 Originality

The objective of this chapter is to present a third decomposed software pipelining technique with two significant originality points, detailed below.

6.1.2.1 Optimization of both makespan and throughput

Existing software pipelining techniques are tailored for optimizing only one real-time performance metric: the processing *throughput* of loops [Yun et al., 2003] (sometimes besides other criteria such as register usage [Govindarajan et al., 1994, Zalamea et al., 2004, Huff, 1993] or code size [Zhuge et al., 2002]). In addition to throughput, we also seek to optimize *makespan*, with makespan being prioritary. Recall that throughput and latency (makespan) are antagonistic optimization objectives during scheduling [Benoît et al., 2007], meaning that resulting schedules can be quite different (an example will be provided in Section 6.2.1.2).

To optimize makespan we employ in the first phase of our approach existing scheduling techniques that were specifically designed for this purpose [Caspi et al., 2003, Zheng et al., 2005, Grandpierre and Sorel, 2003, Potop-Butucaru et al., 2010, Eles et al., 2000], such as the one used to create the scheduling table of Section 4.2.2.3. But the contribution described in this chapter concerns the second phase of our flow, which takes the scheduling table computed in phase 1 and optimizes its throughput while keeping its makespan unchanged. This is done using a new algorithm that conserves all the allocation and intra-cycle scheduling choices made in phase 1 (thus conserving makespan guarantees), but allowing the optimization of the throughput by increasing (if possible) the frequency with which execution cycles are started.

Like retiming, this transformation is a very restricted form of modulo scheduling software pipelining. In our case, it can only change the initiation interval (changes in memory allocation and in the scheduling table are only consequences). By comparison, classical software pipelining algorithms, such as the iterative modulo scheduling of [Rau, 1996], perform a full mapping of the code involving both execution unit allocation and scheduling. Our choice of transformation is motivated by three factors:

- It preserves makespan guarantees.
- It gives good practical results for throughput optimization.
- It has low complexity.

It is important to note that our transformation is not a form of retiming. Indeed, it allows for a given operation to span over several cycles of the pipelined implementation, and it can take advantage of conditional execution to improve pipelining, whereas retiming techniques work in a pure data-flow context, without predication (an example will be provided in Section 6.2.1.2).

6.1.2.2 Predication

For an efficient mapping of our conditional specifications, it is important to allow an independent, predicated (conditional) control of the various computing resources. However, most existing techniques for software pipelining [Allan et al., 1995, Warter et al., 1993, Yun et al., 2003] use hardware models that significantly constrain or simply prohibit predicated resource control. This is due to limitations in the target hardware itself. One common problem is that two different operations cannot be scheduled at the same date on a given resource (functional unit), even if they have exclusive predicates (like the branches of a test). The only exception we know to this rule is *predicate-aware scheduling (PAS)* [Smelyanskyi et al., 2003].

By comparison, the computing resources of our target architectures are not a mere functional units of a CPU (as in classical predicated pipelining), but full-fledged processors such as PowerPC, ARM, *etc.* The operations executed by these computing resources are large sequential functions, and not simple CPU instructions. Thus, each computing resource allows unrestricted predication control by means of conditional instructions, and the timing overhead of predicated control is negligible with respect to the duration of the operations. This means that our architectures satisfy the PAS requirements. The drawback of PAS is that sharing the same resource at the same date is only possible for operations of the same cycle, due to limitations in the dependency analysis phase. Our technique removes this limitation (an example will be provided in Section 6.2.1.3).

To exploit the full predicated control of our platform we rely on a new intermediate representation, namely *predicated and pipelined scheduling tables*. By comparison to the modulo reservation tables of [Lam, 1988, Rau, 1996], our scheduling tables allow the explicit representation of the execution conditions (predicates) of the operations, as we saw in Chapter 4. In turn, this allows the double reservation of a given resource by two operations with exclusive predicates.

6.1.3 Other aspects

A significant amount of work exists on applying software pipelining or retiming techniques for the efficient scheduling of tasks onto coarser-grain architectures, such as multiprocessors [Kim et al., 2012, Yang and Ha, 2009, Chatha and Vemuri, 2002, Chiu et al., 2011, Caspi et al., 2003, Morel, 2005]. To our best knowledge, these results share the two fundamental limitations of other software pipelining algorithms: optimizing for only one real-time metric (throughput) and not fully taking advantage of conditional execution to allow double allocation of resources.

Minor originality points of our technique, concerning code generation and dependency analysis will be discussed and compared with previous work in Sections 6.3.2 and 6.4.1.

6.2 Pipelining technique overview

6.2.1 Representing a pipelined scheduling table

6.2.1.1 A simple example

Recall the example of Fig. 5.2. For this small application, an execution where successive cycles do not overlap in time is clearly sub-optimal. Our objective is to allow the pipelined execution of Fig. 6.3, which ensures a maximal use of the computing resources.

In the pipelined execution, a new instance of operation A starts as soon as the previous one has completed, and the same is true for B and C. The first two time units of the

time	P1	P2	P3	
0	A@true iteration 1			Prologue
1	A@true iteration 2	B@true iteration 1		
2	A@true iteration 3	B@true iteration 2	C@true iteration 1	Steady state
3	A@true iteration 4	B@true iteration 3	C@true iteration 2	
	...	...	...	

Figure 6.3: Pipelined execution trace for the example of Fig. 5.2

time	P1	P2	P3
0	A@true $fst(A) = 0$	B@true $fst(B) = 1$	C@true $fst(C) = 2$

Figure 6.4: Pipelined scheduling table (kernel) for the example of Fig. 5.2

execution are the *prologue* which fills the pipeline. In the *steady state* the pipeline is full and has a throughput of one computation cycle (of the non-pipelined system) per time unit. If the system is allowed to terminate, then completion is realized by the *epilogue*, not pictured in our example, which empties the pipeline.

We represent the pipelined system schedule using the *pipelined scheduling table* pictured in Fig. 6.4. Its length is 1, corresponding to the throughput of the pipelined system. The operation set contains the same operations A, B, and C, but there are significant changes. The start dates of B and C are now 0, as the 3 operations are started at the same time in each pipelined execution cycle. A non-pipelined execution cycle spans over several pipelined cycles, and each pipelined cycle starts one non-pipelined cycle.

To account for the prologue phase, where operations progressively start to execute, each operation is assigned a *start index* $fst(o)$. If an operation o has $fst(o) = n$, it will first be executed in the pipelined cycle of index n (indices start at 0). Due to pipelining, the instance of o executed in the pipelined cycle m belongs to the non-pipelined cycle of index $m - fst(o)$. For instance, operation C with $fst(C) = 2$ is first executed in the 3rd pipelined cycle (of index 2), but belongs to the first non-pipelined cycle (of index 0).

The textual representation of the corresponding schedules (inside the scheduling table) is as follows:

```
ProcSchedule(Processor:0)
SO:0 Condition Test(True) StartDate 0 Duration 1 Block:0
     () -> (o Is Variable:0) Fst 0

ProcSchedule(Processor:1)
SO:1 Condition Test(True) StartDate 0 Duration 1 Block:1
     (i Is Variable:0 On Clock:0) -> (o Is Variable:1) Fst 1
```

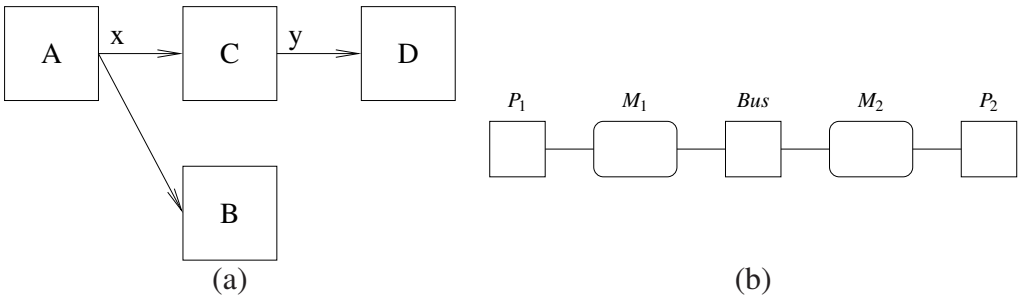


Figure 6.5: Example 2: Dataflow specification (a) and a bus-based implementation architecture (b)

```
ProcSchedule(Processor:2)
SO:2 Condition Test(True) StartDate 0 Duration 1 Block:2
     (i Is Variable:1 On Clock:0) -> () Fst 2
```

where Block:0 corresponds to function A, Block:1 to function B and Block:2 to function C. It includes a new syntactic element which accounts for the start index fst.

6.2.1.2 Example 2: makespan vs. throughput optimization

The example of Fig. 6.5 showcases how the different optimization objectives of our technique lead to different scheduling results, when compared with existing software pipelining techniques.

The architecture is here more complex, involving a *communication bus* that connects the two identical processors. Communications over the bus are synthesized during the scheduling process, as needed. Note that the bus is modeled as a processor performing communication operations. This level of description, defined in Section 5.1.1, is accurate enough to support our pipelining algorithms.

The functional specification is also more complex, involving parallelism and different durations for the computation and communication operations. The durations of A, B, C, and D on the two processors are respectively 1, 2, 4, and 1, and transmitting over the bus any of the data produced by A or C takes 1 time unit.

Fig. 6.6(a) provides the non-pipelined scheduling table produced for this example by the makespan-optimizing heuristic of Chapter 4 (and inspired by [Potop-Butucaru et al., 2009]). Operations A and B have been allocated on processor $P1$ and operations C and D have been allocated on $P2$. One communication is needed to transmit data x from $P1$ to $P2$. The makespan is here equal to the table length, which is 7. The throughput is the inverse of the makespan (1/7).

When this scheduling table is given to our pipelining algorithm, the output is the pipelined scheduling table of Fig. 6.6(b). The makespan remains unchanged (7), but the table length is now 5, so the throughput is 1/5. Note that the execution of operation C

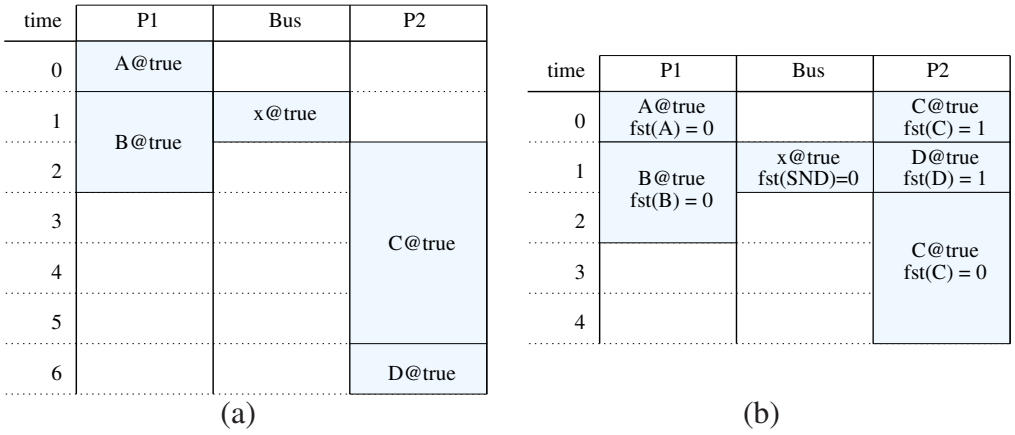


Figure 6.6: Example 2: Non-pipelined scheduling table produced by phase 1 of our technique (a) and pipelined table produced by phase 2 (b) for the example of Fig. 6.5

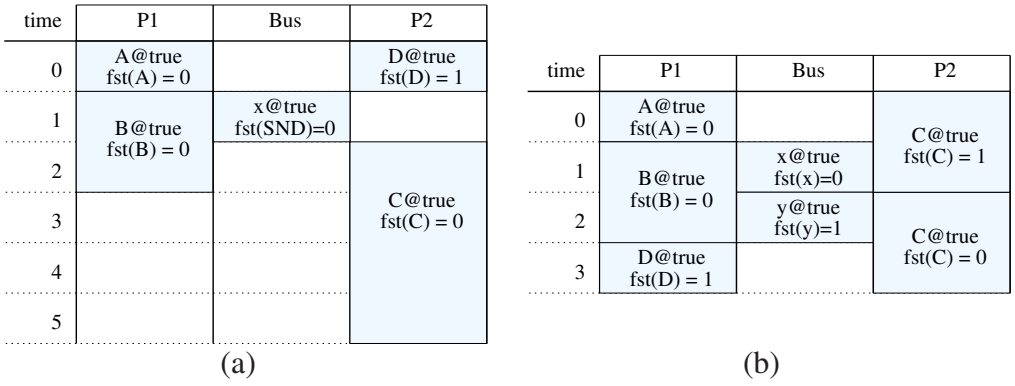


Figure 6.7: Example 2: At left, the result of retiming the scheduling table of Fig. 6.6(a). At right, the result of directly applying throughput-optimizing modulo scheduling onto the specification of Fig. 6.5.

starts in one pipelined execution cycle (at date 2), but ends in the next, at date 1. Thus, operation C has two reservations, one with $fst(C) = 0$, and one with $fst(C) = 1$.

Operations spanning over multiple execution cycles are not allowed in retiming-based techniques. Thus, if we apply retiming to the scheduling table of Fig. 6.6(a), we obtain the pipelined scheduling table of Fig. 6.7(a). For this example, the makespan is not changed, but the throughput is worse than the one produced by our technique (1/6).

But the most interesting comparison is between the output of our pipelining technique and the result of throughput-only optimization. Fig. 6.7(b) provides a pipelined scheduling table that has optimal throughput. In this table, operation D is executed by processor $P1$, so that the bus must perform 2 communications. The throughput is better than in our case (1/4 vs. 1/5), but the makespan is worse (8 vs. 7), even though we chose a schedule with the best makespan among those with optimal throughput.

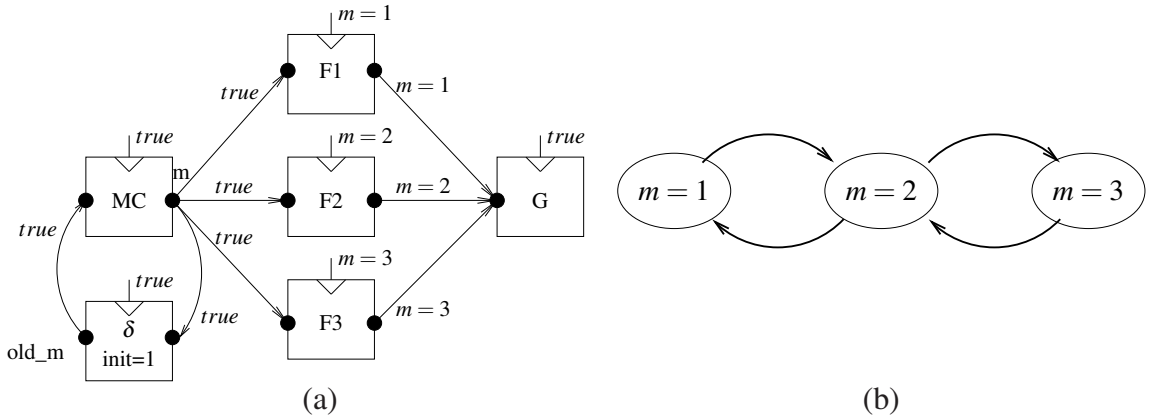


Figure 6.8: Example 3: Dataflow specification with conditional execution (a) and possible mode transitions (b)

6.2.1.3 Example 3: Predication handling

To explain how predication is handled in our approach, consider the example of Fig. 6.8. We only picture here the functional specification (described in the CG formalism). As architecture, we consider two processors $P1$ and $P2$ connected to a shared memory $M1$ (a 2-processor version of the architecture in Fig. 5.1).

This example features a delay, labeled δ. As explained in Chapter 3, delays are the only constructs in our data-flow formalism to represent the system state, and are a source of data dependences between successive execution instants. In this example, we shall assume that the duration of the bookkeeping operation associated with a delay is negligeable (when compared to the durations of the other computations). As a consequence, we will not represent it in the scheduling tables.

The functional specification also makes use of conditional (predicated) execution. Operations $F1$, $F2$, and $F3$ (of length 3, 2, and 1, respectively) are executed in execution cycles where the output m of operation MC equals respectively 1, 2, or 3. Operations MC and G are executed in all cycles. Note that our graphical representation uses, for clarity reasons, integer values for clock definitions (for tests). As explained in Chapter 3, our tool cannot analyze integer tests, which means that integer values and tests over them must be encoded using Boolean variables when writing the CG specification given as input to LoPhT (two variables, in our case).

The output m of operation MC (for *mode computation*) is gives the *execution mode* of our application. This mode is recomputed in the beginning of each execution cycle by MC, based on the previous mode and on unspecified inputs directly acquired by MC. The application has 3 possible modes (1, 2, and 3), and we assume that mode changes can only occur in the directions indicated by arrows in Fig. 6.8(b). This automaton specifies that a direct transition between modes 1 and 3 is forbidden. For instance, if in one execution

cycle $m = 1$, then in the next cycle m cannot have value 3 (only 1 or 2). Such constraints can be specified using the *ensures* clauses introduced in Section 3.2.1.3. Such a clause defines a predicate over the inputs and outputs of various functions and this clause is assumed true in all execution cycles. In our case, this predicate is defined using an *ensures* clause associated with the function of block *MC*. Assuming that the input port of operation *MC* is called *old_m* and that the output port is called *m*, the predicate associated to *MC* is:

$$\texttt{not}\ (m = 1\ \texttt{and}\ old\_m = 3)\ \texttt{and not}\ (m = 3\ \texttt{and}\ old\_m = 1)$$

To represent this predicate using in our CG specification, we shall use a Boolean encoding of *m* and *old_m* using 2 Boolean variables each: `m1` and `m2` for *m* and `om1` and `om2` for *old_m*. The encoding is as follows:

m	m1	m2
1	0	0
2	0	1
3	1	1

Under this encoding, the *ensures* clause associated with the function of *MC* is:

```
ensures And ((Not (And (Not m2) om1)) (Not (And m1 (Not om2))))
```

We assume that operations *MC*, $F1$, $F2$, and $F3$ are executed on processor $P1$, and that *G* is executed on $P2$. Under these conditions, one possible non-pipelined schedule produced by Phase 1 is the one pictured in Fig. 6.9. Note that this table features 3 conditional reservations for operation *G*, even if *G* does not have an execution condition in the data-flow graph. This allows *G* to start as early as possible in every given mode.

This table clearly features the reservation of the same resource, at the same time, by multiple operations. For instance, operations $F1$, $F2$, and $F3$ share $P1$ at date 1. Of course, each time this happens the operations must have exclusive predicates, meaning that there is no conflict at runtime, as we saw in Section 4.2.1.1.

Pipelining this table using the algorithms of the following sections produces the scheduling table of Fig. 6.10. The most interesting aspect of this table is that the reservations *G@m=1,fst=1* and *G@m=3,fst=0*, which belong to different execution cycles of the non-pipelined table, are allowed to overlap in time. This is possible because the dependency analysis of Section 6.3.2 will determine that the two operations have exclusive execution conditions. In our case, this is due to the fact that *m* cannot change its value directly from 1 to 3 when moving from one non-pipelined cycle to the next.

When relations between execution conditions of operations belonging to different execution cycles are not taken into account, the resulting pipelining is that of Fig. 6.11. Here, reservations for *G* cannot overlap in time if they have different *fst* values.

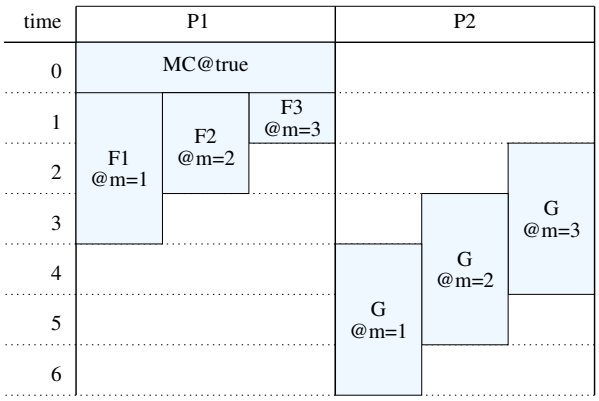


Figure 6.9: Example 3: non-pipelined scheduling table

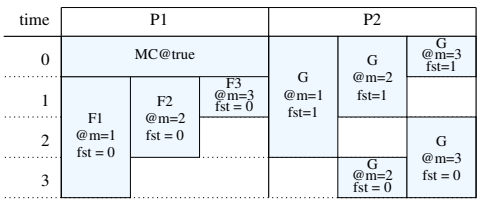


Figure 6.10: Example 3: Pipelined scheduling table produced by our technique

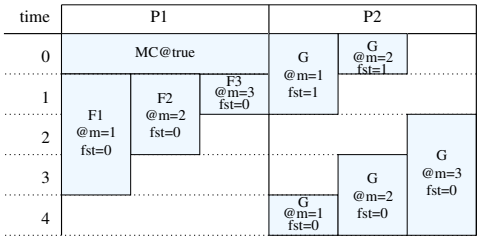


Figure 6.11: Example 3: Pipelined scheduling table where inter-cycle execution condition analysis has not been used to improve sharing.

6.2.1.4 Construction of the pipelined scheduling table

The prologues of our pipelined executions are obtained by incremental activation of the steady state operations, as specified by the *fst* indices (this is a classical feature of modulo scheduling pipelining approaches). Then, the pipelined scheduling table can be fully built using Algorithm 1 starting from the non-pipelined table and from the pipelined initiation interval. The algorithm first determines the start index and new start date of each operation by folding the non-pipelined table onto the new period. Algorithm *AssembleSchedule* then determines which memory cells need to be replicated due to pipelining, using the technique provided in Section 6.4.1.

Algorithm 1 BuildSchedule

Input: $\mathcal{S}$: non-pipelined scheduling table
$\widehat{p}$: pipelined initiation interval
Output: $\widehat{\mathcal{S}}$: pipelined schedule table
for all o in $\mathcal{O}$ **do**
$fst(o) := \lfloor \frac{t(o)}{\widehat{p}} \rfloor$
$\widehat{t}(o) := t(o) - fst(o) * \widehat{p}$
end for
$\widehat{\mathcal{S}} := AssembleSchedule(\mathcal{S}, \widehat{p}, fst, \widehat{t})$

6.3 Optimization algorithms

6.3.1 Dependency graph and maximal throughput

The period of the pipelined system is determined by the data dependences between successive execution cycles and by the resource constraints. If we follow the classification of [Hennessy and Patterson, 2007], we are interested here in true data dependences, and not in *name dependences* such as *anti-dependences* and *output dependences*. A true data dependency exists between two operations when one uses as input the value computed by the other. Name dependencies are related to the reuse of variables, and can be eliminated by variable renaming. For instance, consider the following C code fragment:

```
x := y + z; y := 10; z := y;
```

Here, there is a true data dependence (on variable `y`) between statements 2 and 3. There is also an anti-dependence (on variable `y`) between statements 1 and 2 (they cannot be re-ordered without changing the result of the execution). Renaming variable `y` in statements 2 and 3 removes this anti-dependence and allows re-ordering of statements 1 and 2:

```
x := y + z; y2:= 10; z := y2;
```

In our case, not needing the analysis of name dependences is due to the use of the rotating register files (detailed in Section 6.4.1) which remove anti-dependences, and to the fact that output dependences are not semantically meaningful in our systems whose execution is meant to be infinite.

We represent data dependences as a *Data Dependency Graph (DDG)* – a formalism that is classical in software pipelining based on *modulo scheduling* techniques[Allan et al., 1995]. In this section we define DDGs and we explain how the new period is computed from them. The computation of DDGs is detailed in Section 6.3.2.

Given a scheduling table $\mathcal{S} =< p, \mathcal{O}, Init >$, the DDG associated to $\mathcal{S}$ is a directed graph $DG =< \mathcal{O}, \mathcal{V} >$ where $\mathcal{V} \subseteq \mathcal{O} \times \mathcal{O} \times \mathbb{N}$. Ideally, $\mathcal{V}$ contains all the triples (o_1, o_2, n) such that there exists an execution of the scheduling table and a computation cycle k such that operation o_1 is executed in cycle k, operation o_2 is executed in cycle $k+n$, and o_2 uses a value produced by o_1. In practice, any $\mathcal{V}$ including all the arcs defined above (any over-approximation) will be acceptable, leading to correct (but possibly sub-optimal) pipelinings.

The DDG represents all possible dependences between operations, both inside a computation cycle (when $n = 0$) and between successive cycles at distance $n \geq 1$. Given the static scheduling approach, with fixed dates for each operation, the pipelined scheduling table must respect *unconditionally* all these dependences.

Note the strong similarity between the formalism used to represent DDGs and the non-conditioned dependent task systems of Definition 1 (Section 3.4). The only difference between them is that the arc sets of Definition 1 are fusioned here into a single set of arcs $\mathcal{V}$. This similarity shows the strong ties that exist between various techniques used in compilation and real-time scheduling. Note also the differences: DDGs cannot represent exclusiveness relations due to data dependent control (as they are defined in Definition 2) because they cannot be exploited by the algorithms using them.

For each operation $o \in \mathcal{O}$, we denote with $t_n(o)$ the date where operation o is executed in cycle n, if its guard is true. By construction, we have $t_n(o) = t(o) + n * p$. In the pipelined scheduling table of period $\widehat{p}$, this date is changed to $\widehat{t}_n(o) = t(o) + n * \widehat{p}$. Then, for all $(o_1, o_2, n) \in \mathcal{V}$ and $k \geq 0$, the pipelined scheduling table must satisfy $\widehat{t}_{k+n}(o_2) \geq \widehat{t}_k(o_1) + d(o_1)$, which implies:

$$\widehat{p} \geq \max_{(o_1,o_2,n)\in\mathcal{V}, n\neq 0} \left\lceil \frac{t(o_1) + d(o_1) - t(o_2)}{n} \right\rceil$$

Our objective is to build pipelined scheduling tables satisfying this lower bound constraint and which are well-formed in the sense of Section 5.1.4.

time	P1	P2	P3
0	A@true		
1		B@true	
2			C@true
3	D@true		

Figure 6.12: Dependency analysis example, non-pipelined

time	P1	P2	P3
0	A@true $fst(A) = 0$		C@true $fst(C) = 1$
1	D@true $fst(D) = 1$	B@true $fst(B) = 0$	

Figure 6.13: Dependency analysis example, pipelined

6.3.2 Dependency analysis and main routine

Dependency analysis is a mature discipline, and powerful algorithms have been used in practice for decades [Muchnick, 1997]. However, previous research on inter-iteration dependency analysis has mostly focused on exploiting the regularity of code such as affine loop nests. To the best of our knowledge, existing algorithms are unable to analyze specifications such as our Example 3 (Section 6.2.1.3) with the precision we seek. Doing this requires that inter-iteration dependency analysis deals with data-dependent mode changes (which are a common feature in embedded systems design).

Performing our precise inter-iteration dependency analysis requires the (potentially infinite) unrolling of the non-pipelined scheduling table. But our specific pipelining technique allows us to bound the unrolling, and thus limits the complexity of dependency analysis. By comparison, existing pipelining and predicate-aware scheduling techniques either assume that the dependency graph is fully generated before starting the pipelining algorithm [Rau and Glaeser, 1981], or use the predicates for the analysis of a single cycle[Warter et al., 1993].

The core of our dependency analysis consists in the lines 1-10 of Algorithm 3, which act as a driver for Algorithm 2. The remainder of Algorithm 3 uses DDG-derived information to drive the pipelining routine (Algorithm 1).

Both the data dependency analysis and pipelining driver take as input a flag that chooses between two pipelining modes with different complexities and capabilities. To understand the difference, consider the non-pipelined scheduling table of Fig. 6.12. Resource P_1 has an idle period between operations A and D where a new instance of A can be started. However, to preserve a periodic execution model, A should not be restarted just after its first instance (at date 1). Indeed, this would imply a pipelined throughput of 1, but the fourth instance of A cannot be started at date 3 (only at date 6). The correct pipelining starts A at date 2, and results in the pipelined scheduling table of Fig. 6.13.

Determining if the reuse of idle spaces between operations is possible consists in determining the smallest integer n greater than the lower bound of Section 6.3.1, smaller than the length of the initial table, and such that a *well-formed* pipelined table of length n can be constructed. This computation is performed by lines 14-19 of Algorithm 3. We

do not provide here the code of function *WellFormed*, which checks the respect of the well-formed properties detailed in Section 5.1.4.

This complex computation can be avoided when idle spaces between two operations are excluded from use at pipelining time. This can be done by creating a dependency between any two operations of successive cycles that use a same resource and have non-exclusive execution conditions. In this case, the pipelined system period is exactly the lower bound of Section 6.3.1, and the output scheduling table is produced with a single call to Algorithm 1 (*BuildSchedule*) in line 12 of Algorithm 3. Of course, Algorithm 2 needs to consider (in lines 10-16) the extra dependences.

Excluding the idle spaces from pipelining also has the advantage of supporting a sporadic execution model. In sporadic systems the successive computation cycles can be executed with the maximal throughput specified by the pipelined table, but can also be triggered arbitrarily less often, for instance to tolerate timing variations, or to minimize power consumption in systems where the demand for computing power varies. On the contrary, using the idle spaces during pipelining imposes synchronization constraints between successive execution cycles. For instance, in the pipelined system of Fig. 6.13, the computation cycle of index n cannot complete before operation A of cycle $n+1$ is completed.

The remainder of this section details the dependency analysis phase. The output of this analysis is the lower bound defined in Section 6.3.1, computed as *period_lbound*. The analysis is organized around the **repeat** loop which incrementally computes, for $cycle \geq 1$, the DDG dependences of the type $(o_1, o_2, cycle)$. The computation of the DDG is not complete: we bound it using a loop termination condition derived from our knowledge of the pipelining algorithm. This condition is based on the observation that if $period\_lbound * k \geq len(\mathcal{S})$ then execution cycles n and $n+k$ cannot overlap in time (for all n).

The DDG computation works by incrementally unrolling the non-pipelined scheduling table. At each unrolling step, the result is put in the SSA[2]-like data structure $\overline{\mathcal{S}}$ that allows the computation of (an over-approximation of) the dependency set. Unrolling is done by annotating each instance of an operation o with the cycle n in which it has been instantiated. The notation is o^n. Putting in SSA-like form is based on splitting each memory cell v into one version per operation instance producing it (v_o^n, if $v \in Out(o)$), and one version for the initial value (v_{init}). Annotation and variable splitting is done on a per-cycle basis by the *Annotate* routine (not provided here) which changes for each operation o its name to o^n, and replaces $Out(o)$ with $\{v_o^n \mid v \in Out(o)\}$ (n is here the cycle index parameter). The *Annotate* routine is also responsible for dealing with the ensures clauses that are declared with the functions. For any operation linked to a function containing an ensures

[2]SSA stands for Static Single Assignement representation form [Muchnick, 1997].

Algorithm 2 DependencyAnalysisStep

Inputs: $\mathcal{S}$: non-pipelined scheduling table
l : the list of events of $\mathcal{S}$
n : integer (cycle index)
fast_pipelining_flag : boolean
InputOutputs: $\overline{\mathcal{S}}$: annotated scheduling table
curr : current variable assignments
DDG : Data Dependency Graph

1: $\overline{\mathcal{S}} := Concat(\overline{\mathcal{S}}, Annotate(\mathcal{S}, n))$
2: **while** l not empty **do**
3: $e :=$ head(l) ; $l :=$ tail(l)
4: **if** $e = start(o)$ **then**
5: Replace $Guard(o^n)$ by $\bigvee_{w_i@C_i \in curr(v_i), i=\overline{1,k}} (C_1 \wedge \ldots \wedge C_k) \wedge g_o(w_1, \ldots, w_k)$
where $Guard(o) = g_o(v_1, \ldots, v_k)$.
6: **for all** p operation in $\mathcal{S}$, $u \in Out(p)$, $v \in In(o)$ **do**
7: **if** $u_p^0@C \in curr(v)$ and $\neg Exclusive(C, Guard(o^n))$ **then**
8: $DDG := DDG \cup \{(p, o, n)\}$
9: **end if**
10: **if** *fast_pipelining_flag* **then**
11: **if** $Res(o) \cap Res(p) \neq \emptyset$ **then**
12: **if** $\neg Exclusive(Guard(o^n), Guard(p^0))$ **then**
13: $DDG := DDG \cup \{(p, o, n)\}$
14: **end if**
15: **end if**
16: **end if**
17: **end for**
18: **else**
19: /* $e = end(o)$ */
20: **for all** $v \in Out(o)$ **do**
21: $new\_curr := \{v_o^n@Guard(o^n)\}$
22: **for all** $v_p^k@C \in curr(v)$ **do**
23: $C' := C \wedge \neg Guard(o^n)$
24: **if** $\neg Exclusive(C, Guard(o^n))$ **then**
25: $new\_curr := new\_curr \cup \{v_p^k@C'\}$
26: **end if**
27: **end for**
28: $curr(v) := new\_curr$
29: **end for**
30: **end if**
31: **end while**

clause, the routine maps the annotated variables corresponding to the inputs and outputs of the operation to their counterparts in the ensures formula. Instances of $\mathcal{S}$ produced by *Annotate* are then assembled into $\overline{\mathcal{S}}$ by the *Concat* function which simply adds to the date of every operation in the second argument the length of its first argument. During this assembly the ensures clauses coresponding to the operations of the new instance are concatenated to the already existing ensures clause corresponding to the previous instances. This is done using the conjunction operator between the old and new clauses.

Recall that we are only interested in dependences between operations in different cycles. Then, in each call to Algorithm 2 we determine the dependences between operations of cycle 0 and operations of cycle n, where n is the current cycle. To determine them, we rely on a symbolic execution of the newly-added part of $\overline{\mathcal{S}}$, *i.e.* the operations o^k with $k = n$. Symbolic execution is done through a traversal of list l, which contains all operation start and end events of $\mathcal{S}$, and therefore $\overline{\mathcal{S}}$, ordered by increasing date. For each operation o of $\mathcal{S}$, l contains two elements labeled $start(o)$ and $end(o)$. The list is ordered by increasing event date using the convention that the date of $start(o)$ is $t(o)$, and the date of $end(o)$ is $t(o) + d(o)$. Moreover, if $start(o)$ and $end(o')$ have the same date, the $start(o)$ event comes first in the list.

At each point of the symbolic execution, the data structure *curr* identifies the possible producers of each memory cell. For each cell v of the initial table, $curr(v)$ is a set of pairs $w@C$, where w is a version of v of the form v_o^k or v_{init}, and C is a predicate over memory cell versions. In the pair $w@C$, C gives the condition on which the value of v is the one corresponding to its version w at the considered point in the symbolic simulation. Intuitively, if $v_o^k@C \in curr(v)$, and we symbolically execute cycle n, then C gives the condition under which in any real execution of the system v holds the value produced by o, $n-k$ cycles before. The predicates of the elements in $curr(v)$ provide a partition of true. Initially, $curr(v)$ is set to $\{v_{init}@true\}$ for all v. This is changed by Algorithm 2 (lines 20-29), and by the call to *InitCurr* in Algorithm 3. We do not provide this last function, which performs the symbolic execution of the nodes of $\overline{\mathcal{S}}$ annotated with 0. Its code is virtually identical to that of Algorithm 2, lines 1 and 6-17 being excluded.

At each operation start step of the symbolic execution, *curr* allows us to complete the SSA transformation by recomputing the guard of the current operation over the split variables (line 5 of Algorithm 2). In turn, this allows the computation of the dependences (lines 6-17). Guard comparisons are translated into predicates that are analyzed by a SAT solver. This translation into predicates also considers the predicate holding all the information contained in the ensures clauses of the instanciated dataflow operations, as intuitively explained in Section 6.2.1.3. The translation and the call to SAT are realized by the *Exclusive* function, not provided here.

Algorithm 3 PipeliningDriver

Input: $\mathcal{S}$: non-pipelined schedule table
fast_pipelining_flag : boolean
Output: $\widehat{\mathcal{S}}$: pipelined schedule table

1: $l := BuildEventList(\mathcal{S})$
2: $period\_lbound := 0$
3: $cycle := 0$
4: $\overline{\mathcal{S}} := Annotate(\mathcal{S}, cycle)$
5: $curr := InitCurr(\overline{\mathcal{S}})$
6: **repeat**
7: $cycle := cycle + 1$
8: $(\overline{\mathcal{S}}, curr, DDG) := DependencyAnalysisStep(\mathcal{S}, l,$
$cycle, fast\_pipelining\_flag, \overline{\mathcal{S}}, curr, DDG)$
9: $period\_lbound := max(period\_lbound, \max_{(o_1,o_2,cycle)\in DDG} \lceil \frac{t(o_1)+d(o_1)-t(o_2)}{cycle} \rceil)$
10: **until** $period\_lbound * cycle \geq len(\mathcal{S})$
11: **if** *fast_pipelining_flag* **then**
12: $\widehat{\mathcal{S}} := BuildSchedule(\mathcal{S}, period\_lbound)$
13: **else**
14: **for** $new\_period := period\_lbound$ to $len(\mathcal{S})$ **do**
15: $\widehat{\mathcal{S}} := BuildSchedule(\mathcal{S}, new\_period)$
16: **if** $WellFormed(\widehat{\mathcal{S}})$ **then**
17: **goto** 21
18: **end if**
19: **end for**
20: **end if**
21: **return**

6.3.3 Complexity considerations

The pipelining algorithm *per se* consists in Algorithm 1 and its driver (lines 11-20 of Algorithm 3). The complexity of Algorithm 1 is linear in the number of operations in the scheduling table. As explained above, the complexity of the driver routine (lines 11-20 of Algorithm 3) depends on the value of *fast_pipelining_flag*. When it is set to *true*, a single call to Algorithm 1 is performed. When it is set to *false*, the number of calls to Algorithm 1 is bounded by $len(\mathcal{S})$.

In our experiments, *fast_pipelining_flag* is set for all examples, and the pipelining time is negligible.

But the main source of theoretical complexity in our pipelining technique is hidden in the dependency analysis implemented by Algorithm 2 and its driver (lines 1-10 of Algorithm 3). In Algorithm 3, the number of iterations in the construction of the DDG is up-bounded by $len(\mathcal{S})$ (which can be large). Algorithm 2 is polynomial (quadratic) in the number of operations of the scheduling table, but involves comparisons of predicates with Boolean arguments (instances of the Boolean satisfiability problem SAT). Hence, our algorithm is overall NP-complete.

In practice, however, DDG construction time is negligible for all examples, real-life and synthesized. There are 2 reasons to this:

- In examples featuring predicated execution, the predicates remain simple, so that SAT instances are solved in negligible time. This might not be the case for more complex specifications, but then we also expect that better SAT solving engines bring matching gains (for simplicity reasons, we currently use the very simple SAT engine described in [Conchon and Lescuyer, 2008]).

- The number of iterations in the construction of the DDG is bounded by the condition in line 10 of Algorithm 3. In our experiments the maximal number of iterations is 2.

6.4 Code generation

6.4.1 Memory management issues

Our pipelining technique allows multiple instances of a given variable, belonging to successive non-pipelined execution cycles, to be simultaneously live. For instance, in the example of Figures 5.2 and 6.3 both A and B use memory cell v_1 at each cycle. In the pipelined table, A and B work in parallel, so they must use two different copies of v_1. In other words, we must provide an implementation of the *expanded virtual registers* of [Rau, 1996].

The traditional software solution to this problem is the modulo variable expansion of [Lam, 1988]. However, this solution requires loop unrolling, which would increase the size of our scheduling tables. Instead, we rely on a purely software implementation of *rotating register files* [Rau et al., 1992], which requires no loop unrolling, nor special hardware support. Our implementation of rotating register files includes an extension for identifying the good register to read in the presence of predication. To our best knowledge, this extension (detailed in Section 6.4.1.2) is an original contribution.

6.4.1.1 Rotating register files for stateless systems

Assuming that $\widehat{\mathcal{S}}$ is the pipelined version of $\mathcal{S}$, we denote with $max\_par = \lceil len(\mathcal{S})/len(\widehat{\mathcal{S}}) \rceil$ the maximal number of simultaneously-active computation cycles of the pipelined scheduling table. Note that $max\_par = 1 + \max_{o \in \mathcal{O}} fst(o)$.

In the example of Figures 5.2 and 6.3 we must use two different copies of v_1. We will say that the replication factor of v_1 is $rep(v_1) = 2$. Each memory cell v is assigned its own replication factor, which must allow concurrent non-pipelined execution cycles using different copies of v to work without interference. Obviously, we can bound $rep(v)$ by max_par. We use a tighter margin, based on the observation that most variables (memory cells) have a limited lifetime inside a non-pipelined execution cycle. We set $rep(v) = 1 + lst(v) - fst(v)$, where:

$$fst(v) = \min_{v \in In(o) \cup Out(o)} fst(o) \qquad lst(v) = \max_{v \in In(o) \cup Out(o)} fst(o)$$

Through replication, each memory cell v of the non-pipelined scheduling table is replaced by $rep(v)$ memory cells, allocated on the same memory block as v, and organized in an array $\overline{v}$, whose elements are $\overline{v}[0], \ldots, \overline{v}[rep(v) - 1]$. These new memory cells are allocated cyclically, in a static fashion, to the successive non-pipelined cycles. More precisely, the non-pipelined cycle of index n is assigned the replicas $\overline{v}[n \bmod rep(v)]$ for all v. The computation of $rep(v)$ ensures that if n_1 and n_2 are equal modulo $rep(v)$, but $n_1 \neq n_2$, then computation cycles n_1 and n_2 cannot access v at the same time.

To exploit replication, the code generation scheme must be modified by replacing v with $\overline{v}[(cid - fst(o)) \bmod rep(v)]$ in the input and output parameter lists of every operation o that uses v. Here, cid is the index of the current pipelined cycle. It is represented in the generated code by an integer. When execution starts, cid is initialized with 0. At the start of each subsequent pipelined cycle, it is updated to $(cid + 1) \bmod R$, where R is the least common multiple of all the values $rep(v)$.

This simple implementation of rotating register files allows code generation for systems where no information is passed from one non-pipelined execution cycle to the next (no inter-cycle dependences). Such systems, such as the example in Figures 5.2 and 6.3, are also called stateless.

6.4.1.2 Extension to stateful predicated systems

In stateful systems, one (non-pipelined) execution cycle may use values produced in previous execution cycles. In these cases, code generation is more complicated, because an execution cycle must access memory cells that are not its own.

For certain classes of applications (such as systems without conditional control or affine loop nests), the cells to access can be statically identified by offsets with respect to the current execution cycle. For instance, the *MC* mode change function of Example 3 (Section 6.2.1.3) always reads the variable *m* produced in the previous execution cycle.

But in the general case, in the presence of predicated execution, it is impossible to statically determine which cell to read, as the value may have been produced at an arbitrary distance in the past. This is the case if, for instance, the data production operation is itself predicated. One solution to this problem is to allow the copying of one memory cell onto another in the beginning of pipelined cycles. But we cannot accept this solution due to the nature of our data, which can be large tables for which copying implies large timing penalties.

Instead, we modify the rotating register file as follows: storage is still ensured by the $\overline{v}$ circular buffer, which has the same length. However, its elements are not directly addressed through the modulo counter $(cid - fst(v)) \bmod rep(v)$ used above. Instead, this counter points in an array src_v whose cells are integer indices pointing towards cells of $\overline{v}$. This allows operations from several non-pipelined execution cycles (with different *cid* and $fst(o)$) to read the same cell of $\overline{v}$, eliminating the need for copying.

The full implementation of our register file requires two more variables: an integer $next_v$ and an array of Booleans $write\_flag_v$ of length $rep(v)$. Since $\overline{v}$ is no longer directly addressed through the modulo counter, $next_v$ is needed to implement the circular buffer policy of $\overline{v}$ by pointing to the cell where a newly-produced value can be stored next. One cell of $\overline{v}$ is allocated (and $next_v$ incremented) whenever a non-pipelined execution cycle writes v for the first time. Subsequent writes of v from the same non-pipelined cycle use the already allocated cell. Determining whether a write is the first from a given non-pipelined execution cycle is realized using the flags of $write\_flag_v$. Note that the use of these flags is not needed when a variable can be written at most once per execution cycle. This is often the case for code used in embedded systems design, such as the output of the Scade language compiler, or the outputs of LoPhT that we used for evaluation.If a given execution cycle does not write v, then it must read the same memory cell of $next_v$ that was used by the previous execution cycle.

The resulting code generation scheme is precisely described by the following rules:

1. At application start, for every memory cell (variable) v of the initial specification, $src_v[0]$, $write\_flag_v[0]$, $next_v$ are initialized respectively with 0, *false*, and 1. If $Init(v) \neq nil$ then $\overline{v}[0]$ (instead of v) is initialized with this value.

2. At the start of each pipelined cycle, for memory cell (variable) v of the initial scheduling table, assign to $src_v[(cid - fst(v)) \bmod rep(v)]$ the value of $src_v[(cid - fst(v) - 1) \bmod rep(v)]$, and set $write\_flag_v[(cid - fst(v)) \bmod rep(v)]$ to *false*.

3. In all operations o replace each input and output cell v with $\overline{v}[src_v[(cid - fst(o)) \bmod rep(v)]]$. The same must be done for all cells used in the computation of execution conditions.

4. When an operation o has v as an output parameter, then some code must be added before the operation (inside its execution condition). There are two cases. If v is not an input parameter of o, then the code is the following:

   1: **if not** $write\_flag_v[(cid - fst(o)) \bmod rep(v)]$ **then**
   2: $write\_flag_v[(cid - fst(o)) \bmod rep(v)] = true$
   3: $src_v[(cid - fst(o)) \bmod rep(v)] = next_v$
   4: $next_v = (next_v + 1) \bmod rep(v)$
   5: **end if**

   If v is also an input parameter of o, then only line 2 is needed from the previous code.

6.4.1.3 Accounting for book keeping costs

The software implementation of the rotating register files induces a timing overhead of its own. This overhead is formed of two components:

- Per operation costs, which can be conservatively accounted for in the WCET of the operations, as it is provided to the Phase 1 of our scheduling flow:
  - For each operation and for each input and output parameter, the cost of the indirection of point (3) above. This amounts to 2 indirections, one addition, and one modulo operation.
  - For each operation and for each output parameter, the cost of the book keeping operations defined at point (4) above.
- Per iteration costs, associated to point (2) above, and which must be added to the length of the pipelined scheduling table. This amounts to updating the src_v and $write\_flag_v$ data structures for all v. These costs are also bounded and can be accounted for with worst-case figures in Phase 1.

One important remark here is that our operations are large-grain tasks, meaning that these costs are often considered negligible, even for hard real-time applications.

The use of rotating registers also results in memory usage overheads. These overheads come from the replication of memory cells and from the pointer arrays src_v which must

be stored on the same memory bank as v for all memory cell v. We will assume that the cost of *src* is negligible, and only be concerned with the cost of replication, especially for large data.

If the replication of a large piece of data is a concern, then we can prohibit it altogether by requiring that all accesses to that memory cell are sequenced. This is done by adding a "sequencer" processor to the architecture model, and requiring all accesses to that memory cell to use the sequencer processor (this requires modifications to both the architecture and the functional specification). The introduction of sequencers may limit the efficiency of our pipelining algorithms. However, being able to target specific memory cells means that we can limit efficiency loss to what is really necessary on our memory-constrained embedded platforms. This simple approach satisfies our current needs.

6.5 Experimental results

We have implemented our pipelining algorithms in a prototype tool. We have integrated this tool with an existing makespan-optimizing scheduling tool[Potop-Butucaru et al., 2009] to form the full, two-phase flow of Fig. 6.1.

We have applied the resulting toolchain on 4 significant, real-life examples from the testbench of the SynDEx scheduling tool[Grandpierre and Sorel, 2003][3]. As no standard benchmarks are available in the embedded world, we have also applied our toolchain on a larger number of automatically synthesized dataflow graphs.

Our objective was to evaluate both the standalone pipelining algorithm, and the two-phase flow as a whole. Comparing with *optimal* scheduling results was not possible.[4] Instead, we rely on comparisons with existing scheduling and pipelining heuristics:

1. To evaluate the standalone algorithm, we measure the throughput gains obtained through pipelining, by comparing the initiation intervals of our examples before and after pipelining.

2. To evaluate the two-phase flow as a whole, we compare its output to the output of a classical throughput-optimizing software pipelining technique, namely the FRLC algorithm of [Wang and Eisenbeis, 1993].

[3]The other examples present in the SynDEx benchmark are mainly useful to demonstrate some of the features of SynDEx (such as hierarchy, conditions, etc...), and do not correspond to real, industrial, applications. Moreover, they are usually composed of only a small number of dataflow blocks. We thus decided not to use them in our evaluation process.

[4]Providing optimal solutions to our scheduling problem proved intractable even for small systems with 10 blocks and 3 processors.

example	Example size		Scheduling table length (initiation interval)		
	blocks	processors	initial (makespan)	pipelined (kernel)	gain
knock	5	2	6	3	50%
cycab	40	3	1482	1083	27%
ega	67	2	84	79	6%
robucar	84	3	1093	1053	8%

Figure 6.14: Pipelining gains for the real-life applications. Durations are in time units whose actual real-time length depends on the application (*e.g.* milliseconds).

The testbench The largest examples of our testbench ("cycab" and "robucar") are embedded control applications for the CyCab electric car [Pradalier et al., 2005]. The other two applications are an adaptive equalizer and a simplified model of an automotive knock control application [André et al., 2007].

We have used a script to automatically synthesize 30 examples (of which the first 10 are also presented individually in the result tables). For each example, synthesis is done as follows: we start with a graph containing only one data-flow node and no dependency. We apply a fixed number of expansion steps (3 steps for the examples in Fig. 6.15). At every step, each node is replaced with either a parallel or a sequential composition of newly-created nodes. The sequential and parallel choices are equiprobable. The number of nodes generated through expansion is chosen with a uniform distribution in the interval $[1..5]$. Dependences are also generated randomly at each step, and all previously-existing dependences are preserved. We implement these data-flow graphs on an architecture containing 5 processors and one broadcast bus. To model the fact that the architecture is not homogenous, the durations of the data-flow blocks on the various processors are assigned randomly (uniform distribution in an interval), and we randomly create a small number of placement constraints. We also assume that one of the processors performs input acquisition and another processor controls actuators. This implies placement constraints on data-flow blocks with no inputs and no outputs, respectively.

Pipelining gains The pipelining gains for the real-life and synthesized examples are summarized in Fig. 6.14 and Fig. 6.15, respectively. The figures show improvements on all examples, with a reduction of 27% in cycle time for the large "cycab" example, and an average reduction of 9.31% on the synthesized examples. We conclude that a pipelining stage such as ours should be part of any static scheduling flow.

At the same time, improvement varies greatly among the examples, from 50% for the knock control to 4% for one of the generated examples. We inspected the examples showing poor performance. Some of them, like "ega" have very tight schedules with

Example	Example size		Scheduling table length (initiation interval)		
	blocks	processors	initial (makespan)	pipelined (kernel)	gain
synth3	11	5	171	142	16.9%
synth7	14	5	212	191	9.9%
synth9	15	5	351	320	8.8%
synth5	23	5	375	348	7%
synth10	29	5	318	291	8%
synth1	34	5	463	430	7%
synth2	35	5	315	290	7.9%
synth8	40	5	622	594	4%
synth6	46	5	433	310	28%
synth4	52	5	840	793	5%
Average for 30 examples					**9.31%**

Figure 6.15: Pipelining gains for the synthesized examples

little idle CPU time, and therefore little opportunity for pipelining. More interesting was "robucar", which has significant idle time, but where a critical path in the scheduling table blocks pipelining. For such cases, more powerful pipelining algorithms are needed (as part of future work), able to modify the scheduling of the non-pipelined execution cycles, but without lengthening the makespan.

As explained in Section 6.4.1.3, we did not focus on precisely measuring the overhead in memory consumption induced by our technique.

Example	Our method		[Wang and Eisenbeis, 1993]		Makespan gain	Throughput loss
	makespan	kernel	makespan	kernel		
synth1m	298	276	850	246	64%	12%
synth2m	277	263	570	190	51%	38%
synth3m	142	142	299	142	52%	0%
synth4m	347	328	2380	255	85%	28%
synth5m	155	155	651	155	76%	0%
synth6m	349	288	1774	288	80%	0%
synth7m	130	115	258	115	49%	0%
synth8m	376	357	610	290	38%	23%
synth9m	116	96	198	82	41%	17%
synth10m	300	273	1686	96	82%	184%
Average for 30 examples					**63.63%**	**38.33%**

Figure 6.16: Comparison with a classical software pipelining algorithm

Comparison with a classical software pipelining algorithm To make this comparison, we have implemented the classical FRLC algorithm of [Wang and Eisenbeis, 1993]. We chose this algorithm because of its flexibility. It is easy to extend it to cover aspects taken into account by our tool, such as the presence of operations that have different durations on different functional units, or communication costs. However, we have used here for comparisons its baseline, restricted version. We therefore considered only the synthesized examples, which have a simpler structure, and modified them by removing communication costs and by choosing a single duration for each operation on all processors that can execute it.

On these modified examples we applied our two-phase flow and the FRLC method and compared the results in Fig. 6.16. For each example and scheduling flow we provide the makespan and the kernel length of the generated code. We have also computed the makespan gain and throughput loss when moving from the FRLC technique to ours.

In these figures the makespan-throughput trade-off is clearly visible (as we expected). On average, our method gains 63.63% in makespan while losing 38.33% in throughput. A less expected result is that this trade-off can be identified in each example, even though the algorithms under comparison are heuristics. Indeed, our method is always better in makespan, while the FRLC technique of [Wang and Eisenbeis, 1993] is never worse in throughput.

Given our optimization objective (makespan first, throughput second), we consider that *our choice of two-phase optimization heuristic is justified*.

Predication is present in 3 of our examples: example 3, knock, and a variant of the cycab example. In all these examples, predication is used to encode mode-dependent behavior, and the examples showcase different situations where pipelining is necessary in embedded systems design. Mode-dependent behavior is usually specified at the level of full systems or large subsystems, meaning that the number of predicates is usually low, yet each predicate controls a significant part of the operations of the system. Our examples have only two or three predicates (two in knock and cycab, three in Example 3). To evaluate the contribution of predication analysis to the pipelining results, we have also run our algorithms with the predication analysis disabled. The use of predication does not improve pipelining result for the cycab example. On the contrary, it results in significant gains for Example 3 and knock (20% and 40%, respectively).

6.6 Conclusion

In this section we have presented a new software pipelining technique especially adapted to the needs of embedded applications. We showed that classical software pipelining techniques were not fit for such applications since their main objective is the optimization

of the throughput of loops, and optimizing throughput can lead to unacceptable results in terms of cycle latency/makespan. Our bi-criteria optimization applies in a first step a makespan optimizing scheduling technique. In a second step, it optimizes the throughput of the application without modifying any decision inside the scheduled cycle.

The use of a decomposed software pipelining technique to optimize these two criteria is a major originality of our approach. Another original point is to exploit information concerning the execution conditions (modes) in order to further improve the optimization of these criteria. We demonstrated our technique on various examples (both real-life and synthesized ones) and compared it with an existing decomposed software pipelining technique. The (good) results make us believe that such bi-criteria optimizations are well suited for embedded applications where real-time requirements can be reduced to a simple characterization in terms of cycle latency and throughput.

Nevertheless, as we will see in the next chapters, some modern embedded applications include complex functional and non-functional constraints and requirements (e.g. multiple real-time requirements, partitioning, preemptability), which cannot be tackled efficiently by techniques that optimize only one or two criterions. Consequently, we must adapt our software pipelining technique and integrate it into a scheduling technique whose objective is to respect complex sets of non-functional requirements, as opposed to optimizing one or two metrics. This work is presented in the next chapters.

Part III

Real-time scheduling and code generation under complex non-functional constraints

Chapter 7

Extensions of the Clocked Graph formalism

Contents

In previous chapters we have defined powerful formalisms for the specification of the functionality and of the resources of an embedded system, and for the representation of resource allocation in a table-based off-line scheduling paradigm. These formalisms support scheduling and optimization algorithms able to automatically synthesize efficient implementations providing tight hard real time guarantees onto multi-processor/distributed execution platforms.

However, even though the results of the previous chapters are original, they still follow a classical compilation approach. Indeed, their objective is the *fully automated* generation of implementations that are *functionally correct* and which *optimize* some real-time characteristics (in our case, worst-case latency and throughput, as opposed to the average-case

execution time that is optimized in classical compilation). Such a compilation process never fails, because it is *not constrained*.

But the implementation needs of complex embedded systems are not well captured under the form of such unconstrained optimization problems. Indeed, embedded systems have multiple non-functional requirements that must be satisfied by the final implementation, and which need to be taken into account by the scheduling and code generation algorithms. In the remainder of this thesis we shall extend our specification formalisms to allow the description of complex non-functional requirements, and then provide compilation algorithms able to solve (heuristically) such constrained implementation problems. Thus, we retain the full automation of classical compilation, but extend it to more complex systems.

The choice of non-functional properties that we will consider is not arbitrary. To ensure that the resulting approch has practical applicability, we consider an industrial application class and a case study coming from the aerospace industry. We include in our approach all the aspects (model characteristics and non-functional properties) needed to allow scheduling and code generation for the chosen application class. In particular, considering all details needed to allow actual code generation means that we will need to precisely define the (time-triggered partitioned) execution mechanisms of the target platform. Once this is done, this chapter will focus on the non-functional properties we consider:

- The preemptability[1] of the operations of the application. The initial CG formalism presented in Chapter 3 does not support the preemption of the functions execution. Nevertheless, to improve the schedulability of our systems, we want to allow it. Since we target a totally static implementation, we cannot rely on an online scheduler to deal with preemptions such as in Giotto [Henzinger et al., 2000] or Prelude [Pagetti et al., 2011]. Instead we want our offline technique to build a preemptive static schedule, which includes the potential dates where preemption can happen (in the worst case scenario),

- The temporal partitioning of the application, as specified for TTA [Kopetz and Bauer, 2003], FlexRay [Rushby, 2001] (the static segment), and ARINC653 [ARINC], which allows the static allocation of CPU or bus time slots, on a periodic basis, to various parts (known as partitions) of the application. Also known as static time division multiplexing (TDM) scheduling, partitioning further enhances the temporal determinism of a system,

- Release dates and deadlines for the tasks of the application, which allow the model-

[1] The use of the terms preemptability and preemptable is common in real-time when referring to tasks. The terms “interrupt” and “interruptible” refer to the machine interrupts instead.

ing of constraints coming from the environment and/or the dynamics of the system. A noticeable point is that in order to allow a more natural real-time specification, an improved schedulability and less context changes between partitions (which are notoriously expensive), we allow the use of deadlines that are longer than the periods in the specification.

We also consider allocation constraints, which are already taken into account in the architecture description formalism of Chapter 4.

Industrial case study Our work is based on a launcher (spacecraft) embedded control system case study provided by Airbus DS (formerly ASTRIUM). Such spacecraft systems show very strict real-time requirements, since, for example, the unavailability of the avionics system of a space launcher during a few milliseconds in the atmospheric phase may lead to the destruction of the launcher. In such systems, latencies are defined between the acquisition of data by sensors and the sending of orders to actuators. In the meantime, the orders are computed using control algorithms (GNC, for Guidance, Navigation and Control algorithms). The GNC algorithms are usually emplemented on a dedicated processor in a classical multi-tasking approach. In the last decade, the increase of computational power provided by space processors allows the distribution of the GNC computations on the sensors and actuators processors, and the suppression of the processor that was, until now, dedicated to the GNC computations. In the future, the GNC algorithms could be separated, and for example the navigation algorithm could run on the processor controlling the gyroscope, while the control algorithm would run on the processor controlling the thruster. For the industrials who manufacture the spacecraft systems (space launchers or space transportation vehicles), this means a non-negligeable reduction of costs and of weight and power consumption.

Nevertheless, this practice leads to the sharing of processors by pieces of software having different Design Assurance Levels (such as gyroscope control and navigation for example), and consequently requires the use of an operating system that enforces Time and Space Partitioning (TSP) between them. In such operating systems scheduling is handled by a hierarchic two-level scheduler in which the top level is of static Time-triggered (TDM) type. It is the case, for instance, of ARINC 653 [ARINC]. Moreover, for predictability issues, the processors of the distributed implementation platform share a common time base. As a consequence, the execution platform offers the possibility of a globally time-triggered implementation. Therefore, we aim for distributed implementations that are time triggered at all scheduling levels. This improves system predictability and allows the computation of tighter worst-case bounds on end-to-end latencies.

In the end, this case study allowed us to define the following scheduling problem: given a set of end-to-end latencies defined at spacecraft system level, along with sensing

and actuation operations offsets and safe worst case execution time (WCET) estimations of the computation functions, we wanted to synthesize the time-triggered schedule of the system, including the activation of each partition and each functional node, and the bus frame.

In order to solve this problem, we first needed to increase the expressiveness of the clocked graph formalism, so that it can take into account the new partitioning and real-time constraints that can be specified on such complex systems. We thus extended it with an original real-time characterization that takes advantage of the time-triggered framework to provide a simpler representation of complex end-to-end flow requirements. We also extended our specifications with additional non-functional properties specifying partitioning and preemptability constraints. The CG formalism remains an intermediate representation for code generation, but the increase of its expressiveness allows more tuning, more precision, and more optimizations during the scheduling step of the compilation process, which is described in Chapter 8.

Chapter outline In the remainder of this chapter, we present a detailed state of the art of the models and methods used to tackle separately the various real-time and partitioning problems that we want to solve. Then, we give a precise definition of the time-triggered execution model that we consider, along with a characterization of time partitioning in this framework. Then, using the abstract formalism defined at the end of Chapter 3, we introduce the simplified specification of the aerospace case study that will be used in this chapter and in the next one to present the extensions we made to our formalism, as well as the new scheduling algorithms that we developped. Finally, we provide the extensions made to the CG language in order to include the new non-functional constraints in our models.

7.1 Related work

The main originality of this chapter is to define a complex task model allowing the specification of all the functional and non-functional aspects needed for the efficient implementation of our case study. Of course, *prior work already considers all these functional and non-functional aspects, but either in isolation (one aspect at a time), or through combinations that do not cover our modeling needs*. Our contributions are the non-trivial combination of these aspects in a coherent formal model and the definition of synthesis algorithms able to build a running real-time implementation.

Previous work [Henzinger and Kirsch, 2007, Henzinger et al., 2000, Pagetti et al., 2011, Marouf et al., 2012, Alras et al., 2009] on the implementation of multi-periodic synchronous programs and the work by [Blazewicz, 1977] and [Chetto et al., 1990] on

the scheduling of dependent task systems have been important sources of inspiration. By comparison, our work provides a general treatment of ARINC 653-like partitioning and of conditional execution, and a novel use of deadlines longer than periods to allow faithful real-time specification.

The work of [Caspi et al., 2003] addresses the multiprocessor scheduling of synchronous programs under bus partitioning constraints. By comparison, our work takes into account conditional execution and execution modes, allows preemptive scheduling, and allows automatic allocation of computations and communications. Taking advantage of the time-triggered execution context, our approach also relies on fixed deadlines (as opposed to relative ones), which facilitates the definition of fast mapping heuristics.

Another line of work on the scheduling of dependent tasks is represented by the works of [Pop et al., 1999] and [Zheng et al., 2005]. In both cases, the input of the technique is a DAG, whereas our functional specifications allow the use of delayed dependencies between successive iterations of the DAG. Other differences are that the technique [Zheng et al., 2005] does not take into account ARINC 653-like partitioning or conditional execution, and the technique of [Pop et al., 1999] does not allow the specification of complex end-to-end latency constraints. [Fohler, 1993] does consider conditional control, but does so in a mono-processor, non-partitioned, non-preemptive context.

The off-line (pipelined) scheduling of tasks with deadlines longer than the periods has been previously considered (among others) by [Fohler and Ramamritham, 1997], but this work does not consider, as we do, partitioning constraints and the use of execution conditions to improve resource allocation. This is also our originality with respect to other classical work on static scheduling of periodic systems [Ramamritham et al., 1993].

Compared to previous work by [Isovic and Fohler, 2009] on real-time scheduling for predictable, yet flexible real-time systems, our approach does not directly cover the issue of sporadic tasks, but allows a more flexible treatment of periodic (time-triggered) tasks. Based on a different representation of real-time characteristics and on a very general handling of execution conditions, we allow for better flexibility *inside* the fully predictable domain.

From an implementation-oriented perspective, Giotto [Henzinger et al., 2000, Henzinger and Kirsch, 2007], ΨC [Chabrol et al., 2009], and Prelude [Pagetti et al., 2011, Puffitsch et al., 2013] make the choice of mixing a globally time-triggered execution mechanism with on-line priority-driven scheduling algorithms such as RM or EDF. By comparison, we made the choice of taking *all* scheduling decisions off-line. Doing this complicates the implementation process, but imposes a form of temporal isolation between the tasks which reduces the number of possible execution traces and increases timing precision (as the scheduling of one task no longer depends on the run-time duration of the others). In turn, this facilitates verification and validation. Furthermore, a fully

off-line scheduling approach such as ours has the potential of improving worst-case performance guarantees by taking better decisions than a RM/EDF scheduler which follows an as-soon-as-possible (ASAP) scheduling paradigm. For instance, the transformations of Section 8.3 reduce the number of notoriously expensive partition changes by using a scheduling technique that is not ASAP. These partition changes are not taken into account in the optimality results concerning the EDF scheduling of Prelude [Pagetti et al., 2011].

Compared to classical work on the on-line real-time scheduling of tasks with execution modes (*cf.* [Baruah, 2003]), our off-line scheduling approach comes with precise control of timing, causalities, and the correlation (exclusion relations) between multiple test conditions of an application. It is therefore more interesting for us to use a task model that explicitly represents execution conditions. We can then use *table-based scheduling* algorithms that precisely determine when the same resource can be allocated at the same time to two tasks because they are never both executed in a given execution cycle, such as we explained in Chapter 4.

The use of execution conditions to allow efficient resource allocation is also the main difference between our work and the classical results of [Xu, 1993]. Indeed, the exclusion relation defined by Xu does not model conditional execution, but resource access conflicts, thus being fundamentally different from the exclusion relation we defined in Section 3.4. Our technique also allows the use of execution platforms with non-negligible communication costs and multiple processor types, as well as the use of preemptive tasks (unlike in Xu's paper).

The off-line scheduling on partitioned ARINC 653 platforms has been previously considered, for instance by [Sheikh et al., 2012] and by Brocal *et al.* in Xoncrete [Brocal et al., 2010]. The first approach only considers systems with one task per partition, whereas our work considers the general case of multiple tasks per partition. The second approach (Xoncrete) allows multiple tasks per partition, but does not seem interested in having a functionally deterministic specification and preserving its semantics during scheduling (as we do). For instance, its input formalism specifies not periods, but ranges of acceptable periods, and the first implementation step adjusts these periods to reduce their lowest common multiple (thus changing the semantics). Other differences are that our approach can take into account conditional execution and execution modes, and that we allow scheduling onto multi-processors, whereas Xoncrete does not.

More generally, our work is related to work on the scheduling for precision-timed architectures (*e.g.* [Edwards and Lee, 2007]). Our originality is to consider complex non-functional constraints. The work on the PharOS technology [Chabrol et al., 2009] also targets dependable time-triggered system implementation, but with two main differences. First, we follow a classical ARINC 653-like approach to temporal partitioning. Second, we take all scheduling decisions off-line. This constrains the system but reduces the

scheduling effort needed from the OS, and improves predictability.

References on time-triggered and partitioned systems, as well as scheduling of synchronous specifications will be provided in the following sections.

7.2 Architecture model

7.2.1 Time-triggered systems

In this section we define the notion of time-triggered system that we will be using in the remainder of the thesis. It roughly corresponds to the definition given by [Kopetz, 1991], and is a sub-case of the definition given by [Henzinger and Kirsch, 2007]. We shall introduce its elements progressively, explaining what the consequences are in practice.

7.2.1.1 General definition

By *time-triggered systems* we understand systems satisfying the following 3 properties:

TT1 A system-wide time reference exists, with good-enough precision and accuracy. We shall refer to this time reference as the *global clock*[2]. All timers in the system use the global clock as a time base.

TT2 The execution duration of code driven by interrupts other than the timers (*e.g.* interrupt-driven driver code) is negligible. In other words, for timing analysis purposes, code execution is only triggered by timers synchronized on the global clock.

TT3 System inputs are only read/sampled at timer triggering points.

This definition places no constraints on the sequential code triggered by timers. In particular:

- Classical sequential control flow structures such as *sequence* or *conditional execution* are permitted, allowing the representation of modes and mode changes.
- Timers are allowed to preempt the execution of previously-started code.

This definition of time-triggered systems is fairly general. It covers single-processor systems that can be represented with *time-triggered e-code programs*, as they are defined by [Henzinger and Kirsch, 2007]. It also covers multiprocessor extensions of this model, as defined by [Fischmeister et al., 2006] and used by [Potop-Butucaru et al., 2010]. In particular, our model covers time-triggered communication infrastructures such as TTA

[2]For single-processor systems the global clock can be the CPU clock itself. For distributed multiprocessor systems, we assume it is provided by a platform such as TTA [Kopetz and Bauer, 2003] or by a clock synchronization technique such as the one of Potop *et al.* [Potop-Butucaru et al., 2010].

and FlexRay (static and dynamic segments) [Kopetz and Bauer, 2003, Rushby, 2001], the periodic schedule tables of AUTOSAR OS [AUTOSAR], as well as systems following a multi-processor periodic scheduling model without jitter and drift[3]. It also covers the execution mechanisms of the avionics ARINC 653 standard [ARINC] provided that interrupt-driven data acquisitions, which are confined to the ARINC 653 kernel, are presented to the application software in a time-triggered fashion satisfying property *TT3*. One way of ensuring that *TT3* holds is presented in [Mason et al., 2006], and to our knowledge, this constraint is satisfied in all industrial settings.

7.2.1.2 Model restriction

The major advantage of time-triggered systems, as defined above, is that they have the property of *repeatable timing* [Edwards et al., 2009]. Repeatable timing means that for any two input sequences that are identical in the large-grain timing scale determined by the timers of a program, the behaviors of the program, including timing aspects, are identical. Of course, this ideal property must be amended to take into account the fact that the global clock may not be very accurate or that interrupt-driven driver code does take time and influences the execution of time-triggered code. In practice, however, repeatability can be ensured with good precision. Repeatability is extremely valuable in practice because it largely simplifies debugging and testing of real-time programs. A time-triggered platform also insulates the developer from most problems stemming from interrupt asynchrony and low-level timing aspects.

However, the applications we consider have even stronger timing requirements, and must satisfy a property known as *timing predictability* [Edwards et al., 2009]. Timing predictability means that formal timing guarantees covering *all* possible executions of the system should be computed off-line by means of (static) analysis. The general time-triggered model defined above remains too complex to allow the analysis of real-life systems. To facilitate analysis, this model is usually restricted and used in conjunction with WCET analysis of the sequential code fragments.

In our model we consider a restriction of the general definition provided above. In this restriction, timers are triggered following a fixed pattern which is repeated periodically in time. Following the convention of ARINC 653, we call this period the *major time frame (MTF)*. The timer triggering pattern is provided under the form of a set of fixed offsets $0 \leq t(1) < t(2) < \ldots < t(m) < MTF$ defined with respect to the start of each *MTF* period. Note that the code triggered at each offset may still involve complex control, such as conditional execution or preemption.

This restriction corresponds to the classical definition of time-triggered systems by

[3]But these two notions must be accounted for in the construction of the global clock [Potop-Butucaru et al., 2010].

[Kopetz, 1991, Kopetz and Bauer, 2003]. It covers our target platform, TTA, FlexRay (the static segment), and AUTOSAR OS (the periodic schedule tables). At the same time, it does not fully cover ARINC 653. As defined by this standard, partition scheduling is time-triggered in the sense of Kopetz. However, the scheduling of tasks inside partitions is not, because periodic processes can be started (in normal mode) with a release date equal to the current time (not a predefined date). To fit inside Kopetz's model, an ARINC 653 system should not allow the start of periodic processes *after* system initialization, *i.e.* in normal mode.

7.2.2 Temporal partitioning

Our target architectures follow a strong temporal partitioning paradigm similar to that of ARINC 653[4]. In this paradigm, both system software and platform resources are *statically* divided among a finite set of *partitions* $Part = \{part_1, \ldots, part_k\}$. Intuitively, a partition comprises both a software application of the system and the execution and communication resources allocated to it. The aim of this static partitioning is to limit the functional and temporal influence of one partition on another. Partitions can communicate and synchronize only through a set of explictly-specified inter-partition channels.

In this chapter and the next one, we are mainly concerned with the execution resource represented by the processors. To eliminate timing interference between partitions running on a processor, the static partitioning of the processor time is done using a static time division multiplexing (TDM) mechanism. In our case, the static TDM mechanism is built on top of the time-triggered model of the previous section. It is implemented by partitioning, separately for each processor P_i, the *MTF* defined above into a finite set of non-overlapping *windows* $W_i = \{w_i^1, \ldots, w_i^{k_i}\}$. Each window w_i^j has a fixed start offset tw_i^j, a duration dw_i^j, and it is either allocated to a single partition $partw_i^j$, or left unused. Unused windows are called spares and identified by $partw_i^j = spare$.

Software belonging to partition $part_i$ can only be executed during windows belonging to $part_i$. Unfinished partition code will be preempted at window end, to be resumed at the next window of the partition. There is an implicit assumption that the scheduling of operations inside the *MTF* will ensure that non-preemptive operations will not cross window end points.

For the scheduling algorithms of Section 8.2, the partitioning of the *MTF* into windows can be either an input or an output. More precisely, all, none, or part of the windows can be provided as input to the scheduling algorithms.

[4]Spatial partitioning aspects are not covered in the thesis.

7.2.3 Example

To rapidly present the type of partitioned time-triggered system we synthesize and the underlying execution mechanisms, we rely on the example of Fig. 7.1, which pictures a possible scheduling table for the application described in Fig. 3.18. When compared with the scheduling tables introduced in Chapter 4, our example has new features: partitioning and pre-computed preemption. An extension of the formalism is therefore needed. Through this example we provide here its intuitive semantics, while the formal definitions supporting it (including the corresponding syntax) will come in the following sections.

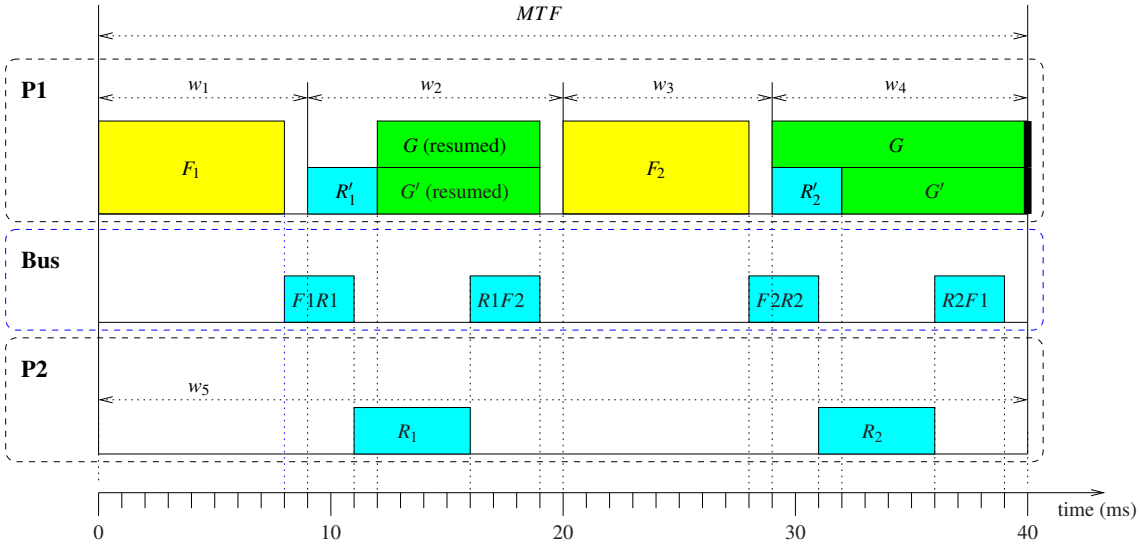


Figure 7.1: An example of time-triggered partitioned system. In this example only processor P1 requires partitioning.

Our system has two processors, denoted P1 and P2, which are connected *via* a bus. As explained in Chapter 4, our execution model is periodic. Our figure provides the execution pattern for one period, under the form of a scheduling table. A full execution of the system is obtained by indefinitely repeating this pattern. In our case, the length of the scheduling table, which gives the MTF (global period) of the system is 40 milliseconds (ms). Note that in this graphical representation, time flows from left to right, and each row in the picture depicts the worst case schedule of a hardware resource.

Recall from Chapter 4 that the scheduling table defines the start dates of all the operations that can be executed during one MTF, be them computations or communications. For operations that can be preempted, it also defines all the dates where they are preempted and resumed (all these dates are pre-computed off-line). Finally, our table also defines the worst-case end dates of the operations. All dates are defined in the time reference of the global clock provided by the time-triggered platform. In our example, this clock (pictured at the bottom of the figure) has a precision of 1 ms. For instance, operation

F_1 starts in each MTF at date 0 and ends (in the worst case) at date 8. Synchronization of operations with respect to the global clock also ensures the correct synchronization between operations. For instance, F_1 ending before date 8 allows the synchronization with the communication *ftor1* which transmits a value from F_1 to R_1.

To comply with Kopetz' time-triggered model, the start/preempt/resume dates of all operations are computed off-line. For instance, on processor P1 operations can only start at dates 0, 9, 20, 29, and 32. However, code triggered at each offset may involve complex control such as conditional execution or preemption. Our example features both. Conditional execution is used here to represent execution modes. There are 2 modes. In *mode1* are executed F_1, F_2, G, R_1, R_2 and the bus communications. The other mode, named *mode2*, is meant to be activated when processor $P2$ is removed from the system due to a hardware reconfiguration. In this mode, operations F_1 and F_2 remain unchanged, but the other are replaced with the lower-duration counterparts G', R'_1, and R'_2 (and no communication operations).

In our example, mode changes occur at the beginning of the MTF. Only operations belonging to the currently-active mode can be started during the MTF. As explained in Chapter 4, operations of different modes can be scheduled on the same processor at the same dates, as is the case for G and G' and G and R'_2. This form of *double reservation* is forbidden if the operations belong to the same mode because our resources (processors and buses) are sequential.

Preemption is needed in the scheduling of long tasks and in dealing with partitioning. In our example, we assumed that the computation operations are divided in 2 partitions: *part1* contains F_1 and F_2, and *part2* all the other operations. During each MTF, the processor time of P1 is statically divided in 4 windows. Windows w_1 and w_3 are allocated to *part1*, and windows w_2 and w_4 are allocated to *part2*. To simplify the presentation of the algorithms we assume that bus time is not partitioned. Processor P2 only executes operations of *part2*, so that its MTF contains one window w_5 spanning over the entire MTF.

Window w_4 starts at date 29 and ends at date 40. Operation G starts at date 29, but its worst-case duration is 18 ms, longer than the 11 ms of w_4. Since window w_1 belongs to another partition, operation G must be pre-empted at the end of w_4, and it can be resumed at date 12 (within w_2) in the next MTF. Our scheduling table therefore contains a precomputed preemption (represented with a thick black bar at the end of G in w_4), and the execution time of G is divided between 2 windows. Similarly, the execution of G' is divided in two by a precomputed preemption.

An important hypothesis in our work is that the scheduling table specifies not only the start date of an operation, but also the dates where an operation can resume after a pre-computed preemption. For instance, operations G and G' resume at date 12, not

at date 9. This hypothesis facilitates scheduling. In time-triggered operating systems such as ARINC 653, where only start dates are specified, this hypothesis can be easily implemented by using simple manipulations of task priorities[5].

Another important hypothesis we make is that once an operation is started, it must be completed. In other words, it can be preempted and resumed, but it cannot be aborted. This hypothesis is key in ensuring the predictability and determinism of our systems. Intuitively, an operation can change the state of sensors, actuators, or internal variables, and it is difficult to determine the system state after an abortion operation.

But the absence of abortion requires much care in dealing with conditional execution and execution mode changes. Assume, for instance, that operation G is started during one MTF, and that the mode changes from *mode1* to *mode2* at the end of that MTF. Then, the scheduling of the next MTF must allow G to resume and complete while ensuring that the operations of *mode2* can be started. This explains why R'_1 and the resumed part of G cannot be scheduled at the same date, even though G and R'_1 belong to different modes.

7.3 Modeling of the aerospace case study

The specification of the space flight application was provided under the form of a set of AADL [international, 2014] diagrams, plus textual information defining specific inter-task communication patterns, determinism requirements, and a description of the target hardware architecture. In this section we use the simpler version of the specification (with fewer tasks), the results on the full example being provided in Section 8.3.

Our first step was to derive a task model in our formalism. In doing this we discovered that the initial system was over-specified, in the sense that real-time constraints were imposed in order to ensure causal ordering of tasks instances using AADL constructs. Removing these constraints and replacing them with (less constraining) data dependencies gave us more freedom for scheduling, allowing for a reduction in the number of partition changes. The resulting specification, described in the formalism of Section 3.4, is presented in Fig. 7.2.

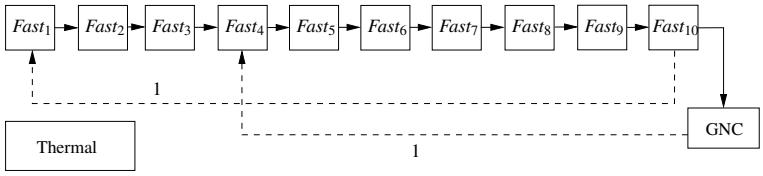


Figure 7.2: The *Simple* example

[5]By using helper tasks that are executed at the time triggering points, and which raise or lower the priority of the other tasks to ensure that the greatest priority belongs to the one that must be executed (started or resumed) in the following time slot

Our model, named *Simple* represents a system with 3 tasks *Fast*, *GNC*, and *Thermal*. The periods of the 3 tasks are 10ms, 100ms, and 100ms, respectively, meaning that *Fast* is executed 10 times for each execution of *GNC* and *Thermal*. The hyperperiod expansion described in Section 3.5 replicates task *Fast* 10 times[6], the resulting tasks being $Fast_i$, $1 \leq i \leq 10$. Tasks *GNC* and *Thermal* are left unchanged because their period equals the hyperperiod. The direct arcs connecting the tasks $Fast_i$ and *GNC* represent regular (intra-cycle) data dependencies of $A(Simple)$. Delayed data dependencies of depth 1 represent the transmission of information from one MTF to the next. In this simple model, task *Thermal* has no dependencies.

7.3.1 From non-determinism to determinism

The design of complex embedded systems usually starts with *sets of requirements* allowing multiple implementations. Space launchers are no exception to this rule. For instance, the requirements for example *Simple* do not impose the presence of a delayed arc connecting *GNC* to $Fast_4$. Instead, they require that there exists an i such that the feedback from *GNC* to $Fast_i$ is performed under a given latency constraint.

But determinism has its advantages: as explained in Section 7.2.1.2, it largely simplifies debugging and testing. Moreover, the determinism of the functional specification allows for a significant decoupling of software development, including verification and validation steps, from allocation and scheduling choices.

This is why a deterministic functional specification is built early in the development process of space launchers. The first step in this direction is made when Giotto-like rules [Henzinger et al., 2000] are used to impose a fixed set of data dependencies, thus creating a deterministic functional specification. These rules only depend on the number of tasks before hyperperiod expansion, their relative periods, and a coarse view on the flow of data from task to task. This information can be easily recovered from the requirements and from early implementation choices.

The Giotto-like rules allow the fast construction of a deterministic specification and are easy to understand and implement. However, they do not take into account latency constraints, nor task durations, and thus the initial functional specification may not allow real-time implementation. In our example, the initial dependency pattern did not allow the respect of the latency constraint on the feedback from *GNC* to *Fast*. When this happens,

[6]Task *Fast* is in fact composed of several functions which can be activated or not depending on the context, and on the considered task instance. In the high-level specification of the problem that we worked on, consisting in multi-periodic tasks, *Fast* is only represented by one task. Nevertheless, this specification also states that there exists a data flow going through ten successive instances of *Fast*, one instance of *GNC*, and which ends by a feedback to one (and only one) instance of *Fast*. The behaviour of the various instances of a given task may thus behave differently, but this special behaviour is specified and fixed before the scheduling phase.

the dependencies are modified manually within the limits fixed by the requirements to allow real-time implementation.

But these manual modifications come at a cost, especially when they must be done late in the design flow, requiring the re-validation of the whole design. This cost can be acceptable when the system is first built, which means once every 20 years or so. But once a deterministic implementation is built, it is strongly desired that subsequent *modifications* of the system preserve unchanged the functionality of sub-systems that are not modified (and in particular their determinism). If this is possible during the system modification, then the confidence in the modified system is improved and less effort is needed for the re-validation of the system.

In other words, design choices made during the initial implementation become desired properties for subsequent modifications, where they are considered as part of the functional specification. In our example, we assume that the feedback from *GNC* to $Fast_4$ is such an implementation choice made in the initial implementation, and which we include in the functional specification.

Our algorithms and tool allow the scheduling of such deterministic specifications. For the cases where the requirements are non-deterministic, our algorithms and tool can be used to speed up an otherwise manual exploration of the possible design choices (but this exploration process is not described in this thesis).

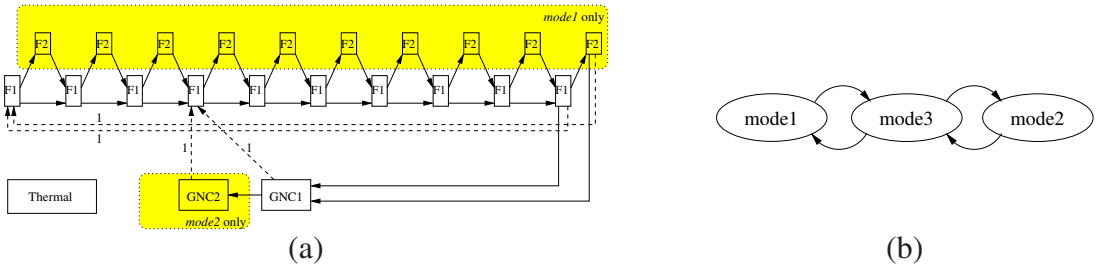
7.3.2 Representing execution modes

The dependent task system of Fig. 7.2 does not represent execution modes, implicitly assuming that for each task the scheduling will always use its worst-case execution time.

But a space launcher application does make use of conditional execution and execution modes, and the scheduling can be optimized by taking them into account. The difficulty is to allow scheduling in a way that takes into account modes and is also compatible with the execution mechanisms of the launcher. Recall that the number of tasks in the launcher is fixed to 3. Mode changes do not trigger here the start or stop of tasks, as in our example of Fig. 7.1. Instead, they are encoded with changes of state variables that enable or disable the execution of various code fragments inside the 3 tasks.

In this approach, the only macroscopic property of a task that changes depending on the mode and can therefore be exploited during scheduling is its duration. Representing mode-dependent durations using our formalism requires a non-trivial transformation, detailed through the example in Fig. 7.3.

We assume that our system has 3 modes (1, 2 and 3), the mode 3 being a transition mode between 1 and 2, as shown in Fig. 7.3(b). We assume that the duration of tasks *Fast* and *GNC* depends on the mode. We denote with $WCET(\tau, P)_m$ the duration of task $\tau \in \{Fast, GNC\}$ on processor P in mode $m \in \{1,2,3\}$. We assume that $WCET(Fast, P)_3 =$



(a) (b)

Figure 7.3: Example 3: Dataflow specification with conditional execution (a) and possible mode transitions (b)

$WCET(Fast,P)_1 < WCET(Fast,P)_2$ and that $WCET(GNC,P)_3 = WCET(Fast,P)_2 < WCET(Fast,P)_1$ for all P. Then, our modeling is based on the use of 2 tasks for the representation of each of *Fast* and *GNC*. The first task represents the amount of computation that is needed in all modes (corresponding to the smaller WCET value), whereas the second task represents the remainder, which is only needed in modes where the duration is longer.

The resulting model is pictured in Fig. 7.3. Here, *Fast* has been split into *F1* and *F2*, the second one being executed only in mode 2. *GNC* has been split into *GNC1* and *GNC2*, the second one being executed only in mode 1. The mode change automaton ensures that $(GNC2, F2_i, 1) \in EX(Simple)$ for $1 \leq i \leq 10$, a property that will be used by the scheduling algorithms of Section 8.2.

The method we intuitively defined here for tasks with 2 durations can be generalized. A task having n durations depending on the mode will need an expansion into n tasks.

7.4 Non-functional properties

Our task model considers non-functional properties of 4 types: real-time, allocation, partitioning, and preemptability.

7.4.1 Period, release dates, and deadlines

The initial functional specification of a system is usually provided by the control engineers, which must also provide a *real-time characterization* in terms of *periods*, *release dates*, and *deadlines*. This characterization is directly derived from the analysis of the control system, and does not depend on architecture details such as number of processors, speed, etc. The architecture may impose its own set of real-time characteristics. Our model allows the specification of all these characteristics in a specific form adapted to our functional specification model and time-triggered implementation paradigm.

7.4.1.1 Period

Recall from Section 3.5 that after hyper-period expansion all the tasks of a dependent task system D have same period. We shall call this period the *major time frame* of the dependent task system D and denote it $MTF(D)$. We will require it to be equal to the *MTF* of its time-triggered implementation, as defined in Section 7.2.1.2.

Throughout this chapter and the next one, we will assume that $MTF(D)$ is an input to our scheduling problem. Other scheduling heuristics, such as those of [Potop-Butucaru et al., 2010] can be used in the case where the MTF must be computed.

7.4.1.2 Release dates and deadlines

For each task $\tau \in T(D)$, we allow the definition of a release date $r(\tau)$ and a deadline $d(\tau)$. Both are positive offsets defined with respect to the start date of the current MTF (period). To signify that a task has no release date constraint, we set $r(\tau) = 0$. To signify that it has no deadline we set $d(\tau) = \infty$.

The main intended use of release dates is to represent constraints related to input acquisition. Recall that in a time-triggered system all inputs are sampled. We assume in our work that these sampling dates are known (a characteristic of the execution platform), and that they are an input to our scheduling problem. This is why they can be represented with fixed time offsets. Under these assumptions, a task using some input should have a release date equal to (or greater than) the date at which the corresponding input is sampled.

End-to-end latency requirements are specified using a combination of both release dates and deadlines. We require that end-to-end latencies are defined on flows (chains of dependent task instances) starting with an input acquisition and ending with an output. Since acquisitions have fixed offsets represented with the release dates, the latency constraints can also be specified using fixed offsets, namely the deadlines.

Before providing an example, it is important to recall that our real-time implementation approach is based on off-line scheduling. The release dates and deadlines defined here are specification objects used by the off-line scheduler alone. These values have no direct influence on implementations, which are exclusively based on the scheduling table produced off-line. In the implementation, task activation dates are always equal to the start dates computed off-line, which can be very different from the specification-level release dates.

7.4.1.3 Modeling of the case study

The specification in Fig. 7.4 adds a real-time characterization to the *Simple* example of Fig. 7.2. Here, $MTF(Simple) = 100$ ms. Release dates and deadlines are respectively represented with descending and mounting dotted arcs. The release dates specify that

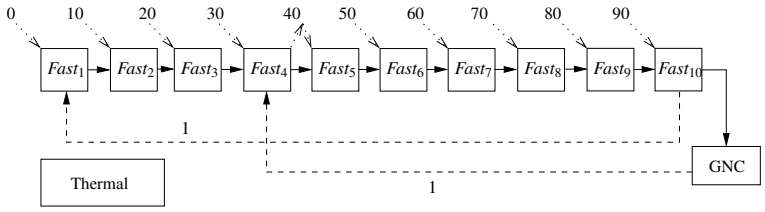


Figure 7.4: Real-time characterization of the *Simple* example (MTF = 100 ms)

task *Fast* uses an input that is sampled with a period of 10ms, starting at date 0, which imposes a release date of $(n-1)*10$ for $Fast_n$. Note that the release dates on $Fast_n$ constrain the start of *GNC*, because *GNC* can only start after $Fast_{10}$. However, we do not consider these constraints to be a part of the specification itself. Thus, we set the release dates of tasks *GNC* and *Thermal* to 0 and do not represent them graphically.

Only task $Fast_4$ has a deadline that is different from the default ∞. In conjunction with the 0 release date on $Fast_1$, this deadline represents an end-to-end constraint of 140ms on the *flow* defined by the chain of dependent task instances

$$Fast_1{}^n \rightarrow Fast_2{}^n \rightarrow \ldots \rightarrow Fast_{10}{}^n \rightarrow GNC^n \rightarrow Fast_4{}^{n+1}$$

for $n \geq 0$. Under the notation for task instances that was introduced in Section 3.4, this constraint requires that no more that 140ms separate the start of the n^{th} instance of task $Fast_1$ from the end of the $(n+1)^{th}$ instance of task $Fast_4$. Since the release date of task instance $Fast_1{}^n$ in the *MTF* of index n is 0, this flow constraint translates into the requirement that $Fast_4{}^{n+1}$ terminates 140ms after the beginning of the *MTF* of index n. This is the same as 40ms after the beginning of *MTF* of index $n+1$ (because the length of one *MTF* is 100ms). The deadline of $Fast_4$ is therefore set to 40ms.

7.4.1.4 Architecture-dependent constraints

The period, release dates and deadlines of Fig. 7.4 represent architecture-independent real-time requirements that must be provided by the control engineer. But architecture details may impose constraints of their own. For instance, assume that the samples used by task *Fast* are stored in a 3-place circular buffer. At each given time, *Fast* uses one place for input, while the hardware uses another to store the next sample. Then, to avoid buffer overrun, the computation of $Fast_n$ must be completed before date $(n+1)*10$, as required by the new deadlines of Fig. 7.5. Note that these deadlines can be both larger than the period of task *Fast*, and larger than the *MTF* (for $Fast_{10}$). By comparison, the specification of Fig. 7.4 corresponds to the assumption that input buffers are infinite, so that the architecture imposes no deadline constraint. Also note in Fig. 7.5 that the deadline constraint on $Fast_3$ is redundant, given the deadline of $Fast_4$ and the data dependency

between $Fast_3$ and $Fast_4$. Such situations can easily arise when constraints from multiple sources are put together, and do not affect the correctness of the scheduling approach.

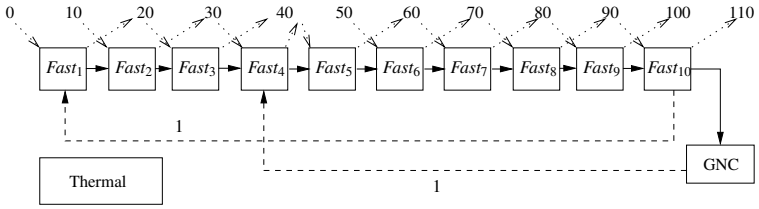


Figure 7.5: Adding 3-place circular buffer constraints to our example

7.4.2 Worst-case durations, allocations, preemptability

We also need to describe the processing capabilities of the various processors and the bus. More precisely:

- For each task $\tau \in T(D)$ and each processor $P \in Procs(Arch)$ we provide the *capacity*, or *duration* of τ on P. Following the model of Chapter 4, we assume this value is obtained through a worst-case execution time (WCET) analysis, and denote it $WCET(\tau, P)$. This value is set to ∞ when execution of τ on P is not possible.

- Similarly, for each data type $type(a)$ used in the specification, we provide a worst-case communication time estimate $WCCT(type(a))$ as an upper bound on the transmission time of a value of type $type(a)$ over the bus. We assume this value is always finite.

In accordance with the architecture description model of Chapter 4, the *WCET* information may implicitly define *absolute allocation constraints*, as $WCET(t, P) = \infty$ prevents t from being allocated on P. Such allocation constraints are meant to represent hardware platform constraints, such as the positioning of sensors and actuators, or designer-imposed placement constraints. *Relative allocation constraints* can also be defined, under the form of *task groups* which are subsets of $T(D)$. The tasks of a task group must be allocated on the same processor. Task groups are necessary in the representation of mode-dependent task durations, as presented in Section 7.3 (to avoid task migrations). They are also needed in the transformations of Section 8.1.

Our task model allows the representation of both preemptive and non-preemptive tasks. A task is non-preemptive when its execution should never be interrupted. Such is often the case for tasks that directly access hardware devices which must be left in a consistent state. Preemptability information is represented for each task τ by the flag $is\_preemptive(\tau)$. To simplify the presentation of our algorithms, we make in this thesis two simplifying assumptions:

- Bus communications are non-preemptable.
- The time costs associated with the preemption of preemptive tasks is negligible (zero). This assumption also covers the partition context switches.

7.4.3 Partitioning

Recall from Section 7.2.2 that there are two aspects to partitioning: the partitioning of the application and that of the resources (in our case, CPU time). On the application part, we assume that every task τ belongs to a partition $part_\tau$ of a fixed partition set $Part = \{part_1, \ldots, part_k\}$.

Also recall from Section 7.2.2 that CPU time partitioning, *i.e* the time windows on processors and their allocation to partitions can be either provided as part of the specification or computed by our algorithms. Thus, our specification may include window definitions which cover none, part, or all of CPU time of the processors. We do not specify a partitioning of the shared bus, but the algorithms can be easily extended to support a per-processor time partitioning like that of TTA [Rushby, 2001].

7.4.4 Syntax extensions

To account for all these extensions in the model, the syntax of the CG formalism must be modified. Non-functional information will be described using new tables and non-functional decorations added to the objects of existing tables.

First of all, our specifications must allow the definition of the set of partitions of the system. This is done using a partition table defined using the syntax of Fig. 7.6. The partition table must be declared before the blocks table. Each partition is defined by a unique index and by a name.

```
partitions_table  :=  Partitions Table partitions_list
patitions_list    :=
                      |partition partitions_list
partition         :=  partition_idx <Identifier>
partition_idx     :=  Partition:  <Int>
```

Figure 7.6: Syntax of the new partition table

The syntax of the block table, initially provided in Fig. 3.12, is extended to allow the definition of the non-functional properties associated with dataflow blocks. The modified syntax is provided in Fig. 7.7. Note that only the original definition of non-terminal `block_definition` is concerned. The extension allows the definition, for each dataflow block, of 4 optional pieces of information:

- It can be specified whether a block is **Preemtible** or **Not Preemptible**. By default a block is considered preemptible.
- A block can have a starting offset. This is specified using the keyword **Offset:**, followed by the integer value of the offset.
- A block can have a deadline offset, specified using the keyword **Deadline:** followed by the integer value of the deadline. Since we accept deadlines superior to the period of the tasks in our specifications, this integer may be superior to the *MTF* of the system.
- A block can belong to a partition. This is specified using the keywords **Belongs To:** followed by the index of the partition to which the block belongs.

```
block_definition  :=  (iport_list)->(oport_list) block_fun
                          extensions
extensions        :=  preemption offset deadline partitions
preemption        :=
                      |Preemptible
                      |Not Preemptible
offset            :=
                      |Offset:  <Int>
deadline          :=
                      |Deadline:  <Int>
partitions        :=
                      |Belongs To:  partition_idx
```

Figure 7.7: Extension of the nodes specification syntax for release dates, deadlines and preemptability

The architecture definition formalism introduced in Chapter 4 must also be extended. We need to allow the specification of the *MTF* of the system. This is an optional parameter of the specification. When it is not specified, we assume it can be synthesized. When it is specified, then the definition of each processor may include partitioning constraints under the form of pre-defined windows associated to the various partitions. Such windows should not be defined when the *MTF* definition is absent. Partitioning constraints are optional, and they may be partial. More precisely, the time windows associated with the partitioning constraints may cover none, all, or part of the duration of the *MTF*. However, all window definitions must define exclusive time intervals. The partitioning constraints are defined through an extension of the processor definitions.

The new architecture definition syntax, which replaces the one of Fig. 4.2, is given in Fig. 7.8. The modifications are the following:

- The introduction of the optional *MTF* declaration.
- The extension of the processor definition syntax to include an optional set of windows. Each window is declared between brackets **[]**, and is composed of:
  - A unique index.
  - A starting date, given by the keyword **Start:** followed by an integer.
  - A finishing date, defined using the keyword **End:** followed by an integer.
  - The partition to which the window is allocated.

```
archi           := Architecture mtf bus_table proc_table
mtf             :=
                   |MTF: <Int>
proc_table      := Processor Table proc_list
proc_list       :=
                   |proc proc_list
proc            := proc_idx <Identifier> comp_durations windows
comp_durations  := comp_duration
                   |comp_duration comp_durations
comp_duration   := Duration(fun_idx)= <Int>
                   |Duration(type_idx)= <Int>
windows         :=
                   |Windows Table windows_list
windows_list    := window
                   |window windows_list
window          := [ window_idx Start:  <Int>
                       End:  <Int> partition_idx ]
window_idx      := Window:  <Int>
```

Figure 7.8: Extension of the architecture description syntax to include MTF and windows declaration

7.4.4.1 Example

After the syntax extensions provided above, the industrial case study of Fig. 7.4 has the following textual specification. We have provided here the full specification file, including functional specification, architecture specification, and non-functional information. We assumed that the application is divided into three partitions: one that contains all the *Fast* tasks, one for the *GNC* task, and one for the *Thermal* task.

```
ClockedGraph

Global Definitions
```

```
Type Table

 Type:0 F_Type Simple
 type:1 GNC_Type

Function Table

 Function:0  Fast_1  (i:Type:0)->(o:Type:0)
 Function:1  Fast_2  (i1:Type:0 i2:Type:1)->(o:Type:0)
 Function:2  GNC     (i:Type:0)->(o:Type:1)
 Function:3  Thermal ()->()

Constant Table

 Constant:0 init_del_1 Type:0
 Constant:1 init_del_2 Type:1

Functional Specification

Variable Table

 Variable:0 Type:0 Single Assignment o@Block:0
 Variable:1 Type:0 Single Assignment o@Block:1
 Variable:2 Type:0 Single Assignment o@Block:2
 Variable:3 Type:0 Single Assignment o@Block:3
 Variable:4 Type:0 Single Assignment o@Block:4
 Variable:5 Type:0 Single Assignment o@Block:5
 Variable:6 Type:0 Single Assignment o@Block:6
 Variable:7 Type:0 Single Assignment o@Block:7
 Variable:8 Type:0 Single Assignment o@Block:8
 Variable:9 Type:0 Single Assignment o@Block:9
 Variable:10 Type:1 Single Assignment o@Block:10
 Variable:11 Type:0 Single Assignment v9_delayed@Block:11
 Variable:12 Type:1 Single Assignment v10_delayed@Block:12

Clock Table

 Clock:0 Tick Primitive

Partition Table

 Partition:0 Fast_part
 Partition:1 GNC_part
 Partition:2 Thermal_part

Block Table

 Block:0 Clock:0 (i Is Variable:11 On Clock:0) -> (o Is Variable:0)
         Function:0 Belongs To: Partition:0
 Block:1 Clock:0 (i Is Variable:0 On Clock:0) -> (o Is Variable:1)
         Function:0 Offset: 10 Belongs To: Partition:0
 Block:2 Clock:0 (i Is Variable:1 On Clock:0) -> (o Is Variable:2)
         Function:0 Offset:20 Belongs To: Partition:0
 Block:3 Clock:0 (i1 Is Variable:2 On Clock:0
                  i2 Is Variable:12 On Clock:0) -> (o Is Variable:3)
```

```
         Function:1 Offset:30 Deadline:40 Belongs To: Partition:0
 Block:4 Clock:0 (i Is Variable:3 On Clock:0) -> (o Is Variable:4)
         Function:0 Offset:40 Belongs To: Partition:0
 Block:5 Clock:0 (i Is Variable:4 On Clock:0) -> (o Is Variable:5)
         Function:0 Offset:50 Belongs To: Partition:0
 Block:6 Clock:0 (i Is Variable:6 On Clock:0) -> (o Is Variable:6)
         Function:0 Offset:60 Belongs To: Partition:0
 Block:7 Clock:0 (i Is Variable:6 On Clock:0) -> (o Is Variable:7)
         Function:0 Offset:70 Belongs To: Partition:0
 Block:8 Clock:0 (i Is Variable:7 On Clock:0) -> (o Is Variable:8)
         Function:0 Offset:80 Belongs To: Partition:0
 Block:9 Clock:0 (i Is Variable:8 On Clock:0) -> (o Is Variable:9)
         Function:0 Offset:90 Belongs To: Partition:0
 Block:10 Clock:0 (i Is Variable:9 On Clock:0) -> (o Is Variable:10)
         Function:2 Belongs To: Partition:1
 Block:11 Clock:0 () -> ()
         Function:3 Belongs To: Partition:2
 Block:12 Clock:0 (i Is Variable:9 On Clock:0) -> (v9_delayed is Variable:11)
         Delay Type:0 Depth:1 Init Const:0 Belongs To: Partition:0
 Block:13 Clock:0 (i Is Variable:10 On Clock:0) -> (v10_delayed is Variable:12)
         Delay Type:1 Depth:1 Init Const:1 Belongs To: Partition:0


Architecture
MTF: 100

Bus Table

Processor Table

Processor:0 Proc0 duration(Function:0)=4
                  duration(Function:1)=4
                  duration(Function:2)=20
                  duration(Function:3)=10

             Windows Table
             [Window:0 Start:0 End:4 Partition:0]
             [Window:1 Start:14 End:20 Partition:1]
```

The declaration of the first elements of the specification follow the initial syntax described in Chapter 3. These elements include:

- two types: one for the outputs of the *Fast* tasks, the other for the output of *GNC*,
- four functions: one for *GNC*, one for *Thermal*, and two for the *Fast* tasks, since the fourth *Fast* task has a particular behaviour, and takes two input variables instead of one,
- two constants for the initialization of the delay blocks,
- the primitive clock,

- thirteen variables that are necessary for the description of the dataflow.

The block table consists in 14 blocks: twelve are function blocks, and two are delays. The ten first blocks correspond to the *Fast* tasks, Block:10 corresponds to *GNC* and Block:11 to the *Thermal* task. Each *Fast* block is specified with a starting offset, except for the first one, since the offset 0 is set by default. Block:3 is the only block to feature a deadline.

The architecture section of the specification defines a monoprocessor partitioned architecture. In this example, the *MTF* of the application is set to 100 time units. The four functions declared in the previous functional specification are allowed to execute on the processor, and two time windows are declared in the specification. The first one is allocated to Partition:0. At each execution cycle, this window starts at date 0 and ends at date 4: the 4 first time units of every cycle are dedicated to Partition:0. The second window lasts from date 14 to date 20 and is reserved for Partition:1.

Note that in this example, the specified windows do not entirely cover the *MTF*. During the scheduling phase, the algorithms will have to respect these specified windows, but can also synthesize new partition windows in the spare time intervals (from date 4 to 14, and from date 20 to date 100) if necessary. As said before, it is possible to declare all, none, or part of the time windows in the specification.

7.5 Conclusion

In this chapter we described the extensions we made to the Clocked Graphs language, in order to allow the modelling of complex non-functional requirements in our specifications. All the requirements we model come from an aerospace case study that was provided by Airbus DS. We modelled here the simple version of this case study. The full application model shall be provided in the next chapter. We defined a particular time-triggered execution framework. Particular features of this framework are ARINC653 temporal partitionning and pre-computed preemptions. We have adapted our specification formalism to this time-triggered framework. In particular, any task can be declared with a starting offset and a deadline, which allows the modelling of periodic data acquisition, of end-to-end latency constraints and of architecture-dependent constraints. Moreover, tasks can be allocated to a particular partition. Finally, we also extended the architecture declaration syntax to allow the specification of time windows on the various processors. These extensions allow the modelling of all non-functional requirements of our case study. The resulting specification is the input of the mapping and code generation algorithms of the next chapter.

Chapter 8

Real-Time scheduling under complex non-functional requirements

Contents

The previous chapter focused on the formal modeling of systems with complex non-functional specification. In this chapter we provide algorithms that take such specifications and automatically transform them into correct-by-construction running implementations.

The transformation is performed in 2 steps: the first one performs the off-line allocation and scheduling of the computations and communications onto the various resources of the platform. This mapping phase produces implementation models (*i.e.* scheduling tables) that are functionally correct and respect the non-functional requirements. This first phase is described in Sections 8.1 and 8.2, which introduce the multi-processor off-line allocation and scheduling algorithms, and in Section 8.3, which details an optimization technique needed to improve the real-time properties of the resulting scheduling tables.

The second step of the transformation is code generation. In our case, it allows fully automated generation of ARINC 653-compliant implementation code which includes: the

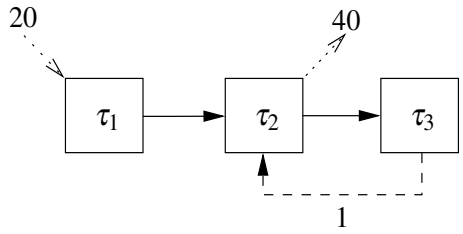


Figure 8.1: Small example for illustrating the potential gain when the recomputation of release dates happens before removing the delayed arcs

OS configuration, the elaboration code of each partition, and the inter-partition communication code. The code generation phase includes some optimizations aimed at minimizing the number of tasks in the final implementation. Code generation is described in Section 8.4.

8.1 Removal of delayed dependencies

The first step in our scheduling approach is the transformation of the initial task model specification into one having no delayed dependency. This is done by a modification of the release dates and deadlines for the tasks related by delayed dependencies, possibly accompanied by the creation of new helper tasks that require no resources but impose scheduling constraints. Doing this will allow in the next section the use of simpler scheduling algorithms that work on acyclic task graphs.

The first part of our transformation ensures that delayed dependencies only exist between tasks that will be scheduled on the same processor, so that associated communication costs are 0. Let $\delta \in \Delta(D)$ and assume that $src(\delta)$ and $dst(\delta)$ are not forced by absolute or relative allocation constraints to execute on the same processor. Then, we add a new task τ^δ to D. The source of δ is reassigned to be τ^δ, and a new (non-delayed) dependency is created between $src(\delta)$ and τ^δ. Relation $EX(D)$ is augmented to place τ^δ in exclusion with all tasks that are exclusive with $src(\delta)$, and at the same depths. Task τ^δ is assigned durations of 0 on all processors where $dst(\delta)$ can be executed, and ∞ elsewhere. Finally, a task group is created containing τ^δ and $dst(\delta)$.

The second part of our transformation performs the actual removal of the delayed dependencies. It does so by imposing for each delayed dependency δ that $src(\delta)$ terminates its execution before the release date of $dst(\delta)$. This is done by changing the deadline of $src(\delta)$ to $r(dst(\delta)) + depth(\delta) * MTF(D)$ whenever this value is smaller than the old deadline.

Delayed dependence removal is now completed, but we also perform as part of this implementation step a re-computation of task deadlines, following the approach of Blazewicz

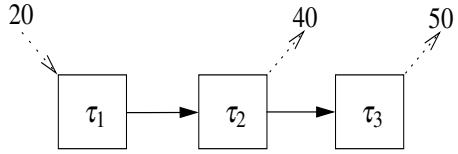
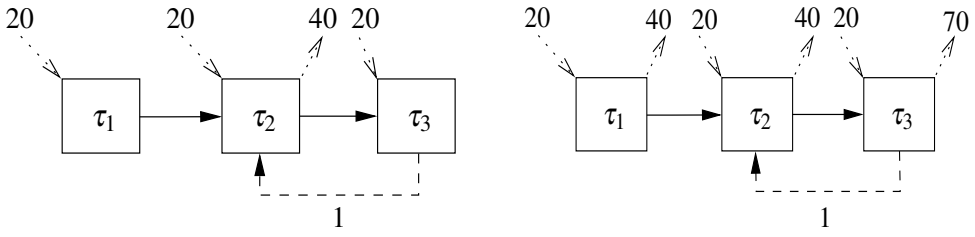


Figure 8.2: Removing the delayed arcs before recomputing the release dates in the example of Fig. 8.1



(a) Recomputing the release dates (b) Recomputing the deadlines

Figure 8.3: Performing the recomputation of release dates and deadlines before removing the delayed arcs in the example of Fig. 8.1

[Blazewicz, 1977], also used by Chetto *et al.* [Chetto et al., 1990]. The deadline of each task is changed into the minimum of all deadlines of tasks depending transitively on it (including itself). One can achieve an increased flexibility for the scheduling process by recomputing the release dates and deadlines even before removing the delayed arcs, using a fixed-point computation. More generally, the recomputation of release dates and deadlines by means of a fixed-point computation can be useful *before* the removal of delayed arcs to provide for longer deadlines. For example consider the example of Fig. 8.1 where three tasks τ_1, τ_2 and τ_3 communicate in the following fashion: there is a direct data dependency between τ_1 and τ_2, and between τ_2 and τ_3, and a delayed dependency between τ_3 and τ_2. The *MTF* is fixed to 50 ms, τ_1 has a release date of 20 ms, and there is an end-to-end constraint of 90 ms to perform the following flow: $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_2$. This example is theoretical, but illustrates a situation that can be encountered inside a subpart of a real system specification . If the delayed arc is removed prior to recomputing the release dates and deadlines of the tasks, the deadline that will be affected to τ_3 is equal (following the formula detailed before) to the value of the *MTF*. τ_3 will thus have a deadline of 50 ms. This is illustrated in Fig. 8.2. Now, if the recomputation of the release dates and deadlines happens before removing the arcs, since τ_1 has a release date of 20 ms, and a direct dependency exists between τ_1 and τ_2, τ_2 will be assigned a release date of 20 ms. In turn, τ_3 will also be affected a release date of 20 ms for the same reason. Now, when computing the deadline that must be affected to τ_3, we obtain the value of the *MTF* added to 20 ms, since it is the release date of τ_2. τ_3 will thus have in this case a deadline

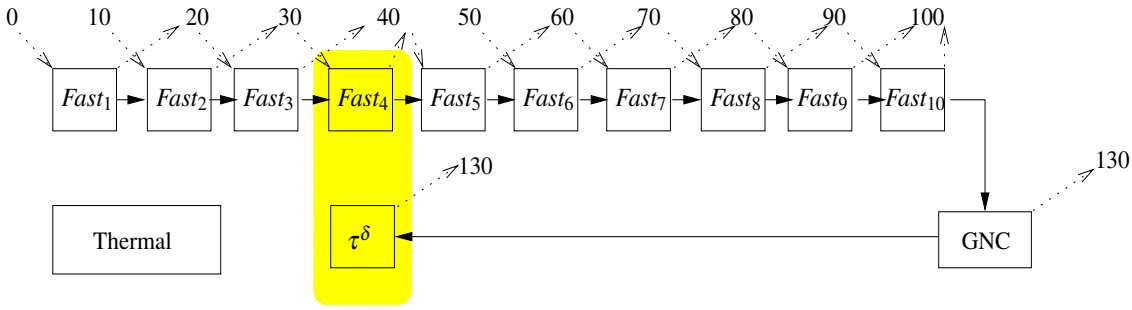


Figure 8.4: Delay removal result for the example in Fig. 7.5

of 70 ms. This widens the exploration space for feasible solutions during the scheduling phase, since it allows τ_3 to finish its computation after the end of the MTF, that is to say at the begining of the next computation cycle. We illustrate this recomputation method in Fig. 8.3.

The result of all these transformations for the example in Fig. 7.5 is pictured in Fig. 8.4. We have assumed that tasks $Fast_4$ and *GNC* can be allocated on different processors, and thus a helper task is needed. The new task group formed of tasks τ^δ and $Fast_4$ is represented by the yellow box. We assume that all tasks $Fast_n$ must be executed on the same processor, due to an absolute allocation constraint. This is why no helper task is needed when removing the delayed dependency from $Fast_{10}$ to $Fast_1$.

Note that this transformation is another source of deadlines larger than the periods. Also note that all the transformations described above are linear in the size of the number of arcs (delayed or not), and thus very fast.

8.2 Offline real-time scheduling

On the transformed task models we apply an *offline scheduling* algorithm whose output is a system-wide *scheduling table* defining the allocation of processor and bus time to the various computations and communications. The length of this table is equal to the *MTF* of the task model.

Our offline scheduling algorithm is a significant extension of the one proposed by Potop *et al.* [Potop-Butucaru et al., 2009]. New features are the handling of *preemptive* tasks, *release dates* and *deadlines*, the *MTF*, and the *partitioning* constraints. The handling of conditional execution and bus communications remains largely unchanged, which is why we do not present these features in detail. Instead, we insist on the novelty points, like partitioning or the use of a deadline-driven criterion for choosing the order in which tasks are considered for scheduling. The deadline-driven criterion was inspired by existing work by Blazewicz [Blazewicz, 1977] and by Chetto *et al.* [Chetto et al., 1990]. By comparison with Blazewicz's works, our algorithm takes into account the *MTF*, the

partitioning constraints, and *conditional execution*.

8.2.1 Basic principles

As explained earlier, our algorithm computes a scheduling table. This is done by associating to each task:

- A *target processor* on which it will execute,
- A set of *time intervals* that will be reserved for its execution. Since we now want to allow a preemptive behaviour, a task may need to reserve several non-contiguous time intervals for its computation,
- A date of *first start*.

The conditional execution paradigm of our task model requires the use of *conditional reservations*: as explained before, multiple time intervals are allowed to overlap if the execution conditions of the corresponding tasks are mutually exclusive, as defined by relation $EX(D)$. A similar reservation model is used for the bus.

Given a task system D, a scheduling table S for D, and $\tau \in T(D)$, we shall denote with $S.proc(\tau)$ the target processor of τ, with $S.start(\tau)$ the date of first start, and with $S.intervals(\tau)$ the set of time intervals reserved for τ. A time interval i is defined by its start date $start(i)$ and end date $end(i)$. It is required that the intervals of $S.intervals(\tau)$ are disjoint, and that the start date of one of them is $S.start(\tau)$ **mod** $MTF(D)$.

Recall from Section 7.2.3 that the execution model is as follows: The n^{th} instance of task τ will start (modulo conditional execution) at date $S.start(\tau) + (n-1) * MTF(D)$. Execution of the task is confined to its reserved time slots (it is suspended between such slots).

The choice of processor, start date, and intervals by the scheduling algorithm must ensure that:

- The intervals reserved for a task allow the complete execution of a task instance before the next instance is started.
- Intervals reserved for two tasks can only overlap if the two tasks belong to the same partition and have exclusive execution conditions. Moreover, an interval allocated to task τ of a partition *part* cannot overlap with windows allocated to other partitions.
- The task and communication execution order imposed by the direct and delayed dependencies is respected.
- The release date of a task precedes its start date, and deadline constraints are respected.

8.2.2 Scheduling algorithm

The scheduling algorithm, whose top-level routine is Algorithm 4, follows a classical list scheduling approach. It works by iteratively choosing a new task to schedule and then scheduling it along with the necessary communications.

Algorithm 4 scheduler_driver

```
Input:  D : dependent task system
        Arch: architecture description
        input_schedule : schedule (complex data structure comprising a scheduling table
                         and a representation of the free intervals)
Output: result_schedule : schedule
  result_schedule := input_schedule
  while T(D) ≠ ∅ and result_schedule ≠ invalid_schedule do
    τ:= choose_task_to_schedule(D)
    D := remove_task(τ, D)
    new_schedule := invalid_schedule
    new_cost := ∞
    for all processor P in Archi do
      if WCET(τ,P) ≠ ∞ and group_ok(τ,P,result_schedule) then
        temp_schedule:= schedule_task_on_proc(τ,P,result_schedule,D)
        if temp_schedule≠invalid_schedule then
          temp_cost := cost_function(temp_schedule)
          if temp_cost < new_cost then
            new_schedule := temp_schedule
            new_cost := temp_cost
          end if
        end if
      end if
    end for
    result_schedule := new_schedule
  end while
  return result_schedule
```

Among the not-yet-scheduled tasks of whom all predecessors have been executed, function *choose_task_to_schedule*, not provided here, returns one of minimal deadline. If several tasks satisfy this criterion, then we determine for each of them the earliest start date in the current scheduling state, and we choose one with maximal earliest start date. For instance, the tasks in Fig. 8.4 are chosen in the order $Fast_1, \ldots, Fast_{10}, GNC, \tau^{\delta}, Thermal$.

The body of the **while** loop allocates and schedules a single task τ, along with the communications needed to gather the input data of τ. It works by attempting to allocate and schedule τ on each of the processors that can execute it. Function *group_ok* determines if the relative allocation constraints and the current scheduling state allow τ to be allocated on P.

Among all the possible allocations of τ, Algorithm 4 chooses the one resulting in

Algorithm 5 schedule_task_on_proc

Input: τ : task to schedule
P : processor on which to schedule
$input\_schedule$: schedule (before adding τ)
D : dependent task system
Output: $result\_schedule$: schedule (after adding τ)

```
(result_schedule, d_earliest) := schedule_bus_communications(τ, P, input_schedule, D)
if d_earliest > d(τ) then
    result_schedule := invalid_schedule
else
    needed_duration := WCET(τ, P)
    interval_set := ∅
    failure := false
    while needed_duration > 0 and not failure do
        /*Search for a new interval*/
        (interval, result_schedule):= get_first_interval(result_schedule, needed_duration,
                                                         d(τ), d_earliest, part_τ, is_preemptive(τ))
        if interval = invalid_interval then
            /*No interval found, attempt to move the search into the next MTF*/
            if deadline ≤ MTF then
                failure := true
            else
                deadline:=deadline-MTF(D)
                d_earliest:=0
            end if
        else
            /*Good interval found*/
            d_earliest:=end(interval)
            needed_length:=needed_length-len(interval)
        end if
    end while
    if failure then
        result_schedule := invalid_schedule
    end if
end if
return result_schedule
```

a -partial- schedule of minimal cost. In our case, *cost_function* chooses the schedule ensuring the earliest termination of τ. If scheduling is not possible on any of the processors Algorithm 4 returns *invalid_schedule* to identify the failure.

The mapping of a task τ onto a processor P is realized through a call to *schedule_task_on_proc*, whose code is provided in Algorithm 5. This algorithm follows a classical ASAP (as soon as possible) scheduling strategy. The scheduling is done as follows. First, the transmission of data needed by τ and not yet present on P is scheduled for communication on the bus using function *schedule_bus_communications*. This function schedules the transmission of both input data of τ and state variables needed to compute the execution condition of τ. We do not provide the function here, interested readers being directed to [Potop-Butucaru et al., 2009].

Once communications are scheduled, we attempt to schedule the task at the earliest date *after* the date where all needed data is available. If this is not possible without missing the deadline, *invalid_schedule* is returned to identify the failure.

Looking for free intervals for the task to schedule is done by function *get_first_interval*, not provided here because it is too complex and because it requires explicit manipulations of Boolean predicates instead of the abstract exclusion relations EX.For non-preemptive tasks, this function looks for the first free interval long enough to allow the execution of the task and satisfying the execution condition and partitioning constraints. For preemptive tasks, this function may be called several times to find the first free intervals satisfying the execution condition and partitioning constraints and of sufficient cummulated length to cover the needed duration. When unable to find a valid interval, this function returns *invalid_interval*. For instance, consider the scheduling of the example of Fig. 3.18 to obtain the scheduling table of Fig. 7.1, and assume that all tasks were scheduled save G. Algorithm 4 will first attempt to schedule G on processor $P1$. We assume that the partitioning of the processors is fixed as described in Section 7.2.3. Therefore, we are looking for time intervals where $P1$ is not used inside the *MTF* windows w_2 and w_4. Search starts at the earliest start date of G, which is 29 (after F_2 terminates and w_4 starts). Given that G is preemptible and that the execution condition of G is exclusive with the execution conditions of R'_2 and G', the first free interval is [29,39]. At the end of this interval G must be preempted and resumed in the next execution iteration of the *MTF* (if the other constraints allow it). In our example, resumption is possible at date 12. It is not possible at date 9, because the execution condition of R'_1 (the instance of the next execution cycle) is not exclusive with that of G.

When the partitioning of the *MTF* is provided (fully or partially), this information is transmitted to Algorithm 4 through parameter *input_schedule*. This initial state contains no task allocation, but may constrain the free interval set due to partitioning.

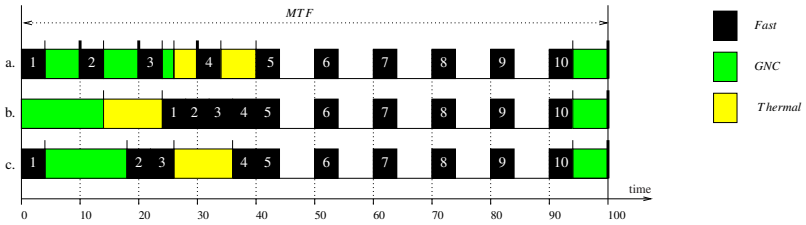


Figure 8.5: Scheduling result for the examples in Figures 7.4, 7.5, and 8.4 on a single-processor architecture (a). The result of applying the post-scheduling slot optimization of Section 8.3 for the example in Fig. 7.4 (b) and for the example in Figures 7.5 and 8.4 (c).

8.2.3 Complexity and optimality considerations

The complexity of Algorithm 4 is linear in the number of tasks and in the number of processors in the architecture, and the complexity of Algorithm 5 is sub-linear in the number of tasks (which bounds the number of calls to *get_first_interval*). But the real complexity of the scheduling algorithm is hidden inside function *get_first_interval*, which is called by Algorithm 5. This function maintains a representation of free intervals. When working on dependent task systems without execution conditions and modes, its complexity is bounded by the number of tasks in the system, making for a globally polynomial complexity. But when execution conditions are taken into account, the representation of free intervals can grow in size exponentially. Moreover, determining if the execution condition of a free interval is compatible with that of a task requires solving instances of the Boolean satisfiability problem (of NP complete complexity). In practice, however, these execution conditions remain quite simple, and both SAT instances and the representation of free intervals remain small. Scheduling time was negligible in all our tests.

From an optimality perspective, our scheduling algorithm is a safe heuristic that never provides an incorrect result, but may fail when a solution exists because it never reconsiders an allocation or scheduling decision done for a task. Simple extensions of this algorithm would allow it to be optimal under restrictive hypotheses[1]. However, in the absence of extensive benchmarks it is unclear how optimality under restrictive hypotheses helps when scheduling problems involving a multiprocessor architecture, a complex control structure, and complex non-functional requirements. We therefore prefered here a compilation-like approach using low-complexity algorithms that can be easily tailored to take into account the previously-mentioned functional and non-functional properties.

When evaluating the software pipelining techniques described in Chapter 6, we considered the use of Integer Linear Programming (ILP) constraint solving engines to find

[1] Single processor, all tasks preemptive, no imposed partitioning, no execution conditions, zero communication and preemption costs. This is the same class of systems handled by Chetto *et al.* [Chetto et al., 1990] and Pagetti *et al.* [Pagetti et al., 2011]. Obtaining optimality for this class of problems requires performing the off-line deadline-driven scheduling for more than one execution cycle of the specification, and thus increases complexity, as explained in [Leung and Merrill, 1980].

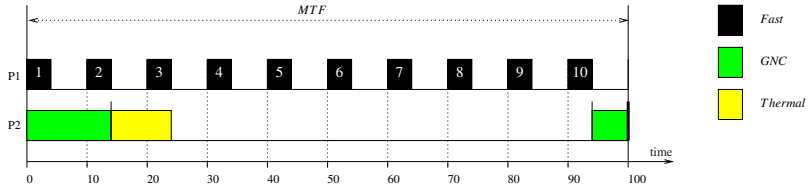


Figure 8.6: Scheduling result for the example in Fig. 7.4 on a two-processor architecture with zero communication costs

optimal solutions to scheduling problems that are simpler than the ones considered here (they involved no conditional execution, no partitioning, and none of the model extensions presented in Chapter 7). As pointed out in footnote 3 of Section 6.5, this technique doesn't scale beyond simple systems when both allocation and scheduling are an output to the scheduling problem. We thus considered that it was not suited for the evaluation of our heuristic.

8.2.4 Scheduling results

We have implemented our scheduling algorithms into a tool, which allowed us to schedule our models of the space launcher application[2]. We have started our evaluation with the reduced model defined by the dependent task system of Fig. 7.4. In our model, all tasks were preemptible.

Our first test performed the scheduling of this task system on an architecture with one processor (P). The durations of the tasks are $WCET(Fast, P) = 4$, $WCET(GNC, P) = 20$, and $WCET(Thermal, P) = 10$. We assumed that the 3 tasks have each its own partition. We also assumed that the partitioning of the *MTF* was not constrained (it was fully synthesized by our tool). The result of the scheduling phase is provided in Fig. 8.5(a). Like in our example of Section 7.2.3, the partitioning of the *MTF* into windows is represented by solid vertical bars. Partition changes are set only at the beginning of an interval when this interval is allocated to a task and the previous allocated interval belongs to another partition. In our example, there are 11 partition changes (counting the final one at the end of the *MTF*). Following the graphical convention of Section 7.2.3, thicker partition separation bars represent partition changes where a task undergoes a precomputed preemption. There are 4 of them in our example. Note how function *get_first_interval* loops over the *MTF* in its search for reservations for tasks *GNC* and *Thermal*. For instance, the scheduling of *GNC* is realized after the one of the $Fast_i$ tasks, and its earliest start date is given by the end of $Fast_{10}$. After reserving the interval [95,100], the search loops over and reserves successively intervals [5,10], [15,20], and [25,26], in order to cover $WCET(GNC, P)$.

In our second test, we scheduled the same dependent task system on an architecture

[2] And some toy examples, like the one in Section 7.2.3.

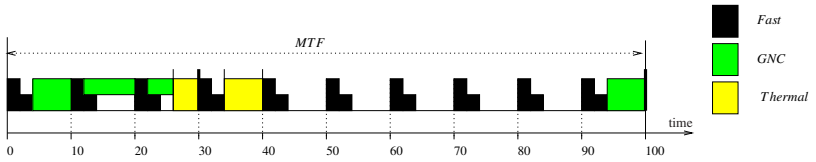


Figure 8.7: Scheduling result for the two-mode example of Fig. 7.3 on a single-processor architecture

with two processors ($P1$ and $P2$) where inter-processor communication takes no time, such as in a shared memory system, when memory access interferences are taken into account in the WCET analysis. We assumed that the two processors are identical and that the durations of the tasks on each processor are the ones provided above for the single-processor. We assumed there are no allocation constraints. The resulting scheduling table is provided in Fig. 8.6. Only 4 partition changes remain (counting the mandatory 4 at the end of the *MTF*), and no pre-computed preemptions.

The third test considered the dependent task system of Fig. 7.3, which features mode-dependent task durations. Scheduling is done here on a single processor P. We assumed that the durations of the various tasks in the example are: $WCET(F1,P) = 20$, $WCET(F2,P) = 20$, $WCET(GNC1,P) = 60$, $WCET(GNC2,P) = 120$, and $WCET(Thermal,P) = 100$. We also assumed that tasks *Fast* ($F1$ and $F2$) and *GNC* belong to one partition, and that task *Thermal* belongs to another. Again, we assumed that partitioning is fully synthesized. The resulting scheduling table is pictured in Fig. 8.7. This example shows how taking into account execution conditions (even in the restricted form allowed by our space launcher application) allows double reservation and (in our example) ensures schedulability.

Finally, we have been able to schedule the large-scale model provided by Astrium. We have pictured in Fig. 8.8 its architecture and the allocation of tasks to partitions and processors. The architecture is formed of 4 processors connected by a broadcast bus. There are 13 tasks divided in 7 partitions. Our figure also provides the periods and durations of the tasks. The *MTF* is 100ms. The tasks are statically allocated to processors. The direct communications between tasks are represented in Fig. 8.8 with directed arcs.

As explained in Section 7.3, the task periods and the information flows definitions allow the construction of a fully deterministic functional specification by using Giotto-like communication rules[3]. For this large example, no manual modification of the dependencies was needed.

[3]The exact rules are the following: direct communication is possible only between two tasks having harmonic periods. In this case, the dependencies between task instances are determined as if the two tasks form a Giotto *mode* of period equal to the largest of the periods of the two tasks (*cf.* [Henzinger et al., 2000], figure 7). The only exception to this rule is when the two tasks belong to the same partition. In this case, if a dependency exists from the fast task to the slow one, then it is realized inside the *round*, between the first instance of the fast task and the instance of the slow task.

Algorithm 6 PostSchedulingOptimizationForMonoprocessor

Input: *input_schedule* : schedule
Output: *result_schedule* : schedule (optimized)

```
/* Initialisation of the interval lists. */
inputIntervalList := SortReservedIntervalsByIncreasingStartDate(input_schedule)
resultIntervalList := empty_list
repeat
  I1 := GetLastInterval(inputIntervalList)
  inputIntervalList := RemoveLastInterval(inputIntervalList)
  resultIntervalList := InsertInterval(I1,resultIntervalList)
  /* Find in inputIntervalList the last interval of the same partition as I1 */
  I2 := FindLast(inputIntervalList,GetPartition(I1))
  if I2 ≠ not_found then
    /* Partition inputIntervalList around the start date of I2 (I2 is in
       neither interval). */
    (intervalsBeforeI2, intervalsAfterI2) := Partition(inputIntervalList,GetStartDate(I2))
    /* If there are no other intervals between I2 and I1, there is no partition change
       and therefore no need to move intervals. */
    if intervalsAfterI2 ≠ empty_list then
      /* Attempt to move I2 after the intervals of intervalsAfterI2,
         if necessary also moving the intervals of intervalsAfterI2 earlier. */
      /* By how much can I2 be delayed if the intervals of
         intervalsAfterI2 are removed from the scheduling table? */
      maxI2Delay := MaxI2Delay(I2,resultIntervalList,GetLength(input_schedule))
      /* By how much can the intervals of intervalsAfterI2 be advanced
         if I2 is removed from the scheduling table? */
      maxAdv := MaxAdvance(intervalsAfterI2,intervalsBeforeI2,GetLength(input_schedule))

      if start(I2) + maxI2Delay ≥ max_{i∈intervalsAfterI2} GetEndDate(i) + maxAdv then
        /* It is possible to move I2 after intervalsAfterI2.
           Perform the move, interval by interval. */
        I2 := MoveInterval(I2,maxI2Delay)
        intervalsAfterI2 := {MoveInterval(i,maxAdvance)|i ∈ intervalsAfterI2}
        inputIntervalList := Concatenate(intervalsBeforeI2,intervalsAfterI2)
        inputIntervalList := Append(inputIntervalList,I2)
      end if
    end if
  end if
until inputIntervalList = empty_list
/* Rebuild the scheduling table. */
resultSchedule := BuildScheduleFromList(resultIntervalList)
```

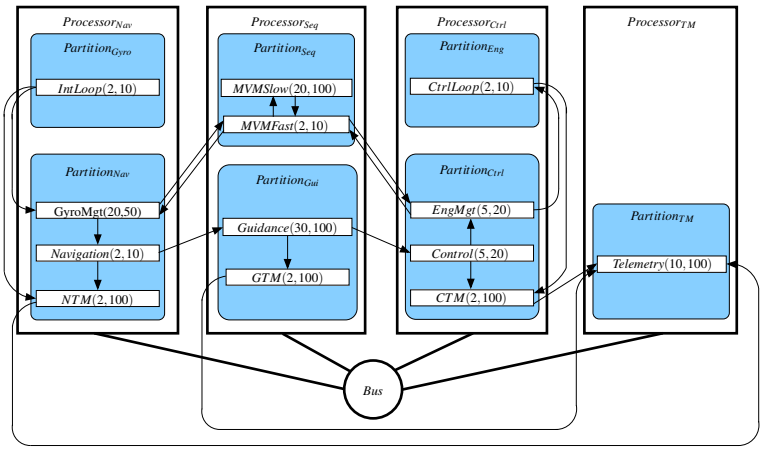


Figure 8.8: Architecture, partitioning and task allocation for the large-scale model of the space launcher. The two integers inside each task define its duration and period (in milliseconds).

The specification defines ten flows (some of them ending with the same task), with end-to-end latency constraints ranging from 100ms to 450ms. Translated into our formalism they amount to the following constraints on the deadlines of tasks (after hyperperiod expansion): $d(GyroMgt_2) = 75$, $d(EngMgt_2) = 75$,$d(MVMSlow) = 50$, $d(CtrlLoop_{10}) = 90$, $d(Telemetry) = 100$,

Our tool built a correct scheduling table for this example, meaning that implementation is possible without manual changes to the dependencies. The 4 processors are loaded respectively at 82%, 72%, 72%, and 10% (the 4th processor is dedicated to telemetry). The bus is loaded at 81%.

8.3 Post-scheduling slot minimization

The algorithm of the previous section follows a classical ASAP deadline-driven scheduling policy, which is good for ensuring schedulability.

However, resulting schedules may have a lot of unneeded preemptions and, most importantly, partition changes which are notoriously expensive. For instance, the scheduling table of Fig. 8.5(a) features no less than 11 partition changes.

To reduce the number of partition changes, we perform a heuristic post-scheduling optimization of our scheduling tables. The algorithm we use in case of mono-processor architectures featuring no conditional execution is Algorithm 6. Intuitively, the transformation we apply is the following: the scheduling table is traversed from end to the beginning. Whenever two intervals I_1 and I_2 allocated to tasks of the same partition are separated by intervals of other partitions, we attempt to group I_1 and I_2 together. Assuming I_2 starts before I_1, our technique attempts to move I_2 just before I_1 while moving

all operations between I_2 and I_1 to earlier dates. The transformation step is only performed when the resulting schedule respects the correctness properties of Section 8.2.1. The complexity of this transformation is quadratic in the number of windows in the initial schedule.

The result of applying this algorithm on our simple example is provided in Fig. 8.5(b). The number of partition changes is significantly reduced (from 11 to 3). Note that all instances of task *Fast* inside an MTF are grouped in a single window inside the MTF. This is possible under the release date and deadline constraints of the dependent task set of Fig. 7.4, where no architectural constraints are taken into account (we assumed that input buffers are infinite).

When the input buffering constrains are taken into account in the dependent task system of Fig. 7.5, there is no change in the output of the scheduling algorithm. However, the supplementary constraints limit the efficiency of slot optimization, leading to the scheduling table of Fig. 8.5(c), which has 6 partition changes. Note that our technique also reduces the number of preemptions. In our example, we move from 4 preemptions in the unoptimized example to only 1.

We have extended the previous algorithm to deal with applications featuring conditional execution (and modes) running on multi-processors, but yet we have not tested these extensions enough to present them in this thesis.

8.4 Partitioned time-triggered code generation

At this point, the only missing piece of our compilation chain is the code generator that takes as input the scheduling table generated by the algorithms of the previous sections and synthesizes the configuration files and the C code of the partitions for an ARINC 653-compliant operating system. Our work targeted the ARINC 653-compliant POK operating system[POK], which has the advantage of being open-source. The work of this section was conducted in collaboration with a French startup called OpenWide. My work was here to provide the guidelines for the code generation, which were implemented and tested by the engineers of OpenWide. The code generation scheme works for distributed targets. However, generated code currently runs only on single-processor partitioned targets (more work is needed at low levels to take into account network card drivers).

The definition of an ARINC 653-compliant and efficient code generation scheme proved unexpectedly difficult. In this section we shall present two of the main challenges and the solutions we proposed to them.

8.4.1 Automatic synthesis of communication channels

The ARINC 653 standard enforces a strict spatial and temporal isolation between its partitions, in order to ensure that all interactions between partitions happen only through explicitly defined *communication channels*. The temporal isolation is enforced through the use of a partitioned time triggered scheduler where the time windows allocated to the partitions do not overlap. Temporal isolation ensures that variations in the run-time behavior of the tasks of one partition have no effect on the behavior of the tasks of the other partitions (so that timing variations cannot introduce hidden communication channels). Spatial isolation consists in ensuring that two partitions share no memory zone or device that could allow the definition of hidden communication channels.

Thus, communication channels are the only way of exchanging information between partitions. Channels are point-to-multipoint communication devices with one source partition and one or more destination partitions. Channels connect to partitions onto ports. Both channels and ports must be explicitly defined in the OS configuration file. Once this is done, the C code of the partitions can use calls to ARINC 653 primitives to send and receive data through the channels.

In our approach based on off-line scheduling, the calls to these send and receive primitives must be explicitly considered as operations and statically scheduled. This is due not only to the fact that these operations may take time, but also (and mainly) to the fact that calls to send operations must be synchronized with the reservations made on the buses.

But our functional specifications have no such send and receive operations. We therefore need a method of modifying a Clocked Graphs specification *prior to scheduling* through the insertion of explicit send and receive operations whenever communications cross the boundaries between partitions. To illustrate the functioning of the method we developed we consider an example. Fig. 8.9 is the graphical representation of the initial Clocked Graphs specification of a simple application, given in the graphical CG formalism which we used in Chapter 3 (cf. Page 53)[4]. This application is composed of four functional blocks:

- block $F1$, which produces variable $v1$,
- block $F2$, which produces variable $v2$,
- block $F3$, which produces variable $v3$,
- block $F4$, which takes as input $v2$ when $v1$ is true, and $v3$ otherwize.

[4]Given that an important step of this transformation consists in taking into account the execution clocks of the functional blocks, we believe the presentation to be clearer if we use this representation formalism instead of the task set formalism defined at the end of Chapter 3 which does not directly displays the information relative to the clocks (cf. Page 62)

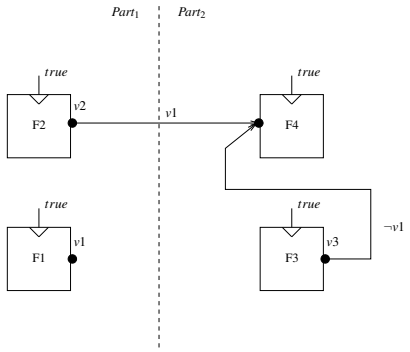


Figure 8.9: Functional specification before the automatic channel generation

The four blocks of the specification are active in any execution cycle, but the two data-flow arcs have clocks different from the base clock. Blocks $F1$ and $F2$ belong to partition $Part_1$, and $F3$ and $F4$ belong to $Part_2$.

In order to comply with the ARINC 653 standard, we must add two communication channels to this specification. Indeed, variable $v2$ is an input to block $F4$ of $Part_2$, but is produced in $Part_1$, and variable $v1$ is needed in $Part_2$ to allow the computation of the clock of $v3$.

The result of adding the needed communication operations is pictured in Fig. 8.10. Adding the channel for $v2$ is done in the following way: we start by adding two blocks called *Channel Write* and *Channel Read* to the specification. The block *Channel Write* belongs to partition $Part_1$. It has the same clock as the producer of $v2$, and thus is activated in all cycles. The block *Channel Read* belongs to $Part_2$. It activates on the same clock as the *Channel Read* block. Once these blocks are created, we remove the communication between $F2$ and $F4$, and add the following communications: $v2$ is connected to the input port of *Channel Write*, on the clock of $F2$ (true). $p1$ is connected to the input port of *Channel Read*, still on the clock of $F2$. Finally, $p2$ is connected to the input port of $F4$, under the condition that $v1$ is true (the initial communication condition).

Now, since $v1$ is also produced in $Part_1$, we need to create a second channel, to make its value available in $Part_2$. The creation process is the same, except that since $v1$ is only needed to compute clocks in $Part_2$, the output port of the *Channel Read* block is not connected to any input port. Instead, we must modify the clocks used in $Part_2$, by replacing any occurence of $v1$ by a reference to $p4$, the output port of the *Channel Read* block.

Our channel generation process works for all correct input specifications. Performing this transformation allows the scheduling process to actually take into account these operations (and their durations). Actual code generation happens after the scheduling phase and takes as input scheduling tables containing the communication operations. Code gen-

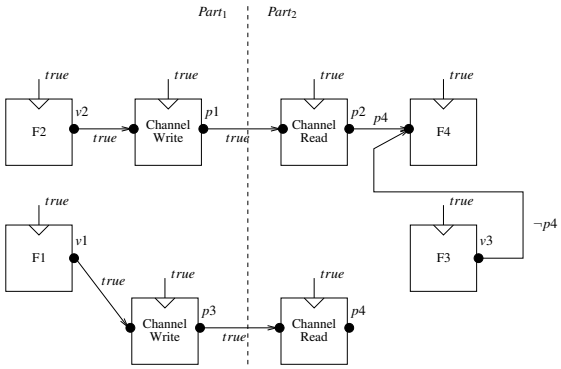


Figure 8.10: Functional specification after the channels are generated

eration uses the *queuing ports* provided by the ARINC 653 standard,[5] which rely on FIFO queues to store the messages until they are consumed. This choice was motivated by the fact that using queuing ports guarantees that no data loss will occur, even when sending multiple variables through the same port before reading them. It thus guarantees functional determinism in the inter-partition communication, regardless of the temporal properties of the application. We also use the multiport channels mandated by ARINC 653 to reduce the number of generated channels.

8.4.2 Process number minimization

We have previously explained that synchronous programs and dependent task models are formally related by a simple abstraction relation. While emphasizing this formal similarity we have hidden a significant difference that exists between the ways the two classes of formalisms are used for system specification in current industrial practice:

- Synchronous languages are used to define complex behavioral descriptions that include the low-level aspects in order to allow the generation of the code of tasks. Intuitively, the data-flow synchronous specification of a complex system will feature a large number of blocks.

- Dependent task systems have few blocks.

Clearly, the blocks of the two classes of specifications are not the same. The blocks of synchronous specifications written in Scade or SynDEx are often simple filters. On the other hand, the blocks of a dependent task specification correspond to the tasks of the future system, and this number must be kept low to simplify OS configuration, real-time analysis, *etc.* Between the two representation levels lies a complex industrial process (often performed manually) that takes small blocks and groups them together into larger tasks.

[5] As opposed to *sampling ports*.

Clearly, the automatic compilation and code generation flow we propose faces the same problem. One solution is to use the same approach of grouping small operations into tasks. Our industrial case study actually uses this approach, so that a full GNC system is represented using 3 tasks. However, this leaves unsolved the problem of grouping operations into tasks, and its results can be significantly improved.

Having a small number of tasks is particularly important in ARINC 653-based systems. Indeed, ARINC 653 imposes that all processes needed in the execution be created in the elaboration phase, and forbids their destruction. This means, for example, that no memory optimization can be performed by creating and destroying processes dynamically during the execution of the system. The memory needed by all the processes of the implementation is statically reserved during the elaboration phase. Depending on the available quantity of memory in the execution platform, this may impose a limitation of the number of threads one can declare for an application.

In the code generation scheme for our POK target, we minimize them by creating only one process per partition window, whenever the specification allows it. This optimization can only be applied when the partition code has no synchronization points during the duration of the window (*e.g.* no sensor reads at dates which are not window start dates). This technique can be generalized by creating one process per point in time where a partition synchronizes with its environment.

8.5 Conclusion

In this chapter we defined our algorithms for the automatic mapping of time-triggered embedded applications with complex non-functional requirements. The algorithms take as input specifications written in the language defined in Chapter 7. This language allows the representation of non-functional requirements of the following types: preemptability, starting offsets, deadlines, execution conditions and partitioning. Our deadline-driven mapping algorithms perform the multi-processor pipelined scheduling under multiple requirements of these types, ensuring their respect in the resulting scheduling table.

The deadline-driven nature of our mapping algorithms results in scheduling tables that contain numerous preemptions and/or partition switches, which are very time-consuming. In order to decrease the number of preemptions and partition switches we defined a post-scheduling optimization which reorganizes the reservations of the scheduling table without affecting its functional correctness or the respect of non-functional requirements.

We were able to successfully use our scheduling algorithms on the full industrial case study provided by Airbus DS, thus demonstrating the efficiency of our specification and scheduling methods.

From the produced scheduling tables, we are also capable of generating executable

code compatible with the ARINC 653 standard and its APEX interface. This automatic code generation process is non-trivial. Two major difficulties that we encountered concerned the automatic synthesis of inter-partition and inter-processor communication channels and associated communication primitives, and the reduction of the number of ARINC 653 processes in the final running application. We presented our solutions to these two problems.

Chapter 9

Conclusion and perspectives

Contents

9.1 Conclusion

In this thesis I have taken inspiration from 3 research fields (real-time scheduling, compilation, and synchronous language design and implementation) to build a compilation technique for complex embedded control systems. The input of my technique consists in:

- A functional specification provided under the form of a synchronous program, and which allows the definition of stateful behaviors involving data-dependent control.
- A specification of the multiprocessor/distributed execution platform defining the computation, communication, and possibly the memory resources.
- Non-functional requirements of various types.

From these 3 inputs, our compilation technique automatically synthesizes correct-by-construction real-time implementations. The compilation process comprises two phases. The first is real-time scheduling, which follows an off-line paradigm and produces a scheduling table. The second is code generation, which takes the scheduling table and builds running code from it.

The main originality of my work is to take inspiration from the 3 fields mentioned above, and thus to bring together the best of the three worlds:

- Our compiler generates code with hard real-time guarantees, and at the same time uses advanced compilation techniques (software pipelining, allocation of variables

to the memory banks) so that the code efficiently uses the resources of the execution platform, thus improving real-time schedulability results.

- Like a compiler, it allows fully automated code generation.
- It considers multiple non-functional requirements of several types.
- It works at system level, and yet considers functional specification details such as conditional control.

To allow the definition of this compilation technique, our work has put in evidence strong links between formal models used in compilation, real-time scheduling, and synchronous programming, which allowed the joint use of scheduling and code generation techniques coming from the 3 fields. Our work has also put in evidence differences between these fields that require non-trivial adaptation of the various models and techniques.

From a more technological point of view, my objective here was to provide models and algorithms that facilitate the development of complex embedded systems by automating the allocation and scheduling step. The yardstick I used to measure success was a space launcher application case study coming from Airbus DS.

From this perspective, the main originality of my work is that it takes into account at the same time multiple complexity elements of both functional and non-functional type. Our off-line scheduling and optimization algorithms take into account these parameters and synthesize scheduling tables under complex non-functional requirements that no existing tool can handle. These scheduling tables can be automatically translated into time-triggered implementations.

We have been able to model and automatically map our aerospace case study. As all scheduling and optimization algorithms used in our tool are fast, we were able to perform significant design space exploration over the chosen case study. I believe that this shows the potential for full automation in the system-level engineering of complex real-time embedded systems.

Defining strong links between the fields of real-time scheduling and compilation has not been easy, and was done in two clearly identifiable phases corresponding to two different levels of integration between the formal models and the methods of the two fields. In both cases, models and techniques from synchronous languages design and implementation served as a formal base for the integration.

The first integration level corresponds to our work on using software pipelining techniques to improve the throughput of real-time applications (Chapter 6). In this work, the objective is the optimization of metrics such as latency and throughput. In this sense, this first integration level remains closer to previous work on compilation, and farther from mainstream work on real-time scheduling where the objective is not the optimization of

some metrics, but the respect of requirements. But while the perspective is closer to compilation, there are significant differences with respect to classical compilation work:

- The main optimization objective is not throughput, but rather latency, with throughput relegated to a second place. We found this objective to be more pertinent in our application classes.
- Our embedded execution platforms allow a simpler exploitation of the execution conditions (modes) present in the applications to improve the mapping results. In particular, we allow the representation and analysis of the evolution of the execution conditions between successive iterations.
- Dealing with the conditional control also requires a complexification of the classical rotating registers to account for the execution conditions.
- Time is accounted for in a conservative fashion, in order to provide hard real-time guarantees. In particular, this timing accounting covers the complex code generation process (memory replication).

The second integration level corresponds to Chapter 7 and Chapter 8. Here, we fully assume a real-time scheduling perspective where the objective is the respect of complex non-functional requirements. While reusing results (models and algorithms) of the first phase, including the use of software pipelining, the second phase significantly complexifies them in all aspects: modelling, real-time scheduling, and code generation by:

- Allowing the modelling of applications in a time-triggered context.
- Building multi-processor scheduling tables by performing a particular version of list scheduling which is deadline-driven and takes into account the time partitioning of the applications and the preemptability of the tasks.
- Including a post-scheduling optimization which aims at decreasing the number of preemptions and of switches between the partitions.
- Automatically synthesizing full ARINC 653-compliant implementations comprising both the C/APEX code of the partitions and the system configuration. We automatically synthesize the inter-partition and inter-processor channels and communication operations. We also minimize during code generation the number of generated implementation tasks.

All these contributions were implemented in the LoPhT compiler, whose general picture has already been pictured in Fig. 1.3 (cf. Page 25).

9.2 Perspectives

The work presented in this manuscript opens many perspectives for future work. We present here the main axes in which we believe this work can, and should be continued in the future:

- Accounting more precisely for the cost of the Operating System. To do so, we could use precise WCET estimates for the cost of preemptions and of partition changes, and apply them every time we schedule a preemption or a partition change during the off-line scheduling process. While remaining conservative, this approach would be more precise than accounting for preemptions/partition switches by applying margins in the WCET estimates of the tasks before knowing exactly how many of them will occur (in the worst case). In the same direction, taking into account the cost of interrupt-driven code such as device drivers is more difficult due to the inherent asynchrony. However, hardware isolation mechanisms can be used here to provide precise and conservative accounting for the supplementary costs.

- Efficiently modelling and scheduling mixed-criticality systems by using the execution conditions, in a scheme similar to the one described in Fig. 7.3 (cf. Page 142). Work must be done on the characterization and modelling of how the system can detect an enforce mode changes related to criticality during execution.

- Going further into building links between compilation and real-time scheduling. We could for example investigate the adaptability of polyhedral compilation methods: these are powerful optimization methods which were first aimed at optimizing the throughput (like software pipelining) of nested loops. Since then they have been extended to optimize various criteria, and have proven very efficient. Other techniques inherited from the compilation domain that could be adapted are for example loop unrolling, that we could automatize in order to exhibit an adequate level of parallelism compared to the capacities of the considered execution platform (for example in the context of many-core architectures). Another interesting approach could be to mix speculative branching with probabilistic real-time approaches in the presence of execution modes accounting for a failure or breakdown inside the system. In this context, it could be possible to attach a low probability of occurence to the tasks corresponding to the fail-safe mode, and to optimize more the schedule of the tasks of the nominal mode, while preserving (probabilistic) real-time guarantees for the whole system.

- Defining a generalist architecture description language tailored for our real-time problems. We are currently capable of handling several kinds of execution archi-

tectures in the LoPhT tool (Noc-based many-core architectures, broadcast buses[1]), but each of them is characterized using a separate language, and thus needs a partially different treatment in the scheduling and code generation phase. Part of our future work should be dedicated to the definition of a unique language capable of modelling execution platforms for a unified treatment for scheduling purposes.

- Proving the correctness of our methods. For instance, if we consider just the scheduling phase, we have two ways of proving the correctness of its result: by proving the compiler itself, or by formally validating its output (the scheduling table). This manuscript provides the complete formalization of input and output models and algorithms, as well as the formal definition of the functional and non-functional correctness properties that must be respected by the resulting scheduling tables. Since our algorithms remain rather simple, we believe it is feasible without major difficulties to use this manuscript as a basis for extending a CompCert-like approach [Leroy, 2009] to LoPhT.

- Increasing the integration level between WCET analysis and scheduling. For example, we could take advantage of our offline scheduling scheme which takes decisions for each task instance one by one without backtracking, and model at each step precise information concerning memory and for example the state of the caches, to use it in turn for the WCET analysis of the future tasks to schedule. This could help the analysis to reach more precise WCET estimates, and thus increase the efficiency of the scheduling technique in terms of schedulability.

- Investigating a way of finding a good tradeoff between simplicity of representation of the specification and efficiency of the scheduling method in terms of schedulability. As we saw, using the hyperperiod expansion technique can lead in the worst case to an exponential explosion in the number of tasks to consider in the scheduling and code generation processes (and thus, a large amount of implementation code). On the other hand, we also explained that using simpler models such as multiperiodic task systems do not yield the same results in terms of schedulability. A very interesting question would be that of finding intermediate representations models which would limit the number of tasks replicas to consider while offering a rather good scheduling freedom to the method.

Last, but not least, we believe that prototypes such as LoPhT pave the way to the construction of industry-strength tools able to support a change of practices in the industry, most notably a move from manual techniques to formally-proven automatic or assisted implementation methods.

[1] Taking into account TTEthernet communication networks for deterministic communications over particular ethernet architectures is current work in progress.

Appendix A

Modeling NoC-based many-cores

Throughout Chapter 4, we explained that our architecture model could only support sequential processing units interconnected by a single message-passing broadcast bus. In fact we extended the model, the scheduling algorithmics and the code generation process in order to take into account Massively Parallel Processor Arrays whose communication is based on a mesh Network-on-Chip. The main differences with the original model are:

- *concerning processors*: the number of processors is largely superior in MPPA than in classical distributed architures. Moreover, the processors are organized in clusters, called tiles, which usually contain up to 16 processors and their shared memory, and are interconnected via the NoC. For tractability reasons, we model the processors and memory inside each tile as one sequential processing resource. This is achieved using results on the precise WCET analysis for distributed architectures and parallel code [Potop-Butucaru and Puaut, 2013]. By parallelizing regular code on the various processors of a given tile, and using precise WCET analysis, we can obtain a good bound for the execution of the parallel code, and thus amalgamate the processors inside one abstract sequential computation resource.

- *concerning the transmission media*: the broadcast bus is replaced by a complex network on chip (NoC) which links the computation tiles using FIFO-like unidirectional resources and routers. Switching follows a wormhole paradigm instead of the store-and-forward switching of classical embedded networks. In store and forward switching, each data packet arriving on a router is entirely stored in the router before being forwarded again. Wormhole switching is suited for architectures where the routers have limited buffering capabilities which make it impossible to store full packets inside the router. In this approach, packets are seen as sequences of *flits* which are sent on the network in a pipelined fashion, each router of the communication path being able to buffer only a few flits (typically 2 or 3). The first flit of a packet contains the routing information which sets the path for all other packets. As the first flit progresses through the network towards its destination, by hopping from

one router to the next, all the other flits must follow it and occupy the routers located on the communication path. Starting from the first flit, one single packet may stretch over several routers[1]. The flits of a packet advance synchronously (no gap exists between successive flits), which strongly synchronizes the functioning of the routers storing its flits. By consequence, the resources of the NoC (the inter-router links) must be reserved in a synchronized fashion [Djemal et al., 2012].

- *concerning the target execution model*: the code generated for MPPA architectures is not time triggered. Instead LoPhT generates traditional C code, with a completely static memory allocation for the variables of the system. The synchronizations between the various computations running on different resources are managed through the use of hardware locks. Finally, LoPhT generates specific code for the NoC routers, corresponding to the communications schedule.

In this context, LoPhT was able to generate schedules close to the optimum on our examples. Moreover, the corresponding code which was generated automatically by LoPhT executes in times that are very close to the end-to-end latency bounds derived from the schedule. In other terms, the computed conservative bounds are very tight. These results are one of the main contributions of Manel Djemal's thesis, and are presented in [Carle et al., 2013].

[1] If packets are limited to 20 flits and the buffering capacity of a router along a path is of 3 flits, then a packet may stretch along 7 routers.

List of Publications

1. Carle,T., Potop Butucaru,D. **Predicate-aware, makespan-preserving software pipelining of scheduling tables**. In *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 11, 1, Feb. 2014.

2. Carle,T., Djemal,M., Potop Butucaru,D., de Simone,R., Zhang,Z. **Static mapping of real-time applications onto massively parallel processor arrays**. In *Proceedings of the IEEE international conference on Application of Concurrency to System Design (ACSD'14)*, Tunis, Tunisia

3. Carle,T., Djemal,M., Potop Butucaru,D., de Simone,R., Zhang,Z., Pechêux,F., and Wajsbürt, F. **Reconciling performance and predictability on a many-core through off-line mapping**. In *Reconciling Performance and Predictability Workshop (REPP'14)*, Grenoble, France

4. Carle,T., Djemal,M., Genius,D., Pechêux,F., Potop Butucaru,D., de Simone,R., Wajsbürt,F., and Zhang,Z. **Reconciling performance and predictability on a many-core through off-line mapping**. In *9th International Symposium on Reconfigurable Communication-centric Systems-on-chip (ReCoSoC'14)*, Montpellier, France

Bibliography

V. Allan, R. Jones, R. Lee, and S. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3), September 1995. 61, 62, 97, 103, 112

M. Alras, P. Caspi, A. Girault, and P. Raymond. Model-based design of embedded control systems by means of a synchronous intermediate model. In *Proceedings ICESS*, pages 3–10, Zhejiang, China, May 2009. 131

C. André, F. Mallet, and M.-A. Peraldi-Frati. A multiform time approach to real-time system modeling; application to an automotive system. In *Proceedings SIES*, Lisbon, Portugal, July 2007. 97, 98, 123

ARINC. ARINC 653: Avionics application software standard interface. `www.arinc.org`, 2005. 72, 82, 97, 99, 129, 130, 135

AUTOSAR. Autosar (automotive open system architecture), release 4. `http://www.autosar.org/`, 2009. 72, 97, 135

S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24:93–128, 2003. 133

A. Benoît, V. Rehn-Sonigo, and Y. Robert. Multi-criteria scheduling of pipeline workflows. In *Proceedings of the International Conference on Cluster Computing*, Austin, TX, USA, Sep 2007. 102

A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the signal language. *Automatic Control, IEEE Transactions on*, 35(5):535–546, May 1990. ISSN 0018-9286. doi: 10.1109/9.53519. 32

A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003. 11, 22, 29

G. Berry. The constructive semantics of pure esterel, 1996. 30, 32

J. Blazewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In *Proceedings of the International Workshop organized by the Commision of the European Communities on Modelling and Performance Evaluation of Computer Systems*, pages 57–65, Amsterdam, The Netherlands, The Netherlands, 1977. North-Holland Publishing Co. ISBN 0-7204-0554-8. URL http://dl.acm.org/citation.cfm?id=647407.724282. 61, 131, 154, 155

V. Brocal, M. Masmano, I. Ripoll, A. Crespo, and P. Balbastre. Xoncrete: a scheduling tool for partitioned real-time systems. In *Proceedings ERTS*, Toulouse, France, 2010. 133

G. Butazzo. *Hard real-time computing systems. Predictable scheduling, algorithms and applications*. Kluwer, 2002. 7, 19

P.-Y. Calland, A. Darte, and Y. Robert. Circuit retiming applied to decomposed software pipelining. *Parallel and Distributed Systems, IEEE Transactions on*, 9(1):24–35, 1998. 98, 100

T. Carle and D. Potop-Butucaru. Predicate-aware, makespan-preserving software pipelining of scheduling tables. *ACM Trans. Archit. Code Optim.*, 11(1):12:1–12:26, February 2014. ISSN 1544-3566. doi: 10.1145/2579676. URL http://doi.acm.org/10.1145/2579676. 72

T. Carle, D. Potop-Butucaru, Y. Sorel, and D. Lesens. From dataflow specification to multiprocessor partitioned time-triggered real-time implementation. Technical report, INRIA, October 2012. URL http://hal.inria.fr/hal-00742908. 72, 100

T. Carle, M. Djemal, D. Potop-Butucaru, R. De Simone, and Z. Zhang. Off-line mapping of real-time applications onto massively parallel processor arrays. Rapport de recherche, to appear in ACSD '14 RR-8429, INRIA, December 2013. URL http://hal.inria.fr/hal-00919411. 13, 24, 177

T. Carle, M. Djemal, D. Genius, F. Pecheux, D. Potop Butucaru, R. De Simone, F. Wajsburt, and Z. Zhang. Reconciling performance and predictability on a many-core through off-line mapping. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*, pages 1–8, May 2014. doi: 10.1109/ReCoSoC.2014.6861367. 13, 24, 72

P. Caspi, A. Curic, A. Magnan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Proceedings LCTES*, San Diego, CA, USA, June 2003. 61, 97, 98, 103, 104, 132

D. Chabrol, C. Aussaguès, and V. David. A spatial and temporal partitioning approach for dependable automotive systems. In *Proceedings ETFA*, Mallorca, Spain, 2009. 32, 132, 133

K. S. Chatha and R. Vemuri. Hardware-software partitioning and pipelined scheduling of transformative applications. *IEEE Trans. Very Large Scale Integr. Syst.*, 10:193–208, 2002. 104

H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2, 1990. ISSN 0922-6443. 61, 62, 65, 131, 154, 155, 160

Y.-S. Chiu, C.-S. Shih, and S.-H. Hung. Pipeline schedule synthesis for real-time streaming tasks with inter/intra-instance precedence constraints. In *DATE*, Grenoble, France, 2011. 62, 104

A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous kahn networks: a relaxed model of synchrony for real-time systems. In *Proceedings POPL'06*. ACM Press, 2006. 65

S. Conchon and J. Kanigand S. Lescuyer. SAT-MICRO: petit mais costaud ! In *JFLA (Journées Francophones des Langages Applicatifs)*, pages 91–106, Etretat, France, 2008. URL `http://hal.inria.fr/inria-00202831`. 118

M. Cordovilla, F. Boniol, J. Forget, E. Noulard, and C. Pagetti. Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset. In *19th International Conference on Real-Time and Network Systems*, Nantes, France, September 2011. Irccyn. URL `http://hal.inria.fr/inria-00618587`. 81

J. B. Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376. Springer Berlin Heidelberg, 1974. ISBN 978-3-540-06859-4. doi: 10.1007/3-540-06859-7_145. URL `http://dx.doi.org/10.1007/3-540-06859-7_145`. 57

M. Djemal, R. De Simone, F. Pêcheux, F. Wajsbürt, D. Potop-Butucaru, and Z. Zhang. Programmable routers for efficient mapping of applications onto noc-based mpsocs. In *DASIP*, pages 1–8, 2012. 177

J. C. Doyle, B. A. Francis, and A. R. Tannenbaum. *Feedback Control Theory*. Macmillan Publishing Co., 1990. 6, 18

S.A. Edwards and E.A. Lee. The case for the precision timed (pret) machine. In *Proceedings DAC*, 2007. 133

S.A. Edwards, S. Kim, E.A. Lee, I. Liu, H.D. Patel, and M. Schoeberl. A disruptive computer design idea: Architectures with repeatable timing. In *Proceedings ICCD*. IEEE, October 2009. Lake Tahoe, CA. 135

P. Eles, A. Doboli, P. Pop, and Z. Peng. Scheduling with bus access optimization for distributed embedded systems. *IEEE Transactions on VLSI Systems*, 8(5), Oct 2000. 97, 98, 103

S. Fischmeister, O. Sokolsky, and I. Lee. Network-code machine: Programmable real-time communication schedules. In *Proceedings RTAS 2006.*, april 2006. 134

G. Fohler. Changing operational modes in the context of pre run-time scheduling, 1993. 132

G. Fohler and K. Ramamritham. Static scheduling of pipelined periodic tasks in distributed real-time systems. In *In Procs. of EUROMICRO-RTS97*, pages 128–135, 1997. 132

G. Fohler, A. Neundorf, K.-E. Årzén, C. Lucarz, M. Mattavelli, V. Noel, C. von Platen, G. Butazzo, E. Bini, and C. Scordino. EU FP7 ACTORS project. Deliverable D7a: State of the art assessment. Ch. 5: Resource reservation in real-time systems. `http://www3.control.lth.se/user/karlerik/Actors/d7a-rev.pdf`, 2008. 97

J. Forget, F. Boniol, D. Lesens, and C. Pagetti. A real-time architecture design language for multi-rate embedded control systems. In *In Procs. of SAC '10*, SAC '10. ACM, 2010. 32

M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979. 98

F. Gasperoni and U. Schwiegelshohn. Generating close to optimum loop schedules on parallel processors. *Parallel Processing Letters*, 4(4):391–404, December 1994. 98, 100

K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 22(5):414–421, Sept 2005. 82

R. Govindarajan, E. Altman, and G. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, 1994. 102

T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives. In *Proceedings MEMOCODE*, Mont St Michel, France, 2003. 98, 103, 122

P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal, a data-flow oriented language for signal processing. *Acoustics, Speech, and Signal Processing (see IEEE Transactions on Signal Processing)*, 34:362–374, 1986. 32

P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers*, April 2003. Special Issue on Application Specific Hardware Design. 32, 63

N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer academic Publishers, 1993. 29

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991. 32

D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987. 7, 19

J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4^{th} edition, 2007. 97, 111

T.A. Henzinger and C. Kirsch. The embedded machine: Predictable, portable real-time code. *ACM Transactions on Programming Languages and Systems*, 29(6), Oct 2007. 131, 132, 134

T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of the IEEE*, pages 166–184. Springer-Verlag, 2000. 32, 129, 131, 132, 140, 162

R.A. Huff. Lifetime-sensitive modulo scheduling. In *In Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 258–267, 1993. 102

SAE international. The AADL formalism page: http://www.aadl.info/, 2014. 139

D. Isovic and G. Fohler. Handling mixed sets of tasks in combined offline and online scheduled real-time systems. *Real-Time Systems*, 43:296–325, 2009. 132

A. Kent and J. G. Williams, editors. *Encyclopedia of Computer Science and Technology: Volume 45 - Supplement 30*, chapter Real-Time Constraints, pages 285–309. CRC Press, 2002. 7, 19

W. Kim, D. Yoo, H. Park, and M. Ahn. Scc based modulo scheduling for coarse-grained reconfigurable processors. In *Field-Programmable Technology (FPT), 2012 International Conference on*, Seoul, Korea, 2012. 104

H. Kopetz. Event-triggered versus time-triggered real-time systems. In *LNCS 563*, volume 563 of *Lecture Notes in Computer Science*, pages 87–101, 1991. 134, 136

H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91 (1):112–126, 2003. 72, 129, 134, 135, 136

M. Lam. Software pipelining : An effective scheduling technique for vliw machines. In *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 318–328, 1988. 101, 104, 119

E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 36(1):24–35, 1987a. URL `http://doi.ieeecomputersociety.org/10.1109/TC.1987.5009446`. 81

E. A. Lee and S. A. Seshia. Introduction to embedded systems, a cyber-physical systems approach, 2011. URL `http://LeeSeshia.org`. 6, 18

E.A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceeding of the IEEE*, 75 (9):1235–1245, Sep. 1987b. 61

C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991. 100

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. URL `http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf`. 175

J.Y.-T. Leung and M.L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115 – 118, 1980. 160

C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973. 10, 21, 34, 82

M. Marouf, L. George, and Y. Sorel. Schedulability analysis for a combination of non-preemptive strict periodic tasks and preemptive sporadic tasks. In *Proceedings ETFA'12*, Kraków, Poland, September 2012. 131

J.F. Mason, K. R. Luecke, and J.A. Luke. Device drivers in time and space partitioned operating systems. In *25th Digital Avionics Systems Conference, IEEE/AIAA*, Portland, OR, USA, Oct. 2006. 135

A. Monot, N. Navet, F. Simonot, and B. Bavoux. Multicore scheduling in automotive ECUs. In *Proceedings ERTSS*, Toulouse, France, 2010. 97

L. Morel. *Exploitation des structures régulières et des spécifications locales pour le developpement correct de systèmes réactifs de grande taille*. PhD thesis, Institut National Polytechnique de Grenoble, 2005. 104

S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman, 1997. 113, 114

A. Munier. The basic cyclic scheduling problem with linear precedence constraints. *Discrete Applied Mathematics*, 64(3):219 – 238, 1996. 65, 66

C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3):307–338, 2011. 65, 129, 131, 132, 133, 160

Q. Pan, T. Gautier, L. Besnard, and Y. Sorel. Signal to syndex: Translation between synchronous formalisms. internal report, 2003. Internal report, INRIA, Rocquencourt, France, 2003. URL `http://www-rocq.inria.fr/syndex/publications/pubs/signalSyndex03/sig` 57

L.T.X. Phan, S. Chakraborty, and P. S. Thiagarajan. A multi-mode real-time calculus. In *Real-Time Systems Symposium, 2008*, pages 59–69, Nov 2008. 82

POK. Pok, a partitionned operating system. `http://pok.tuxfamily.org/`, 2008. 165

P. Pop, P. Eles, and Z. Peng. Scheduling with optimized communication for time-triggered embedded systems. In *Proceedings CODES'99*, 1999. 132

D. Potop-Butucaru and I. Puaut. Integrated Worst-Case Execution Time Estimation of Multicore Applications. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of

OpenAccess Series in Informatics (OASIcs), pages 21–31, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-54-5. doi: http://dx.doi.org/10.4230/OASIcs.WCET.2013.21. URL http://drops.dagstuhl.de/opus/volltexte/2013/4119. 176

D. Potop-Butucaru and Y. Sorel. Chapter 12: Approche synchrone et ordonnancement. In *Ordonnancement dans les systèmes temps réel*, pages 325–364, 2014. URL http://iste-editions.fr/products/ordonnancement-dans-les-systemes-t
29

D. Potop-butucaru, R. De Simone, and J. p. Talpin. The synchronous hypothesis and synchronous languages, in embedded systems handbook, 2005. 29

D. Potop-Butucaru, R. De Simone, Y. Sorel, and J.-P. Talpin. Clock-driven distributed real-time implementation of endochronous synchronous programs. In ACM, editor, *EMSOFT '09 Proceedings of the seventh ACM international conference on Embedded software*, pages 147–156, 2009. 37, 52, 84, 106, 122, 155, 159

D. Potop-Butucaru, A. Azim, and S. Fischmeister. Semantics-preserving implementation of synchronous specifications over dynamic TDMA distributed architectures. In *Proceedings EMSOFT*, Scottsdale, Arizona, USA, 2010. 97, 98, 103, 134, 135, 143

C. Pradalier, J. Hermosillo, C. Koike, C. Braillon, P. Bessière, and C. Laugier. The CyCab: a car-like robot navigating autonomously and safely among pedestrians. *Robotics and Autonomous Systems*, 50(1), 2005. 123

W. Puffitsch, E. Noulard, and C. Pagetti. Mapping a multi-rate synchronous language to a many-core processor. In *Proceedings RTAS*, 2013. 132

K. Ramamritham, G. Fohler, and J. M. Adan. Issues in the static allocation and scheduling of complex periodic tasks. In *In Proc. 10th IEEE Workshop on Real-Time Operating Systems and Software*, 1993. 132

B.R. Rau. Iterative modulo scheduling. *International Journal of Parallel Programming*, 24(1):3–64, 1996. 99, 103, 104, 118

B.R. Rau and C.D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th annual workshop on Microprogramming, IEEE*, 1981. 97, 113

B.R. Rau, M. Lee, P.P. Tirumalai, and M.S. Schlansker. Register allocation for software pipelined loops. In *Proceedings PLDI'92*, San Francisco, CA, USA, June 1992. 119

P. Richard, F. Cottet, and C. Kaiser. Précédences généralisées et ordonnançabilité des tâches de suivi temps réel d'un laminoir. *Journal européen des systèmes automatisés*, 35, 2001. 65

J. Rushby. Bus architectures for safety-critical embedded systems. In *Proceedings EMSOFT'01*, volume 2211 of *LNCS*, Tahoe City, CA, USA, 2001. 97, 129, 135, 146

A. Al Sheikh, O. Brun, P.-E. Hladik, and B.J. Prabhu. Strictly periodic scheduling in ima-based architectures. *Real-Time Systems*, 48(4):359–386, 2012. URL `http://dx.doi.org/10.1007/s11241-012-9148-y`. 133

M. Smelyanskyi, S. Mahlke, E. Davidson, and H.-H. Lee. Predicate-aware scheduling: A technique for reducing resource constraints. In *Proceedings CGO*, San Francisco, CA, USA, March 2003. 103

Y. Sorel. From modeling/simulation with scilab/scicos to optimized distributed embedded real-time implementation with syndex. In *Proceedings of the International Workshop On Scilab and Open Source Software Engineering, SOSSE'05*, Wuhan, China, October 2005. 57, 69

L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 4, pages 101–104 vol.4, 2000. 81

J. Wang and C. Eisenbeis. Decomposed software pipelining. `http://hal.inria.fr/inria-00074834`, 1993. 98, 100, 102, 122, 124, 125

N.J. Warter, D. M. Lavery, and W.W. Hwu. The benefit of predicated execution for software pipelining. In *HICSS-26 Conference Proceedings*, Houston, Texas, USA, 1993. 103, 113

J. Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *Software Engineering, IEEE Transactions on*, 19(2):139–154, 1993. ISSN 0098-5589. 133

H. Yang and S. Ha. Pipelined data parallel task mapping/scheduling technique for mpsoc. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, Nice, France, 2009. 104

H.-S. Yun, J. Kim, and S.-M. Moon. Time optimal software pipelining of loops with control flows. *International Journal of Parallel Programming*, 31(5):339–391, October 2003. 102, 103

J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Register constrained modulo scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 15(5):417–430, 2004. 102

W. Zheng, J. Chong, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. Extensible and scalable time-triggered scheduling. In *Proceedings ACSD*, St. Malo, France, June 2005. 65, 97, 98, 103, 132

Q. Zhuge, Z. Shao, and E.H. Sha. Optimal code size reduction for software-pipelined loops on dsp applications. In *Proceedings of the International Conference on Parallel Processing*, 2002. 102