THESE

Pour l'obtention du grade de

DOCTEUR DE L'UNIVERSITE DE POITIERS

Ecole Nationale Supérieure de Mécanique et d'Aérotechnique

Faculté des Sciences Fondamentales et Appliquées

Ecole Doctorale des Sciences pour l'Ingénieur

(Diplôme National – Arrêté du 30 Mars 1992)

Secteur de Recherche : INFORMATIQUE

Présentée et soutenue publiquement par

Khanh Hieu NGO

Le 04 Novembre 2008

Aide au développement de systèmes temps réel à l'aide d'un langage graphique flots de données

---

Directeurs de Thèse : Francis COTTET, Emmanuel GROLLEAU

JURY

Rapporteurs :	**Frédéric BONIOL, IRIT-ENSEEIHT, Université de Toulouse**
	Patrick MARTINEAU, LI-Ecole Polytech, Université de Tours
Examinateurs :	**Isabelle PUAUT, IRISA, Université de Rennes**
	Pascal RICHARD, LISI, Université de Poitiers
	Francis COTTET, LISI, ENSMA
	Emmanuel GROLLEAU, LISI, ENSMA

LABORATOIRE D'INFORMATIQUE SCIENTIFIQUE ET INDUSTRIELLE
Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Téléport 2 - 1, avenue Clément Ader - BP 40109 - 86961 Chasseneuil-Futuroscope Cédex
Tél: 05.49.49.80.63 - Fax: 05.49.49.80.64

Remerciements

Je remercie tout d'abord Monsieur Francis Cottet qui m'a permis d'intégrer l'équipe temps réel du LISI en acceptant diriger mes travaux de thèse. Je lui exprime tout ma reconnaissance pour les conseils et le soutien tant sur le plan administratif que scientifique qu'il m'a apporté durant mes années de thèse.

Je remercie Monsieur Doan Kim Son pour avoir appuyé ma demande de bourse Gouvernement Français, et le Poste MCA2008 pendant les travaux de ma thèse. Je le remercie aussi pour les connaissances et les conseils pour ma formation de chercheur.

J'exprime les vifs remerciements à Monsieur Emmanuel Grolleau pour son encadrement et les nombreux conseils inestimables qu'il m'a donnés tout au long au long de mes travaux de thèse. Mes résultats obtenus dans cette thèse sont principalement grâce à des occasions de voir Manu pour moi de lui manifester toute ma gratitude pour les connaissances acquises auprès de lui. Mille fois merci, Manu. Pour moi, tu es non seulement mon professeur, mais aussi mon « frère ».

Je présente mes remerciements à Monsieur Guy Pierra pour m'avoir accepté dans le laboratoire qu'il a dirigé.

Je présente mes remerciements à Monsieur Yamine Ait Ameur pour l'ensemble des moyens mis à ma disposition pour le déroulement de ma soutenance de thèse.

J'adresse mes remerciements aux Monsieur Frédéric Boniol, Monsieur Patrick Martineau, et Madame Isabelle Puaut qui ont bien voulu consacrer leur temps à la relecture et à la correction en tant que rapporteurs de ma thèse. Je remercie par la même occasion Monsieur Pascal Richard pour avoir accepté d'être membre de mon jury de thèse.

Je tiens à remercier également tous les membres du LISI, mes collègues du LISI et de l'ENSMA pour leur sympathie et les bons moments passés ensemble.

Mes remerciements s'adressent aussi à tous ceux qui m'ont permis d'obtenir la bourse. Il s'agit de Monsieur Denis Gamby, Monsieur Lionel Buthaud du LMPM, et de Monsieur Nguyen Thien Tong du Département d'Aérotechnique de l'IPHCM, Vietnam.

Je remercie le Gouvernement Français pour m'avoir donné l'opportunité de poursuivre ces travaux de thèse en m'attribuant une bourse.

Enfin, je mille fois remercie mon père (qui a décédé lorsque j'ai été en France), ma mère, ma femme, ma petite sœur.

A ma famille,

Table des Matières

Introduction générale 1
I. Introduction 3
I.1. Contexte de l'étude 3
I.1.1. Définitions des systèmes de contrôle-commande 3
I.1.2. Caractéristiques des systèmes de contrôle-commande 4
I.1.3. Architecture des systèmes de contrôle-commande 4
I.2. Cycle de développement d'un logiciel 6
I.2.1. Le cycle de vie en «Cascade» 7
I.2.2. Le cycle de vie en «V» 8
I.2.3. Les autres cycles de développement d'un logiciel 9
I.3. Le cycle de vie Logiciel/Matériel 13
I.3.1. L'approche de développement «Logiciel/Matériel» traditionnelle 13
I.3.2. L'approche de développement «Logiciel/Matériel» en Co-design 14
I.3.3. Le cycle de développement en «W» dans l'approche traditionnelle 16
I.3.4. «Time-to-market», un point clé à succès 17
I.4. Support d'exécution pour le temps réel 18
I.4.1. Exécutif ou noyau temps réel 18
I.4.2. Concepts des tâches 19
I.4.3. Gestion des tâches 20
I.4.4. Outils de communication et de synchronisation 21
I.4.5. Gestion du temps 25
I.4.6. Principales normes temps réel 25
I.5. Méthodes de développement d'un logiciel 29
I.5.1. JSD (Jackson System Development) 33
I.5.2. SA-RT (Structured Analysis For Real-Time Systems) 37
I.5.3. Statecharts 49
I.5.4. DARTS (Design Approach for Real-Time Systems) 54
I.5.5. UML-RT 63
I.5.6. Conclusion sur les méthodes de développement 74
I.6. Validation temporelle des systèmes temps réel 77
I.6.1. Modèles d'exécution 77
I.6.2. Modélisation des tâches 78
I.6.3. Ordonnancement temps réel 83
II. DARTS: du concept au code 87
II.1. Utilisation de la méthode DARTS 87
II.1.1. Critères de création d'une structure des tâches DARTS 87
II.1.2. Communication/Synchronisation des tâches 91
II.1.3. Critères de création d'une structure de modules de données 96
II.1.4. Intégration de tâches et de modules 98
II.2. Implémentation DARTS sur LabVIEW 98
II.2.1. Présentation de LabVIEW 98
II.2.2. Outils pour le parallélisme dans LabVIEW 104
II.2.3. LabVIEW, un outil d'implémentation des concepts DARTS 112
II.2.4. Rôle de LabVIEW dans le développement d'un logiciel 126
II.3. Implémentation DARTS sur langages de programmation impératifs 127
II.3.1. Implémentation DARTS sur la norme POSIX 127

II.3.2. Implémentation DARTS sur exécutif VxWorks 133
II.3.3. Implémentation DARTS sur Ada 137
III. Architecture DARTSVIEW 145
III.1. Présentation de la palette DARTSVIEW 145
III.1.1. Sous-palette «Tâche Matérielle» 145
III.1.2. Sous-palette «Tâche Logicielle» 149
III.1.3. Sous-palette «Outil Communication» 149
III.1.4. Sous-palette «Générer Code» 151
III.1.5. «Modèle DARTSVIEW du Système» 151
III.1.6. «Informations Enregistrées» 152
III.2. Palette DARTSVIEW et le premier V du cycle de vie en W 152
III.3. Introduction à XML 153
III.3.1. Fichier XML bien formé 153
III.3.2. Validation des fichiers XML 154
III.4. Palettes XML de LabVIEW 155
III.4.1. Palette «XML» dans la palette «Manipulation de données» 155
III.4.2. Sous-palette «XML» dans le Toolkit Internet de LabVIEW 156
III.4.3. Palettes XML de LabVIEW: Désavantages 157
III.5. Palette DARTSVIEW et le second V du cycle de vie en W 161
III.5.1. Schéma de la génération de code 162
III.5.2. Variable globale dans le schéma de la génération de code 162
III.5.3. Fichier XML de description du modèle DARTSVIEW 168
III.5.4. Génération de code d'un langage particulier 171
III.5.5. Conclusion 190
IV. Cas d'étude 191
IV.1. Diagramme DARTSVIEW du système de chauffage 191
IV.1.1. Simulateur du système de chauffage d'une maison 192
IV.1.4. Modèle de tâche «Acquérir Etat des Appareils de Chauffage» 195
IV.1.5. Modèle de la tâche «Acquérir Thermostat» 195
IV.1.6. Modèle de la tâche «Mesurer Température» 196
IV.1.7. Modèle de la tâche «Contrôle Chauffage» 197
IV.1.8. Modèle des tâches de commande 200
IV.2. Génération de code du système de chauffage d'une maison 200
IV.2.1. Fichier XML de description du système de chauffage 200
IV.2.2. Code des normes temps réel du langage C du système de chauffage 201
IV.2.3. Code de la norme Ada du système de chauffage 201
IV.3. DARTSVIEW vs les modifications de modèle DARTS du système de chauffage 201
V. Conclusion 206
Bibliographie 208
Annexe A 215
Entités 215
IGNORE et INCLUDE 216
Eléments 216
Attributs des éléments 218
DTD (Déclaration de Type de Document) 219
XML-DTD: limitations 220

Introduction générale

Les systèmes de contrôle/commande et temps réel ont envahi aussi bien l'industrie, que le domaine du transport, et s'insèrent rapidement dans notre quotidien. L'un des enjeux importants pour les ingénieurs développant ces applications est le *time-to-market* : il faut en effet développer rapidement tout en garantissant fiabilité, tolérance aux fautes, robustesse, et lorsque cela est nécessaire, valider temporellement et fonctionnellement le logiciel.
Nous nous sommes donc intéressés aux cycles et méthodes de développement pour les systèmes temps réel et de contrôle/commande. La première question posée est la relation logiciel/matériel : deux familles d'approches sont utilisées, l'approche en co-design et l'approche traditionnelle visant à séparer initialement logiciel et matériel. Diverses discussions avec des industriels nous ont montré qu'ils appréciaient le côté rassurant du choix de matériel parmi un ensemble de matériels dont les équipes de recherche et développement ont une expérience, plutôt que de partir de zéro. Nous avons donc privilégié dans ces travaux l'approche traditionnelle, répondant mieux aux habitudes de nombres d'ingénieurs.
Pour une approche traditionnelle, deux grandes familles de méthodes de développement se côtoient : les approches orientées objet de type UML, et les approches fonctionnelles de type flot de données. Les méthodes orientées objet sont plutôt orientées par les données alors que les approches fonctionnelles sont dirigées par les traitements. Dans le cadre de ma thèse, nous avons choisi une approche fonctionnelle, bien adaptée aux ingénieurs de formation généraliste qui travaillent dans le domaine des systèmes embarqués et de contrôle/commande.
L'idée générale est de proposer une méthode simple et facilement accessible de développement d'applications embarquées et de contrôle-commande, permettant d'arriver rapidement à un prototypage rapide en utilisant le langage graphique flot de données LabVIEW, mais permettant ensuite simplement le passage à un langage impératif traditionnel pour l'embarqué (langage basé sur C et Ada notamment). Le résultat est une bibliothèque (palette) de fonctionnalité pour LabVIEW appelée DARTSVIEW. Sa prise en main très simple permet cependant une implémentation multitâche des applications de contrôle commande sur station de travail – elle est d'ailleurs utilisée depuis 3 ans en travaux pratiques pour des élèves ingénieurs de 2ème année de l'ENSMA. Puis nous l'avons améliorée de façon à la rendre utile pour l'embarqué dans le cadre d'un cycle de développement en W. L'idée est d'obtenir rapidement une application test en environnement LabVIEW, qui est un langage très adapté et répandu pour développer des applications de contrôle/commande. On peut mener des tests fonctionnels et d'intégration en connectant ce prototype logiciel à un simulateur numérique du procédé. Ensuite, DARTSVIEW permet de générer du code, en C, Ada ou autre, permettant de porter rapidement ce système sur un microcontrôleur pour l'embarqué. Les tests fonctionnels ayant déjà eu lieu lors de la première phase, les tests nécessaires sont alors réduits sur l'architecture finale, connectée physiquement au procédé. La dernière phase concerne la validation temporelle du procédé : lorsque le système est à contraintes de temps, une analyse d'ordonnançabilité est nécessaire, là encore DARTSVIEW utilise la génération de code pour générer un modèle de tâches qui sera passé à des outils de validation temporelle.
Les langages cibles sont mouvants, de même que les outils de validation cible. Par conséquent, l'accent a été porté sur l'évolutivité de la génération de code, puisqu'un ingénieur utilisant DARTSVIEW n'aura qu'à définir un fichier XML pour adapter la génération de code à un langage cible spécifique. Cette démarche a d'ailleurs été effectuée par un stagiaire ingénieur qui a ajouté à DARTSVIEW la possibilité de générer du code vers OSEK/VDX. Ce générateur a ensuite été utilisé pour aider au développement du système embarqué du mini-drone AMADO [Tra07].

Ce mémoire est organisé en 4 chapitres.
Le premier chapitre introduit les systèmes de contrôle/commande, puis met l'accent sur la relation entre logiciel et matériel. Différents cycles classiques de développement sont présentés et utilisés sur une application de contrôle/commande simple utilisée comme fil conducteur du manuscrit. Nous introduisons le cycle en W et montrons son intérêt dans le contexte étudié. Après une brève présentation des éléments de programmation classiques, nous présentons différentes méthodes de spécification et conception utilisables dans un cycle de développement en W. Enfin, nous donnons quelques éléments d'ordonnancement temps réel, dernière phase du cycle de vie en W.
Le second chapitre présente en détail la méthode DARTS, qui est la méthode de conception sur laquelle s'appuie DARTSVIEW, et montre comment implémenter ses concepts en LabVIEW, C (norme POSIX et bibliothèque VxWorks), et dans la norme Ada.
Le troisième chapitre présente DARTSVIEW en détails et montre comment nous avons utilisé XML pour rendre DARTSVIEW évolutif en regard de la multitude de langages de programmation cible pour l'embarqué et pour la validation temporelle.
Le quatrième chapitre est un cas d'étude complet illustrant le fonctionnement de notre outil.

I. Introduction

I.1. Contexte de l'étude

I.1.1. Définitions des systèmes de contrôle-commande

Un système de contrôle-commande est un système informatique en relation avec l'environnement physique réel externe par l'intermédiaire de capteurs et/ou actionneurs, contrairement aux systèmes d'informatiques scientifiques (gestion de base de données, CAO, etc.) qui ont des entrées constituées de données fournies par des fichiers ou éventuellement un opérateur. Ainsi une définition générale d'un système de contrôle-commande est donnée ci-dessous (voir Figure I-1): «Un système de contrôle-commande reçoit des informations sur l'état du procédé externe, traite ces données et, en fonction du résultat, évalue une décision qui agit sur cet environnement extérieur afin d'assurer un état stable» [CG05]. Il est important de noter que la notion de l'état stable peut être différente selon les applications ou procédés (il dépend du cahier des charges de l'application, par exemple maintien d'une température de consigne, régime moteur, qualité de service…).

Un procédé est une entité physique dont on peut connaître l'état par le biais de capteurs et qu'on peut commander par le biais d'actionneurs. Un système de contrôle commande est le « cerveau » du procédé. Son rôle est de recevoir des informations en provenance de l'environnement à l'aide des capteurs, et de commander des changements d'état du procédé via des actionneurs. L'interaction du système de contrôle-commande avec le procédé extérieur à piloter se décompose en deux parties:

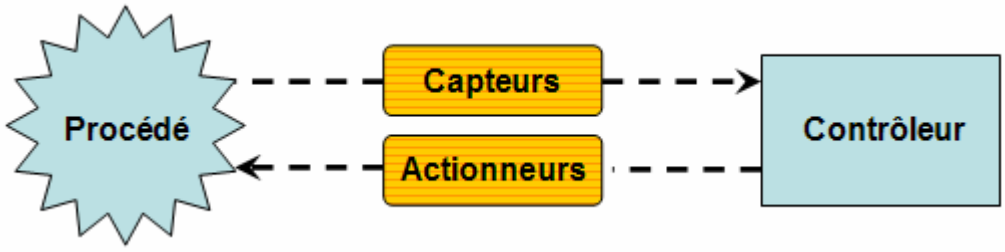


Figure I-1: Interaction entre le procédé et le contrôleur

- observations par l'intermédiaire de capteurs (*sensors*) qui permettent d'obtenir des informations sous la forme d'«interruptions» (information tout ou rien), ou des «mesures» (information continue) en provenance du procédé physique,
- actions réalisées par l'intermédiaire d'actionneurs (*actuators*) qui permettent d'agir sur le procédé physique sous la forme de «commandes» (modification d'état physique du système), ou simplement sous la forme d'un «affichage» (diodes, afficheurs, écrans, etc.).

En qualifiant des systèmes de contrôle-commande avec des spécifications particulières, on peut classifier les systèmes de contrôle-commande en différentes catégories :

- Le système temps réel: un système de contrôle-commande dans lequel l'exactitude des applications ne dépend pas seulement du résultat mais aussi du temps auquel ce résultat est produit. Si les contraintes temporelles de l'application ne sont pas respectées, on parle de défaillance du système [CDKM00]. Ces contraintes temporelles peuvent être de deux types:
  - *contraintes temporelles strictes ou dures*: dans cette catégorie, le système se compose de tâches à contraintes strictes. Chaque occurrence de ces tâches est associée à une échéance que le système doit impérativement respecter. Une faute temporelle peut avoir des conséquences catastrophiques (humaines, matérielles…),
  - *contraintes temporelles relatives ou lâches*: dans cette catégorie, le système se compose de tâches ayant des échéances, comme dans le cas des systèmes à contraintes strictes.

Cependant, une violation d'une contrainte temporelle est tolérable et a un impact sur la qualité de service que l'on cherche à minimiser.

- Le système de contrôle-commande embarqué (*embedded real-time system*): un système, piloté par un logiciel, est complètement intégré au système qu'il contrôle (donc, pas de l'intervention humaine directe, pas de modification du programme ou des paramètres du programme).
- Le système de contrôle-commande dédié (*dedicated real-time system*): c'est un système dont les architectures matérielles ou logicielles sont spécifiques à l'application (noyau, processeur, etc.).
- Le système de contrôle-commande distribué ou réparti (*distributed real-time system*): c'est un système dont l'architecture matérielle est constituée de plusieurs processeurs reliés entre eux par un bus ou un réseau.

Il est évident que ces différentes spécifications d'un système de contrôle-commande peuvent se combiner comme par exemple un système de contrôle-commande dédié, distribué et à contraintes temporelles strictes (application pour un véhicule automobile).

I.1.2. Caractéristiques des systèmes de contrôle-commande

Les caractéristiques représentées dans cette section sont les principales caractéristiques des systèmes de contrôle-commande. Certaines d'entre elles peuvent être aussi caractéristiques des autres systèmes (par exemple, systèmes transformationnels, systèmes interactifs).

- *Grande diversité des dispositifs d'entrées/sorties*: les données à acquérir qui sont fournies par les capteurs et les données à fournir aux actionneurs sont de types très variés (continu, discret, tout ou rien ou analogique).
- *Prise en compte des comportements concurrents*: c'est une caractéristique spécifique aux systèmes de contrôle-commande, car l'ensemble de ces données physiques qui arrivent de l'extérieur ne sont pas synchronisés au niveau de leurs évolutions.
- *Respect des contraintes temporelles*: c'est une caractéristique requise pour tous les systèmes de contrôle-commande temps réel. Par conséquent, le système informatique possède non seulement une réactivité suffisante pour prendre en compte tous les comportements concurrents et en réponse à ceux-ci, mais doit envoyer des commandes en respectant des délais compatibles avec la dynamique du système.
- *Sûreté de fonctionnement*: la sûreté de fonctionnement est l'évaluation du niveau de confiance justifié qu'on peut lui attribuer dans son bon fonctionnement. La sûreté de fonctionnement regroupe les activités d'évaluation de la fiabilité, de la maintenabilité, de la disponibilité, et de la sécurité (FMDS). Les systèmes de type contrôle-commande demandent un niveau important de sécurité pour raisons de coût ou de vies humaines. Pour répondre à cette demande, il est nécessaire de mettre en œuvre toutes les réponses de la sûreté de fonctionnement (développement sûrs, tests, méthodes formelles, prévisibilité, déterminisme, continuité de service, tolérance aux fautes, redondance, etc.).

I.1.3. Architecture des systèmes de contrôle-commande

Une application de contrôle-commande est constituée de deux composantes fondamentales. La première composante est matérielle. Il s'agit des interfaces permettant de piloter le procédé et de recueillir des informations en provenance de ce dernier (i.e., des cartes d'entrée/sorties analogiques ou numériques,…). La deuxième composante est logicielle. Elle assure le bon fonctionnement du procédé. C'est-à-dire que ce dernier puisse satisfaire toute sa spécification sans aucune défaillance. Cette double composante implique une spécification et un développement

conjoints des parties matérielles et logicielles ainsi que des tests d'intégration lors de la mise en œuvre effective sur le procédé.

I.1.3.a. Architecture logicielle des systèmes de contrôle-commande

Un système informatique de contrôle-commande doit satisfaire a priori la cohérence entre des changements de l'environnement et des réactivités du système face à ces changements. A cause du comportement concurrent des événements et des données externes, il est nécessaire de décrire l'environnement en tant qu'un système fortement parallèle. Il en résulte qu'une architecture adaptée pour répondre à ce comportement parallèle du procédé externe est une «architecture multitâche». Cette architecture permet de faciliter la conception de logiciel en définissant les tâches comme «les entités d'exécution et de structuration de l'application» [CG05]. En effet, dans cette architecture logicielle multitâche, nous pouvons découper cet ensemble de tâches selon les groupes suivants:

- *Tâches d'entrées/sorties*: permettent d'accéder aux données externes fournies par des capteurs, ou de générer des commandes aux actionneurs par l'intermédiaire de cartes d'entrées/sorties. Ces tâches peuvent être exécutées de façon régulière ou par interruption.
- *Tâches de traitement*: constituent le cœur de l'application. Elles peuvent être considérées comme des boîtes noires, exécutant des algorithmes issus par exemple du traitement du signal, du traitement d'images, ou de l'automatique…
- *Tâches de gestion de l'interface utilisateur*: permettent de présenter l'état du procédé ou de sa gestion à l'utilisateur. Donc, l'utilisateur peut modifier les consignes données ou changer les commandes.
- *Tâches de communication*: ces tâches sont destinées à gérer les messages envoyés ou reçus à travers un ou plusieurs réseaux ou bus de terrain. Si ce type de tâches existe, l'application est dite distribuée ou répartie.
- *Tâches de sauvegarde*: permettent de stocker l'état du système à des instants fixés afin de pouvoir être utilisé a posteriori en cas de l'analyse de son fonctionnement, ou de reprise d'exécution à une étape précédente.

Les tâches obtenues, qui constituent le système informatique de contrôle-commande, ne sont pas des entités d'exécution indépendantes. Certaines tâches dès leur activation communiquent et/ou se synchronisent avec les autres tâches, ou sont connectées à l'extérieur pour les entrées/sorties. Ainsi, dans l'architecture logicielle multitâche, il faut que le concepteur adresse des problèmes des relations entre tâches, qui sont représentées typiquement par la synchronisation, la communication, le partage de ressources (ces types de relations sont décrits en détails aux paragraphes suivants de ce manuscrit), afin d'assurer le comportement concurrent des systèmes de contrôle-commande.

I.1.3.b. Architecture matérielle des systèmes de contrôle-commande

L'aspect matériel a une très grande importance dans les applications de contrôle-commande. Cette implication est liée d'une part à la connexion directe avec le monde physique réel à l'aide d'une grande diversité de systèmes d'entrées/sorties, et d'autre part au matériel informatique parfois spécifique et développé pour une application donnée. En fait, l'architecture matérielle décrit l'agencement de composants électroniques ainsi que leur interaction. Elle est souvent grandement masquée aux utilisateurs par le biais du système d'exploitation, qui donne une abstraction unifiée de tous les composants physiques. Dans ces matériels, nous pouvons trouver des processeurs du type microcontrôleur, des mémoires, des horloges, des composants spécifiques (i.e., ASIC: *Application Specific Integrated Circuit*), des alimentations…, disciplines qui débordent largement le contexte de ce manuscrit. Pour les lecteurs qui s'intéressent à l'architecture matérielle, consulter le chapitre 4 de [CG05], ou la section II de [Zur05].

I.2. Cycle de développement d'un logiciel

Le génie logiciel est *«l'application pratique de la connaissance scientifique dans la conception et l'élaboration de programmes informatiques et de la documentation associée nécessaire pour les développer, les mettre en œuvre et les maintenir»* [Boe76]. Le génie logiciel consiste à appliquer des méthodologies, c'est-à-dire à développer et à utiliser des méthodes et des outils dans le but de produire des logiciels de qualité en respectant des contraintes de temps et de coût. En effet, on cherche à développer des logiciels qui correspondent aux besoins des utilisateurs de ces logiciels. Le cycle «problème-solution» en est un exemple. Dans ce cas, on est souvent en quête de la solution. Mais il n'existe pas de solution définitive parce que la solution proposée peut être inadaptée (le problème a été mal compris) ou l'application de cette solution peut entraîner des nouveaux problèmes ou mettre à jour des problèmes déjà existants. Donc, le fait de trouver une solution n'implique pas la fin des problèmes [Bro87].

Il existe des «critères» de qualité dont un développement peut être fait pour satisfaire tout ou partie d'un ensemble de ces critères.

- Exactitude: le plus important des critères de qualité. Exigence d'un logiciel à être exempt d'erreur; à fournir des mêmes résultats dans les mêmes conditions normales d'utilisation. Notons que ce déterminisme et cette reproductibilité des résultats ne sont pas adaptés au temps réel.
- Robustesse: Capacité d'un logiciel à maintenir ses performances malgré des changements dans les conditions de fonctionnement ou la présence d'incertitudes liées à ses paramètres. Il s'agit du fait qu'un logiciel doit bien réagir lorsque l'on s'écarte des conditions normales d'utilisation.
- Extensibilité: Possibilité pour un logiciel d'utiliser tout ou certaines parties de ce logiciel pour développer un autre logiciel répondant à d'autres besoins, sans modification ou avec des modifications mineures.
- Portabilité: Facilité avec laquelle on peut exploiter un logiciel dans différentes implémentations. On distingue la portabilité de la compatibilité. Le terme *compatibilité* est plutôt employé pour désigner la capacité d'un ordinateur et du matériel associé à communiquer entre eux et à échanger des données ou de deux ordinateurs à exécuter les mêmes programmes.
- Efficience: Caractéristique d'un logiciel dans laquelle il y a un gain de temps, de taille,… pour satisfaire des besoins des utilisateurs. Ce critère est très important dans les applications embarquées.

Le développement d'un logiciel se fait suivant un cycle, appelé le cycle de vie d'un logiciel. Le cycle de développement du logiciel est un ensemble de phases permettant de transformer à travers un logiciel les besoins en des processus de traitement automatique d'information en provenance de son utilisateur, de son environnement ou de lui-même pour répondre à ces besoins. La décomposition en phases de développement permet aussi de faciliter le suivi, la vérification et la validation du projet par le concepteur et l'utilisateur.

Il existe de nombreux modèles de cycle de vie: modèle en «Cascade», modèle en V, modèle en «Spirale», modèle en W,… Malgré des différences de ces modèles, tous se composent des différentes phases qui sont échelonnées dans le temps:

- Spécifications des besoins
- Conception préliminaire
- Conception détaillée
- Implémentation
- Tests unitaires et tests d'intégration

- Validation
- Exploitation/Recette

Une phase se termine par la remise d'un (ou plusieurs) document(s) validé(s) conjointement par l'utilisateur et le concepteur. Une phase est terminée lorsque la revue de cette phase est faite. Et une phase ne peut commencer que lorsque la précédente est terminée.

I.2.1. Le cycle de vie en «Cascade»

Etant proposé par W. Royce [Roy70], le modèle en «Cascade» est considéré comme le modèle classique de développement d'un logiciel. A l'origine, le cycle de vie en «Cascade» n'est pas itératif. Il est vu comme un processus séquentiel comprenant les phases suivantes: la Spécification, la Conception Préliminaire, la Conception Détaillée, l'Implémentation, des Tests unitaires et tests d'intégration. Si une erreur se produit dans une de ces phases, elle va se propager à des phases successives. Et des changements ne sont pris en compte qu'à la fin de la dernière phase. Ce qui pose un surcoût de développement et de validation d'un logiciel.

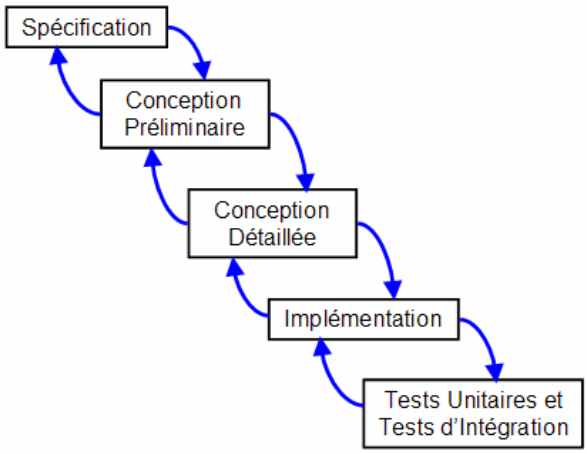


Figure I-2: Modèle en Cascade affiné

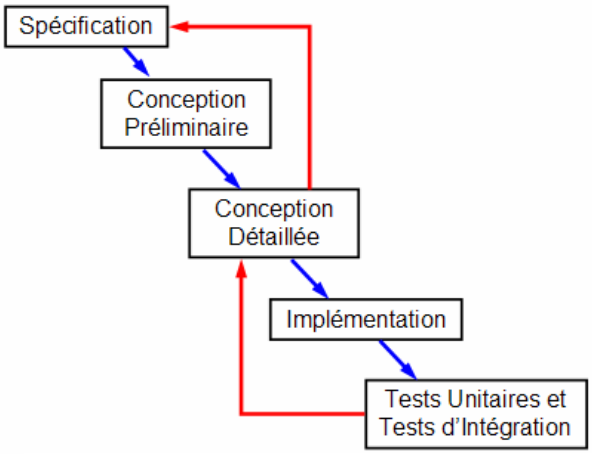


Figure I-3: Itérations dans la phase de Conception

Le modèle de vie en «Cascade» affiné est présenté sur la Figure I-2. Dans ce modèle, chaque phase peut mener à la précédente ou la suivante mais rarement aux phases plus éloignées dans la séquence. La vertu de ce modèle dans le développement est qu'il existe une liaison entre les phases après la phase de Spécification, ce qui permet de retourner au point pour lequel une difficulté imprévue survient. Cependant, dans certains cas, une telle démarche peut être risquée et échouer. Ainsi, si une erreur détectée dans la phase de Tests est liée à une mauvaise Conception ou à une Spécification imprécise, un effort pour refaire le code dans la phase d'Implémentation peut ne pas remédier au problème. En effet, le développement peut retourner de nouveau à l'origine de la phase de Spécification (voir Figure I-3).

Un autre modèle en «Cascade» affiné est proposé dans [Bal94]: il permet de revenir depuis une étape à l'une quelconque des étapes précédentes. W. Boehm dans [Boe88] propose un modèle en «Cascade» raffiné vers une étape "build it twice" en incorporant un prototype à chaque étape pour le feed-back.

Dans le schéma de la Figure I-2, on retrouve:

- La Spécification: l'appréhension du système passe par une vue totale de toutes ses caractéristiques pour répondre à la question "De quoi s'agit-t-il?". Ce sont des aspects exhaustifs du système qui décrivent non seulement des besoins fonctionnels (les services attendus du système), mais également des besoins non fonctionnels, c'est-à-dire les

services comportementaux, matériels et les qualités requises. En effet, pour les systèmes temps réel, il faut s'étendre sur des contraintes temporelles, d'interfaces de communication, etc. La Spécification peut être exprimée de manière informelle en langue naturelle, mais elle peut être complétée sous forme de notations ou de diagrammes formels comme le proposent Z, B, SART, UML, Statechart,…

- La Conception: cette phase peut se diviser elle-même en deux étapes, la Conception Préliminaire et la Conception Détaillée. Elle a pour objectif de définir une structure interne du système vis-à-vis des spécifications imposées dans la phase précédente pour répondre à la question "Comment fait le système?". La Conception Préliminaire produit une architecture générale du système en le décomposant en composants constitutifs, notamment des modules et des tâches concurrentes. Un module est une unité regroupant un ensemble de fonctions ou procédures destinée à être utilisées en conjonction avec d'autres composants. Une tâche concurrente est une tâche pouvant s'exécuter en parallèle d'autres tâches. Le découpage en modules et tâches concurrentes impose une différence entre les systèmes temps réel et les autres. Elle constitue le point de vue statique et le point de vue dynamique de la structure interne du système. La Conception Détaillée permet à partir du résultat de la Conception Générale de poursuivre le découpage du système jusqu'à arriver à une description détaillée de chacun des composants. Cette phase consiste à décrire les choix algorithmiques, à préciser la signature des tâches et des fonctions, à définir les structures de données, et à spécifier les protocoles de communication. Comme la phase de Spécification, il est possible d'utiliser une méthode de conception lors de cette phase. DARTS, UML, etc. font partie des méthodes de conception.
- L'Implémentation: cette étape permet d'implémenter les concepts proposés dans la phase de Conception. Lors de cette étape, chacun des composants est encodé en langage de programmation. En effet, pour les systèmes concurrents, soit un langage concurrent (Ada, Module 2,…) soit un langage séquentiel utilisé conjointement à un système d'opération multitâche ou un noyau temps réel (i.e., C et exécutif temps réel). En outre, un langage de programmation graphique peut être utilisé. Grâce à sa programmation graphique, ce langage permet non seulement d'implémenter des concepts de Conception mais aussi de faire intégrer les composants continus d'une application réelle dans un environnement homogène de manière intuitive. LabVIEW (Laboratory Virtual Instrument Engineering) est un tel langage qui sera utilisé principalement dans la suite pour le Développement d'un logiciel.

I.2.2. Le cycle de vie en «V»

Le cycle de vie en «V» est une variation du modèle en «Cascade». Contrairement au cycle de vie en «Cascade», le cycle de vie en «V» se focalise sur la vérification et la validation. Un processus en «V» s'accomplit en deux pentes: une pente descendante et une pente ascendante. La pente descendante du «V» reprend les étapes progressives du modèle en «Cascade» à partir de la Spécification jusqu'à l'Implémentation. La pente ascendante du «V» porte un ensemble de tests qui permettent de vérifier et/ou de valider le système développé en faisant correspondre chaque étape de tests ou de validation à une étape de l'analyse et de la conception (voir Figure I-4).

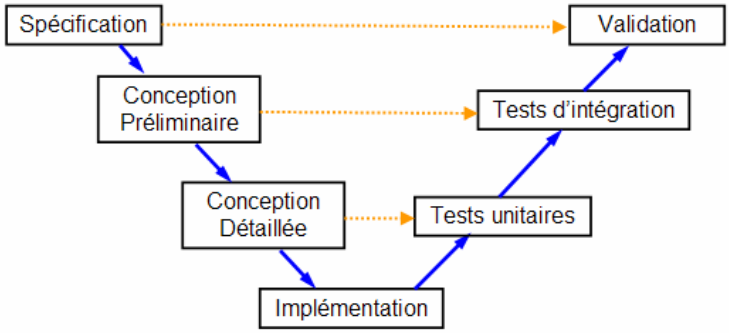


Figure I-4: Modèle en «V»

- Les tests unitaires: permettent de vérifier que les composants indépendants du système répondent chacun à leurs spécifications et fonctionnent correctement en toutes circonstances.
- Les tests d'intégration: sont des tests qui se déroulent suivant les tests unitaires servant à vérifier que les composants réalisés indépendamment fonctionnent bien ensemble. Il s'agit de tester des relations d'utilisation entre les composants, le fonctionnement de tâches qui s'appuient sur les composants, des mécanismes de communication et de synchronisation entre des tâches, etc.
- La validation: permet de vérifier que le logiciel développé répond aux besoins exprimés dans la phase de Spécifications.

Le canevas du modèle en «V» améliore celui du modèle en «Cascade» avec une limitation des retours aux étapes précédentes. Les phases de la partie montante doivent renvoyer des informations sur les phases en vis-à-vis lorsque des défauts sont détectés afin d'améliorer le logiciel. Cependant les difficultés fondamentales du modèle linéaire que l'on affronte dans le modèle en «Cascade» existent encore. Parce que chaque phase est traitée complètement avant que la suivante ne soit commencée, les tests de vérification et de validation ne sont renvoyés qu'en fin du processus d'implémentation. Et parce que la plupart des erreurs sont relevées pendant les spécifications, les retours coûtent chers.

I.2.3. Les autres cycles de développement d'un logiciel

Pour le modèle en «V» en particulier, le modèle linéaire en général s'appuie sur des documents complètement élaborés comme un critère de transition entre des étapes. Un tel modèle est bien adapté pour des systèmes dont on pourrait obtenir la totalité des besoins au départ. Ce qui nous permet d'une part de préparer tardivement des phases de tests, d'autre part de figer ces besoins dans toutes les étapes successives. Pourtant, avec un modèle «dirigé par les documents» [Boe88], plusieurs projets sont produits avec des spécifications détaillées de services qu'ils ne maîtrisent pas encore (i.e., l'interface utilisateur et les opérations fournies). Par conséquence, il y a de grandes chances que dans la conception et le développement de ces projets qui vont suivre, une grande quantité de code soit inutilisable.

Pour des systèmes à risques, il est favorable d'utiliser d'autres techniques de prototypage qui permettent de valider tout ou une partie du cahier des charges et de prendre en considération le «feed-back» entre le concepteur et l'utilisateur. Parmi ces modèles, on retrouve:

I.2.3.a. Le modèle en «Prototype Rapide» (Rapid Prototyping Model)

Ce modèle [Scha02][Som98][Cor07] focalise l'objectif du logiciel à la première étape du cycle de développement: une conception et un prototype seront construits rapidement, ce qui permet de le vérifier avant la phase de spécification et d'exiger moins de boucles de «feed-back» par rapport au modèle en «Cascade» (voir Figure I-5). Ce modèle est très utilisé dans l'industrie automobile, où

le constructeur peut être amené à faire à l'appel d'offre aux instrumentiers en se basant sur un prototype tournant.

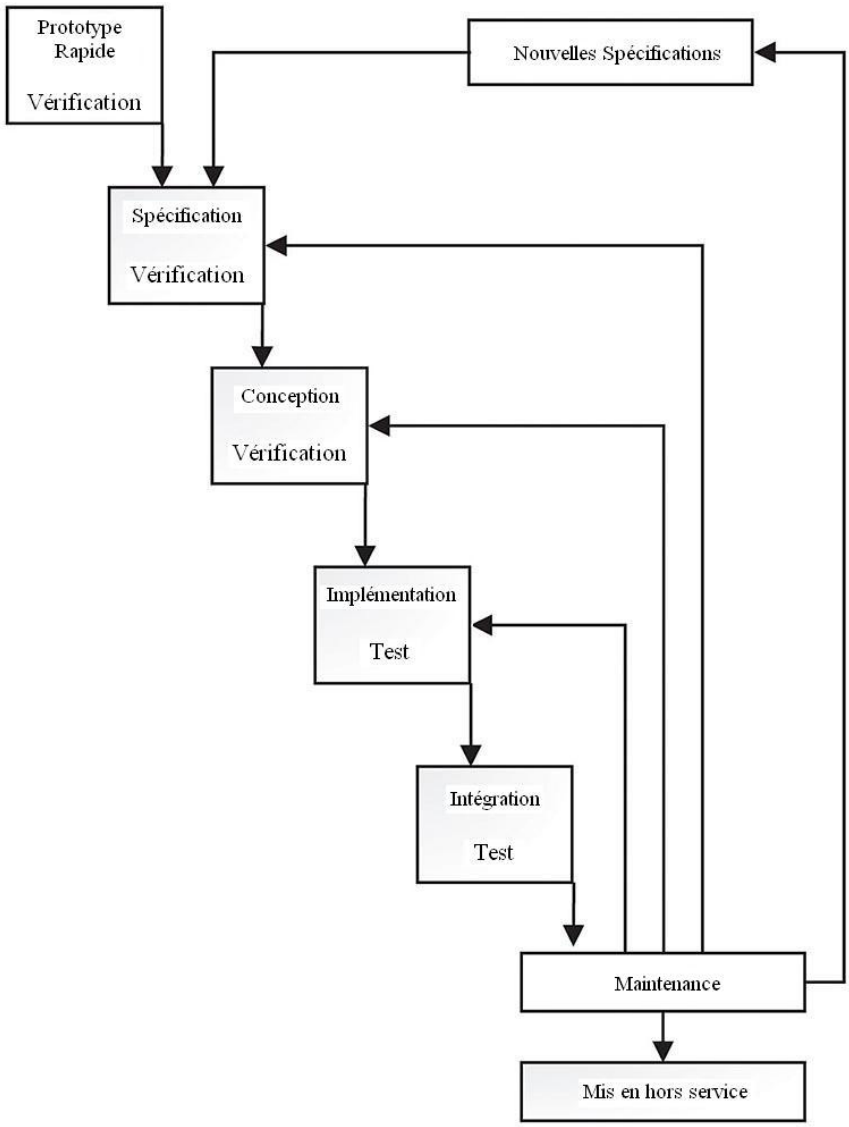


Figure I-5: Modèle en Prototype Rapide

I.2.3.b. Le modèle Incrémentiel et Itératif (Incremental and Iterative Model)

Avec ce modèle [Jac99], le développement est découpé en étapes quand le logiciel se divise en morceaux dont chacun correspond à un ensemble de besoins. A chaque étape, on emploie un modèle de vie linéaire pour élaborer un ou plusieurs morceaux du système simultanément. Puis ces morceaux sont intégrés et testés ensemble. Le produit final est une série de modèles incrémentiels (voir Figure I-6).

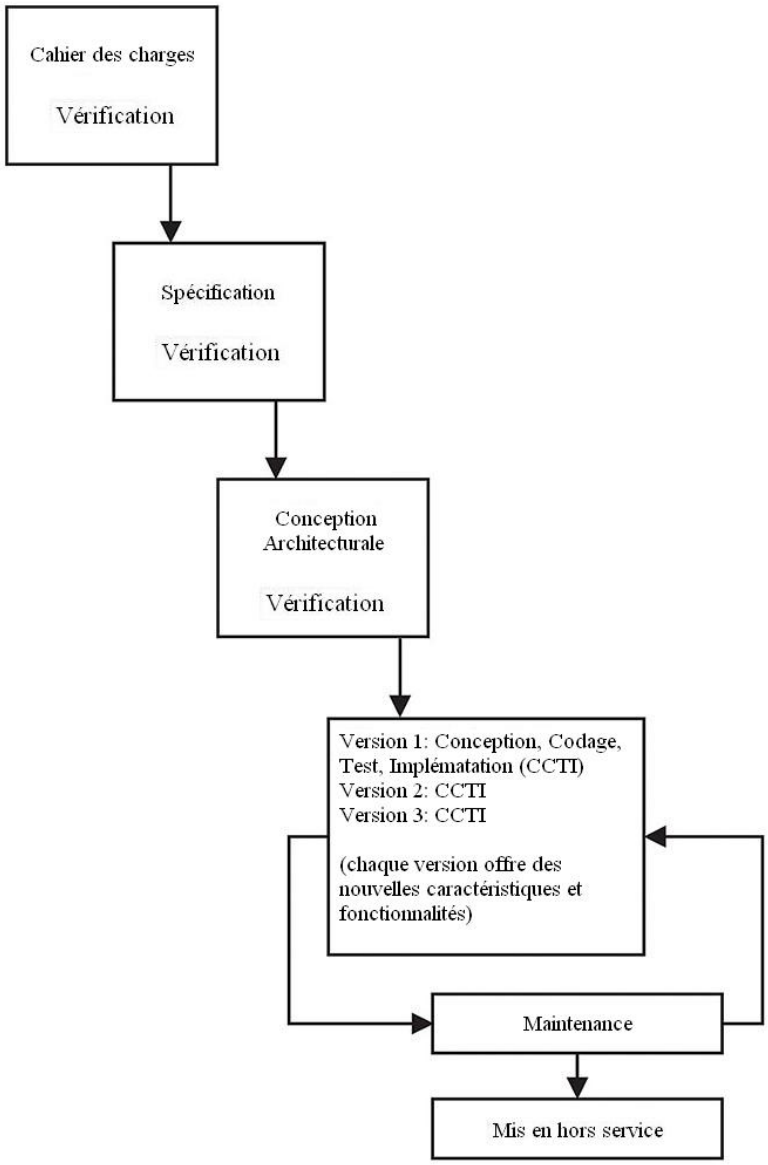


Figure I-6: Modèle Incrémentiel

I.2.3.c. Le modèle en «Spirale»

Le modèle en «Spirale» [Boe88] (est un modèle dirigé par les risques, proposé par W. Boehm. Dans ce modèle, la *dimension* radicale représente le coût cumulatif, la dimension angulaire représente le progrès atteint à chaque cycle de spirale (phase). Une phase commence par l'identification des objectifs de la phase, des alternatives pour obtenir ces objectifs et leurs contraintes. Puis, à la suite, chaque alternative est analysée sous un point de vue «risques». Des prototypes, des simulations,... seront préparés afin de résoudre chacun des potentiels risques. Si tous les risques sont résolus, le concepteur peut poursuivre l'étape suivante. Cette étape correspond à un modèle de vie classique (en «Cascade» ou en «V» avec Recherche des fonctionnalités, Analyse des besoins, Conception Préliminaire, Conception Détaillée, etc.) modifié pour incorporer au développement évolutif. Enfin, les résultats du cycle actuel sont évalués et un plan est vérifié pour les cycles successifs (voir Figure I-7).

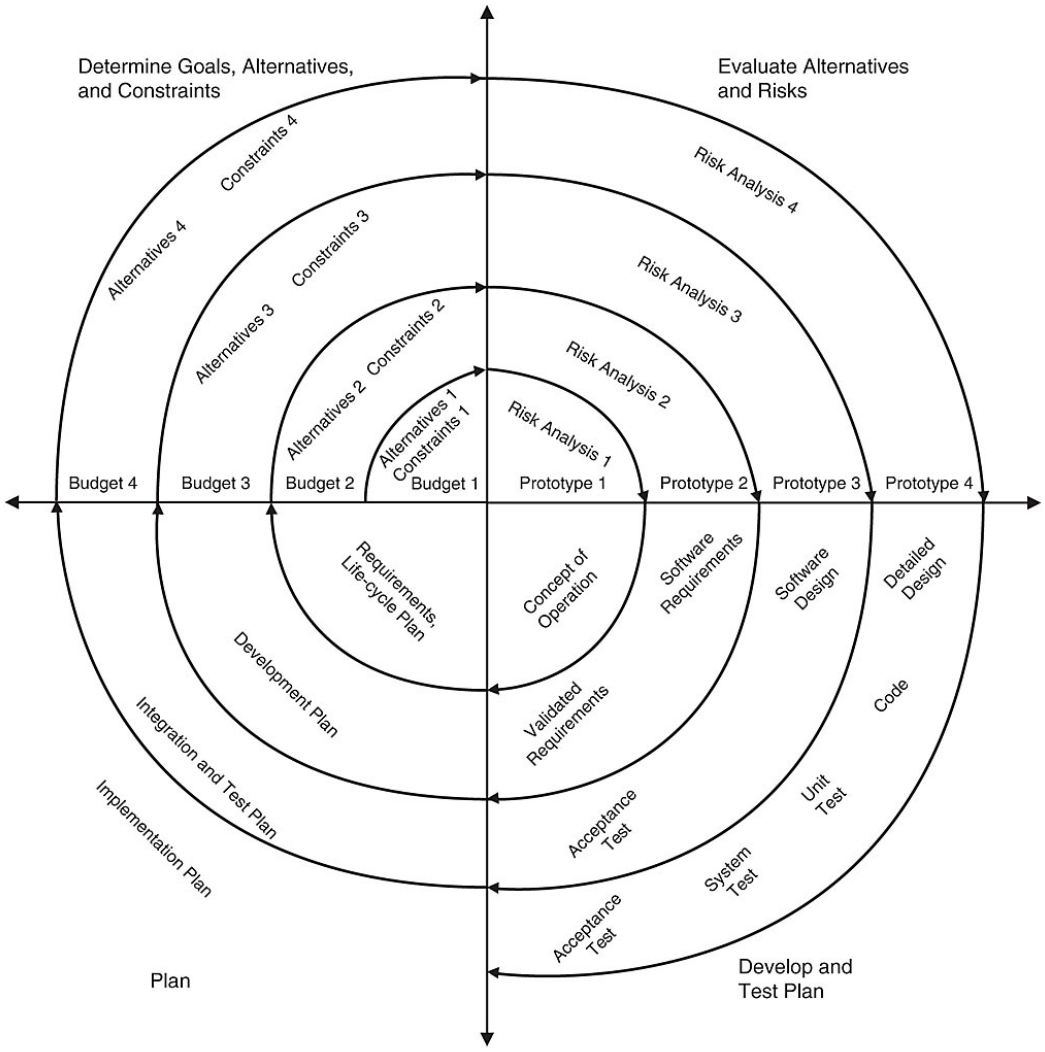


Figure I-7: Modèle en Spirale

I.2.3.d. Programmation orientée-objet (Object-Oriented Programming)

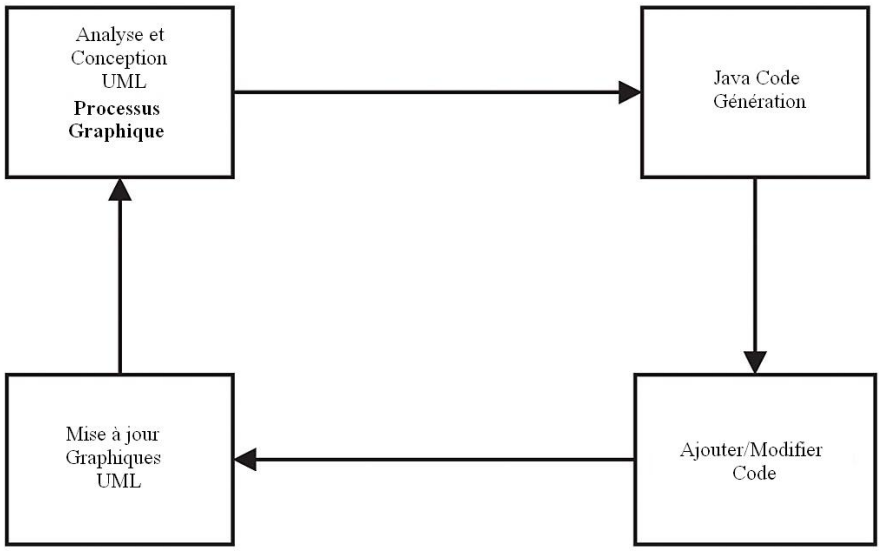


Figure I-8: Modèle «Round-Tripping» de Booch

La programmation orientée-objet (OOP ou POO) n'est pas un modèle de développement d'un logiciel à proprement parler. Elle fait appel notamment à l'encapsulation et au polymorphisme, ce qui permet de réduire la complexité et d'augmenter la réutilisabilité des programmes. Avec le développement des langages de programmation orientée-objet tels que C++, Smalltalk, Java, C#, etc. de nos jours, la POO devient un nouveau style de programmation, ce qui la fait se différencier par rapport à la programmation procédurale traditionnelle. Diverses méthodes utilisées lors du cycle de développement se basent directement sur ce modèle de développement, comme le modèle «Round-Tripping» de Booch [Boo94] représenté sur la Figure I-8. A gauche, nous voyons que les diagrammes UML sont utilisés pour l'analyse et la conception d'un système, tandis qu'à droite le langage Java est utilisé pour l'implémentation. Le code Java est généré à partir des diagrammes UML, et le «feed-back» des changements de code est renvoyé aux diagrammes UML. Le domaine des applications de ce modèle est très vaste, car il est basé sur UML.

Tous ces modèles prennent en charge la totalité du développement du logiciel en prenant en compte le cycle de décision et de l'analyse de risques. Dans le cadre de cette thèse, où nous nous focalisons sur la conception, ce sujet n'est pas traité en détail.

I.3. Le cycle de vie Logiciel/Matériel

En informatique industrielle, on parle d'un système temps réel lorsque ce système informatique contrôle un procédé physique à une vitesse adaptée à l'évolution du procédé contrôlé. Ce système doit réagir continuellement à des événements en provenance du procédé contrôlé avec des contraintes temporelles dont le respect est aussi important que l'exactitude du résultat. Une réaction ne respectant pas son échéance pourrait causer une catastrophe humaine et financière. Un tel système comporte deux parties fondamentales: le matériel et le logiciel. Le matériel est la partie physique (le microprocesseur, la carte d'acquisition, le convertisseur A/D ou D/A, etc.) permettant de piloter le procédé et de recueillir des informations en provenance de ce dernier. Le logiciel est un ensemble des éléments informatiques assurant le fonctionnement du procédé contrôlé. Une part significative du développement d'un système temps réel consiste à décider des composants matériels et des composants logiciels du système, ainsi que du choix de quelles parties peuvent être implémentées comme des logiciels s'exécutant sur des composants programmables et lesquelles peuvent être implémentées sur des matériels spécifiques [Edw 97].

Il existe deux approches: l'approche de développement «Matériel/Logiciel» traditionnelle et l'approche de développement «Matériel/Logiciel» en Co-design.

I.3.1. L'approche de développement «Logiciel/Matériel» traditionnelle

Dans cette approche, le développement d'un système temps réel est divisé en deux parties, une partie matérielle et une partie logicielle (voir Figure I-9). Les deux cycles en «V» dans la Figure I-9 représentent les deux développements distincts de l'architecture matérielle et de l'architecture logicielle. A partir de la conception préliminaire, le concepteur partitionne le système en des composants matériels et des composants logiciels. Parce que la distinction des composants matériel/logiciel à ce niveau n'est pas claire. Le choix dépend de l'expérience du concepteur, du savoir-faire, du coût de production et de maintenance. Le choix du matériel se déroule souvent a priori en fonction des expériences passées du concepteur. Puis, à la suite, les architectures sont développées séparément dans leur propre cycle de vie en «V» avec des phases séquentielles (la conception détaillée, l'implémentation, des tests unitaires). Lorsque le matériel et le logiciel satisfont chacun leur fonctionnement, on les intègre ensemble dans la phase d'intégration et la phase de validation afin de vérifier la communication entre ces deux parts, et aussi le fonctionnement du système considéré (i.e., des tests d'opération, des évaluations des contraintes temporelles,…).

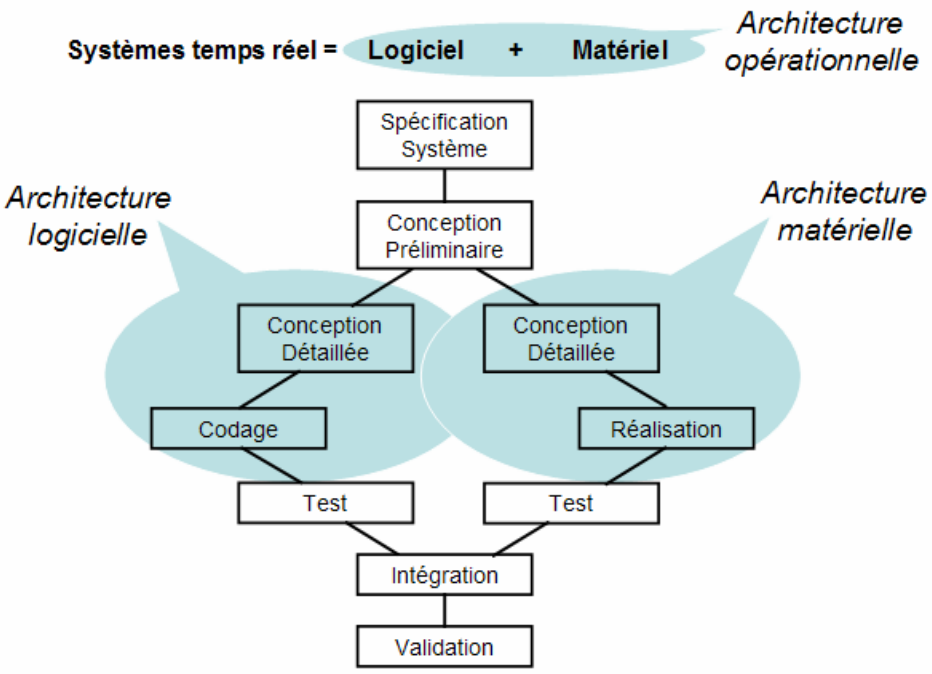


Figure I-9: Modèle de développement «Matériel/Logiciel» traditionnel [CG05]

L'approche développement «Matériel/Logiciel» traditionnelle est une approche très répandue dans le monde des systèmes embarqués temps réel. En séparant un tel système en une partie matérielle et une partie logicielle, l'approche traditionnelle donne une vue claire de l'architecture opérationnelle d'un système temps réel. Cependant, cette approche a quelques inconvénients. Premièrement, la partition de matériel/logiciel a lieu aussitôt à partir de la conception générale, ce qui ne permet pas une séparation évidente des composantes matérielles/logicielles. Le choix de matériel a priori est empirique et dépend de l'expérience du concepteur. De plus, des impacts entre le matériel et le logiciel ne sont pas bien définis. Par conséquent, des problèmes de matériel détectés tardivement dans la phase de l'intégration peuvent entraîner un nombre de modifications de logiciel. Cela va diminuer la qualité et augmenter le coût de développement d'un système. Enfin, la validation des composantes logicielles sur des exécutifs réels n'est pas aisée. Il faut vérifier ces dernières sur les deux aspects, l'aspect fonctionnel et l'aspect temporel. Il existe de nombreuses techniques qui permettent la vérification de l'aspect fonctionnel des composantes logicielles (i.e., des langages formels, des méthodes formelles,…). Mais, pour leurs aspect temporel, des analyses de codes générés sur des exécutifs réels sont souvent longues et coûteuses (humaines et matérielles). Donc, la lacune de l'interaction entre le matériel et le logiciel dans l'approche traditionnelle peut causer des influences imprévues sur le temps acquis pour le développement d'un système au point de la livraison («time-to-market»).

Malgré ces inconvénients, l'approche traditionnelle est encore très utilisée pour les applications embarquées.

I.3.2. L'approche de développement «Logiciel/Matériel» en Co-design

«Matériel/Logiciel» en co-design signifie la rencontre des objectifs au niveau du système en exploitant la synergie de matériel et de logiciel par leur conception concurrente [Mic97]. Contrairement à l'approche traditionnelle, l'approche de développement «Matériel/Logiciel» en Co-design se concentre sur la combinaison des perspectives de matériel et de logiciel dès les premières phases de développement. Alors, un développement va commencer avec une notation abstraite telle que chaque composante ou module est indépendant(e) de sa réalisation finale en matérielle ou en logicielle [Fra91]. Puis, la séparation des composantes matérielles/logicielles peut être effectuée de manière la plus appropriée. L'intégration «Matériel/Logiciel» est ensuite

complétée par un ensemble des instructions permettant des tests et des co-simulations. Un modèle typique de développement en Co-design est présenté sur la Figure I-10 ci-dessous.

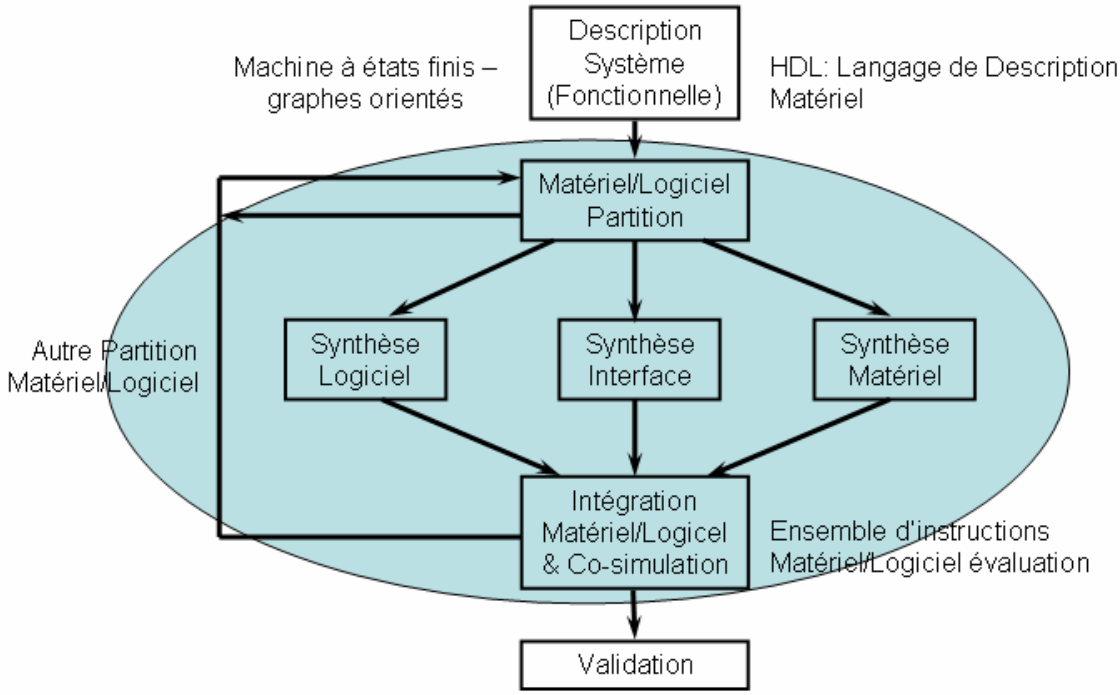


Figure I-10: Modèle de développement «Matériel/Logiciel» en co-design [RASSP]

L'approche de développement en Co-design est une approche encore récente pour les systèmes temps réel. Cette approche est bien adaptée à des applications dont la spécification peut changer continuellement et dont le «Time-to-market» affecte fortement le succès (i.e., systèmes de communication, électroménager,…). L'approche de développement en Co-design se base sur la communication synchrone dirigée par les événements. Dans cette approche, la description fonctionnelle du système est exprimée sous la forme de graphes orientés, appelés des CFSMs (Machines à états finis en Co-design ou Co-design Finite State Machine en anglais). Un CFSM interprète un ensemble d'entrées en un ensemble de sorties avec un nombre fini d'états internes. Mais, contrairement à un FSM traditionnel, un CFSM ne change pas tous ses états exactement en même temps. La communication entre des CFSMs est réalisée par des événements unidirectionnels, non zéro et sans bornes de temps (asynchrone globalement). Il existe aussi des traductions qui permettent de convertir des spécifications en provenance d'un langage synchrone (i.e., ESTEREL [Ber91], SIGNAL [Ben86], HardwareC,…) en des CFSMs. Les CFSMs sont supposés indépendant des choix matériels ou logiciels. Cela permet un choix libre de l'implémentation dans l'étape de «Matériel/Logiciel Partition», tel que l'on transforme des CFSMs en un format de description du matériel dans lequel chaque transition est implémentée par un circuit combinatoire verrouillant ses sorties afin d'assurer leurs non zéro retard de réaction; ou bien que l'on transforme des CFSMs en une structure de logiciel qui comporte des procédures et un système d'exploitation simple exprimés sous la forme d'un s-graphe (une forme réduite de graphe flot de contrôle contenant les nœuds, nommés «Begin», «End», «Call», «Test», «Assign»). Un s-graphe peut être interprété en des codes pour être exécutables par un microprocesseur [Chi94]. Des interfaces entre des composantes matérielles et des composantes logicielles sont générées automatiquement grâce à des techniques de mémoire partagée (pour des communications dans lesquelles l'événement émis par le logiciel est consommé par le matériel), de «scrutation» ou d'«interruption», de communication directe entre des matériels ou des logiciels par «bus» ou «drapeau» respectivement. Cela permet de faciliter la vérification de l'intégration des matériels/logiciels et la validation dans les étapes suivantes. Les co-simulations s'exécutent sur des matériels simulés (i.e., des ASICs, Application Specific Integrated Circuit), ce qui permet de

réduire le coût de production et d'augmenter la fiabilité, mais d'élever également le coût et le temps de développement. Par ailleurs, l'implémentation des logiciels possède un temps de réponse pouvant être plus grand que celui des matériels. Notamment, si le logiciel est basé sur un système d'exploitation avec le mécanisme de préemption, le temps de réaction d'un logiciel ne peut être représenté exactement par un modèle basé sur une hypothèse synchrone.

De nos jours, avec des outils de conception assistés par ordinateur (CAO), l'introduction du FPGA (Field-Programmable Gate Array), et des HDLs (Hardware Description Language), le développement «Matériel/Logiciel» en Co-design devient un vaste domaine de recherche et une clé de technologie pour des systèmes numériques [Mic97], [Stau97] [Cosyma] pour le COSYMA (COSYnthesis for eMbedded micro Achitectures), [Roz94] pour le Ptolemy, [POLIS] [Bala99] pour le POLIS, etc.

Bien que ces méthodes soient bien adaptées pour le développement d'applications très intégrées, il nécessite de la part du concepteur une maîtrise globale du système. Or il est assez classique, notamment dans le monde automobile, et afin de réduire le coût et le «time-to-market», d'avoir un choix de matériels précis (1 ou 2 microcontrôleurs au choix, récupérant des informations sur un réseau de terrain). Dans ce cas, les développements matériels peuvent se réduire à la mise sur circuits de certaines fonctions ne pouvant être exécutées de façon logicielle pour des problèmes de puissance de calcul. Dans ce cas, une approche traditionnelle pourra réduire le temps de développement. Il convient cependant que l'impact du matériel sur le logiciel soit bien connu. De plus, il convient de pouvoir tester le logiciel indépendamment du matériel. Dans ce cas, il est fréquent d'appliquer un cycle de développement en «W».

I.3.3. Le cycle de développement en «W» dans l'approche traditionnelle

Le modèle en «W» [CG05] (voir Figure I-11) est une extension du modèle en «V». Dans ce modèle, le premier V correspond au cycle de vie en «V» classique sur simulateur logiciel (à développer), le deuxième V étend le premier pour prendre en considération les propriétés du matériel dans le développement d'un logiciel.

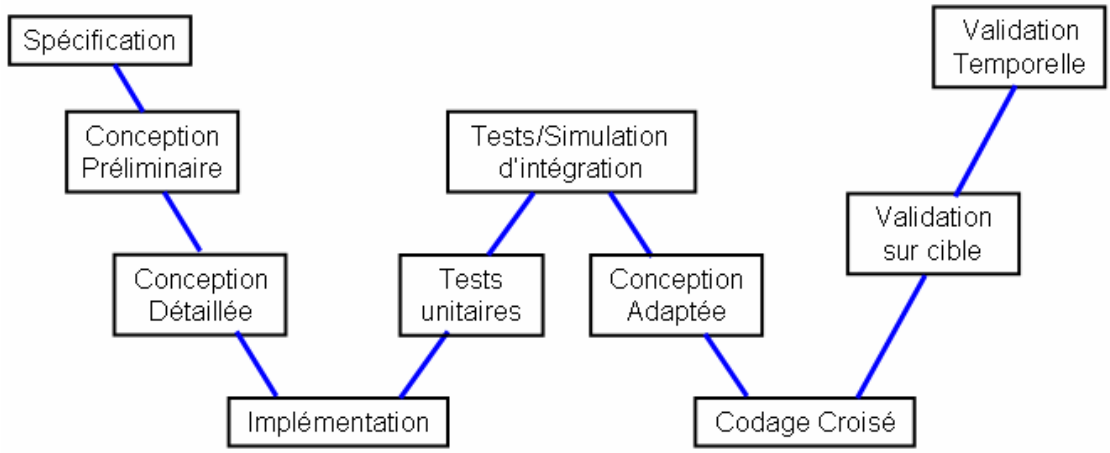


Figure I-11: Modèle en «W»

- La conception adaptée: consiste à prendre en compte les modifications dues à la prise en compte du matériel (i.e., initialisation, alimentation, etc.).
- Le codage croisé: consiste à adapter le code exécutif spécifique, pilotes de périphérique des composants logiciels conformes à une cible implémentée (i.e., norme POSIX, VxWorks, OSEK/VDX, etc.).
- La validation sur cible: vérifie l'intégration de matériel/logiciel pour voir si tout s'exécute bien sur la cible.

- La validation temporelle: vérifie l'ordonnancement du modèle des tâches implémentées sur la cible.

Le modèle en «W» permet une flexibilité du choix du matériel par rapport au modèle en «V» classique. En facilitant des modifications de l'architecture logicielle lorsqu'il y a des changements de l'architecture matérielle, la lacune de l'interaction entre le logiciel et le matériel dans l'approche traditionnelle (voir Figure I-9) peut être résolue. Par conséquent, on peut diviser l'architecture logicielle en deux parties, donc une part indépendante du matériel et une part étant spécifique à une cible réelle (voir Figure I-12). A partir de la conception préliminaire, on prend la première part (le premier V) pour développer le logiciel en des composantes «pures». Ce sont des tâches, des modules, des outils de communication ou synchronisation, des tests et des simulations vérifiant son aspect fonctionnel. Puis dans la part suivante (le deuxième V), en associant des informations en provenance de l'architecture matérielle, on peut générer des codes des composantes logicielles qui s'adaptent bien à une cible. Cela nous permet de tester l'intégration matériel/logiciel et la validation temporelle. Alors, un changement de l'architecture matérielle n'entraîne que des modifications sur la deuxième partie de l'architecture logicielle.

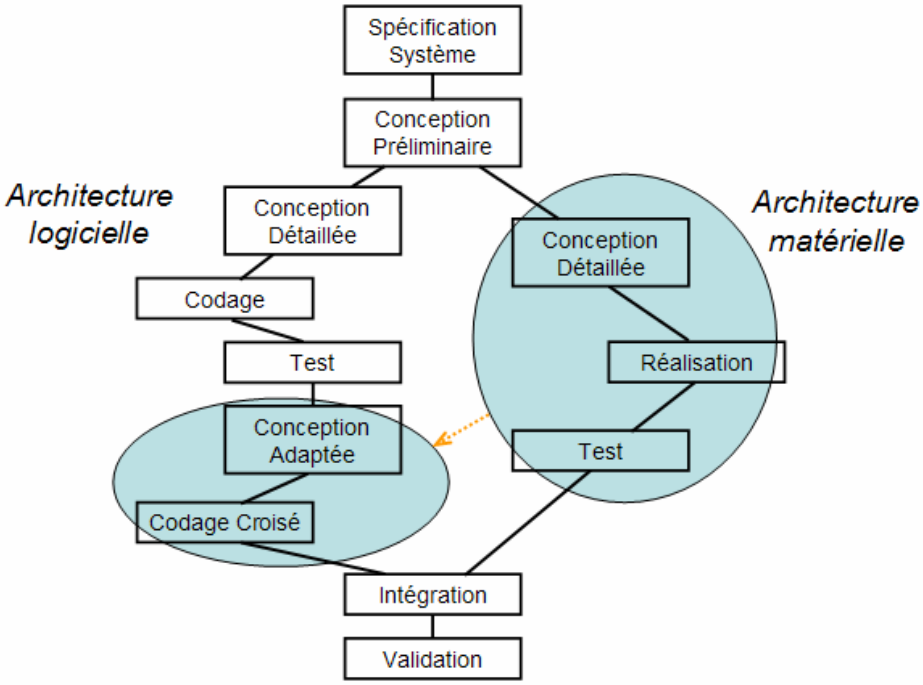


Figure I-12: Modèle en «W» dans l'approche traditionnelle

Dans la suite, le cycle de vie en «W» sera utilisé comme un modèle de développement dans lequel les différentes étapes sont échelonnées et les différentes méthodes sont employées pour produire des logiciels de qualité en respectant les contraintes de temps et de coût.

I.3.4. «Time-to-market», un point clé à succès

«Time-to-market» est le temps séparant la mise en conception d'un produit à sa mise sur le marché. De nos jours, avec le développement très rapide de la technologie, ou bien l'augmentation de demandes de la part des utilisateurs, le «Time-to-market» devient un critère de plus en plus important de réussite d'un produit, notamment dans les domaines de haute technologie, ou dans les domaines où leur environnement et leurs matériels d'input changent rapidement. Si une chaîne de fabrication d'un produit ou un cycle de développement d'un logiciel nécessite plus de temps pour la mise en œuvre ou la mise à jour, ce produit/logiciel peut être démodé à cause des nouvelles demandes en provenance des utilisateurs, ou à cause d'un ou de plusieurs concurrents.

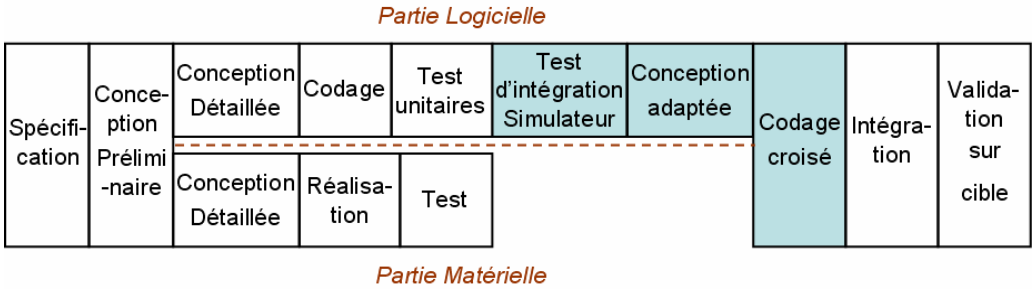


Figure I-13: «Time-to-market» et modèle en W dans l'approche traditionnelle

Pour les systèmes temps réel, le choix d'un cycle de développement adéquat influence considérablement le «Time-to-market». Ainsi, si l'on compare le cycle de développement Matériel/Logiciel traditionnel avec le modèle de développement de logiciel en «W» (voir Figure I-13), une modification dans la partie matérielle peut être mise à jour aisément par la partie logicielle correspondante grâce à des phases dans le deuxième V du cycle de développement en W (au lieu de toutes les phases de ce cycle). En conséquence, le «Time-to-market» pour la mise à jour d'un produit à cause de cette modification se voit réduit. Autrement dit, ce cycle de développement peut s'adapter aux changements des composants matériels. En fait, il est difficile de dire quel est le meilleur cycle de vie pour le «Time-to-market», car il faut relier cela aux domaines d'applications, ou bien aux caractéristiques spécifiques de chaque cycle. Donc, le choix d'un cycle de vie convenable pour le développement d'un système devient une «tâche» vitale, il est souvent choisi en fonction des expériences de concepteur.

I.4. Support d'exécution pour le temps réel

I.4.1. Exécutif ou noyau temps réel

Un noyau de système d'exploitation (*kernel*) est la partie fondamentale de certains systèmes d'exploitation. Le rôle du noyau est de gérer les ressources matérielles et permet aux différents composants, matériels et logiciels, de communiquer entre eux. Un noyau fournit des mécanismes d'abstraction du matériel («Gestion du matériel»), notamment de la mémoire («Gestionnaire de mémoire»), du (ou des) processeur(s) («Ordonnanceur»), et des échanges d'informations entre logiciels et périphériques matériels. Il autorise aussi diverses abstractions logicielles et facilite la communication entre les processus.

- Ordonnanceur d'un système d'exploitation: n'a de sens qu'en système multitâche. Il gère l'ordre dans lequel les instructions des différentes tâches sont exécutées, est assisté par le *dispatcher* qui est responsable de la sauvegarde et de la restauration du contexte des tâches (commutation de contexte).
- Gestionnaire de mémoire: alloue de la mémoire à des processus lorsqu'ils en ont besoin. Si un processus tente d'utiliser des zones de mémoire ne lui appartenant pas, il peut être évincé automatiquement, c'est-à-dire que le noyau lui-même peut prendre la décision de suspendre ou détruire immédiatement le processus fautif.
- Gestion du matériel: se fait par l'intermédiaire de pilotes de périphériques, qui sont généralement inclus dans l'espace noyau et communiquent avec l'espace utilisateur via les appels système (des fonctions fournies par le noyau, et utilisées par les programmes s'exécutant dans l'espace utilisateur).

Exécutif ou noyau temps réel sont souvent assimilés, dans le sens où l'exécutif est souvent vu comme un noyau possédant des pilotes de périphériques ou des services supplémentaires (pile réseau,…) par rapport à un noyau. Un noyau est généralement léger, et on parle d'empreinte mémoire d'un noyau (occupation mémoire) pour caractériser son encombrement. Les noyaux temps réel fournissent au moins deux types de modules, l'un spécifique pour le temps réel

(gestion des tâches, communication, synchronisation, gestion du temps) et l'autre plus classique (librairies standard). Les applications temps réel font alors appel à la partie temps réel du noyau.

Contrairement au système d'exploitation classique (politiques d'ordonnancement des activités basées sur le partage équitable du processeur; gestion de temporisateurs ou de l'horloge pas assez fine; la concurrence de l'application temps réel avec le système d'exploitation toujours actif…), les principales fonctions d'un noyau temps réel peuvent être scindées en trois groupes ci-dessous. Donc, les tâches de même que les processus concourent à l'obtention du ou des processeurs, et peuvent communiquer ou se synchroniser. La différence est qu'une tâche est caractérisée par une pile et un pointeur d'instruction, mais partage la mémoire avec les autres tâches du même processus; tandis que la mémoire pour les processus est privée (il en résulte que le passage par l'exécutif pour la communication entre processus est plus lourd mais plus sécurisé que pour les tâches):

- Gestion des entrées/sorties.
- Ordonnancement des tâches.
- Relations entre les tâches (synchronisation, communication, accès à une ressource critique en exclusion mutuelle, gestion de temps…).

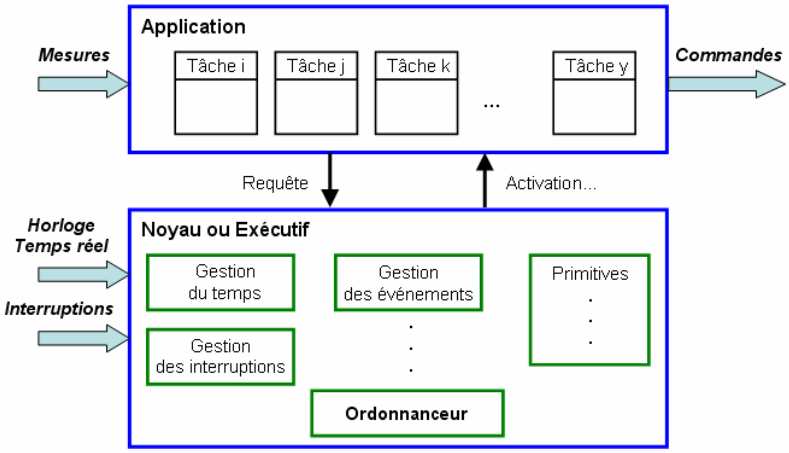


Figure I-14: Architecture de l'application: tâches et noyau temps réel [CG05]

Il est important de noter que les tâches sont les unités actives du système; le noyau temps réel n'est actif que lors de leur appel: donc, une tâche activée peut appeler le noyau temps réel sous forme d'une requête, les différentes requêtes sont servies par des modules du noyau temps réel appelées primitives (partie temps réel du noyau). Ensuite le noyau temps réel réactive une tâche de l'application selon l'algorithme d'ordonnancement utilisé (partie générique du noyau) (voir Figure I-14).

Une autre architecture souvent retenue est un noyau hybride qui s'appuie sur la combinaison d'un noyau temps réel spécialisé, allouant du temps d'exécution à un noyau de système d'exploitation non spécialisé. Cette technique permet d'assurer le fonctionnement temps réel des applications, tout en maintenant la compatibilité avec des environnements préexistants. Nous trouvons une telle architecture dans RTLinux [DM02].

I.4.2. Concepts des tâches

Une tâche est une entité élémentaire d'exécution et de structuration de l'application. Une tâche est localisée dans le temps par une date de début et/ou une date de fin, dont la réalisation nécessite une durée et une échéance (la date à laquelle la tâche doit avoir terminé son exécution). Dans les noyaux temps réel, les tâches peuvent être exécutées par morceaux, ou doivent être exécutées sans interruption (on parle alors respectivement de problèmes préemptibles et non préemptibles).

- Une tâche non préemptible ne peut être interrompue qu'à des endroits spécifiques et à la demande de la tâche elle-même, par exemple fin_de_tâche, attente_signal…
- Une tâche préemptible peut être interrompue à n'importe quel instant et le processeur affecté à une autre tâche.

Lorsque les tâches ne sont définies que par leurs paramètres temporels, on dit alors qu'elles sont des tâches indépendantes. En effet, certaines tâches doivent communiquer avec les autres tâches, ou doivent être connectées vers l'extérieur pour les entrées/sorties. Par conséquent, elles peuvent être liées par un des types des relations suivant:

- Synchronisation: se traduit par une relation de précédence d'exécution entre les tâches,
- Communication: se traduit par une relation de précédence comme la synchronisation mais avec l'échange de données entre les tâches,
- Partage de ressources: les ressources sont les moyens techniques nécessaires à l'exécution des tâches. Au niveau d'un système, les tâches utilisent des ressources mises en commun comme des zones de mémoire, des cartes d'entrées/sorties, etc. Certaines des ressources n'autorisent que des accès en exclusion mutuelle, c'est-à-dire pas plus d'une tâche peut y accéder à la fois, pour avoir un fonctionnement correct. Elles sont appelées ressources critiques.

I.4.3. Gestion des tâches

Les noyaux temps réel gèrent les tâches suivant le même principe que les systèmes d'exploitation non spécialisés. Alors, au cours de la vie d'une application, les tâches peuvent atteindre les états suivants (voir Figure I-15):

- *Tâche «inexistante»*: la tâche n'est pas créée,
- *Tâche «passive»*: la tâche est créée, et initialement elle doit être endormie (passive) pour l'attente de son réveil. De plus, une tâche peut devenir «passive» à partir de l'état «élue» pendant un certain temps ou jusqu'à une certaine date, on parle de «suspension»,
- *Tâche «prête»*: la tâche attend d'être élue. Dans cet état elle requiert un processeur pour s'exécuter,
- *Tâche «élue»*: lorsque le noyau le décide, une tâche en état «prête» alloue un processeur afin d'être exécutée. Cependant, de l'état «élue», une tâche peut être préemptée par une autre tâche. Dans ce cas, elle passe dans l'état «prête» en attendant le processeur,
- *Tâche «en attente»*: de l'état «exécutée», une tâche peut se mettre en attente d'un message, d'un événement, ou de l'accès à une ressource (on parle de «blocage»). Lorsque le message, l'événement (ou la ressource) est arrivé (ou libre), la tâche passe dans l'état «prête» pour requérir le processeur.

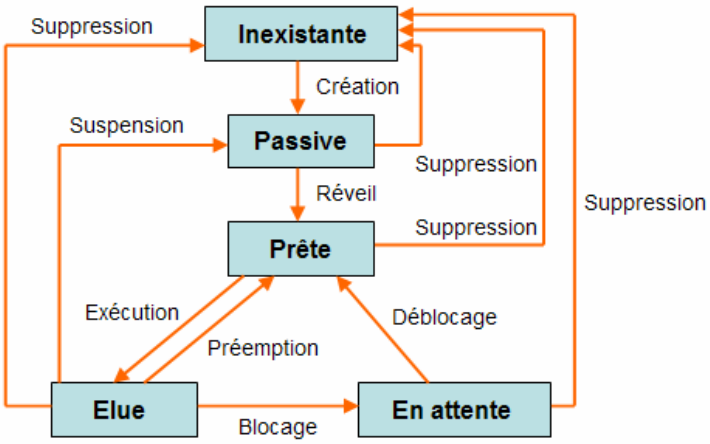


Figure I-15: Graphe des états possibles d'une tâche

A tout instant, une tâche peut être supprimée. Dans ce cas, elle passe dans l'état «inexistante». Etant donné que les tâches sont générées finement dans les noyaux temps réel, certains noyaux distinguent la création et l'initialisation d'une tâche.

I.4.4. Outils de communication et de synchronisation

Ce paragraphe présente les concepts de base des outils de communication et de synchronisation, permettant aux tâches d'interagir. Effectivement, les exécutifs temps réel ne proposent pas toujours tous les outils de communication et de synchronisation, et il faut que dans certains cas le concepteur d'une application les implémente lui-même en utilisant ces concepts de base.

I.4.4.a. Sémaphore

Le sémaphore est un outil introduit par Dijkstra en 1965, donc l'un de ses rôles est de protéger une zone de programme dite critique lorsque le nombre de tâches qui accèdent simultanément à une ressource critique est supérieur à un nombre limité d'instances de cette ressource. En effet, un sémaphore est une variable soit binaire (avec deux états possibles: libre et pris), soit n-aire.

Dans le cas des sémaphores binaires, un sémaphore est caractérisé par une valeur booléenne et une file d'attente de tâches en attente du sémaphore (voir Figure I-16(a)). Tandis que dans le cas des sémaphores n-aire, appelés «sémaphores à compte» ou «sémaphores compteur», un sémaphore est caractérisé par un nombre entier naturel (donc la valeur 0 indique que le sémaphore est pris, et une valeur différente de zéro indique qu'il y a un certain nombre d'instances libres), et une file d'attente de tâches en attente du sémaphore (donc, chaque tâche est couplée avec un nombre entier strictement positif indiquant le nombre d'instances requises) (voir Figure I-17(a)).

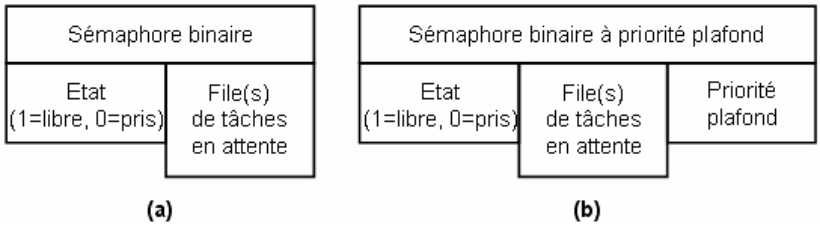


Figure I-16: Caractérisation d'un sémaphore binaire [CG05]
(a) sans protocole de gestion de ressource ou avec protocole à priorité hérité
(b) avec protocole à priorité plafond

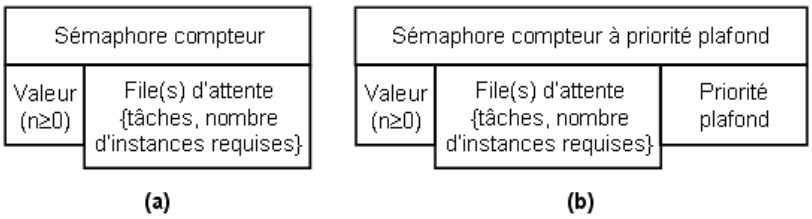


Figure I-17: Caractérisation d'un sémaphore compteur [CG5]
(a) sans protocole de gestion de ressource ou avec protocole à priorité hérité
(b) avec protocole à priorité plafond

La file d'attente peut être gérée, en fonction de l'exécutif sous-jacent, soit de façon FIFO, soit sous la forme de plusieurs files FIFO gérées par priorité des tâches désirant accéder au sémaphore. Lorsque les sémaphores sont utilisés pour assurer l'exclusion mutuelle, ou encore pour permettre un accès de type lecteur/écrivain (soit un lecteur/un écrivain en cas de sémaphore binaire, soit multiple lecteur/un écrivain en cas de sémaphore compteur), ces sémaphores peuvent être couplés, selon les exécutifs, avec un protocole à priorité héritée ou bien un protocole à priorité plafond afin d'éviter le phénomène d'inversion de priorité. Dans le cas du protocole à priorité plafond, il est nécessaire de caractériser chaque type des sémaphores par une «priorité plafond» (voir Figure I-16 (b), Figure I-17 (b)).

I.4.4.b. Moniteurs

Le moniteur est un outil utilisé pour synchroniser deux ou plusieurs tâches qui utilisent des ressources partagées. En fait, l'utilisation de la ressource ne peut avoir lieu qu'à travers le moniteur, et les primitives du moniteur permettant de manipuler cette ressource sont non réentrantes. Le moniteur est strictement plus puissant que le sémaphore. Un moniteur est constitué de:

- un ensemble de primitives permettant l'interaction avec la ressource partagée,
- un verrou d'exclusion mutuelle,
- des variables associées à la ressource.

En pratique, il existe deux types de moniteur, l'un de moniteur classique et l'autre de moniteur à la Ada.

- Le moniteur classique, appelé «moniteur de Hoare»: est un mécanisme dans lequel la non réentrance des primitives est couplée avec un «wait» et un «signal» permettant respectivement de mettre en attente et de réveiller une tâche accédant au moniteur sous certaines conditions. Ce type de moniteur peut être implémenté par exemple dans la norme POSIX 1003.1 à l'aide de variables conditionnelles. Lorsqu'une tâche doit être mise en attente sur le moniteur tant qu'une condition n'est pas réalisée, charge au programmeur d'utiliser les outils «wait» et «signal» pour implémenter les conditions (gardes) d'accès.
- Le moniteur à la Ada (objet protégé): permet de mettre une garde (une barrière logique) à l'entrée de chaque primitive du moniteur, ce qui offre des traitements plus fins et plus transparents des condition d'accès par rapport au moniteur de Hoare. Par exemple, dans le cas où il y a plusieurs conditions de réveil, seule la tâche concernée est réveillée par le changement d'état du moniteur, contrairement au cas du moniteur de Hoare dans lequel toutes les tâches en attente sur «wait» sont réveillées sur un «signal», puis doivent évaluer leur condition afin de se remettre en attente ou de poursuivre leur exécution. De plus, le moniteur à la Ada possède un mécanisme natif de l'accès de type lecteur/écrivain, c'est-à-dire plusieurs tâches peuvent accéder au moniteur en lecture en même temps, mais une seule tâche peut y accéder en écriture à la fois. En outre, le protocole à priorité plafond existe également pour le moniteur à la Ada.

I.4.4.c. Signaux

Les signaux sont des événements (sans données) pouvant être envoyés à une ou plusieurs tâches simultanément. Il y a deux types de signaux: les «signaux synchrones», internes à une tâche ou un processus, et les «signaux asynchrones», provenant d'autres tâches ou processus, ou bien de source matérielle externe [CG05].

- Un signal synchrone est le résultat d'un événement interne à une tâche. Il s'agit lorsqu'un événement interne a lieu, qu'une occurrence du signal synchrone associé ait lieu. Et ce signal est traité immédiatement de façon synchrone à son occurrence par la tâche. Normalement, à chaque signal synchrone est associée une action par défaut, qu'une tâche peut effectuer lorsqu'elle reçoit un signal synchrone.
- Les signaux asynchrones sont des signaux provenant d'une source externe à l'adresse d'une tâche ou un processus (signaux privés), ou bien à plusieurs tâches et/ou plusieurs processus (signaux publics). Lorsqu'un signal privé est envoyé à un processus, la dernière tâche (chronologiquement) à s'être sensibilisé à l'événement peut percevoir l'événement. Lorsqu'un signal public est envoyé à un processus, toutes les tâches sensibilisées au signal peuvent le percevoir. Alors dans les cas de signaux public, un répartiteur d'occurrences s'occupera de prévenir chaque tâche sensibilisée qu'un signal public est envoyé.

Si une tâche n'est pas prête à recevoir un signal au moment où l'occurrence de ce signal a lieu, il existe trois possibilités dont les occurrences non prises en compte peuvent être gérées: «signal fugace» (l'occurrence non prise en compte est perdue, la tâche doit attendre l'arrivée de cette occurrence de nouveau); «signal mémorisé» (l'occurrence non prise en compte est mémorisée en mettant le «drapeau» à 1, par conséquent si une autre occurrence du même événement a lieu d'ici à sa prise en compte, elle est ignorée); «signal à compte» (les occurrences non prises en compte immédiatement sont comptées pour utilisation ultérieure).

I.4.4.d. La boîte aux lettres

Une boîte aux lettres permet une communication asynchrone entre des tâches. Une boîte aux lettres est constituée typiquement d'une zone d'échange tampon (buffer) dans laquelle une tâche dite «émettrice» peut déposer des données, et d'un mécanisme de géstion des données (messages) stockées dans la boîte aux lettres.

- La taille de la zone d'échange est déterminée par le nombre de messages maximum multiplié par la taille de chaque message.
- Les données de la boîte aux lettres sont gérées généralement par une file de type FIFO (i.e., premier déposé/premier retiré). Donc, une tâche dite «réceptrice» retire les données dans l'ordre d'arrivée des données. En fonctions de l'implémentation, les données de la boîte aux lettres peuvent être gérées par niveau de priorité. Ainsi, la priorité de données peut être déterminée d'une façon soit orientée message (chaque message est muni d'une priorité qui influence l'ordre dans lequel il va être lu), soit orientée tâche (la priorité du message est liée à la priorité de la tâche «émettrice»). Il en résulte que la zone d'échange tampon possède plusieurs files d'attente des messages à l'ordre de priorité, et les messages ayant la même priorité sont gérés dans la même file d'attente des messages de type FIFO.

Il est important de noter que la communication par la boîte aux lettres est asynchrone, car la tâche «émettrice» n'a pas besoin d'attendre que la tâche «réceptrice» soit à l'écoute pour lui envoyer des données.

- Si la tâche «réceptrice» désire recevoir des données au moment où la boîte est vide, elle est mise en attente jusqu'à ce que des données aient été déposées dans la boîte (on parle de blocage en lecture).

- Lorsque la tâche «émettrice» désire déposer des données dans une boîte pleine (car la taille de la zone d'échange tampon est normalement bornée), en règle générale, cette tâche elle-même, est mise en état «en attente» jusqu'à ce que des données aient été retirées de la boîte (on parle alors de blocage en écriture, ou de la boîte aux lettres sans écrasement, qui dans ce cas ne fonctionne plus de façon asynchrone). Cependant, en fonction de l'implémentation choisie, le message le plus ancien de la boîte peut être écrasé par le nouveau, en conséquence la tâche «émettrice» ne se bloque jamais en écriture (on parle de la boîte avec écrasement).

La Figure I-18 illustre une représentation récapitulative de la composition d'une boîte aux lettres (sans écrasement et avec écrasement).

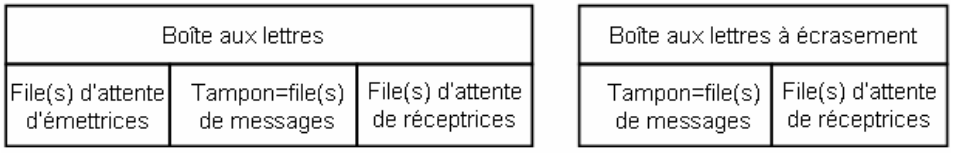


Figure I-18: Composition d'une boîte aux lettres [CG05]

I.4.4.e. Le rendez-vous

Le rendez-vous est une communication synchrone asymétrique des tâches avec passage de données où une tâche dite «acceptante» attend un rendez-vous, et une tâche dite «appelante» demande à effectuer un rendez-vous avec une tâche «acceptante». Lorsque un rendez-vous est établi (donc, la tâche «acceptante» est sur une instruction de type «accept» et la tâche «appelante» demande l'obtention du rendez-vous), la tâche «acceptante» exécuté le code du rendez-vous, tandis que la tâche «appelante» est bloquée jusqu'à la fin du rendez-vous. Des données peuvent alors être passées de la tâche «acceptante» à la tâche «appelante».

Le rendez-vous peut se ramener à l'utilisation de deux boîtes aux lettres, nommées respectivement «bal_demande_rdz» et «bal_fin_rdz» par exemple. Remarquez que dans cette implémentation, la boîte aux lettres de demande de rendez-vous est nécessairement publique, car toute tâche peut demander un rendez-vous, alors que la boîte aux lettres utilisée pour signifier la fin du rendez-vous est spécifique à chaque tâche demandant le rendez-vous.

I.4.4.f. Le tableau noir

Le tableau noir est la forme la plus simple des outils de communication asynchrone. En effet, un tableau noir est une zone de mémoire commune pouvant contenir un message où l'écriture d'un message écrase le message précédent, et la lecture est non bloquante et non destructive (c'est-à-dire, une valeur déjà lue peut être relue plusieurs fois tant qu'elle n'a pas été écrasée par une écriture).

Il est nécessaire d'utiliser un mécanisme de protection pour garantir l'exclusion mutuelle: un accès à la donnée ne doit pas être interrompu par un autre accès. Cependant, il n'est pas gênant que plusieurs accès simultanés en lecture aient lieu. Nous voyons que le mécanisme de protection d'un tableau noir s'apparente au problème du «lecteur/écrivain». La Figure I-19 représente un modèle de la communication par tableau noir.

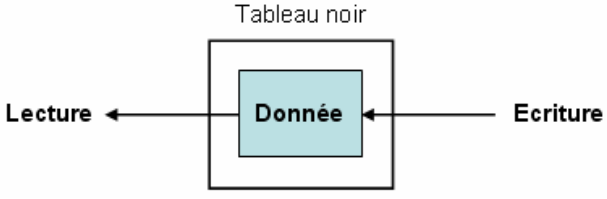


Figure I-19: Communication par tableau noir [CG05]

I.4.5. Gestion du temps

Le temps informatique est compté par des horloges. Il existe deux types d'horloges: le premier type d'horloge est le plus simple qui dépend de l'alimentation électrique et génère une interruption à chaque période de la tension dont la fréquence est souvent de 50 Hz ou 60 Hz (il s'agit d'une horloge de faible résolution de l'ordre de la milliseconde. La plupart des systèmes d'exploitation généralistes sont munies de ce type de l'horloge); le deuxième type d'horloge est plus complexe et peut générer un signal périodique dont la fréquence correspond à la fréquence d'activation de l'ordonnanceur et peut être maitrisée par un logiciel (il s'agit des horloges programmables de haute résolution de l'ordre de la micro seconde que les exécutifs temps réel utilisent).

En pratique, les systèmes d'exploitation temps réel utilisent un ou plusieurs circuits programmables spécialisés dans la gestion du temps. Ces circuits peuvent être programmés de sorte à:

- déclencher périodiquement une interruption, ce type de programmation est utilisé typiquement pour déclencher périodiquement l'ordonnanceur,
- déclencher une interruption à une certaine date, ce type de programmation permet typiquement à une tâche de se réveiller périodiquement, avec une résolution temporelle importante (période),
- déclencher une interruption au bout d'un certain temps, ce type de programmation permet typiquement à une tâche de s'endormir pendant un certain temps, ou bien de surveiller la durée d'un traitement par «chien de garde». Il est associé normalement à une ou plusieurs actions, qui sont lancées immédiatement au cas où le traitement ou l'attente prendrait un temps plus élevé que prévu.

Pour terminer cette section, je parle des modèles de la gestion du temps sur lesquels les exécutifs ou les systèmes d'exploitation temps réel sont actuellement basés. Il est l'un des deux modèles suivants:

- Noyau dirigé par le temps: seule la notion de «tick» est utilisée pour gérer le temps. Donc, le noyau prend à main à chaque «tick», et les tâches ou les processus ne peuvent être réveillés que sur des «ticks».
- Noyau dirigé par les événements: les horloges programmables sont utilisées à leur granularité pour réveiller une tâche ou un processus. Ce modèle permet non seulement d'obtenir le fonctionnement similaire au modèle dirigé par le temps, mais aussi d'atteindre une résolution d'horloge de l'ordre de la micro seconde avec un surcoût processeur moindre par rapport au modèle dirigé par le temps.

I.4.6. Principales normes temps réel

I.4.6.a. La norme POSIX

POSIX (Portable Operating System Interface) est le nom d'une famille des standards définie depuis 1988 par l'IEEE et formellement désignée P1003. Le but au début de cette norme est de normaliser l'accès aux services offerts à différents niveaux par les systèmes d'exploitation UNIX. De nos jours, beaucoup de systèmes d'exploitation, d'exécutifs temps réel ou de noyaux temps réel se disent conformes à la norme POSIX. Dans les paragraphes suivants, je parle de quelques éléments sur les normes POSIX orientées temps réel.

- IPC (Interprocess Communication): proposé dans l'amendement POSIX 1003.1b (1993) corrigée par le 1003.1i en 1995, regroupe les trois outils définis pour la communication entre processus, ce sont la boîte aux lettres, le sémaphore et le tableau noir.

Les boîtes aux lettres sont dites «boîtes aux lettres nommées», car chaque boîte est identifiée par un nom unique dès la création de cette boîte. La manipulation d'une boîte aux lettres nommée se fait à l'aide de fonctions dont le nom est préfixé par «mq_» pour *message queue*, par exemple «mq_open» crée une nouvelle boîte ou bien ouvre une boîte aux lettres existante; «mq_send» et «mq_receive» servent respectivement à envoyer et à attendre des messages d'une boîte aux lettres, etc. Il est important de noter qu'une boîte aux lettres nommée est une boîte aux lettres bornée sans écrasement, avec possibilité d'affecter une priorité aux messages.

Les sémaphores dits «sémaphores nommés» sont des sémaphores à compte. La manipulation d'un sémaphore se fait à travers des fonctions dont le nom est préfixé par «sem_», par exemple «sem_open» crée ou ouvre un sémaphore; «sem_wait» prend un sémaphore; «sem_post» libère un sémaphore, etc. Aucun protocole de gestion de ressource n'est disponible pour les sémaphores nommés.

Les tableaux noirs dits «shared memories», sont les zones des mémoires spécifiques partagées, car la mémoire de chaque processus est protégée. En utilisant la fonction «shm_open» nous pouvons créer ou ouvrir une zone de mémoire partagée. Après avoir ouvert/créé, on peut calquer cette zone de mémoire partagée sur une variable d'un processus avec la fonction «mmap». Notons que tout accès à la variable « mappée » est alors un accès à la zone de mémoire partagée, et non pas l'accès à une variable interne au processus. Ce mécanisme se couple généralement à un sémaphore pour garantir l'exclusion mutuelle.

Pour les boîtes aux lettres nommées et les sémaphores nommés, l'amendement 1003.1j ajoute la possibilité de borner des attentes lors des envois ou réceptions de messages, ou des demandes de sémaphores. Par conséquent, dans ce cas, si le délai ou bien la date maximale est dépassé, la primitive bloquante se termine en renvoyant un code d'erreur.

- Tâches POSIX: définies dans l'amendement POSIX.1c. Des tâches POSIX appartenant à un processus partagent le même espace mémoire global, et se distinguent par une pile différente, ainsi que des attributs particuliers (priorité, type d'ordonnancement…).

  Une tâche POSIX est créée par la fonction «pthread_create». Cette fonction comporte des arguments principaux suivants: des attributs de tâche, un pointeur de fonction (cette fonction est le code de la tâche), et un pointeur pouvant contenir le ou les arguments à passer à la fonction.

  Une tâche POSIX peut être exécutée en mode «détaché» ou «attaché». Donc, en mode «attaché», chaque tâche s'exécute de façon synchrone (join) avec son créateur, dans ce cas, le créateur de la tâche attend la terminaison de celle-ci avant de continuer son exécution. Au contraire, en mode «détaché», les tâches peuvent s'exécuter de façon indépendante de leur créateur (« detach state »). Il en résulte que si les tâches sont créées par la procédure principale d'un processus, afin d'éviter la terminaison prématurée du processus avant la terminaison des tâches, on pourra attacher ces tâches à la procédure principale du processus.

- Synchronisation et communication des tâches: définies aussi dans l'amendement 1003.1c, ce sont des outils de communication et de synchronisation entre des tâches POSIX.

  Sémaphores binaires (ou mutex) : sont dédiés à l'exclusion mutuelle. Les attributs d'un «mutex» permettent de choisir le protocole de gestion de ressource associé au «mutex» (soit priorité plafond immédiate, soit priorité héritée). Les tâches en attente d'un sémaphore sont gérées par les files d'attente FIFO à priorités.

  Sémaphores en lecture/écriture (ou rwlocks) comportent pour chacun les deux sémaphores, nommés «rdlock» et «wrlock», donc la fonction «pthread_rwlock_rdlock» permet de requérir le sémaphore en lecture, alors que «pthread_rwlock_wrlock» requiert le sémaphore en écriture. Aucun protocole de gestion de ressources n'est défini sur les sémaphores en lecture/écriture.

Variable conditionnelle (ou pthread_cond) peuvent être utilisées conjointement avec des «mutex» pour synchroniser des tâches. La fonction «pthread_cond_init» crée une variable conditionnelle. Au cas où plusieurs tâches sont bloquées simultanément sur la même variable conditionnelle, un appel à l'équivalent de «signal(pthread_cond_signal)» ne réveille qu'une seule tâche bloquée sur la variable conditionnelle. Donc, on peut utiliser la fonction «pthread_cond_broadcast» pour réveiller toutes les tâches bloquées sur une variable conditionnelle.

Notons qu'il est possible d'effectuer des attentes bornées sur les instructions bloquantes des objets de synchronisation et de communication (soit en durée, soit jusqu'à une certaine date).

- Horloges: il existe deux types d'horloge, appelés respectivement «CLOCK_REALTIME» et «CLOCK_MONOTONIC».

  «CLOCK_REALTIME» représente le temps écoulé depuis le 1er Janvier 1970 à 00 h 00. Cette horloge permet de donner une date absolue à un «timer». Elle a une résolution au pire de 20 ms, mais en fonction de l'architecture matérielle sous-jacente, on peut attendre une résolution de l'ordre de la micro seconde ou de la dizaine de micro secondes.

  «CLOCK_MONOTONIC» (cette horloge est optionnelle dans les implémentations) a une propriété de monotonie: elle est modifiée par l'horloge physique, mais ne peut pas être modifiée par le programme. Elle permet de donner un délai

  Deux autres horloges peuvent optionnellement exister: «CLOCK_CPU_TIME», donnant le temps processeur consacré à un processus, et «CLOCK_THREAD_CPU_TIME_ID», donnant le temps processeur consacré à une tâche.

  POSIX définit les exécutifs comme étant dirigés par les événements.

I.4.6.b. La norme OSEK/VDX

OSEK est l'acronyme de «Offene Systeme und deren Schnittstellen für die Elektronik im Krafttfahrzeug», en français Systèmes ouverts et interfaces correspondantes pour l'électronique des véhicules automobiles. La norme OSEK/VDX est née en 1995 de la fusion d'un consortium de constructeurs d'automobiles allemands et d'un consortium de constructeurs d'automobiles français (VDX est l'acronyme de «Vehicle Distributed eXecutive»). Bien qu'initialement conçue pour le domaine automobile, cette norme est très bien adaptée aux applications de contrôle-commande.

Contrairement à POSIX où tout est dynamiquement créé, la plupart des objets OSEK/VDX sont définis de façon statique à l'élaboration de l'exécutif et de l'application: tous les services utilisés (tâches, objets de communication et de synchronisation, etc.) sont définis de façon statique dans le langage OIL (OSEK Implementation Language). Cela entraîne une faible empreinte mémoire, notamment des avantages indéniables en ce qui concernent les techniques de validation.

- Les tâches: Une tâche est une portion de code séquentiel. OSEK/VDX distingue deux types de tâches, les tâches basiques et les tâches étendues.

  Les tâches basiques ne peuvent pas se bloquer en attente d'événement, de messages, de ressource… en cours d'exécution. Elles ne possèdent pas d'état «en attente». Elle peut prendre trois états différents: «passive», «prête» et «élue». Etant donné qu'une tâche basique ne peut être préemptée que par une tâche plus prioritaire qui se terminera forcément avant elle, il est assez fréquent que dans l'implémentation du système, toutes les tâches basiques soient exécutées en utilisant une seule pile.

  Les tâches étendues ont les mêmes caractéristiques que les tâches basiques avec en plus la possibilité de se bloquer durant leur exécution. Typiquement, une tâche étendue se bloque en attendant une synchronisation ou une ressource. Par conséquent, les tâches étendues peuvent

prendre quatre états différents, donc les trois mêmes états que les tâches basiques avec l'état «en attente» additionnel.

Comme dans la norme POSIX, une tâche OSEK/VDX est définie par une fonction. Cependant, contrairement à celle-ci, elle est créée de façon statique, il n'y a pas d'état «inexistante». L'initialisation, quand à elle, est effectuée automatiquement au démarrage de l'exécutif.

- Synchronisation et communication des tâches:

  Exclusion mutuelle : assurée par la définition de ressources (de façon statique dans le langage OIL). OSEK/VDX fournit deux primitives, «GetResource» et «ReleaseResource» pour prendre et vendre une ressource. Tenter de prendre une ressource non disponible a pour effet de placer une tâche (forcément une tâche étendue) dans l'état «en attente». Notons que chaque ressource est gérée avec le protocole à priorité plafond immédiat afin d'éviter l'inversion de priorité, avec un mécanisme pour borner la durée de blocage d'une tâche en attente d'une ressource.

  Synchronisation par événements : synchronisations binaires de type n/1: chaque tâche étendue est liée à un certain nombre d'événements, qu'elle seule peut attendre. OSEK/VDX propose une primitive de déclenchement d'événement («SetEvent»), une primitive d'attente d'événement («WaitEvent»), et une primitive d'effacement d'événement («ClearEvent»). Seule la tâche propriétaire d'un événement a le droit de l'attendre et de l'effacer.

  Communication par message : de type communication n/m (n émettrices, m réceptrices possibles pour un message). Dans le cas d'une communication de type boîte aux lettres, si un message a m réceptrices, il est déposé dans chacune des files d'attente de messages de réceptrice de ce message, et il est consommé par chaque réceptrice de façon indépendante des autres réceptrices. Dans le cas d'une communication de type tableau noir, la lecture est non bloquante et non destructive, et le message lu est le message le plus récemment écrit. Des messages vides peuvent être transmis, dans ce cas-là, ils servent de synchronisation entre tâches. En conséquence, l'exécutif fournit trois primitives d'envoi d'un message: «SendMessage» pour l'envoi d'un message de taille fixe; «SendDynamic-Message» pour l'envoi d'un message de taille dynamique; et «SendZeroMessage» pour l'envoi d'un message vide. OSEK/VDX permet de définir des messages périodiques, qui seront automatiquement envoyés à chaque période. Chaque message peut être muni d'une échéance: si le message n'est pas reçu dans le délai imparti, une notification de violation d'échéance est envoyée.

- Horloges: de même que la norme POSIX, OSEK/VDX impose la présence d'au moins une horloge, mais en fonction de l'architecture sous-jacente, plusieurs horloges peuvent être disponibles. La gestion des horloges dans la norme OSEK/VDX est similaire à la norme POSIX. Les «timers» sont définis de façon statique dans le fichier OIL. Il est important de noter qu'OSEK/VDX est un exécutif dirigé par les événements.

I.4.6.c. La norme Ada

Le langage Ada lui-même est une norme, supportant différentes implémentations. Les implémentations de la norme Ada reposent sur un exécutif ou noyau, pouvant être lui-même conforme à POSIX ou encore OSEK.

- Les tâches Ada: une tâche Ada est déclarée en deux parties: une spécification de tâche («task»), et un corps de tâche («task body»). Une tâche peut être instanciée de façon globale ou dynamiquement. Lorsqu'elle est instanciée de façon globale (en tant que variable de paquetage par exemple), cette tâche est lancée automatiquement au démarrage du programme principal. Alors que lorsqu'elle est instanciée dynamiquement (une instance d'un type tâche, ou allouée dynamiquement par «new»), elle est lancée au moment de l'instanciation. Les

tâches Ada sont toujours attachées automatiquement au programme principal (i.e. le programme principal ne se termine pas tant qu'il y a au moins une tâche encore vivante).

- Communication et synchronisation: depuis Ada95, le moniteur de haut niveau (objet protégé) est introduit. Le moniteur Ada permet d'implémenter tout type de synchronisation et de communication, synchrone et asynchrone. La spécification d'un objet protégé comporte un ensemble de primitives et une partie privée, correspondant aux variables internes du moniteur. Il y a trois types de types de primitives:

  Les procédures («procedure») sont des primitives non réentrantes par rapport aux autres primitives de l'objet protégé (i.e., une procédure ne peut pas avoir lieu tant qu'une autre primitive de l'objet protégé est exécutée, et empêche toute autre primitive de s'exécuter). Elles sont non bloquantes au sens de l'objet protégé auquel elles appartiennent, c'est-à-dire qu'elles n'intègrent pas la possibilité d'attendre que le moniteur soit dans un certain état. Les procédures peuvent avoir les paramètres en entrée et/ou en sortie.

  Les procédures gardées («entry») sont des procédures dont l'accès est conditionné (gardé). Une barrière se trouve au début de chaque primitive. Si la barrière d'une procédure gardée n'est pas satisfaite lors d'un appel de cette procédure, la tâche appelante est bloquée jusqu'à ce que cette condition de réveil soit vraie.

  Les fonctions («function») sont des primitives en lecture seule sur l'objet protégé. Elles sont réentrantes les unes par rapport aux autres. Cela permet d'affiner les accès aux objets protégés en leur donnant des primitives de type lecteur/écrivain.

  La norme Ada propose le protocole à priorité plafond immédiat pour éviter l'inversion de priorité lors d'accès concurrents à un objet protégé.

- Le profil Ravenscar: pour le langage Ada, une norme de développement a été proposée, sous le nom de profil Ravenscar, normalisé par l'ISO en 1999. La norme préconise les principes suivants:
  - pas de déclaration de tâches dynamiques: tout est déclaré au début de l'application,
  - les tâches ne se terminent pas ou alors toutes ensemble,
  - toutes les tâches sont au même niveau (pas de hiérarchie mère/fille entre les tâches),
  - tous les outils de communication et de synchronisation sont déclarés initialement, et pas de façon dynamique,
  - utilisation de dates et pas de durées pour les réveils de tâches périodiques, afin d'éviter une dérive des horloges,
  - utilisation d'un protocole de gestion de ressources à priorité plafond,
  - pas d'utilisation du rendez-vous mais uniquement des objets protégés, avec une seule primitive de type «entry» par objet protégé, les autres pouvant être des procédures ou bien des fonctions,
  - un et un seul mécanisme de déclenchement pour une tâche (attente de synchronisation, de message ou d'interruption, ou bien attente périodique).

I.5. Méthodes de développement d'un logiciel

Les méthodes: elles définissent le comment, c'est-à-dire ce qu'il faut faire pour produire un logiciel. Chaque méthode a besoin d'un langage pour la production d'une spécification, et donc à chaque méthode est attaché un langage propre.

Les outils: ils constituent le support pour les méthodes.

Les systèmes temps réel doivent produire des résultats corrects dans des intervalles de temps spécifiés. Si un résultat est incorrect ou arrive trop tard, un tel système est voué à l'échec. Kopetz et al. dans [Kop91] ont élaborées des caractéristiques principales pour une méthode de développement d'un système temps réel, ce sont:

- Prévisibilité: la possibilité d'une méthode de prévoir la performance de l'architecture matérielle, notamment l'analyse de temps, pour assurer le respect des contraintes temporelles.
- Testabilité: la possibilité d'une méthode de vérifier l'opération dynamique d'un système selon ses paramètres d'entrée. Il s'agit d'indiquer à quelle fraction des scénarios du système réel peuvent être couverts par des scénarios de test correspondant; de déterminer si le système sous des tests s'exécute correctement (par exemple, l'ordre et le temps des événements, l'observation des actions et des sorties du système pendant le test, …); de contrôler l'exécution du système afin d'assurer le déterminisme de réactions si le contexte d'un scénario est changé.
- Tolérance aux pannes: permet de réaliser la fiabilité requise d'un service. Foncièrement, il existe deux types d'applications temps réel: l'un de «défaillance-arrêt» et l'autre de «défaillance-opérationnelle». Avec le deuxième, le système doit assurer une continuation de l'application dans toutes les situations opérationnelles, même s'il y a des erreurs ou des défaillances.
- Conception de système: la possibilité d'une méthode de pourvoir des mécanismes qui permettent les développements simultanés de matériel et de logiciel de manière succincte et consistante.
- Décomposition systématique: une possibilité de faire décomposer graduellement un système en des composantes qui sont faciles à comprendre et peuvent être traitées relativement indépendamment.
- Evaluabilité: avant le test l'évaluation sur un exécutif temps réel, il est désirable d'évaluer la conception du système considéré. Cette caractéristique permet au concepteur de commencer une telle évaluation le plus tôt possible dans un cycle de développement d'un logiciel afin d'éviter une révision coûteuse dans la phase de test.

Il existe plusieurs méthodes pour le développement d'un logiciel, parmi elles on peut distinguer deux types: les méthodes se basant sur des descriptions mathématiques formelles du sens d'un programme donné par un langage de programmation informatique de haut niveau permettant d'obtenir un programme pour un ordinateur (il s'agit des «méthode formelles»); et les méthodes se basant sur des descriptions semi-formelles du sens des conceptions fonctionnelles données par des diagrammes, des notations graphiques maîtrisés par les clients, et facilitant les échanges entre clients et concepteurs.

Les méthodes formelles permettent d'obtenir une très forte assurance de l'absence de «bogue» dans les logiciels. Cependant, elles sont des méthodes «constructives» et «universelles» [Jack98]. Donc elles sont généralement coûteuses en ressources (humaines et matérielles) et actuellement réservées aux logiciels les plus critiques.

Pour les approches semi-formelles, Shaw dans [Sha95] les a classifiées en quatre groupes principaux selon le style de l'architecture utilisé: *l'Architecture orientée objet* (focalise la décomposition d'un système en des objets distincts, en encapsulant leurs états et leurs opérations. Les objets interagissent par invocation des opérations des autres); *l'Architecture état-transition* (se concentre sur des états d'un système et sur des événements qui provoquent des transitions entre ces états); *l'Architecture asservissement* (une forme spéciale de diagramme de flot de données adaptée pour des systèmes embarqués où le procédé doit s'adapter régulièrement afin de compenser des perturbations en provenance de l'extérieur); et *l'Architecture temps réel* (se

concentre sur des système temps réel dans lesquels le respect des contraintes de temps doit être satisfait). Les descriptions semi-formelles offrent des mécanismes de structuration et de décomposition très riches tout en proposant une représentation intuitive et synthétique du système à construire. Mais, il n'existe pas d'outils associés permettant d'analyser, de vérifier ou de simuler, de valider de telles spécifications.

Effectivement, aucune des méthodes existantes ne qualifie tous les aspects présentés au-dessus [Kop91]. En s'apercevant que chaque méthode possède certaines caractéristiques spécifiques pour certaine domaine d'application, s'il existe une méthode qui peut regrouper les caractéristiques de chacune des méthodes existantes, la nouvelle méthode peut être qualifiée d'un nombre maximal des critères, et surtout ce principe peut s'appliquer à chercher une méthode convenable pour tous les domaines de problème ou tous les environnements de développement à des développeurs. Saeki dans [Sae94] a proposé un modèle pour construire une telle méthode, nommée «l'ingénierie des méthodes» (voir Figure I-20). L'idée principale de ce modèle est la suivante: en se basant sur la «réutilisation» des «méthodes fragmentaires», appelées «méthode de base», les développeurs récupèrent les méthodes fragmentaires convenables d'abord, puis les adaptent, et les s'intègrent à la nouvelle méthode.

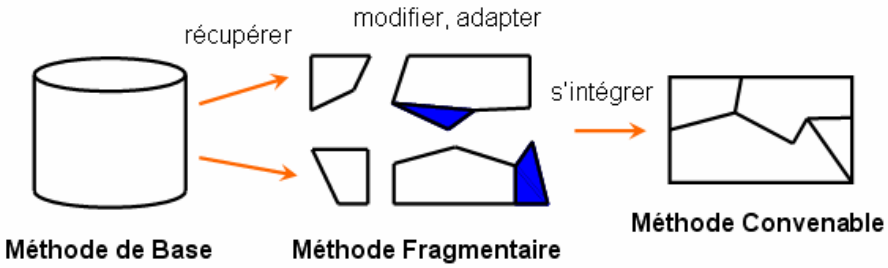


Figure I-20: L'ingénierie des méthodes [Sae94]

En effet, on peut voir la méthode OMT de Rumbaugh (Object Modeling Technique) comme l'intégration de quatre fragments: le Diagramme de Classe, le Diagramme de Séquence, le Diagramme des états-transitions, et le Diagramme flots de données. SA-RT (Structured Analysis For Real-Time Systems) est une méthode de spécification des systèmes temps réel construite par incorporer les diagrammes états-transitions dans les diagrammes flots de données de la méthode SA (Structured Analysis) afin d'exprimer l'aspect dynamique du système. En outre, le développeur peut construire un diagramme de classes, et puis raffiner les détails en utilisant une technique formelle telle que le langage B [Mar00].

Les difficultés dans l'emploi de cette approche résultent de la complexité des concepts que les méthodes possèdent. Il en résulte que la consistance de l'intégration des méthodes doit être vérifiée.

L'idée de l'intégration de plusieurs méthodes dans une seule méthode n'est pas nouvelle, mais pour répondre à la question "Est-ce que la nouvelle méthode est la meilleure, permettant à la fois de satisfaire un nombre maximal de critères, et d'obtenir le «Time-to-market» le plus court ?". Les travaux présentés dans ce manuscrit ne cernent que cet objectif. En se focalisant sur l'approche semi-formelle, dans les paragraphes suivants, quelques méthodes semi-formelles pour le développement de systèmes temps réel sont présentées. Puis en s'inspirant de l'idée d'ingénierie des méthodes de Saeki, une nouvelle approche, appelée DARTSVIEW, est créée pour tenter de résoudre les problèmes énoncés en-dessus. J'utilise le «Système de chauffage d'une maison» comme un exemple typique pour comparer les avantages et les inconvénients des méthodes semi-formelles, ainsi que des modifications nécessaires de méthodes elles-mêmes afin d'être mieux adaptées aux systèmes temps réel.

L'énoncé du système de chauffage est donné ci-dessous:

«Le résidant d'une maison choisit une température à l'aide d'un thermostat. Le système informatique doit s'assurer que la variation totale de température permise dans la maison ne dépasse pas 4 degrés C. Précisément, si le résidant désire une température de 21°C, la température de la maison devra osciller entre 19°C et 23°C, dans la mesure où la température extérieure ne dépasse pas 23°C (il s'agit d'un chauffage, non d'une climatisation).

Le chauffage utilisé est un appareil de chauffage au fuel (carburant) qui possède un moteur qui pilote un ventilateur pour fournir l'air comburant, et pilote aussi une pompe à fuel. Les règles précises de son fonctionnement sont données ci-dessous:

Allumage

- *le ventilateur est mis en route afin de permettre l'alimentation en air,*
- *la vanne d'arrivée du fuel est ouverte,*
- *puis lorsque la vitesse du ventilateur et l'arrivée en carburant sont suffisants, on procède à l'allumage du mélange air/fuel par le déclenchement d'une étincelle.*

Extinction

- *fermeture de la vanne d'arrivée de fuel,*
- *extinction du ventilateur,*

Parallèlement au contrôle de température via la commande de chauffage, le contrôleur temps réel doit s'assurer que le système fonctionne correctement en mode sécuritaire. Précisément, lorsque le chauffage est en marche, il doit vérifier que le ventilateur tourne à vitesse suffisante, que l'arrivée de fuel s'effectue correctement, et que la combustion a lieu. Si ce n'est pas le cas, le système de chauffage doit être immédiatement arrêté. La maintenance pourra alors réparer le ventilateur, remettre du fuel et réinitialiser le système à l'aide d'un disjoncteur principal. La combustion est alternativement activée et désactivée par le contrôleur pour maintenir la température dans les limites voulues. Un indicateur de circulation de fuel et un capteur optique de combustion permettant au contrôleur de détecter les anomalies.

Le système informatique chargé de contrôler la température fonctionne sur les règles suivantes:

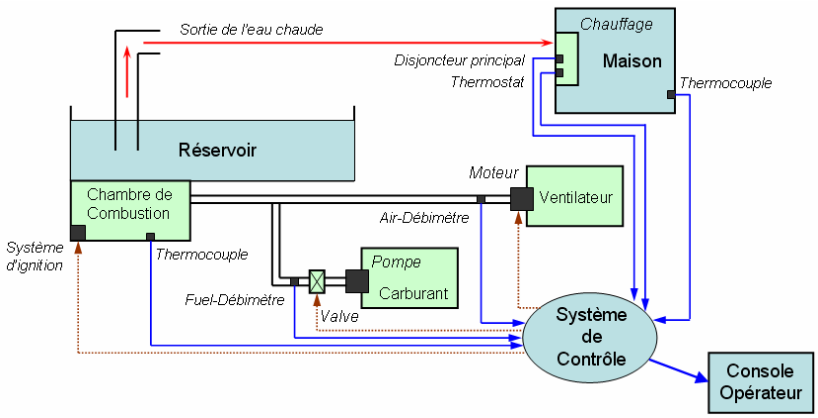


Figure I-21: Schéma du système du chauffage d'une maison

Informations en entrée:

- *l'état du disjoncteur principal (Marche/Arrêt),*
- *la température captée dans la maison ($T_{Courante}$),*
- *la température désirée dans la maison ($T_{Choisie}$),*
- *l'état du ventilateur (Normal/Autre),*

- *l'état de la circulation du fuel (Normal/Autre),*
- *l'état de la combustion (Allumée/Eteinte).*

Action sur le procédé:

- *commande du ventilateur (Démarrage/Arrêt),*
- *commande de la vanne de fuel (Ouverture/Fermeture),*
- *signal d'allumage,*
- *affichage de l'état du procédé (Normal/Panne).*

Contrôle de température:

Quand le disjoncteur principal est en Marche et qu'on se trouve en dehors des limites de température: lorsque $T_{Courante} \leq T_{Choisie} - 2$*, le contrôle de température passe en mode «Allumage»; lorsque* $T_{Courante} \geq T_{Choisie} + 2$*, il est en mode «Arrêt». Quand la chaudière est allumée, toute défaillance de celle-ci entraînant son extinction doit être détectée par le système. Ce dernier doit alors passer en mode «Panne».*

Mode «Allumage»:

- *le système commence par envoyer des commandes au moteur et à la vanne de fuel pour démarrer le ventilateur et ouvrir la vanne respectivement. Si 3 secondes après les commandes de démarrage du ventilateur et de l'ouverture de la vanne, ces deux éléments ne sont toujours pas en fonctionnement, le système doit passer en mode panne (panne du ventilateur ou panne de la circulation de fuel ou toutes les deux),*
- *lorsque la vitesse d'écoulement du fuel atteint un niveau adéquat, le signal d'allumage est déclenché. Si après 5 secondes, la chaudière n'est pas allumée, le système doit passer en mode panne (panne de la combustion).*

Mode «Arrêt»:

- *le système est en mode «Arrêt» dans l'un des cas suivants: le disjoncteur principal est mis à Arrêt; ou le système est en mode «Allumage» et on se trouve en dehors de limite supérieure de la température (*$T_{Courante} > T_{Choisie} + 2$*),*
- *lorsqu'on passe en mode arrêt, il faut d'abord fermer la vanne d'arrivée du carburant, puis arrêter le moteur et l'allumage.*

Mode «Panne»:

- *lorsque le système passe en mode panne, il faut passer en mode arrêt, puis afficher la cause de la panne,*
- *on ne sort du mode panne après une coupure et un allumage du disjoncteur principal. Durant la coupure, un service de maintenance peut intervenir.*

…».

I.5.1. JSD (Jackson System Development)

JSD (Jackson System Development) est une approche de développement de logiciels orientée objet développée par Michael A. Jackson et John Cameron. JSD est une extension de JSP (Jackson Structured Programming), une méthode de conception de logiciels que Jackson a proposée dans son livre, «Principles of Program Design», en 1975. La première version de JSD a été publiée en 1983.

Un concept fondamental de JSD est que le développeur modèle la réalité d'abord [Jack83] avant de considérer les fonctions du système. Au début, le système est considéré comme une simulation du vrai monde, dans laquelle des Entités représentent les objets réels. Les entités sont représentées sous la forme d'une séquence ordonnée par temps des événements reçus. Jackson appelle une telle structure le «Diagramme Entité Structure». Les notations graphiques utilisées dans le diagramme

entités structure sont similaires à celles dans JSP. Donc, l'entité est représentée par une boîte rectangulaire étiquetée, et l'événement est illustré par une ligne. Il existe trois structures élémentaires dans un diagramme: la *Séquence* (représente une ou plusieurs parties d'une entité composite dans laquelle elles surviennent chacune de la gauche vers la droite); la *Sélection* (illustre un choix entre plusieurs parties d'une entité composite où une seule est sélectionnée. Le symbole 'o' apparaît en haut à droite de chaque partie de la sélection); l'*Itération* (dans cette structure, une partie d'une entité composite peut se répéter une ou plusieurs fois. Le symbole '*' apparaît en haut à droite de chaque partie de l'itération). Le diagramme entité structure du système de chauffage d'une maison est présenté sur la Figure I-22.

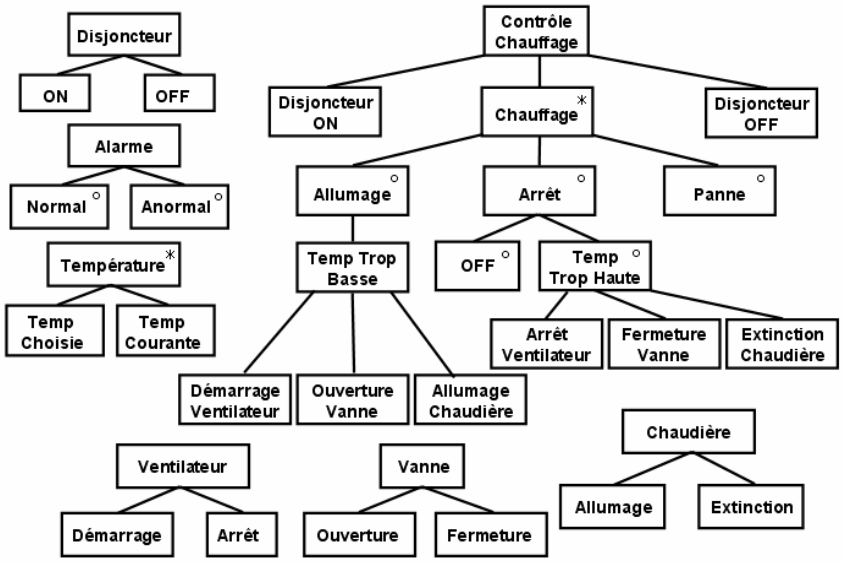


Figure I-22: Digramme Entité Structure du système de chauffage

Nous voyons que le chauffage est modélisé par une structure des entités du monde réel:

- L'entité *Disjoncteur* représente le disjoncteur principal utilisé par le résident pour activer ou désactiver le chauffage. Les événements sont «ON» et «OFF».
- L'entité *Alarme* représente la surveillance du système afin de détecter des pannes pendant son fonctionnement. Donc, quand le système est en opération, il est soit en état «Normal», soit en état «Anormal».
- *L'entité Température* modélise l'indicateur de température. Cette entité reçoit d'abord une valeur fournie par le résident pour la température désirée «Temp Choisie», puis une autre valeur venue d'un thermocouple pour la température actuelle «Temp Courante». Ces événements sont générés régulièrement.
- *L'entité Ventilateur* modélise l'opération du moteur. Au «Démarrage», le ventilateur tourne à vitesse suffisante afin de fournir l'air comburant. A l'«Arrêt», le moteur s'arrête, et l'apport du comburant est arrêté.
- *L'entité Vanne* représente l'opération de la vanne du fuel. A l'«Ouverture», la vanne est ouverte pour la circulation de fuel. A la «Fermeture», la vanne est fermée.
- L'entité *Chaudière* simule la combustion du mélange air/fuel. A l'«Allumage», le déclenchement d'une étincelle a lieu pour allumer ce mélange. A l'«Extinction», l'allumage du mélange air/fuel est arrêté.
- L'entité *Contrôle Chauffage* modélise le fonctionnement d'un système de chauffage. Elle donne une vue générale de tous les modes d'opération possibles du système en fonction des événements reçus. En effet, le système s'exécute à l'ordre dans un des trois modes d'opération fondamentaux ci-après: «ON», «Chauffage», et «OFF», donc le premier et le dernier dépendent de l'état du Disjoncteur. Quand le disjoncteur est mis à «ON», le «Contrôle

Chauffage» passe à l'état «Chauffage» où en fonction de la différence entre la température courante et la température désirée (soit «Trop Basse», soit «Trop Haute»), le chauffage est mis soit en état «Allumage», soit en état «Arrêt», soit en état «Panne» (voir Figure I-22). Car le système doit manifester son opération en réponse à un changement de température, le «Chauffage» s'exécute d'une manière régulière jusqu'à ce que le disjoncteur soit mis en «OFF». Dans ce cas-là, le «Contrôle Chauffage» passe à l'état «OFF», et l'exécution du système se termine.

A la fin de cette étape, la décomposition en tâches des entités dessus aura lieu. Il en résulte que chaque entité sera représentée par une tâche avec uniquement ses événements d'entrée (voir la Figure I-23).

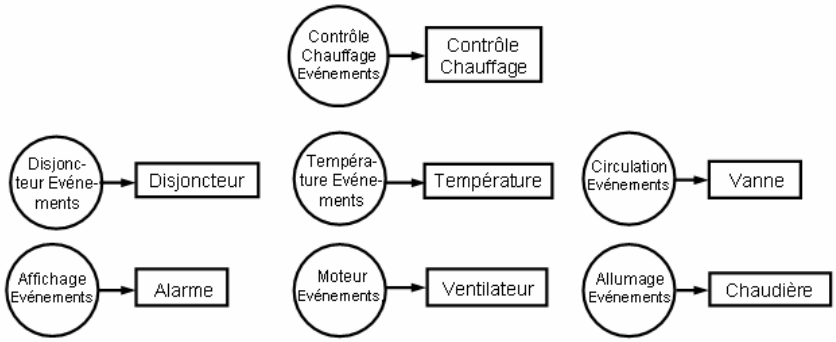


Figure I-23: Modèle des tâches JSP du système de chauffage

Ensuite, la communication entre ces tâches est définie. Selon l'approche JSP, il existe les deux types de communication de base: la communication par flot de données (représentée par un cercle) où la tâche «productrice» envoie un message à la tâche «consommatrice»; la communication par l'inspection d'un vecteur-état (State vector inspections), représentée par un diamant, où l'écriture se fait seulement sur la tâche qui tient le vecteur-état, tandis que les autres tâches n'effectuent que la lecture sur ce vecteur. Le réseau de communication du système de chauffage est illustré ci-après.

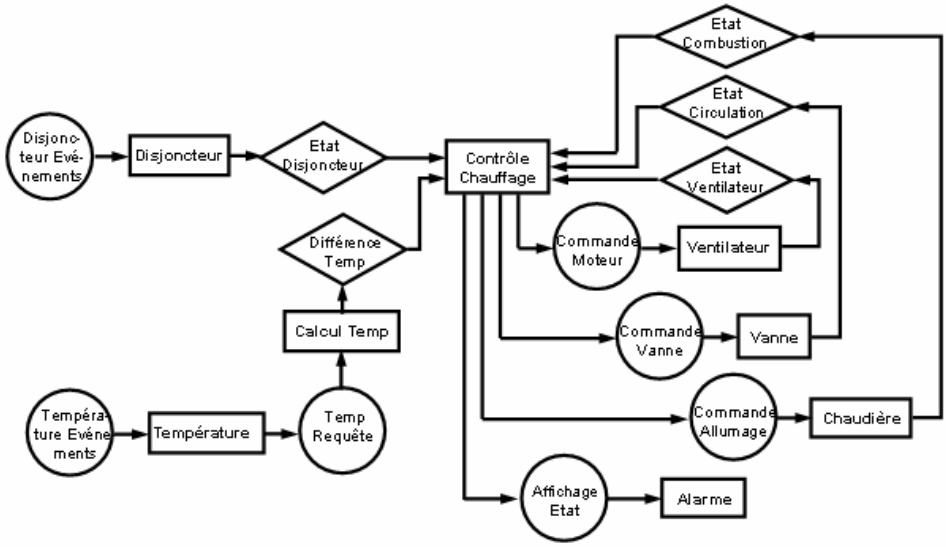


Figure I-24: Réseau communication du système de chauffage

Dans ce diagramme, une nouvelle tâche est ajoutée pour calculer la différence entre la température courante et la température désirée. C'est la tâche «Calcul Temp», une tâche asynchrone qui se lance lorsqu'il y a une entrée «Temp Requête» disponible. Il en va de même pour les tâches «Ventilateur», «Vanne», «Chaudière», et «Alarme». Ainsi, la tâche «Ventilateur», déclenchée par l'arrivée d'une entrée «Commande Moteur», va faire démarrer ou éteindre le

moteur, et faire l'écriture sur son vecteur-état, nommé «Etat Ventilateur», etc. Au contraire, le «Contrôle Chauffage» est un objet passif. Il est activé par une interruption externe ou une horloge. A chaque exécution, cette tâche demande une inspection sur chacun des vecteurs-états, puis en fonction de ces valeurs reçues, elle génère des messages vers des tâches asynchrones. Les contraintes temporelles peuvent être aussi considérées d'une manière que certaines sorties doivent être générées dans un intervalle de temps spécifié dès l'arrivée de certaines entrées. Par exemple, si 3 secondes après la commande de démarrage du ventilateur, cet élément n'est toujours pas en fonctionnement, le «Contrôle Chauffage» doit envoyer un message à la tâche «Alarme» pour afficher l'état anormal du système (panne de moteur dans ce cas).

Dans la phase d'implémentation, à partir du diagramme «Réseau Communication», Jackson propose un concept d'inversion-programme. L'implémentation consiste en une tâche, nommée «Ordonnanceur» qui joue le rôle d'une procédure centrale du système considéré, et s'exécute successivement. Il en résulte que chaque tâche qui reçoit des entrées provenant de l'extérieur est implémentée par une procédure dont l'appel est effectué par l'«Ordonnanceur» s'il y a une entrée pour elle. La communication entre des tâches par messages est transformée en un appel de fonction, où la «consommatrice» est mappée, à partir de la «productrice» avec un paramètre auquel le contenu du message est inséré. La communication des tâches par vecteur-état est transformée en un appel d'une fonction à partir d'un objet passif avec un paramètre du type «sortie» pour effectuer la lecture sur un vecteur-état. Le modèle d'une telle implémentation du système de chauffage est illustré sur la Figure I-25. Nous voyons clairement que dans cette solution, au lieu de la réception des messages envoyés par «Contrôle Chauffage», «Ventilateur», «Vanne», «Chaudière» et «Alarme» tous les quatre sont appelés par leur propre procédure à partir de «Contrôle Chauffage» lorsqu'il y a un message pour eux.

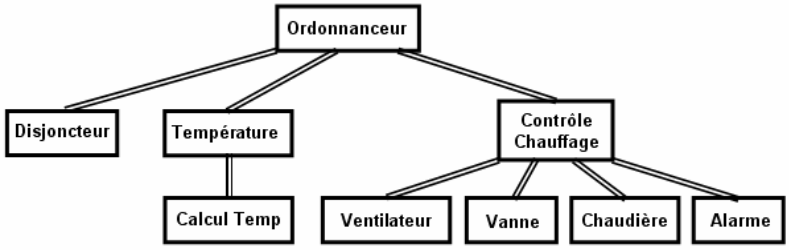


Figure I-25: Modèle d'implémentation invertie du système de chauffage

Il existe une autre solution pour implémenter le diagramme Réseau Communication de JSP. Cette approche est basée sur le concept de tâches du langage Ada. Donc, chaque tâche JSP correspond à une tâche Ada. Pour la communication entre des tâches JSP, on peut utiliser deux objets protégés, un objet pour la gestion des flots d'entrée et un autre pour des vecteurs-états. Lorsque la tâche «productrice» veut envoyer un message à la tâche «consommatrice», elle appelle une «procedure» de cet objet, nommé «Envoyer» par exemple, puis ceci va déclencher une occurrence à la tâche «consommatrice» qui est en attente de ce message dès un appel de l'entry «Recevoir» afin de la réveiller. Le modèle d'un tel objet protégé en Ada est présenté ci-après:

```
-------------------------------------------------------------
----
protected Flot_Input is
   procedure Envoyer(message: in Type_Message);
   entry Recevoir(message: out Type_Message);
private
   tampon: Type_Message;
   Arrivée: Boolean:= False;
end Flot_Input;

protected body Flot_Input is
   procedure Envoyer(message: in Type_Message) is
   begin
```

```
            tampon:= message;
            Arrivée:= True;
         end Envoyer;
         entry Revevoir(message: out Type_Message) when (Arrivée)
      is
         begin
            message:= tampon;
            Arrivée:= False;
         end Recevoir;
      end Flot_Input;
      ------------------------------------------------------------
      ----
```

Il en va de même pour la gestion de l'objet protégé des vecteurs-états. Cependant dans ce cas-là, il n'existe pas de barrière pour des appels de la primitive «Recevoir» (qui sera une fonction), donc plusieurs tâches peuvent être simultanément en lecture. Au cas où nous ne souhaitons pas utiliser la mémoire tampon, nous pouvons la remplacer aisément par l'outil rendez-vous du langage Ada (il s'agit de la synchronisation des tâches). Chaque vecteur-état sera encapsulé dans la tâche qui le possède sous la forme d'une variable privée. Et la lecture d'un vecteur-état aura lieu en demandant un rendez-vous avec cette tâche. Pour de plus amples informations concernant le mécanisme de rendez-vous dans Ada, le lecteur peut se référer à [Bur98].

Remarques de la méthode JSP (Jackson System Development):

JSP est une méthode de spécification et de conception des systèmes, favorisant la décomposition d'un système en des entités orientées objets réels. Le diagramme Entité Structure et le Diagramme Réseau Communication de JSP donnent une vue générale sur le fonctionnement statique d'un système: des entités sont ordonnées par des événements reçus, la communication entre des entités est réalisée soit par un flot d'entrées soit par une inspection d'un vecteur-état. Pour la phase d'implémentation, JSP facilite également l'implémentation des entités JSP en des tâches informatiques grâce à la technique inversion-programme (une implémentation hautement successive), ou grâce aux mécanismes de l'objet protégé et de rendez-vous du langage Ada (une implémentation hautement concurrente).

Puisque la structure des tâches JSP se base principalement sur un modèle des objets du monde réel, un petit changement du dernier peut causer des modifications exhaustives de la première. De plus, la présentation des événements en séquence dans le diagramme réseau communication de JSP n'est pas nécessairement évidente et efficace pour exprimer l'aspect fonctionnel d'un système complexe par rapport au digramme état-transition ou au «statechart» (l'état-transition et le statechart seront présentés dans les paragraphes suivants). JSP ne propose aucune instruction pour définir la structure interne d'une tâche. Ceci peut entraîner des difficultés lorsque l'on ajoute des fonctions dans la phase de mise en réseau des tâches JSP puisque la différentiation entre ces fonctions et des tâches est obscure. Enfin, les notations proposées par la méthode JSP ne permettent pas de distinguer parmi des tâches JSP, d'une part les tâches qui doivent être activées périodiquement par une horloge ou par une interruption externe, d'autre part les tâches qui sont déclenchées par une autre tâche. La communication des tâches JSP par l'inspection vecteur-état au début est simplement une communication asynchrone par l'échange de messages sous la forme de la variable globale que nous trouvons dans des langages de programmation. Alors une implémentation de cette communication par des objets protégés dans Ada est une extension de la méthode JSP où le concept du module de données peut s'appliquer pour être bien adaptée à des applications temps réel.

I.5.2. SA-RT (Structured Analysis For Real-Time Systems)

SA-RT (Structured *Analysis* for Real-Time Systems) est une méthode de spécification des systèmes temps réel. SA-RT est une extension de la méthode SA (Structured Analysis ou Analyse Structurée proposée par Tom Demarco en 1978), une méthode de spécification permettant de

décrire un système en terme de processus fonctionnels de façon indépendante de l'implémentation. La méthode SA-RT étend la méthode SA avec des sémantiques et des notations additionnelles pour développer les événements et le contrôle, les deux parts les plus imprécises et ambigües dans SA. Il existe deux variations de SA-RT, l'une de Ward et Mellor en 1986, l'autre de Hatley et Pirbhai en 1987. Dans le cadre de ce mémoire, mes travaux utilisent la deuxième, SA-RT de Hatley et Pirbhai.

I.5.2.a. SA de Demarco

La méthode *SA* de Demarco [Dema78] décompose un système en un ensemble de diagrammes flots de données hiérarchiques, ce qui permet de tracer pas à pas la génération des flots de sortie à partir des flots d'entrée. La transformation des flots s'effectue dans des processus fonctionnels. A l'aide du diagramme de flot de données d'un système, le concepteur peut créer un plan pour le traitement d'un programme qui satisfait les spécifications. La méthode SA se compose d'un Diagramme de Contexte de Donnée (DCD), un ou plusieurs Digrammes de Flot de Données (DFD), une Dictionnaire de Données, et une Mini-Spécification (PSPEC) pour chaque processus non décomposé.

- Diagramme de Contexte de Données (DCD)

Le DCD représente les bornes entre le système et l'environnement externe. Il identifie les entités extérieures avec lesquelles le système doit communiquer. Chaque entité est représentée sous la forme d'une terminaison par un rectangle étiqueté. Le système est représenté par un processus unique qui expose le but principal (c'est-à-dire le cercle contenant l'identification du rôle central du système). Toutes les entrées et les sorties du système sont représentées sous la forme des flots de données qui font des interfaces entre les terminaisons et le processus principal: donc tous les flots qui circulent vers le processus principal sont en entrée, et tous les flots qui circulent à partir du processus principal sont en sortie. Notons que la méthode SA utilise le terme «flot de donnée» dans un sens plutôt large. Il peut couvrir tout type de processus agissant sur tout type d'objet [Hat91]. Ainsi chacun des flots de données dans un DCD est représenté par un arc orienté avec un nom qui désigne la donnée de manière assez générale. Le diagramme de contexte de données du système de chauffage présenté sur la Figure I-26 illustre clairement le contexte de l'application. Dans ce DCD, le chauffage se compose de neuf terminaisons (dont les cinq à gauche sont en entrée, les quatre à droite sont en sortie) et d'un processus central unique, nommé «Contrôler le Chauffage d'une Maison». Chacune des terminaisons correspond à une information en entrée («Etat Global», «Etat Circulation», «Etat Ventilateur», «Etat Combustion», «Température Courante») ou une action sur le procédé («Commande Allumage», «Commande Vanne», «Commande Ventilateur», «Affichage Etat»), ce qui fait l'interface entre cette terminaison et le processus central.

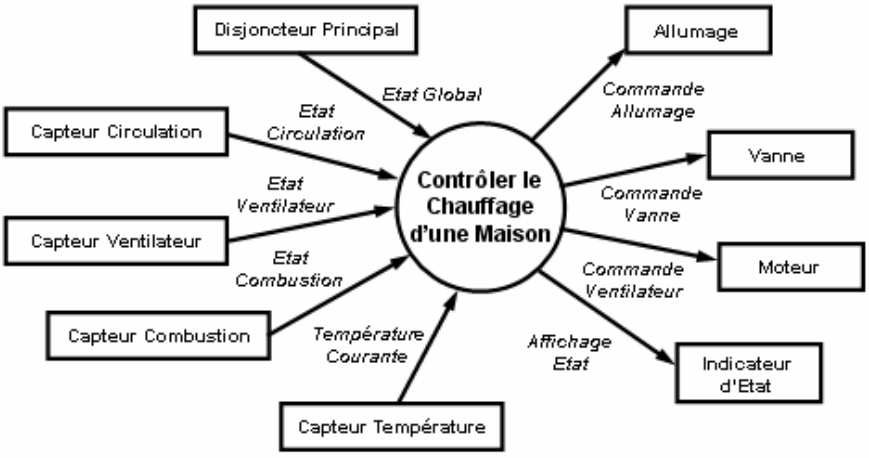


Figure I-26: Diagramme de contexte de données du système de chauffage

- Diagramme de Flot de Données (DFD)

Le DFD décompose le processus central unique et les flots d'entrée/sortie dans le DCD en sous-processus et en sous-flots respectivement (les sous-flots sont donc introduits entre les sous-processus) afin de détailler le fonctionnement du système. Le DFD contient seulement des processus, des flots de données et des zones de stockage. Les processus et les flots de données sont similaires à ceux du diagramme de contexte de données. Les stockages, indiqués par des paires de lignes de droite parallèles avec des noms à l'intérieur, représentent des données stockées pour une utilisation ultérieure par un processus. Ils sont semblables à des vecteurs-états de la méthode JSP. Tout DFD d'un niveau est une décomposition d'un processus du niveau supérieur. Un DFD est appelé fils du processus qu'il décompose, et ce processus est appelé père du DFD dans le sens contraire. Chaque processus du DFD d'un certain niveau est attribué une dénomination décrivant sa fonction, et un numéro constitué du numéro de son processus père, suivi d'un ou plusieurs chiffres. Car le processus unique du diagramme de contexte de donnée a le numéro 0 (qui n'est généralement pas indiqué), les processus du DFD de niveau 1 ont des numéros à un nombre (par exemple 1, 2, 3 etc.). Cette décomposition des DFD en des diagrammes de plus en plus détaillés est appelée décomposition en niveaux. Et elle s'arrête lorsque la spécification du système atteint un certain niveau de détail. Un DFD possible du système de chauffage de niveau 1 est présenté sur la figure ci-après.

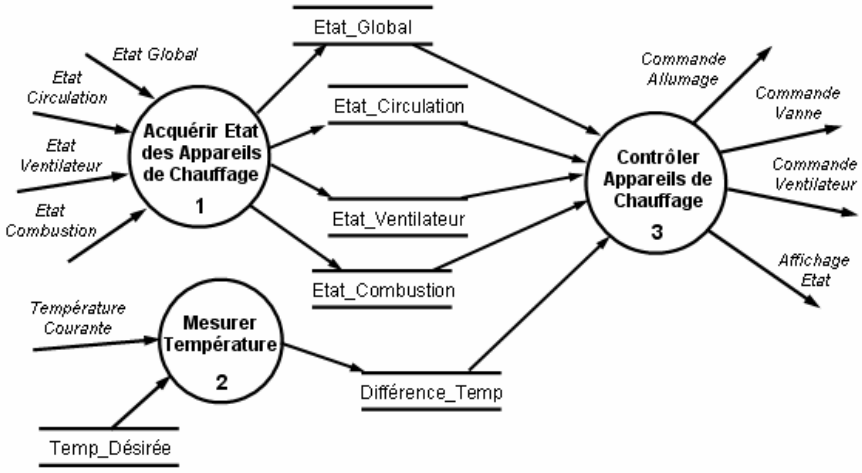


Figure I-27: Diagramme de flot de données du système de chauffage

Puisqu'un DFD et son processus père représentent la même information à des niveaux différents de détail, leurs entrées et sorties doivent être identiques. Ainsi, à la suite, la décomposition des deux processus, «Acquérir Etat des Appareils de Chauffage» (numéro 1) et «Contrôler Appareils de Chauffage» (numéro 3), en des DFD de niveau 2 est effectuée. Nous pouvons vérifier que les sources et les destinations des entrées/sorties qui apparaissent comme n'arrivant de nulle part et ne partant vers nulle part, peuvent être trouvées en se référant au processus père (voir la Figure I-28 et la Figure I-29).

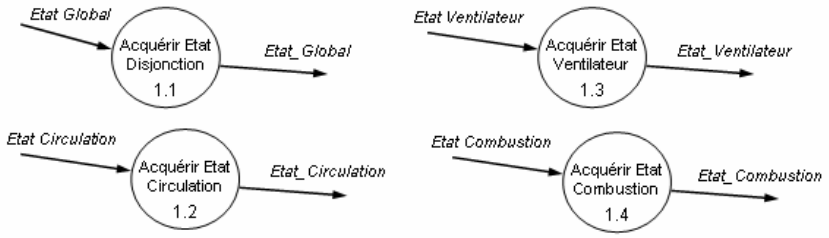


Figure I-28: DFD du processus «Acquérir Etat des Appareils de Chauffage»

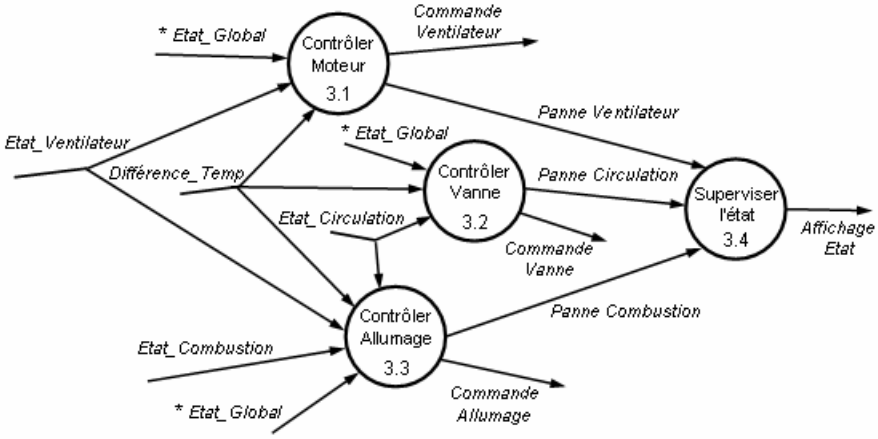


Figure I-29: DFD du processus «Contrôler Appareils de Chauffage»

Dans la Figure I-29, les flots de données, «Panne Ventilateur» «Panne Circulation» «Panne Combustion», sont les sous-flots qui font des interfaces entre le sous-processus «Superviser l'état» et les sous-processus «Contrôler Moteur», «Contrôler Vanne», «Contrôler Allumage». Leurs rôles sont de surveiller le fonctionnement des appareils du chauffage. Si le chauffage fonctionne en état anormal (par exemple, au cas où 3 secondes après la commande de démarrage du ventilateur, cet élément n'est toujours pas en fonctionnement), un sous-flot se lance (le «Panne Ventilateur» dans ce cas) vers le sous-processus «Superviser l'état» afin d'afficher cette anomalie à des utilisateurs. Des erreurs et des exceptions seront cernées dans la mini-spécification de processus.

- Dictionnaire de Données

Le dictionnaire de données regroupe la sémantique et la structure des données présentées dans le système. Le dictionnaire fournit une définition exacte de chaque flot de données et de tous les stockages. Comme nous le voyons dans les DFD, il existe deux types de base de flots: un flot primitif est un élément d'information qui a ses propres valeurs intrinsèques et attributs; un flot non primitif est constitué d'un groupe de flots primitifs. De plus, les flots primitifs sont soit continus soit discrets (donc les flots discrets peuvent prendre une valeur parmi un nombre fini de valeurs connues dans un ordre indifférent, et les flots continus peuvent prendre une valeur parmi un nombre arbitrairement grand de valeurs numériques ordonnées [Hat91]). Les attributs des flots dans les DFD ne se composent en général que d'une petite définition, et d'une unité pour chaque flot. Les flots discrets peuvent nécessiter un attribut additionnel sur les noms des valeurs. Il existe aussi d'autres attributs qui ne font pas partie de la spécification. Ils sont assignés dans la phase de conception (par exemple, les fréquences de données, etc.). Le dictionnaire de données du système de chauffage est montré ci-dessous:

```
-------------------------------------------------------------
Etat Global= \l'Etat actuel du disjoncteur principal\
                      Valeurs: ON, OFF
                      Fréquence: 200 ms
Etat_Global= \le Stockage de donnée «Etat Global»\
                      Valeurs: ON, OFF
Etat Ventilateur= \l'Etat actuel du moteur fournissant
       l'alimentation en air\
      Valeurs: Marche, Arrêt
      Fréquence: 200 ms
Etat_Ventilateur= \le Stockage de donnée «Etat Ventilateur»
                      Valeurs: ON, OFF
...
Température Courante= \la Température actuelle de la maison\
                      Unités: dégrée C
```

```
                        Fréquence: 5000 ms
Différence_Temps= \le Stockage d'information de la différence
                    entre température courante et température
                    désirée\
                      Valeurs: Trop Basse, Trop Haute
Panne Ventilateur   = \l'Etat en panne du ventilateur\
                      Valeurs: Normal, Panne
...
Panne=([Panne Ventilateur|Panne Circulation|Panne Combustion])
Commande Allumage   = \l'Action sur le procédé d'allumage\
                      Valeurs: Allumée, Eteinte
...
-------------------------------------------------------------
```

- Mini-Spécification de Processus (PSPEC)

La PSPEC raffine les explications sous la forme d'une définition textuelle afin de détailler un peu plus la dénomination de chacun des processus primitifs (processus fonctionnels élémentaires, donc non décomposés en DFD). Ainsi, le but de la PSPEC est de montrer comment les flots de sortie sont générés à partir des flots d'entrée. Dans la PSPEC, nous pouvons utiliser des variables internes pour lier des équations. Et des commentaires peuvent être inclus pour aider le concepteur à mieux comprendre le comportement de ces processus. Un exemple de la mini-spécification de processus est illustré ci-après.

```
-------------------------------------------------------
PSPEC 2: Mesurer Température
% Mesurer la différence de température courante et température
% désirée. Ce processus est déclenché pour chaque impulsion de
% la donnée TEMPERATURE COURANTE
     mesurer TEMPERATURE COURANTE
     lire TEMP_DESIREE
     si TEMPERATURE COURANTE – TEMP_DESIREE > 2, alors:
          mettre DIFFERENCE_TEMP = Trop Haute
          ecrire DIFFERENCE_TEMP
     sinon si TEMPERATURE COURANTE – TEMP_DESIREE < -2, alors:
          mettre DIFFERENCE_TEMP = Trop Basse
          ecrire DIFFERENCE_TEMP
     autrement:
          % ne rien faire
-------------------------------------------------------
```

I.5.2.b. Remarques sur la méthode SA (Analyse Structurée ou Structured Analysis)

SA est une méthode de spécification des systèmes basée sur le modèle des processus. Un système est partitionné en deux parties: une partie se composant de l'information et des processus, l'autre partie se composant des entités qui représente des éléments dans le monde réel. Le DCD montre très clairement les frontières entre ces deux parties, et également les flots d'information qui font leurs interfaces. Un ensemble de DFD représente les spécifications des besoins à un certain niveau de détail, ce qui permet de décrire explicitement le fonctionnement statique du système tel que chacun des processus peut être vu comme un processeur déclenché par les données, indépendant et infiniment rapide [Hat91] (hypothèse synchrone).

Contrairement à la méthode JSP, la méthode SA propose le mécanisme d'abstraction de données. Les diagrammes flots de données hiérarchiques sont décomposés de manière que chaque processus père est un modèle abstrait de ses sous-processus et de ses sous-flots. Cependant, si cette décomposition atteint un plus bas niveau de détail, le modèle des processus peut être vu comme un plus énorme réseau de processus terminaux liés ensemble par leurs flots de données. Ceci peut entraîner une situation d'explosion du nombre des flots de données (voir la Figure I-29).

En fait, pour des grands systèmes, cette représentation devient trop complexe et inapplicable. Dans ce cas, une autre représentation telle que le diagramme état-transition ou le diagramme de statechart sera plus adaptée.

Le stockage de données est une autre implémentation du réservoir de données que nous avons trouvé dans les vecteurs-états de la méthode JSP. Donc, le contenu du stockage est présent et peut être lu et relu n'importe quel nombre de fois par un processus «récepteur», jusqu'à ce qu'il soit remplacé par de nouvelle donnée par un processus «émetteur» (principe du tableau noir). Enfin, le modèle des processus de SA est destiné uniquement à représenter les spécifications de besoins, et non pas à la conception ou à l'implémentation.

I.5.2.c. SA-RT de Hatley et Pirbhai

Pour pallier certains inconvénients de la méthode SA, Ward et Mellor en 1986, Hatley et Pirbhai en 1987 ont proposé deux variations d'une nouvelle méthode. C'est la *méthode SA-RT (Structured Analysis for Real-Time Systems)*, une méthode de spécification se basant sur le modèle des processus de la méthode SA. Le principe de la méthode SA-RT est d'abord de décomposer un système en processus fonctionnels, et de décrire le passage d'information sous forme de flots de données. Puis un processus de contrôle joue le rôle de chef d'orchestre en activant/désactivant ou en déclenchant ces processus fonctionnels grâce à des événements prédéfinis: «Enable» (Activation), «Disable» (Désactivation), et «Trigger» (Déclenchement). La spécification SA-RT comporte généralement un diagramme de contexte de données, un ensemble de diagrammes de flots de données/flots de contrôle hiérarchique, et un diagramme Etat-Transition. Les éléments utilisés dans des diagrammes SA-RT sont présentés sur la figure ci-après:

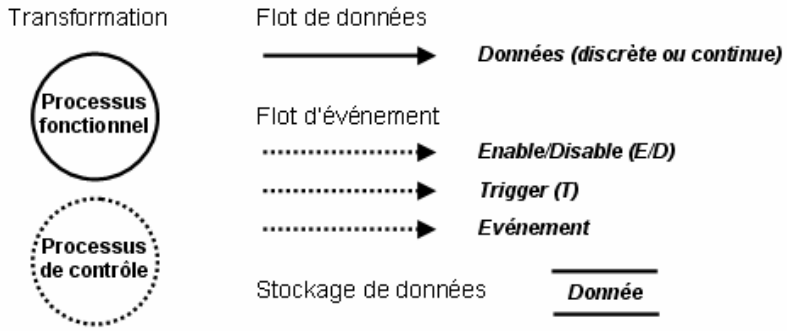


Figure I-30: Eléments des diagrammes SA-RT

Le processus de contrôle représente la logique du pilotage des processus fonctionnels. Les flots de contrôle (ou Flots d'Evénement) transportent les événements ou informations qui conditionnent directement ou indirectement l'exécution des processus de transformation de données. Alors que les données peuvent prendre soit des valeurs discrètes, soit le plus souvent des valeurs continues, les signaux de contrôle prennent toujours des valeurs discrètes (voir la Figure I-30). Comme dans SA, des DFD hiérarchiques sont créés en partant du diagramme de contexte de données en respectant les règles suivantes : un seul ou aucun processus de contrôle par niveau de diagramme (diagramme préliminaire ou diagrammes de décomposition); les flots de données ne sont traitées que dans les processus primitifs, les flots de contrôle ne sont que dans le processus de contrôle et font des interfaces entre le processus de contrôle et des processus fonctionnels.

Dans le cas où un diagramme de flots de données possède un processus de contrôle, la compréhension de ce diagramme de flot de données nécessite une description ou une spécification du processus de contrôle (CSPEC). Cette spécification représente l'aspect comportemental de l'application. Avant tout son rôle est exactement le même que celui de PSPEC (Spécification du processus de donnée): montrer comment sont générées ses sorties à partir de ses entrées. La PSPEC et la CSPEC diffèrent dans la manière dont elles agissent, et dans le rôle qu'elles jouent dans la structure d'ensemble. La CSPEC assure le lien entre le DFC et le DFD. Ses entrées sont les flots de contrôle du DFC et certaines de ses sorties contrôlent des processus dans le DFD. Elle

a d'autres sorties qui rétroagissent en entrée des DFC comme flots de contrôle [Hat91]. Ainsi avec les spécifications de contrôle, le modèle des besoins de la méthode SA-RT peut être considéré comme une boucle de rétroaction, comme le montre la Figure I-31.

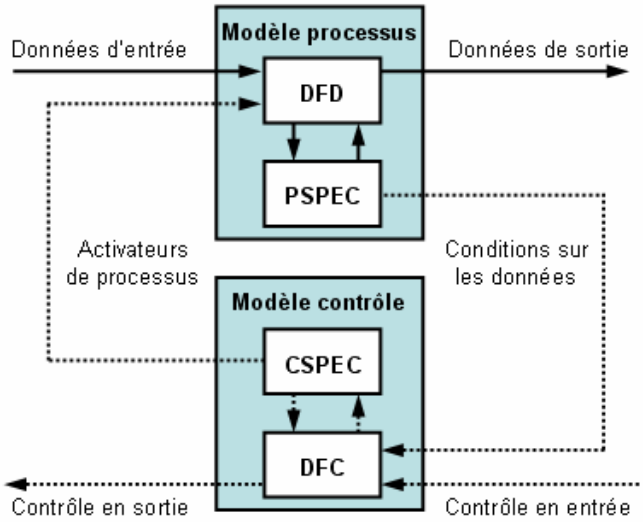


Figure I-31: Modèle des besoins SA-RT [Hat91]

Dans cette figure, nous voyons une information qui provient d'une PSPEC du modèle des processus vers un DFC du modèle de contrôle. Cette information est un flot d'événement dérivé d'une donnée pour assurer le lien unique des DFD vers des DFC. Il s'agit que dans un diagramme de flot de données, tracé à un certain niveau d'analyse, des processus primitifs selon des conditions sur leurs flots de données reçus vont générer des flots d'événements. Ces flots d'événement n'apparaissent pas comme les sorties des processus dans le DFD, mais bien dans le processus de contrôle correspondant du DFC. Ceci entraîne des modifications dans des PSPEC. Et pour vérifier qu'une PSPEC est cohérente, nous devons vérifier le DFD et le DFC.

Le diagramme de flot de contrôle (DFC) et le diagramme préliminaire SA-RT du système de chauffage d'une maison sont donnés sur la Figure I-32 et la Figure I-33 (notez que le diagramme de contexte de données du système est identique que celui de la méthode SA).

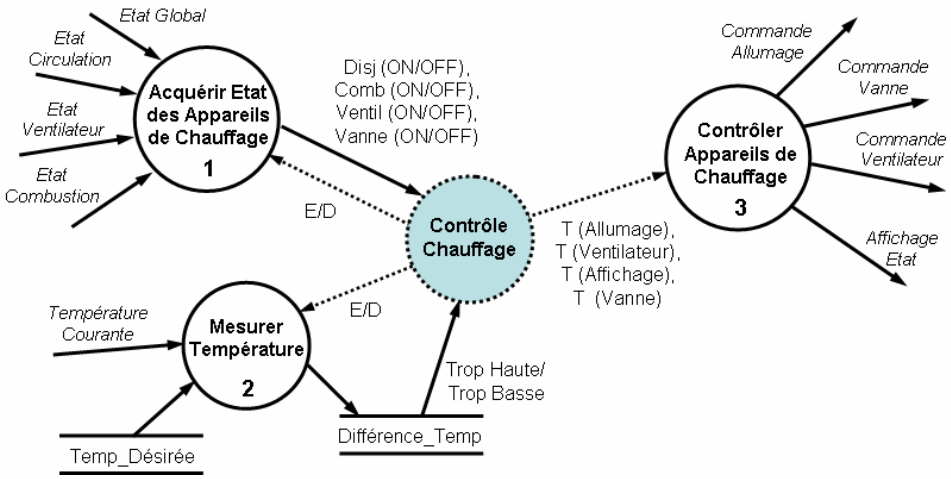


Figure I-32: Diagramme de flots de contrôle du système de chauffage

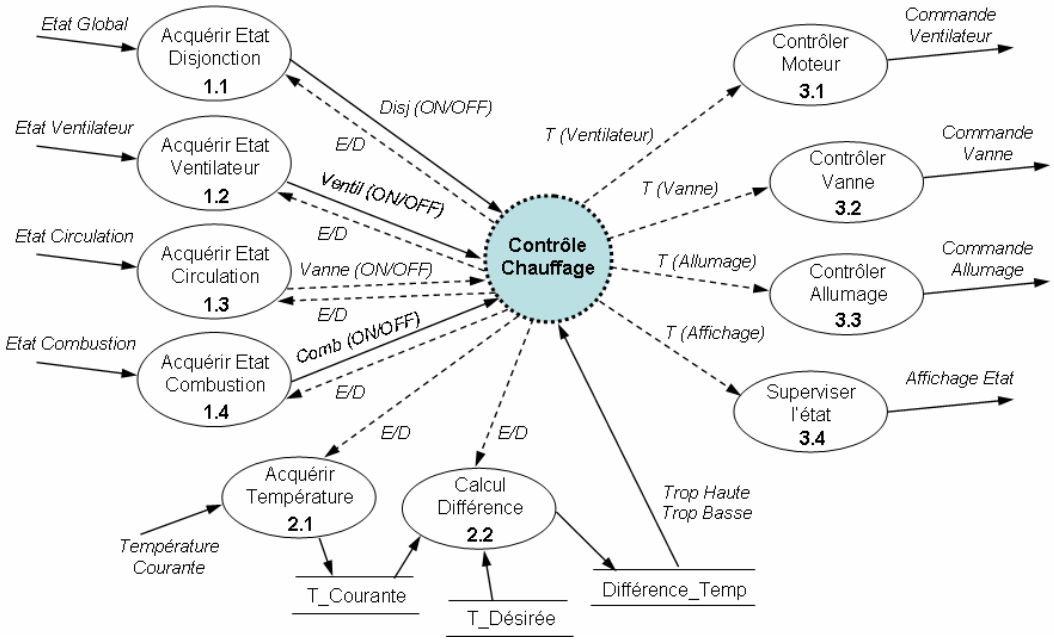


Figure I-33: Diagramme préliminaire SA-RT du système de chauffage

Dans le diagramme préliminaire, nous voyons très clairement la fusion entre le modèle de contrôle et le modèle des processus dans la spécification des besoins de la méthode SA-RT. Le processus de contrôle, appelé «Contrôle Chauffage», est le contrôleur du système. Il dirige le fonctionnement de tous les processus fonctionnels grâce à des flots d'événement appropriés. Donc les processus d'acquisition sont activés ou désactivés par des événements «Enable/Disable» (ou E/D), et les processus de commande sont déclenchés par des événements «Trigger» (ou T). D'autre part chacun des processus d'acquisition après qu'il scrute l'état du bord des appareils de chauffage, envoie un flot d'événement au contrôleur. Selon ces flots d'événement, la CSPEC du «Contrôle Chauffage» va modifier la réponse de processus de données en accord avec les conditions passées et présentes à l'intérieur et à l'extérieur du système. C'est-à-dire l'automate de «Contrôle Chauffage». Les stockages de données qui font des interfaces entre des processus primitifs dans le DFD de la méthode SA (voir la Figure I-27), sont remplacés par les conditions sur les données vers le processus de contrôle «Contrôle Chauffage». Un stockage d'événements, nommé «Différence_Temp», est utilisé. Ce stockage permet de mémoriser un événement généré par le processus «Calcul Différence» de façon à ce que son occurrence soit utilisée plusieurs fois par le contrôleur. Dans ce cas-là, en fonction de la différence entre «T_Courante» et «T_Désirée», l'événement stocké dans ce stockage sera soit «Trop Haute», soit «Trop Basse».

Il ne reste à donner la spécification du processus de contrôle, «Contrôle Chauffage». Nous savons que dans le cas où un diagramme de flot de données contient un processus de contrôle, il est nécessaire de générer une description ou une spécification du processus de contrôle afin de représenter l'aspect comportemental de l'application. Cette spécification peut être faite sous la forme d'un diagramme état-transition. Le diagramme état-transition est la représentation la plus familière d'une machine séquentielle (une machine séquentielle est définie comme une machine à états finis dont les sorties sont déterminées à la fois par les entrées courantes et passées [Hat91]). Dans un diagramme état-transition, nous précisons tous les états possibles d'un système lors de son exécution. Les états sont représentés par des boîtes rectangulaires avec les noms des états. A un instant donné, le système doit être toujours dans l'un de ses états spécifiés. Le changement d'un état courant à un autre état est déterminé par un arc de transition avec une flèche vers l'état suivant. Des événements causant la transition, et des actions engendrées lorsque la transition est franchie sont notés directement sur l'arc de transition. Les deux sont séparés par une ligne (voir Figure I-34).

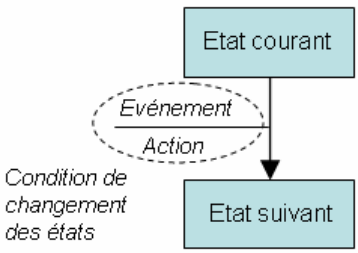


Figure I-34: Etat-transition

Dans tout état donné, quand un événement associé à cet état survient, le système ira dans l'état indiqué par cet arc de transition et exécutera simultanément l'action associée. Si un événement provient, et qu'il n'est pas associé à une transition de cet état, rien n'arrive. Le diagramme état-transition du processus de contrôle «Contrôle Chauffage» est présenté sur la Figure I-35.

- *OFF*: Dans cet état, le disjoncteur principal est mis à l'état OFF. Tous les processus d'acquisition sont inactifs, sauf le processus fonctionnel scrutant le disjoncteur.
- *Allumage_1*: Quand le résidant met le disjoncteur à ON, le système de chauffage arrive à cet état et active en même temps les processus d'acquisition. L'état «Allumage_1» est aussi atteint à partir de l'état «Combustion» lorsque la température actuelle est plus haute que celle de l'extérieur (Trop Haute). Mais dans ce cas, le système arrête le ventilateur et ferme la vanne d'arrivée de fuel.

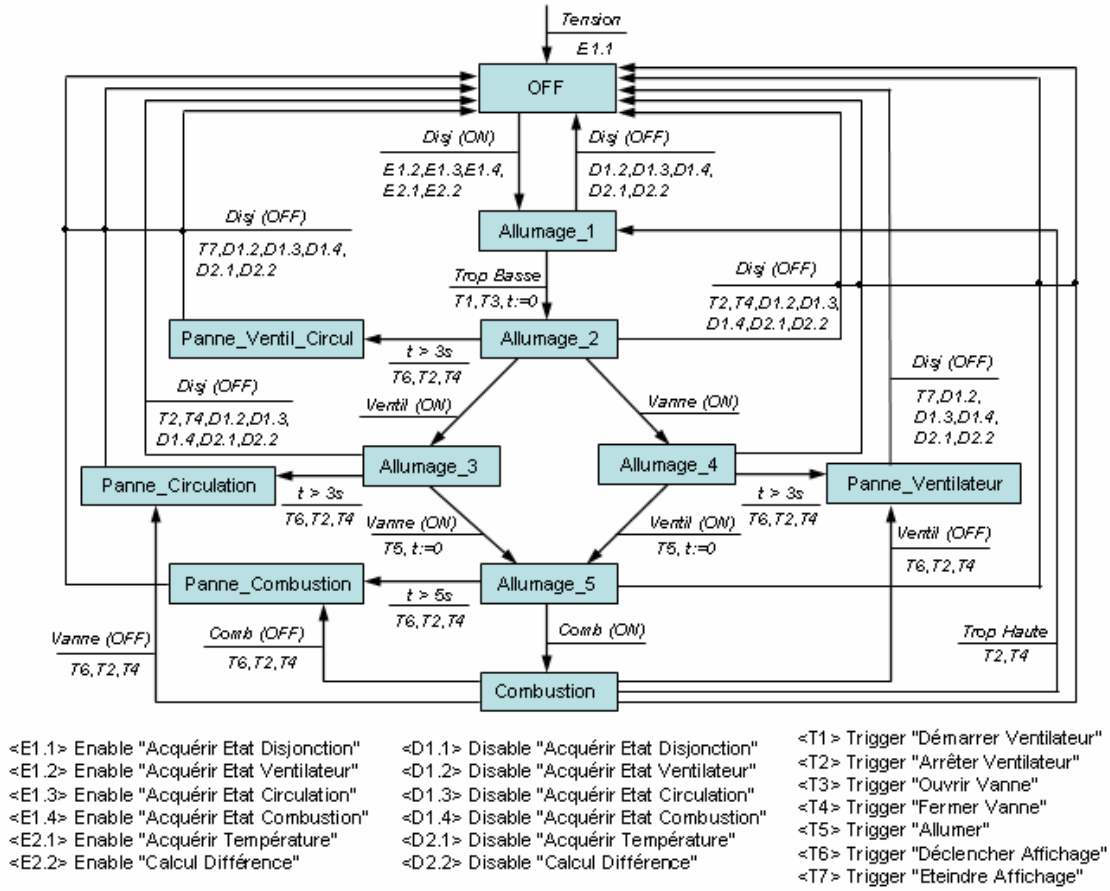


Figure I-35: Diagramme état-transition du procédé «Contrôle Chauffage»

- *Allumage_2*: A l'état «Allumage_1» si la température à l'intérieur de la maison est plus basse que celle à l'extérieur (Trop Basse), le système ira dans cet état et enverra deux événements afin de mettre en route le ventilateur et ouvrir la vanne du fuel. Le chien-de-garde surveillant le changement de l'état est mis à zéro.

- *Allumage_3*: Cet état est atteint lorsque le système est dans l'état «Allumage_2», et lorsque le ventilateur est tourné (Ventil (ON)).
- *Allumage_4*: Cet état est atteint lorsque le système est dans l'état «Allumage_2», et lorsque la vanne du fuel est ouverte (Vanne (ON)).
- *Allumage_5*: Cet état est atteint lorsque la vanne est ouverte (Vanne (ON)) dans l'état «Allumage_3» ou lorsque le ventilateur est mis en route (Ventil (ON)) dans l'état «Allumage_4». Le procédé d'allumage du mélange air/fuel est déclenché, et le temps d'attente d'un changement d'état vers l'état «Combustion» à partir de l'état «Allumage_5» est remis à zéro.
- *Combustion*: A l'état «Allumage_5», si la combustion est détectée (Comb (ON)), l'état «Combustion» est atteint.
- *Panne_Ventil_Circul*: Si 3 secondes après les commandes d'allumage du ventilateur et d'ouverture de la vanne, ces deux éléments ne sont pas en fonctionnement, le système passe à cet état (panne ventilateur et panne circulation).
- *Panne_Ventilateur*: Cet état est atteint si 3 secondes après la commande d'allumage du ventilateur, cet élément ne fonctionne pas toujours, ou si le système étant dans l'état «Combustion», le ventilateur ne marche plus (Ventil (OFF)).
- *Panne_Circulation*: Cet état est atteint si 3 secondes après la commande d'ouverture de la vanne du fuel, cet élément ne fonctionne pas toujours, ou si le système étant dans l'état «Combustion», la vanne est fermée (Vanne (OFF)).
- *Panne_Combustion*: Cet état est atteint si 5 secondes après la commande d'allumage du mélange air/fuel, ce procédé n'est toujours pas en fonctionnement, ou si le système étant dans l'état «Combustion», la combustion est éteinte (Comb (OFF)).

Quand le système est dans mode panne, les commandes «Arrêter Ventilateur», «Fermer Vanne», et «Déclencher Affichage» sont signalées afin d'éteindre le ventilateur, de fermer la vanne du fuel et d'afficher l'état en panne du système à des utilisateurs.

I.5.2.d. Remarques de la méthode SA-RT (Structured Analysis for Real-Time Systems)

SA-RT est une méthode de spécification des systèmes basée sur le concept de la rétroaction. Dans le modèle des besoins de la méthode SA-RT, nous voyons que le système est décomposé non seulement en processus primitifs, mais également en états de façon à ce que la réponse de ces processus primitifs est dirigée par un processus de contrôle en fonction des conditions passées et présentes à l'intérieur et à l'extérieur du système. Tandis que l'activation des processus fonctionnels dans les diagrammes de flot de données de la méthode SA est réalisée par leurs données, le fait que des processus fonctionnels dans les diagrammes de flot de données sont activés par des événements rend la méthode SA-RT bien adaptée à des systèmes de contrôle/commande ou temps réel.

Du point de vue de la spécification des besoins, les diagrammes état-transition (ou tables état-transition, matrices état-transition) nous permettent de décrire distinctement l'aspect comportemental des systèmes de manière dynamique (par rapport aux méthodes SA et JSP). De plus, une spécification des contraintes de temps peut être associée sous la forme de temps de réponse entrée-sortie (une représentation simple du domaine des temps de réponse admissibles entre chaque événement survenant sur les terminaux d'entrée, et chaque événement résultant sur les terminaux de sortie). Cependant, dans le cas des grands systèmes, en particulier des systèmes complexes, la décomposition en états peut provoquer le problème de l'explosion des états où les diagrammes état-transition peuvent être inappropriés en comparaison avec les statecharts ou les tables état-transition.

Quand plusieurs événements associés à un état surviennent simultanément, le choix de la transition à exécuter et de l'état suivant correspondant dans un diagramme état-transition devient non-déterministe. Etant une méthode de spécification, la méthode SA-RT ne propose aucun mécanisme pour résoudre ce problème. Il laisse cela aux étapes suivantes (l'étape de conception et d'implémentation). La communication entre le processus de contrôle et ses processus de données dans SA-RT est la communication synchrone faiblement couplée réalisée uniquement par des flots d'événement (soit un Enable/Disable (E/D) événement, soit un Trigger (T) événement). Mais la communication synchrone fortement couplée entre les processus de données peut être faite par des flots de données discrets de façon à ce que le processus appelant va envoyer une requête au processus appelé, attendre une donnée de ce dernier.

Gomaa dans [Gom 93] propose une méthode de spécification basée sur des objets concurrents, appelée COBRA (Concurrent Object-Based Real-Time Analysis). En fait, c'est une extension de la méthode SA-RT. COBRA utilise les mêmes notations que SA-RT, mais utilise une stratégie alternative pour développer l'aspect comportemental d'un système. En fait, le diagramme de contexte de données dans COBRA se base sur des objets qui peuvent s'exécuter en concurrence (un peu similaire à la méthode JSP, mais différent de la méthode SA-RT). Il fournit aux développeurs des instructions pour décomposer ces objets en des sous-systèmes qui sont indépendants l'un de l'autre, et communiquent par l'échange de données de façon asynchrone. COBRA est une méthode mieux adaptée à des systèmes distribués de grande taille. Pour les plus amples informations de la méthode COBRA, le lecteur pourrait consulter au chapitre 13 de [Gom93].

Sachant que SA-RT est une méthode de spécification, il faut que la coupler à d'autres méthodes pour la conception et l'implémentation. Parmi des méthodes possibles, SD (Structured Design ou Conception Structurée) et DARTS (Design Approach for Real-Time Systems) sont deux méthodes familiales.

I.5.2.e. SD

SD est une méthode d'implémentation et DARTS est une méthode de conception des systèmes temps réel (La méthode DARTS sera présentée aux paragraphes suivants et en détail au chapitre II de ce mémoire). SD, une méthode d'implémentation, est basée sur une idée de M. Constantine [YC79]. Il a constaté que les logiciels qui sont les plus aisément d'implémenter et à changer sont ceux composés de modules simples, indépendants. La raison de ceci est que problème résolvant est plus rapide et plus facile quand le problème peut être subdivisé en pièces qui peuvent être considérées séparément [Ste99]. Le terme module est utilisé dans la méthode SD pour se référer à un ensemble d'instructions contiguës de programme ayant un nom par lequel les autres parties du système peuvent l'invoquer, et ayant de préférence son propre ensemble de variables locales. SD possède ses notations graphiques permettant de représenter un système en forme de graphes structurels. La Figure I-36 montre un graphe structurel typique dans SD: A est un module appelant le module B; B est subordonné à A; B reçoit un paramètre d'entrée X (nommée dans A) et renvoie le paramètre de sortie Y (nommée dans A).

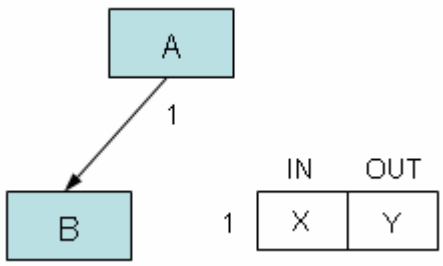


Figure I-36: Graphe structurel

Les modules dans un graphe structurel sont principalement des modules fonctionnels bien que quelques-uns soient des modules d'encapsulation de données. Il est important de noter que SD ne fournit aucune instruction pour structurer un système en tâches concurrentes. Par conséquent, le système est structuré dans SD comme une seule tâche (un programme) dans laquelle les liens entre ses modules sont bien établis (par exemple, grâce au couplage par les données, au couplage par la commande, au couplage par partage d'un environnement commun, etc.). La communication asynchrone des tâches concurrentes est implémentée dans SD par des appels des fonctions ou des expéditions des commandes associées à un module à partir d'un autre. La méthode SD montre comment un problème inhérent à la concurrence peut être résolu en utilisant une méthode de conception sans concurrence.

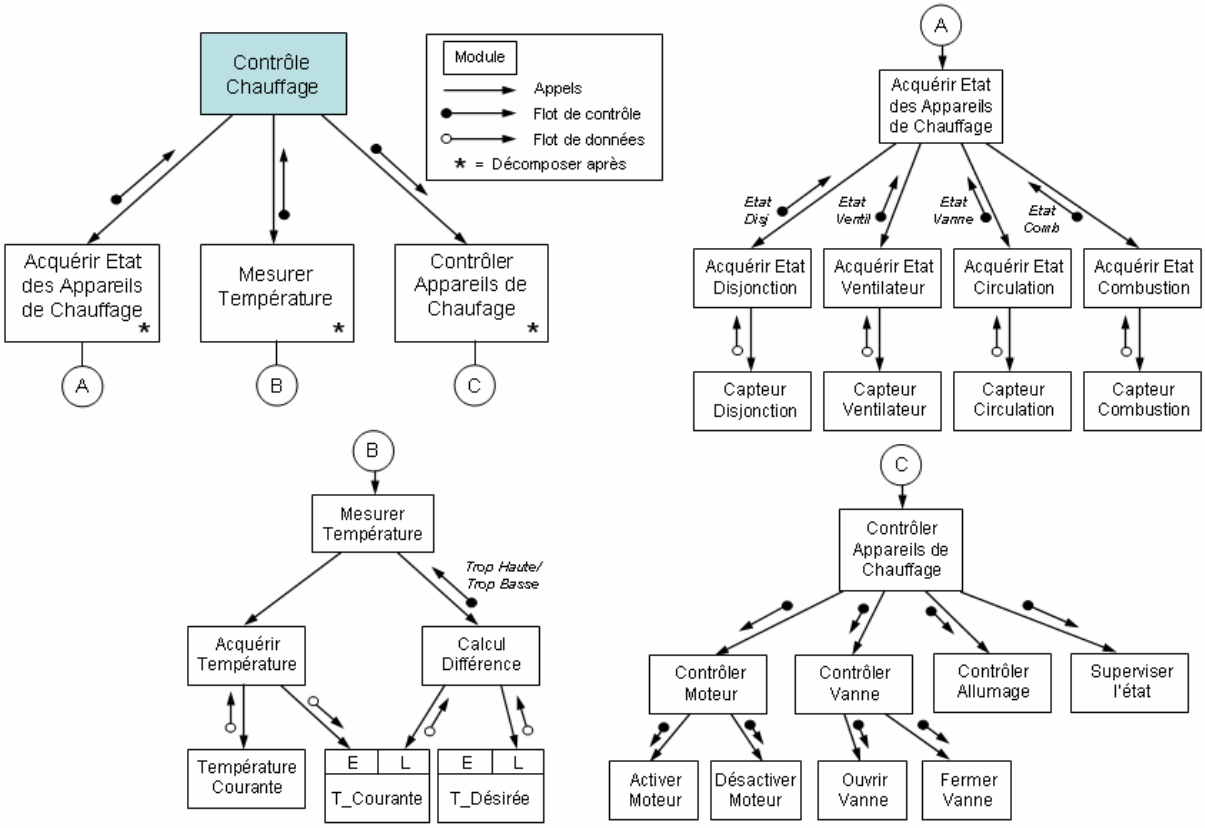


Figure I-37: Graphes Structurels du système de chauffage d'une maison

Les graphes structurels du système de chauffage d'une maison sont donnés sur la Figure I-37. Grâce à l'approche de l'inspection I/O (Poll I/O), les états des appareils du chauffage (le disjoncteur principal, le ventilateur, la vanne, et la combustion) sont échantillonnés périodiquement par les modules appelés par le module «Acquérir Etat des Appareils de Chauffage». Par exemple, le module «Capteur Ventilateur» renvoie un flot de données indiquant le fonctionnement actuel du ventilateur si une demande est reçue depuis la dernière fois à ce que ce module est appelé par le module «Acquérir Etat Ventilateur». Sinon, il renvoie une demande nulle. Puis, le module «Acquérir Etat Ventilateur lui-même va renvoyer un flot de contrôle s'il y a une demande à partir du module «Acquérir Etat des Appareils de Chauffage». Les deux modules d'encapsulation de données, nommés «T_Courante» et «T_Désirée», sont utilisés pour stocker les valeurs de la température dans la maison et la température choisie par l'utilisateur. Chacun de ces deux modules possède des opérations permettant des autres modules accèdent à ses données (donc dans ce cas-là, ce sont l'opération «Ecrire» et l'opération «Lire»). Le module d'encapsulation de données est introduit dans SD afin d'alléger les résultats indésirables des variables globales.

La *Figure I-37* nous montre comment réaliser le modèle d'implémentation d'un système à partir de son modèle des besoins SA-RT avec la méthode SD (Structured Design ou Conception

Structurée). Malgré le fait que le choix de la méthode SD dans ce cas-là soit instructif, la méthode SD limite la conception des systèmes concurrents ou temps réel à cause de la faiblesse dans la structuration en tâches. L'application d'un autre modèle de conception tel que celui de la méthode DARTS s'adapte beaucoup. Pour plus d'informations concernant les détails de la méthode SD, le lecteur pourrait consulter [Ste99][Gom93].

I.5.3. Statecharts

Les *Statecharts* ont été proposés par David Harel en 1983 comme un formalisme visuel pour la spécification comportementale des systèmes réactifs [Har87][Har88]. L'aspect fonctionnel d'un système peut être représenté par un ensemble de diagrammes de flots de données hiérarchiques dans lesquels le fonctionnement des composants du système est capturé par des données et des signaux qui font des interfaces entre ces composants. Pour l'aspect comportemental, la compréhension des diagrammes de flot de données/flot de contrôle peut être décrite dynamiquement par des diagrammes états-transition. Mais dans le cas des systèmes complexes ou de grande taille, le choix de diagrammes état-transition devient infaisable car le nombre des états possibles peut augmenter exponentiellement. Les statecharts avec des caractéristiques avancées, permettent d'alléger les inconvénients des diagrammes états-transition. De plus, en intégrant les statecharts dans une approche de spécification convenable, nous pouvons obtenir des modèles exécutables, des analyses dynamiques et des synthèses de code, ce qui facilite la vérification comportementale et la génération de code à partir de la spécification ou de la conception.

A la différence d'un diagramme états-transition classique, dans un statechart il existe trois types d'état différents: l'état de type OR (OR-state), l'état de type AND (AND-state) et l'état élémentaire (Basic-state). Chaque état est représenté par une boîte rectangulaire arrondie étiquetée par le nom de l'état.

- *«OR-state»*: un état du type OR peut avoir des sous-états qui sont liés l'un à l'autre par l'opération logique OU-exclusif. C'est-à-dire qu'à un instant donné, si le système est dans un OR-état, il doit être dans un des sous-états de OR-état. L'état D sur la Figure I-38 est un exemple. Il comporte deux sous-états A et C. Par défaut, quand le système est dans l'état D, il est immédiatement dans son sous-état A. L'événement β entraîne le changement d'état à partir de soit l'état A, soit l'état C vers l'état B. L'utilisation de OR-état dans ce cas-là nous permet de diminuer les transitions entre les deux états A, C et l'état B par rapport au diagramme état-transition original donné à gauche.

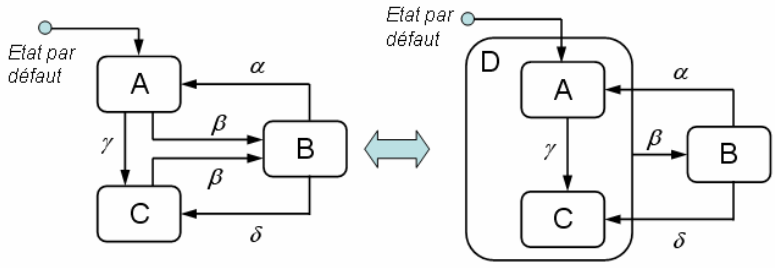


Figure I-38: Un OR-état

- *«AND-state»*: un état du type AND peut avoir des composants orthogonaux qui sont liés par l'opération logique AND. A un instant donné, si le système est dans un AND-état, il doit être dans tous les composants de AND-état. L'état X sur la Figure I-39 est un AND-état qui se compose de deux composants, appelés A et D, séparés par une ligne en pointillé. Quand le système est dans l'état X, il doit posséder un état dans les deux composants, donc au début il passe à l'état B de A et à l'état F de D. Nous voyons que le produit orthogonal est une généralisation de produit usuel d'un automate. Ceci nous permet d'améliorer considérablement le problème d'explosion des états dans les diagrammes états-transition des systèmes complexes.

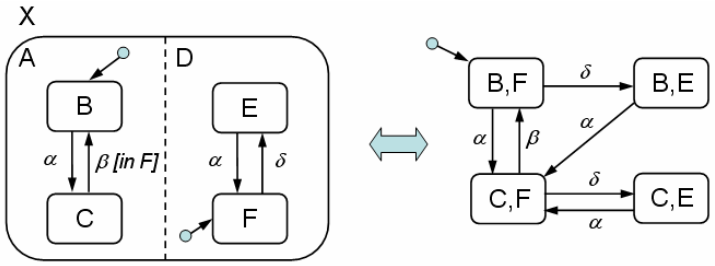


Figure I-39: Un AND-état

- *«Basic-state»*: un état élémentaire est un état que nous ne pouvons pas décomposer en sous-états. Cette sémantique est similaire à celle de l'état dans un diagramme état-transition. Donc, sur la Figure I-39, B, C, E et F sont des états élémentaires.

De même que pour un diagramme états-transition, la transition d'un état à un autre dans un statechart est déterminée par un arc avec des événements causant la transition et des actions s'exécutant lorsque la transition est exécutée. Cependant, la syntaxe de l'expression étiquetant une transition dans un statechart est un peu différente car elle permet d'ajouter une opération booléenne aux événements pour exprimer efficacement une condition d'activation. La syntaxe de la transition dans un statechart est «événement [condition]/action». Par conséquent, une transition n'a lieu que si tous les événements sont arrivés, et que si le test de l'opération booléenne est vrai. Sinon, le système reste dans le même état. La transition de l'état C à l'état B du composant orthogonal A sur la Figure I-39 est un exemple. Cette transition devient franchissable si un événement β survient et si le système se trouve dans l'état F du composant orthogonal D au moment du déclenchement de cet événement.

Chacun des états d'un statechart peut être associé à un nombre de réactions statiques qui sont exécutées en fonction de certains événements et conditions, dont la syntaxe est similaire à celle d'une transition, lorsque le système se trouve dans cet état. Dans un état donné, certains événements sont requis pour provoquer des réactions, mais non pour changer cet état.

Supposons que le système soit dans l'état A, et qu'un événement provoquant la transition t_1 (donc, la syntaxe de t_1 est «événement/action») à partir de l'état A vers l'état B s'est produit. La réponse du système est :

- la transition t_1 devient permise,
- les éventuels événements ayant lieu lorsqu'on quitte l'état A et qu'on entre dans l'état B sont générés,
- la condition spéciale indiquant l'état dans A du système, appelé «dans(A)», devient fausse et l'autre condition pour l'état B, appelé «dans(B)», devient vraie,
- des actions spécifiées sur la sortie de l'état A sont exécutées (par exemple, [not A]/action),
- des actions spécifiées sur l'arrivée dans l'état B sont exécutées (par exemple, [B]/action),
- des réactions statiques de l'état S (l'état père des deux états A et B) vont continuer à s'exécuter si les événements déclenchant ces réactions sont vrais (car la transition de l'état A vers l'état B ne fait pas sortir de l'état S),
- toutes les activités qui sont exécutées lorsque le système est dans l'état A sont désactivées, alors que celles qui sont définies lorsque le système est dans l'état B sont activées.

Par conséquent, le système ne peut pas entrer et puis sortir d'un état donné vers un autre état dans un même pas. En fait, il a besoin de deux pas, donc un pas pour l'entrée de l'état donné, et l'autre pas pour la sortie de cet état. Dans un statechart, Harel a appelé une configuration un ensemble maximal des états dans lesquels le système peut être présent simultanément. Puis, la transition actuelle entre les configurations peut consister en un nombre de transitions séparées apparues dans les composants orthogonaux différents. De plus, chacune de ces transitions peut consister en un

nombre de segments-transitions qui sont représentés par des flèches étiquetées vers des états, et un connecteur de l'un des deux types suivants: soit un AND-connecteur, soit un OR-connecteur. La Figure I-40 nous donne un exemple d'un OR-connecteur, nommé © (un connecteur- condition). Etant donné que [T_1 ou T_2] est un ensemble de trois segments-transitions, donc segment-transition à partir de [T_1 ou T_2] doit contenir exactement un segment de T_1 et un segment de T_2.

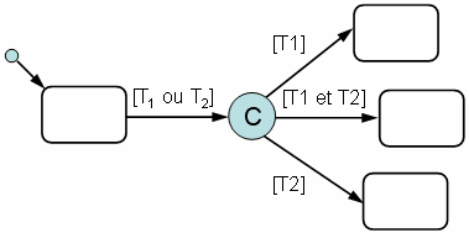


Figure I-40: OR - connecteur

Un point important qui fait différencier un diagramme état-transition d'un statechart est que dans un statechart la portée des transitions (Scope of transitions) peut être déterminée. La portée d'une transition permet de définir la priorité des transitions. La transition ayant la portée la plus grande est la transition la plus prioritaire. Par conséquent, si plusieurs transitions ont lieu simultanément à un état donné, la transition qui possède la portée la plus prioritaire sera exécutée, contrairement à un état-transition où le choix de la transition dans ce cas-là se produit de façon arbitraire. Le problème de non-déterminisme des transitions est donc allégé. Par exemple, sur la Figure I-41: la portée de la transition t_1 est l'état A, la portée de la transition t_2 est l'état V. Puisque l'état A est un sous-état de l'état V, la transition t_2 possède une priorité supérieure que celle de la transition t_1. Et il en résulte que la transition t_2 aura lieu en priorité par rapport à t_1, si toutes les deux surviennent en même temps.

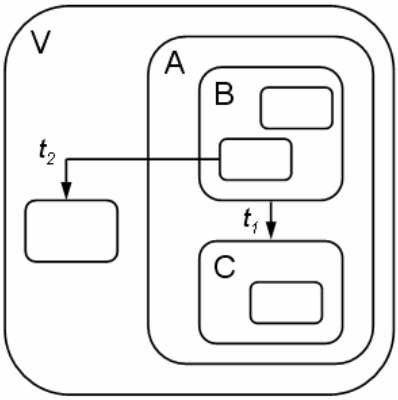


Figure I-41: Portée de transitions

Grâce à toutes ces caractéristiques, les statecharts de Harel permettent de capturer non seulement les états des objets comme une pré-condition pour des requêtes, mais également l'aspect comportemental des objets en répondant à ces requêtes et en maintenant les relations avec d'autres objets. Les statecharts génèrent un modèle de spécifications des besoins d'un système orienté objets concurrents. Pour la suite, afin d'éclaircir cette pensée, l'aspect comportemental du système de chauffage d'une maison sera représenté en utilisant les statecharts.

Du point de vue des développeurs, le système de chauffage peut être considéré comme un objet à quatre composants: le Contrôle-chauffage, le Moteur, la Vanne et la Chambre de combustion. Donc le Contrôle-chauffage peut être décomposés en quatre composants orthogonaux, c'est-à-dire le Contrôleur, le Disjoncteur principal, la Température et la Surveillance. Le statechart du «Contrôleur» est donné sur la Figure I-42 ci-dessous.

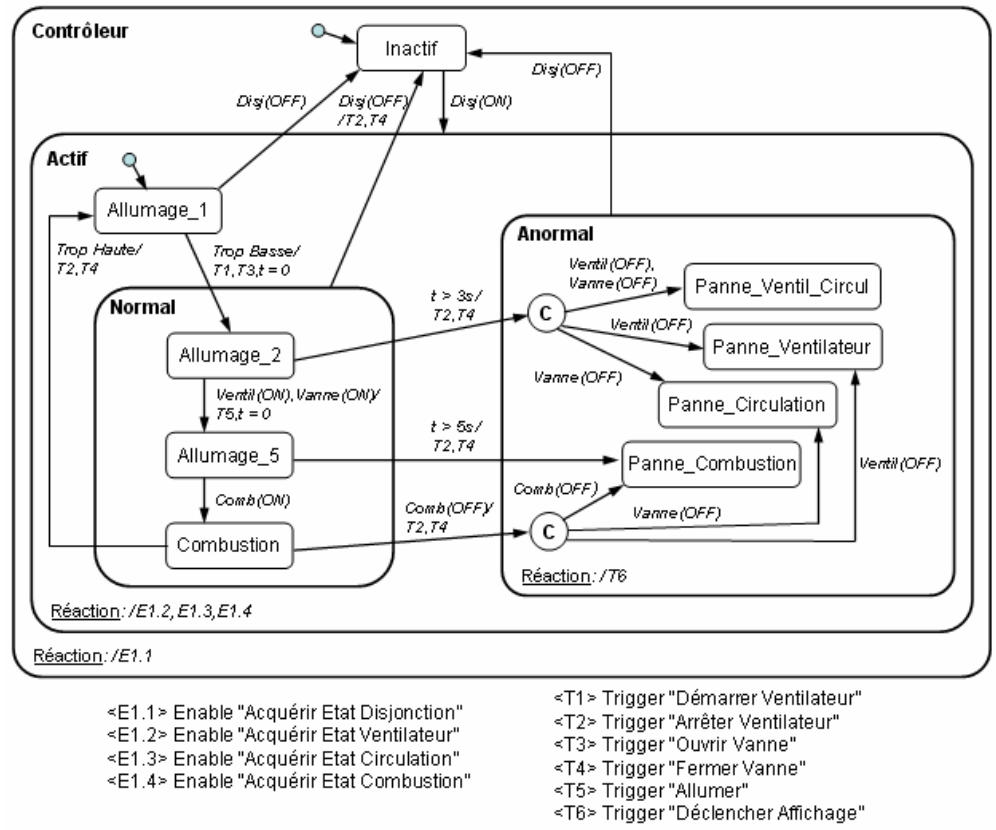


Figure I-42: Statechart du «Contrôleur»

Au début quand une tension de 220 volts par exemple est fournie, le contrôleur est mis en état par défaut «Inactif», et la réaction statique «Acquérir Etat Disjoncteur» est activée afin d'observer l'état du disjoncteur principal. Lorsque l'utilisateur appuie sur le bouton «Mise en Marche» du Disjoncteur, l'événement «Disj(ON)» est signalé. Et par la suite le contrôleur devient actif en changeant d'abord de l'état actuel à l'état «Allumage_1», l'état par défaut de l'état «Actif», puis en activant les trois réactions pour acquérir l'état des appareils du chauffage («Acquérir Etat Ventilateur», «Acquérir Etat Circulation», «Acquérir Etat Combustion»). De même, le composant orthogonal «Température» change de l'état par défaut «Inactif» à l'état «Mesurer Température», et fait déclencher les deux réactions «Acquérir Température», «Calcul Différence», donc la dernière va calculer régulièrement la différence entre la température dans la maison et la température désirée. Elle provoque l'événement «Trop Basse» si cette différence est inférieure à -2, ou l'événement «Trop Haute» si celle-ci est supérieure à 2 (le statechart de la «Température» est donné sur la Figure I-43). Si la température devient trop basse, la transition de l'état «Allumage_1» vers l'état «Allumage_2» se produit. Il en résulte que le démarrage du ventilateur et l'ouverture de la vanne sont déclenchés. Si 3 secondes après les commandes de ces déclenchements, le ventilateur et la vanne ne sont toujours pas en fonctionnement correct, le contrôleur passe à l'état «Anormal» et engendre deux actions pour arrêter le ventilateur et fermer la vanne. Dans l'état «Anormal», en fonction de la condition qui provoque cette transition, le contrôleur est soit dans l'état «Panne_Ventilateur», soit «Panne_Circulation», soit «Panne_Ventil_Circul» (voir Figure I-43). De plus, une alarme est lancée grâce à la réaction «Déclencher Affichage» afin de signaler l'état en panne du système à l'utilisateur. Il en va de même pour l'état «Allumage_5» et l'état «Combustion». Etant dans l'état «Anormal», l'utilisateur doit arrêter le fonctionnement courant du système en appuyant sur le bouton «Mise en Arrêt» du disjoncteur principal pour qu'un service de maintenance puisse intervenir ensuite. Notons que dans le cas de fonctionnement normal, à partir de tous les états un appui sur la «Mise en Arrêt» du disjoncteur va causer une terminaison immédiatement du chauffage (le contrôleur va retrouver son état «Inactif», et les acquisitions de l'état des appareils sont désactivées).

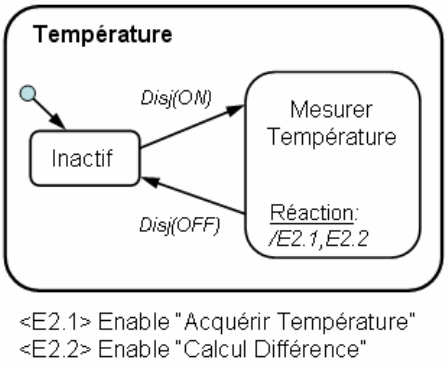


Figure I-43: Statechart de la température

Dans le cas contraire, si pendant 3 secondes après les commandes de démarrage du ventilateur et d'ouverture de la vanne, ces deux éléments fonctionnent correctement, le contrôleur passe à l'état «Allumage_5», et l'action d'allumage de mélange air/fuel est déclenchée. Si l'allumage est achevé dans 5 secondes, le contrôleur passe à l'état «Combustion» (sinon il passe à l'état «Panne_Combustion»). Puis il reste dans cet état jusqu'à ce que la température dans la maison devienne trop haute par rapport à celle désirée (ceci va causer la transition vers l'état «Allumage_1», l'arrêt du ventilateur et la fermeture de la vanne). Le fonctionnement du système de chauffage va continuer de nouveau comme celui décrit dessus.

Remarques des statecharts de Harel:

Le statechart de Harel est une extension de l'état-transition traditionnel. Grâce aux concepts de «OR-state» et «AND-state», les statecharts permettent d'une part d'encapsuler des sous-états en états abstraits en niveaux différents afin de favoriser la représentation des comportements dynamiques d'un système (le «Contrôleur» du chauffage sur la Figure I-42 est un exemple. Au niveau le plus abstrait de son statechart, le «Contrôleur» peut être soit dans l'état «Inactif», soit dans l'état «Actif» à tout instant), d'autre part de décomposer un système en objets de manière à ce que les statecharts décrivent non seulement comment ces objets communiquent et collaborent, mais aussi comment ils réalisent leurs propre comportement interne (par exemple, la collaboration entre le statechart du composant orthogonal «Contrôleur» et celui de «Température» sur les Figure I-42 et Figure I-43). De nos jours, les statecharts jouent un grand rôle dans les approches de développement des systèmes orienté-objet telles que le UML (Unified Modeling Language), le OOSE (Object-Oriented Software Engineering), etc.

Etant un langage de description des spécifications de besoins, les statecharts de même que les diagrammes état-transition ne proposent aucun mécanisme pour les traitements de l'interaction entre des objets sauf le déclenchement d'événements. Mais, en associant des conditions aux événements qui provoquent les transitions, les statecharts permettent de borner aisément le déclenchement de ces transitions. De plus, grâce au concept de la portée d'une transition, dans les statecharts la priorité de chaque transition peut être déterminée. Le problème de non-déterminisme dont nous avons parlé dans les digrammes état-transition auparavant est ainsi résolu. Par exemple, sur la Figure I-42 quand le «Contrôleur» est dans l'état «Allumage_5», si les deux événements, nommés «Comb(ON)» et «t>5s», surviennent simultanément, la transition provoquée par l'événement «t>5s» vers l'état «Panne_Combustion» sera exécutée car sa portée (l'état «Actif») est supérieure que celle de la transition provoquée par l'événement «Comb(ON)» (l'état «Normal» dans ce cas-là).

Enfin, dans les statecharts le fait que des entrées/sorties d'un état donné puissent causer des activations/ désactivations de certaines réactions statiques associées à cet état facilite considérablement le contrôle des réactions (ou des processus primitifs) au lieu de l'utilisation des flots d'événement E/D (Enable/Disable) en comparaison avec l'état-transition. En outre dans les étapes de conception et d'implémentation après, le développeur peut implémenter rapidement ces réactions comme des opérations internes d'un objet de façon à ce que chaque opération pendant

son exécution déclenche des événements, ou invoque des opérations des autres objets selon certaines conditions. La réaction «Calcul Différence» sur la Figure I-43 est un exemple. Cette réaction n'est activée que si le composant orthogonal «Température» est dans l'état «Mesurer Température». Puis lors de son activation, elle va générer selon la différence entre les deux valeurs de température, «T_Courante» et «T_Choisie», une occurrence de l'événement «Différence_Temp» qui vaut soit «Trop Haute», soit «Trop Basse», afin de provoquer les changements des états dans le composant orthogonal «Contrôleur».

Dans le cadre de ce mémoire, la représentation de statecharts est limitée à certaines de leurs caractéristiques de base. Pour de plus amples informations concernant les statecharts de Harel, le lecteur pourrait consulter [Har90][Har96][Har97].

I.5.4. DARTS (Design Approach for Real-Time Systems)

Un système concurrent est un système où plusieurs activités peuvent s'exécuter en parallèle. Puisque chaque activité peut être représentée par une tâche devant pouvoir communiquer ou se synchroniser avec les autres, une méthode de conception doit être capable de créer à partir d'une spécification un modèle des tâches d'un système temps réel en fournissant des outils pour adresser leurs problèmes de communication et/ou synchronisation.

Proposée par H. Gomaa dans [Gom84][Gom93], la méthode DARTS (Design Approach for Real-Time Systems) est une telle méthode : elle fournit une approche permettant de structurer un système en tâches concurrentes, ainsi qu'un mécanisme permettant de définir des interfaces entre les tâches. Les notations des éléments d'un diagramme DARTS sont représentées sur la Figure I-44 ci-dessous: ce sont celles les tâches, boîtes aux lettres, synchronisations, rendez-vous, événements d'activation, ou mémoire partagée. Ces éléments sont les éléments de base fournis par un exécutif multitâche.

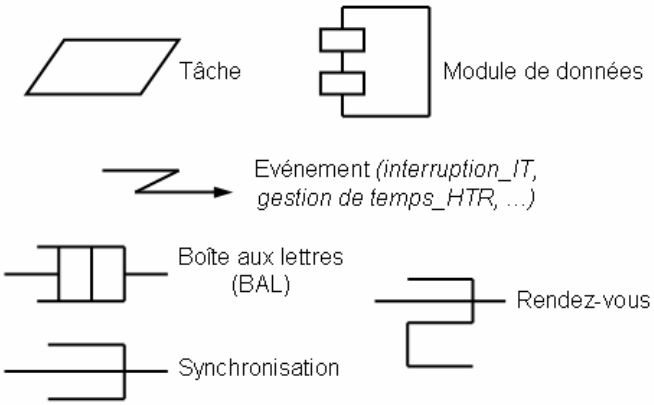


Figure I-44: Eléments d'un diagramme DARTS

I.5.4.a. Concept de tâches DARTS

Dans un diagramme DARTS, *une tâche* est représentée par un parallélogramme. Elle peut être de l'un des deux types suivants:

- *tâche matérielle*: une tâche périodique (déclenchée par l'horloge temps réel HTR), soit une tâche apériodique (déclenchée par interruption ou chien de garde). Elle est donc activée par une interruption interne (horloge temps réel, chien de garde), ou par une interruption externe provenant d'un procédé, via des dispositifs d'acquisition (entrée/sortie numérique déclenchant une interruption, arrivée d'un message sur un bus de communication, etc.).
- *tâche logicielle*: une tâche activée par une autre tâche (soit tâche matérielle, soit tâche logicielle). Les caractéristiques temporelles d'une tâche logicielle sont héritées des

caractéristiques temporelles de la tâche déclenchante, en fonction des relations temporelles spécifiques entre la tâche logicielle et la tâche déclenchante [BC98].

I.5.4.b. Synchronisation

La synchronisation représente une communication sans données où une tâche, appelée «productrice», génère un événement qui aura pour effet de déclencher une tâche, appelée «consommatrice». La « consommatrice » se bloque en attendant cet événement. La synchronisation est utilisée pour satisfaire une contrainte de précédence entre deux ou plusieurs tâches. La Figure I-45 représente une synchronisation entre deux tâches, appelées «Tâche Productrice» et «Tâche Consommatrice» : la première est susceptible de déclencher l'activation de la seconde, qui est bloquée en attendant ce déclenchement. La tâche «consommatrice» ne peut en aucun cas être exécutée avant l'exécution de la tâche «productrice», la contrainte de précédence entre ces deux tâches est donc satisfaite. En pratique, il existe de plusieurs outils permettant d'implémenter la synchronisation. Parmi ceux-ci, le Sémaphore et l'Evénement sont des outils utilisés fréquemment. On peut noter que DARTS ne permet pas d'exprimer la mémorisation ou non (au sens des événements) des déclenchements.

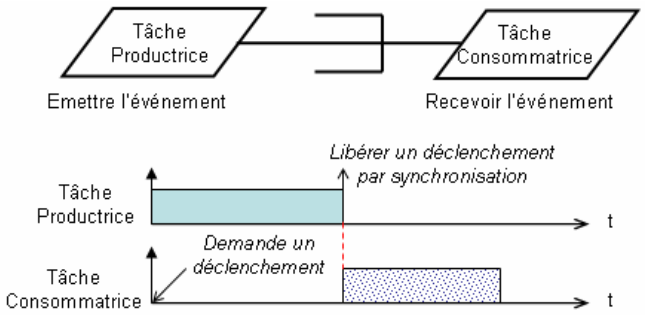


Figure I-45: Synchronisation entre deux tâches (modèle dirigé par le temps)

I.5.4.c. Communication

La communication permet le passage d'informations d'une tâche à une autre. Ce type d'interface est utilisé quand une tâche, appelée «émettrice», a besoin d'expédier un message à une tâche, appelée «réceptrice». La communication peut être de l'un des deux types communs suivants:

- *la communication synchrone (Closely-coupled communication)*: représentée typiquement par le *Rendez-vous (Ada) ou «Barrier» de la norme POSIX*. L'expédition de message et la réception de la réponse sont effectuées pour chacune sur une queue de taille 1, l'une pour le message et l'autre pour la réponse. Une fois que la tâche «émettrice» envoie un message à la tâche «réceptrice», elle se bloque en attente jusqu'à la réception d'une réponse de la tâche «réceptrice» (voir Figure I-46),

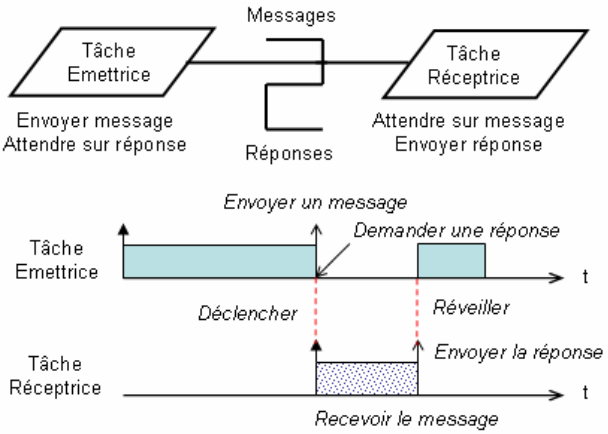


Figure I-46: Communication synchrone entre deux tâches

- *la communication asynchrone (Loosely-coupled communication)*: représentée typiquement par la *Boîte aux lettres (BAL)*, est une queue des messages dans laquelle l'expédition et la réception des messages sont indépendantes. La réceptrice est en attente de message qu'elle consomme pour s'activer. Elle reste bloquée tant que la file de messages est vide. Lorsque la file de messages est bornée, et qu'il n'y a pas d'écrasement, la tâche «productrice»se bloque si la file de messages est plaine (paradigme producteur/consommateur). La productrice ne peut pas se retrouver bloquée si la boîte est non-bornée ou si il y a écrasement de messages lorsque la file est pleine. Il est important de noter que lorsqu'un message est bien envoyé, la tâche «émettrice» n'a pas à se bloquer en attente d'une réponse de la réception du message à la tâche «réceptrice» (voir Figure I-47).

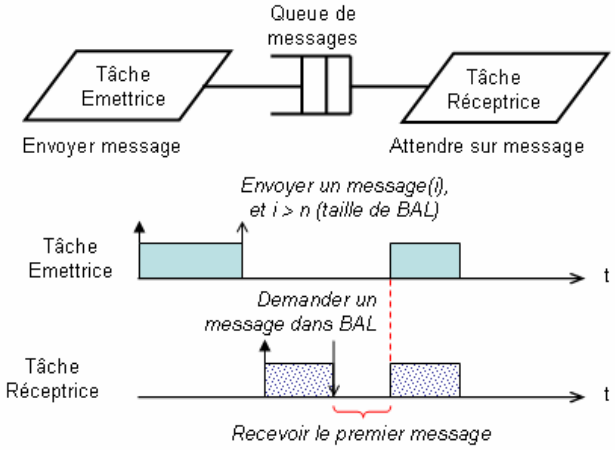


Figure I-47: Communication asynchrone par BAL bornée entre deux tâches

I.5.4.d. Module de donnée

Le Module de Données (Information Hiding Module) est une extension du mécanisme d'abstraction de données de Parnas [Par72]. C'est un stockage de données où l'accès aux données n'est effectué par des tâches qu'à travers une interface en appelant ses opérations (ou fonctions) non ré-entrantes les unes par rapport aux autres (respect de l'exclusion mutuelle). Chaque module de données comporte deux parties: une partie visible aux utilisateurs qui joue le rôle de l'interface abstraite avec des opérations du module, et une autre partie invisible aux utilisateurs qui décrit le comportement du module. La Figure I-48 représente un modèle typique de module de donnée utilisé fréquemment dans les diagrammes DARTS. Ce module possède une interface de deux opérations, nommées «Ecrire» et «Lire», permettant de manipuler les données encapsulées. La mise à jour de données est réalisée par une tâche «Ecriture», qui utilise l'opération «Ecrire», tandis que la récupération de données est effectuée par une autre tâche, appelée «Lecture», qui utilise l'opération «Lire». L'atomicité de ces deux opérations est garantie par un outil de synchronisation inclus dans ce module (par exemple, un sémaphore d'exclusion mutuelle).

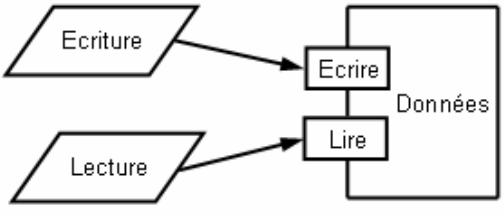


Figure I-48: Module de données dans un diagramme DARTS

I.5.4.e. Démarche DARTS

Puisque la méthode DARTS se focalise sur une analyse structurée des flots de données d'un système, le développement des diagrammes flots de données du système peut être considéré

préalablement: le système est décomposé en des processus primitifs (ou des fonctions) d'abord, et puis des flots de données qui forment des interfaces sont identifiés. Les diagrammes flots de données (DFD) obtenus dans la méthode DARTS sont assez semblables à ceux dans la méthode SA de Demarco (réf. I.1.4.2). La seule différence est que par rapport au DFD de la méthode SA, certains stockages de données sont remplacés par des flots de données qui proviennent des processus d'acquisition vers le processus de contrôle dans les diagrammes DARTS. Cela permet de faciliter la définition des interfaces entre les processus dans l'étape de décomposition du système en des tâches concurrentes DARTS après. Le DFD de la méthode DARTS du système de chauffage est représenté sur la Figure I-49.

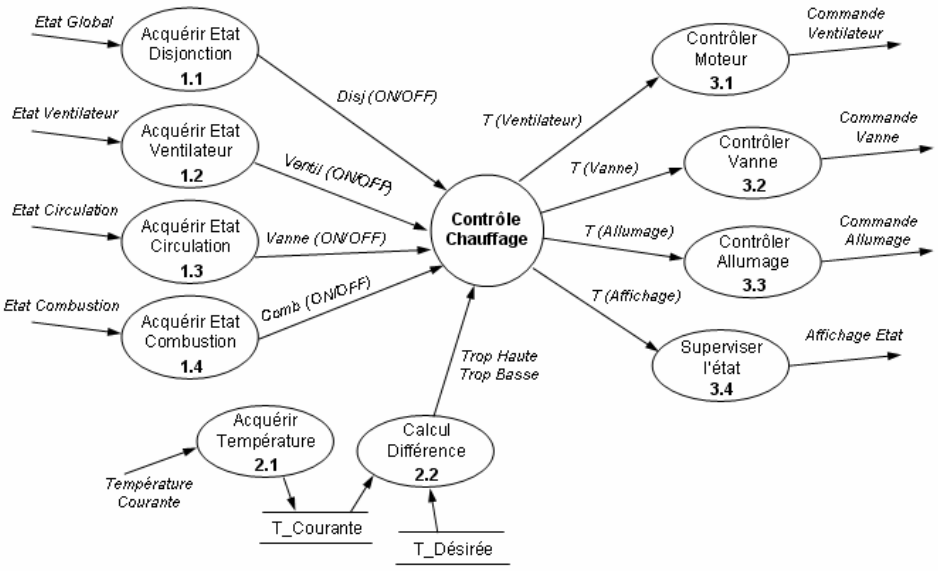


Figure I-49: DFD du système de chauffage dans la méthode DARTS

Ensuite, à partir de DFDs du système, les tâches concurrentes sont identifiées en fonction de la nature asynchrone de chaque processus primitif: quels processus peuvent s'exécuter concurremment, et quels processus doivent s'exécuter séquentiellement. Un processus peut être une tâche, ou une tâche peut comporter plusieurs processus. Par conséquent, les diagrammes flots de données sont repris afin de montrer les tâches et leurs interfaces.

Les critères permettant d'identifier si un processus primitif (ou une fonction) est une tâche séparée ou groupé(e) avec les autres processus dans une tâche sont les suivantes [Gom84]:

- Dépendance de l'entrée/sortie (I/O): quand il concerne une entrée physique ou une sortie physique du système, un processus fonctionnel est souvent exécuté à la même vitesse que la vitesse du dispositif avec lequel il exerce une interaction. En général un tel processus doit être identifié en tant qu'une tâche séparée.
- Fonctions en temps critique: une fonction en temps critique doit s'exécuter à un niveau de priorité plus élevée, donc doit être une tâche séparée.
- Processus de calcul (Computational requirements): une tâche identifiée à partir d'un ensemble des processus (ou fonctions) peut s'exécuter comme une tâche moins prioritaire en utilisant le temps oisif de processeur.
- Cohésion fonctionnelle: les processus qui représentent un ensemble de fonctions enchaînées peuvent être groupés en une tâche. Cela permet de diminuer le trafic entre les fonctions par rapport à si elles étaient développées en tant que tâches séparées. Par conséquent, chaque fonction sera implémentée comme un module distinct dans la même tâche afin d'assurer la cohésion fonctionnelle entre elles au niveau de module.

- Cohésion temporelle: certains processus représentent des fonctions qui s'exécutent simultanément. Ces fonctions peuvent être groupées dans une tâche. Cela permet de déclencher ces processus en même temps dès que la tâche reçoit un stimulus.
- Exécution périodique: un processus qui a besoin d'être exécuté périodiquement peut être structuré en tant qu'une tâche distincte qui est déclenchée par une horloge ou un événement dans des intervalles réguliers.

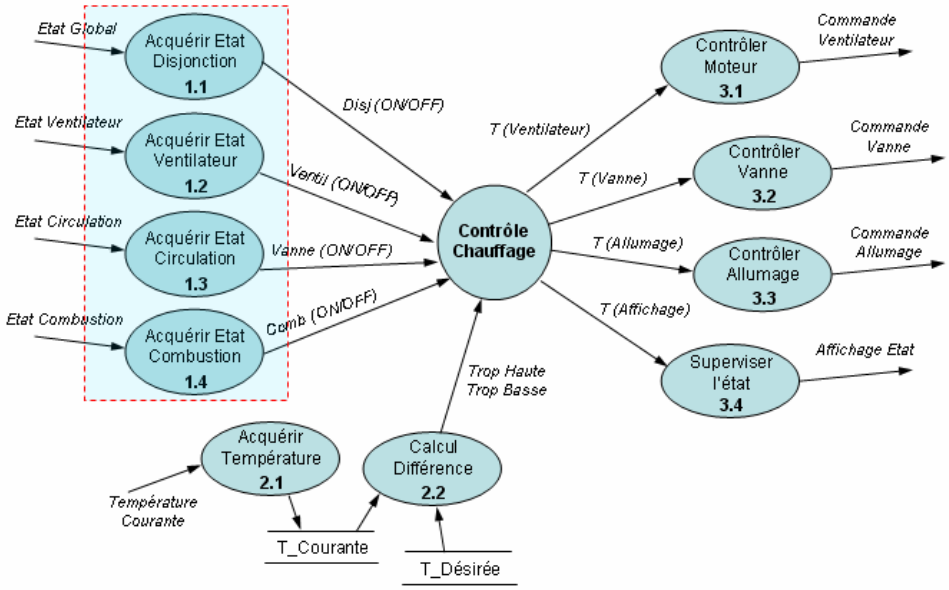


Figure I-50: Identification de tâches

Le diagramme «Identification de Tâches» du système de chauffage est donné sur la Figure I-50. Comme toutes les tâches d'acquisition sont déclenchées par des interruptions en provenant d'une horloge temps réel (HTR), chacune de ces tâches est identifiée en tant qu'une tâche périodique. Au cas où les fréquences de leur activation sont les mêmes, nous pouvons les grouper dans une seule tâche puisqu'il y a cohésion temporelle. Et comme toutes les tâches de commande sont activées par le processus «Contrôle Chauffage», selon la règle de la dépendance de l'entrée/sortie, elles sont identifiées chacune en tant qu'une tâche logicielle. Suivant la cohésion fonctionnelle, «Calcul Différence» est une tâche matérielle activée par HTR qui comporte trois fonctions, donc les deux premières pour la lecture des températures dans deux stockages, nommés «T_Courante» et «T_Désirée», la dernière pour le calcul de la différence entre ces deux valeurs. Il en va de même pour la tâche «Contrôle Chauffage». C'est une tâche logicielle déclenchée par les tâches d'acquisition et la tâche «Calcul Différence». Elle joue le rôle de contrôleur dans le système: en fonction des entrées reçues, cette tâche génère des sorties appropriés à des tâches de commande afin de contrôler les activations/ désactivations des appareils du chauffage, ou d'afficher un état anormal lorsqu'il est détecté.

A la troisième étape, les interfaces entre les tâches sont définies. Dans DARTS, les interfaces des tâches sont déterminées selon les règles suivantes:

- une boîte aux lettres si une tâche doit expédier un message à une autre tâche de façon asynchrone (c'est-à-dire que l'émettrice peut poursuivre son exécution après l'envoi d'un message sans attendre une réponse de la réception du message à la réceptrice).
- un Rendez-vous si un message ou un événement est expédié à partir d'une tâche à une autre, mais la première ne peut continuer son exécution jusqu'à ce qu'elle reçoit une réponse de la deuxième. Il s'agit de la communication synchrone entre deux tâches.
- une synchronisation si seulement une notification de l'arrivée d'un événement est requise. Aucune donnée n'est transférée dans ce cas.

- un module de donnée au cas où deux ou plusieurs tâches peuvent accéder simultanément à une donnée de façon complètement asynchrone. Dans ce cas-là la structure de données est encapsulée dans le module, et une interface avec des opérations d'accès à cette structure de données est définie.

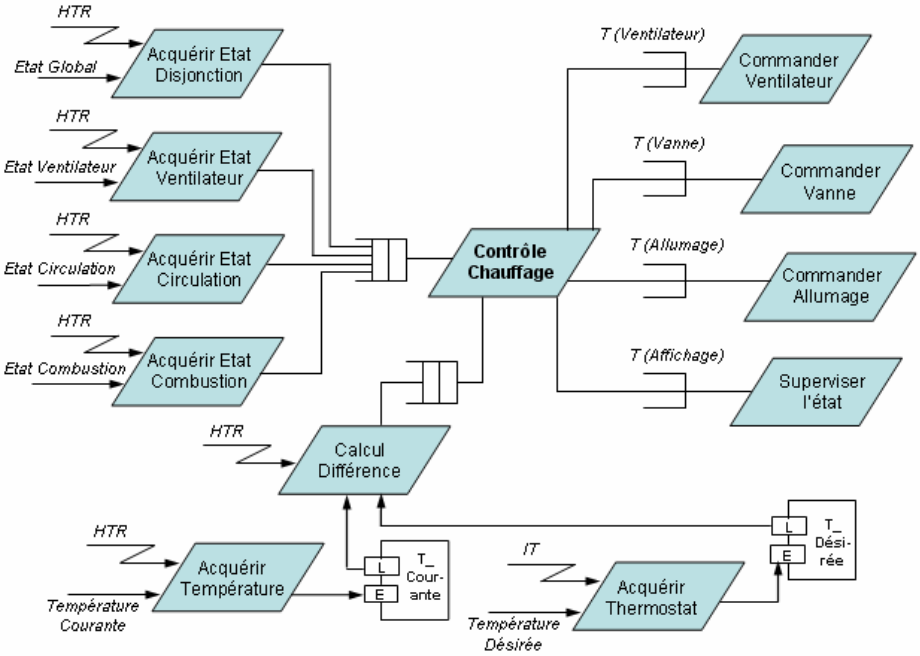


Figure I-51: Diagramme Architecture de tâches pour le système de chauffage

Avec les règles ci-dessus, le diagramme «Architecture de Tâches» pour le système de chauffage d'une maison est représenté sur la Figure I-51. Les boîtes aux lettres font les interfaces entre la tâche «Contrôle Chauffage» et les tâches d'acquisition de données («Acquérir Etat Disjonction», «Acquérir Etat Ventilateur», «Acquérir Etat Circulation», «Acquérir Etat Combustion», et «Calcul Différence»). Puisqu'il n'y a pas de passage d'information entre le contrôleur et les tâches de commande («Commander Ventilateur», «Commander Vanne», «Commander Allumage», et «Superviser l'Etat»), la synchronisation est choisie. Deux modules de données, nommés «T_Courante» et «T_Désirée», sont utilisés pour stocker des valeurs de la température désirée et la température courante dans la maison. Chaque module possède une interface de deux opérations, appelées «Lire» et «Ecrire», ce qui permet à trois tâches («Calcul Différence», «Acquérir Température», et «Acquérir Thermostat») d'accéder à sa propre valeur de température. Notons que «Acquérir Thermostat» est une tâche matérielle activée par une interruption externe provenant de l'utilisateur pour chaque mise à jour de la température désirée.

Remarquons que si la période d'acquisition est la même, on pourrait envisager de regrouper les tâches d'acquisition en une seule. L'un des points importants à souligner est le fait que «Contrôle Chauffage» attend un message en provenance de 2 boîtes aux lettres différentes avant de s'exécuter. Cela pose 2 problèmes : le premier concerne le fait que peu d'exécutifs possèdent des mécanismes d'implémentation d'attente alternative (attente en OU avec exécution si un message ou un autre est reçu). Par conséquent, cette attente sera vraisemblablement implémentée par une attente ET, donc le rythme d'activation de la tâche est le plus lent des rythmes des productrices de messages. Cela peut être très pénalisant pour le système. Le second problème est qu'il est très difficile de dégager une période d'activation de cette tâche, et nous verrons que la plupart des modèles de validation temporelle de tâches temps réel reposent sur des modèles périodiques ou sporadiquement périodiques. Par conséquent, un tel système posera des problèmes de validation temporelle. Il est donc vraisemblable que le concepteur soit amené à remplacer la boîte aux lettres ayant le rythme le plus lent par un module de données.

De plus, le système obtenu utilise un nombre conséquent de tâches, ce qui a pour effet d'utiliser plus de mémoire et causer plus de surcoût processeur du aux nombreux changements de contexte et à la gestion des tâches par l'exécutif. Afin de minimiser le nombre de tâches dans un système, une autre démarche que celle proposée par Gomaa peut être adoptée : il s'agit d'identifier les rythmes différents : dans l'exemple du système de chauffage, on trouvera donc un rythme par période d'acquisition différente, plus un rythme lié à l'interruption matérielle du thermostat. Ensuite les processus fonctionnels dégagés lors de l'analyse SA-RT sont placés dans les tâches correspondant à leur rythme. Toutes les communications entre les tâches sont alors asynchrones. Ensuite, on pourra décomposer les rythmes en chaînes de tâches logicielles communicant par boîtes aux lettres ou synchronisation afin d'augmenter la sûreté de fonctionnement (par exemple, diminution de la diminution du service lors du «plantage» d'une tâche) ou bien encore de la priorité.

Dans la dernière étape, l'étape de l'implémentation des tâches DARTS, Gomaa a proposé dans ses livres et ses articles la méthode SD (Structured Design ou Conception Structurée). Nous savons que SD ne fournit aucune instruction pour structurer un système des tâches concurrentes. Donc, la communication asynchrone dans le diagramme Architecture de tâches DARTS sera implémentée sans concurrence par des appels de fonctions ou des envois des commandes associées à un module à partir d'un autre (réf. I.1.4.2, page 40-41). Il en résulte que les graphes structurels du système de chauffage dans la méthode DARTS sont identiques à ceux présentés dans la méthode SD (voir Figure I-37).

Puisque SA (Structured Analysis ou Analyse Structurée) est une méthode de spécification basée profondément sur l'aspect fonctionnel d'un système, dans la spécification de la méthode SA, les actions dépendent seulement des flots de données d'entrée. Etant une méthode de conception reposant en général sur une spécification préalable SA, la méthode DARTS de Gomaa offre un module, appelé Gestionnaire Etat-Transition (State Transition Manager – STM), comme une alternative. La STM est définie comme un module de donnée qui maintient un diagramme états-transitions (ou une table des états-transitions) invisible aux tâches appelantes. Il contient aussi les procédures d'accès permettant de vérifier la validité des requêtes, et d'effectuer une transition. La STM s'exécute dans la tâche qui invoque ce module. Pour le système de chauffage, le diagramme états-transitions du module «Gestionnaire Etat-Transition» est semblable à celui de la méthode SA-RT présenté sur la Figure I-35. Il est exécuté dans la tâche «Contrôle Chauffage». De manière précise, lorsqu'il y a une entrée sur chacun des flots de données en entrée, la tâche «Contrôle Chauffage» est activée, puis en fonction des entrées reçues, elle fait un appel au «Gestionnaire Etat-Transition» afin de tester si un changement d'état est survenu.

I.5.4.f. Remarques sur la méthode DARTS

La méthode DARTS proposée par Gomaa premièrement dans [Gom 84] est une extension de la méthode SA/SD pour adresser les besoins des systèmes temps réel en fournissant une approche permettant à partir d'une spécification de la méthode SA de structurer un système en tâches concurrentes, et de définir les interfaces entre ces tâches.

Par rapport à la méthode SA/SD, la méthode DARTS grâce au module «Gestionnaire Etat-Transition» souligne l'utilisation des diagrammes état-transition, ce qui est particulièrement important dans le développement des systèmes temps réel.

Puisque la méthode DARTS dans [Gom 84] ne fait aucune distinction entre flots de données et flots de contrôle, le choix de quels flots de données sont actuellement flots de données et lesquels sont utilisés en tant que flots de contrôle est différé jusqu'à la troisième étape, surtout après la définition des interfaces entre les tâches. Les diagrammes DARTS dans [Gom 93] utilisent les spécifications de la méthode SA-RT au lieu de la méthode SA. Cela permet d'une part de distinguer les flots de donnée de ceux de contrôle, d'autre part de se concentrer sur l'aspect comportemental du système aussitôt dans les diagrammes de spécification. Cependant plusieurs

concepteurs trouvent que si la phase de la spécification SA-RT n'est pas bien faite, la phase de la structuration en tâches DARTS peut être plus difficile. La raison en est que la spécification SA-RT n'offre aucune approche pour la décomposition d'un système. Le diagramme préliminaire SA-RT du système de chauffage dans la Figure I-33 (réf. I.1.4.2, page 36) en est un exemple. Si chacun des processus d'acquisition est structuré en tant qu'une tâche séparée selon le critère de dépendance des entrées/sorties de la méthode DARTS, le concepteur peut se trouver en difficulté, car le modèle de ces tâches d'acquisition n'existe pas dans DARTS. Un tel modèle représente des tâches périodiques dont les activations/désactivations sont dirigées par d'autres tâches (notamment par le processus de contrôle qui pourra activer/désactiver une tâche en utilisant l'événement E/D). Pour ce cas-là, Gomaa recommande avec la méthode DARTS de développer les diagrammes état-transition avant les diagrammes de flot de données.

Les mécanismes de synchronisation/communication proposés dans DARTS favorisent considérablement la définition des interfaces entre les tâches en assurant leur concurrence. Ils sont fournis dans l'une de trois façons suivantes: par le système d'exploitation; par l'utilisation d'un langage de programmation convenable qui est capable d'implémenter la communication et la synchronisation multitâche (i.e., Ada, …); ou au moyen d'un module spécifique pour la gestion des messages entre les tâches, qui se substitue les primitives de synchronisation fournies normalement par le système d'exploitation.

Dans le Diagramme Architecture de tâches DARTS pour le système de chauffage présenté sur la Figure I-51, les interfaces entre la tâche «Contrôle Chauffage» et les tâches d'acquisition sont des BALs (Boîte aux lettres bornée, la communication asynchrone des tâches). Car les tâches d'acquisition sont périodiques, la périodicité de ces tâches peut être donc être violée dans le cas où la BAL est pleine (la tâche émettrice se bloque en attente jusqu'à ce qu'un message soit retiré par la tâche réceptrice). Notons que le même problème survient pour les tâches d'acquisition activées par des interruptions externes dont la vitesse de l'activation est plus rapide que celle de la tâche «Contrôle Chauffage». Le fait de rendre la BAL non bornée peut résoudre ce problème. Mais il exige beaucoup de mémoire et de temps pour la gestion des messages dans BAL. De plus, le message reçu par la tâche réceptrice est le plus ancien message dans la file d'attente de la BAL. Et ceci peut entraîner un retard inattendu dans la réponse aux messages plus récents. La boîte aux lettres avec écrasement (BAL_Ecr) sera une solution adéquate pour ce problème. La BAL_Ecr est une extension de la BAL classique avec le mécanisme de l'écrasement. Contrairement à la BAL, la BAL_Ecr permet d'écraser l'ancien message si la queue de messages est pleine à l'instant que la tâche émettrice envoie un message. Par conséquent, la tâche émettrice ne se bloque jamais en écriture, et la tâche réceptrice peut recevoir toujours des messages plus frais. Mais pour la lecture, il en va de même que la BAL, si la BAL_Ecr est vide, la tâche réceptrice doit se bloquer en attente jusqu'à l'arrivée d'un nouveau message. La notation de la boîte aux lettres avec écrasement est donnée sur la Figure I-52 (cette notation n'existe pas dans la méthode DARTS mais est proposée dans [CG05]).

Le diagramme Architecture des tâches pour le système de chauffage avec les BAL_Ecrs est donné sur la Figure I-53.

DARTS ne nous fournit aucuns mécanismes pour tester et/ou vérifier le comportement temporel d'un système sous le développement [Ale 04]. Nous trouvons dans DARTS qu'il y a des critères favorisant la décomposition d'un système en un ensemble de tâches concurrentes, mais qu'il n'existe aucunes instructions pour l'analyse des paramètres temporels de ces tâches (par exemple, la fréquence d'activation, le temps de calcul, le temps de réponse, la configuration du système sous-jacent, etc.).

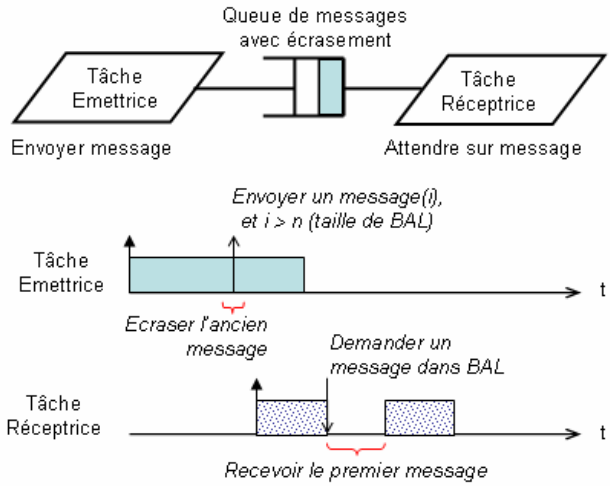


Figure I-52: Communication par BAL bornée avec écrasement

De par sa proximité des outils de programmation classiques en asynchrone (programmation multitâche), DARTS permet d'obtenir un squelette des tâches qui peuvent être exécutées en concurrence, de façon indépendante du langage sous-jacent (c'est-à-dire des langages de programmation, des normes temps réel). Il faut associer DARTS à un ou des outils d'implémentation approprié(s) pour les tests et les analyses des paramètres temporels vers une implémentation explicite dans les étapes suivantes. L'association entre DARTS et LabVIEW, un langage de programmation graphique basé sur le langage G, que je présente au chapitre III (LabVIEW est donc présenté au chapitre II) est un exemple d'une telle association. De même, Gomaa dans son livre «Design Approach for Real-Time Systems» [Gom 93] présente ADARTS (Ada-based Design Approach for Real-Time Systems), une extension de la méthode DARTS tournée vers le langage Ada.

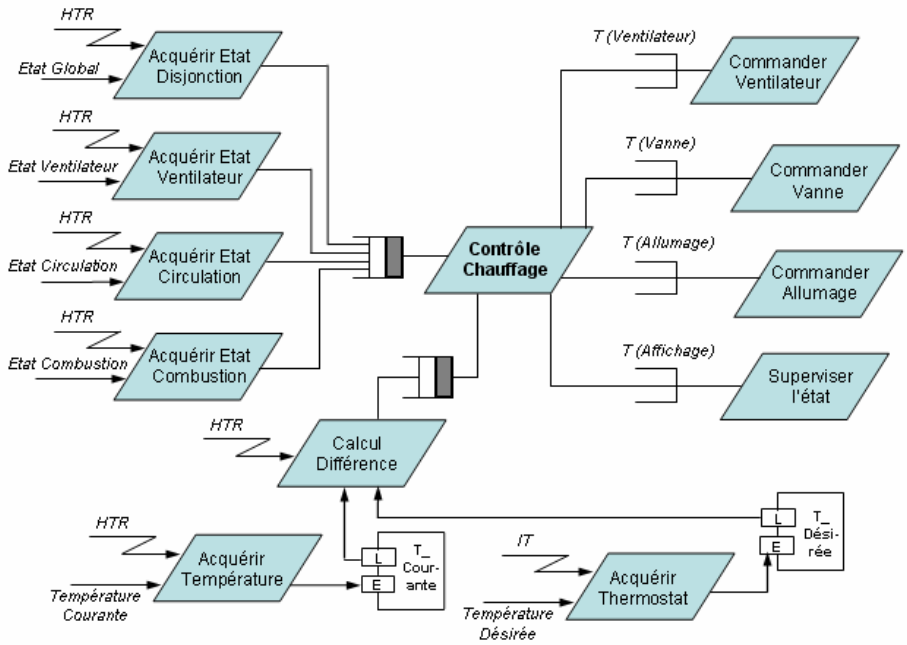


Figure I-53: Diagramme Architecture de Tâches du système de chauffage avec les BAL_Ecrs

Avec ADARTS, Ada devient le langage cible, et l'activité centrale dans cette étape est de développer une architecture de tâches basées sur les tâches Ada. Chaque tâche DARTS correspond à une tâche Ada. Un appel à une «entry» de la tâche appelée va causer une attente de la tâche appelante sur cet appel jusqu'à ce que la tâche appelée termine son exécution (c'est-à-dire

la communication synchrone entre des tâches). Pour adresser la communication asynchrone et/ou la synchronisation des accès à une donnée entre des tâches, il faut définir une bibliothèque pour tous les outils de communication/synchronisation DARTS. Grâce aux objets protégés, cette bibliothèque est aisément construite. Ainsi, la BAL classique est implémentée par un objet protégé qui possède une queue de message et deux «entries» (une pour l'expédition de message, nommée «Envoyer», l'autre pour la réception de message, nommée «Recevoir»). «Envoyer» est appelée par la tâche «Emettrice» afin d'envoyer un message à la queue (les paramètres du message dans ce cas deviennent l'input de cet «entry»). «Recevoir» est appelée par la tâche «Réceptrice» afin de recevoir un message à partir de la queue (les paramètres du message deviennent la sortie de cette «entry»). Les gardes peuvent être utilisées pour empêcher d'une part l'émettrice d'envoyer un message quand la queue est pleine, et d'autre la réceptrice de recevoir un message quand la queue est vide.

Il en va de même pour la communication asynchrone par le module d'encapsulation de donnée (MDD), la synchronisation par sémaphore, la BAL avec écrasement. Enfin, afin de rendre plus réutilisables les composantes Ada, il faut regrouper les tâches en paquetages. Par conséquent,, pour le système de chauffage, toutes les tâches peuvent être groupées dans le paquetage, appelé «Modèle». La périodicité des tâches périodiques est gérée grâce au paquetage prédéfini «Ada.Real_Time». Pour de plus amples informations sur la méthode ADARTS et le langage Ada, consultez à [Gom 93] [Bur 98].

Une implémentation de la BAL à une place sans écrasement en Ada est représentée ci-après:

```
generic
  type T is private;
package BAL is
  protected type Bal is
    entry Envoyer (Msg_In: in T);
    entry Recevoir (Msg_Out: out T);
  private
    Msg: T;
    Pleine: Boolean := False;
  end Bal;
end BAL;
```

```
package body BAL is
  protected body Bal is
    entry Envoyer (Msg_In: in T) when (not
Pleine) is
    begin
      Msg:= Msg_In;
      Pleine:= True;
    end Envoyer;
    entry Recevoir (Msg_Out: out T) when Pleine
is
    begin
      Msg_Out:= Msg;
      Pleine:= False;
    end Recevoir;
  end Bal;
end BAL;
```

I.5.5. UML-RT

UML (Unified Modeling Language), un standard défini par l'OMG (Object Management Group), est un langage graphique de modélisation des données et des traitements. C'est une formalisation très aboutie et non-propriétaire de la formalisation objet utilisée en génie logiciel. Il est l'accomplissement de la fusion des précédents langages de modélisation objet Booch, OMT (Object Modeling Technique), et OOSE (Object Oriented Software Engineering). La version la plus récente de l'UML est UML 2.1 recommandée par l'OMG en 2007.

Le formalisme UML est composé de 13 types de diagrammes (voir Figure I-54), dont les six premiers sont les diagrammes statiques qui décrivent le comportement statique du système, tandis que les trois suivants sont utilisés pour exprimer le comportement dynamique, et les autres sont les diagrammes d'interactions:

- Diagramme de classes (Class Diagram): il représente les classes intervenant dans le système considéré.

- Diagramme d'objets (Object Diagram): il sert à représenter les instances de classes (donc, chaque objet correspond à une instance d'une classe) utilisées dans le système.
- Diagrammes de composants (Component Diagram): il permet de montrer les composants du système d'un point de vue physique, tels qu'ils sont mis en œuvre.
- Diagramme de déploiement (Deployment Diagram): il sert à représenter les éléments matériels (ordinateurs, périphériques, réseaux, systèmes de stockage…), et la manière dont les composants du système sont répartis sur ces éléments matériels et interagissent avec eux.
- Diagramme des paquetages (Package Diagram): un paquetage étant un conteneur logique permettant de regrouper et d'organiser les éléments dans le modèle UML. Le diagramme de paquetage sert à représenter les dépendances entre paquetages, c'est-à-dire les dépendances entre ensembles de définitions.
- Diagramme de structure composite (Composite Structure Diagram): permet de décrire sous forme de boîte blanche les relations entre composants d'une classe.

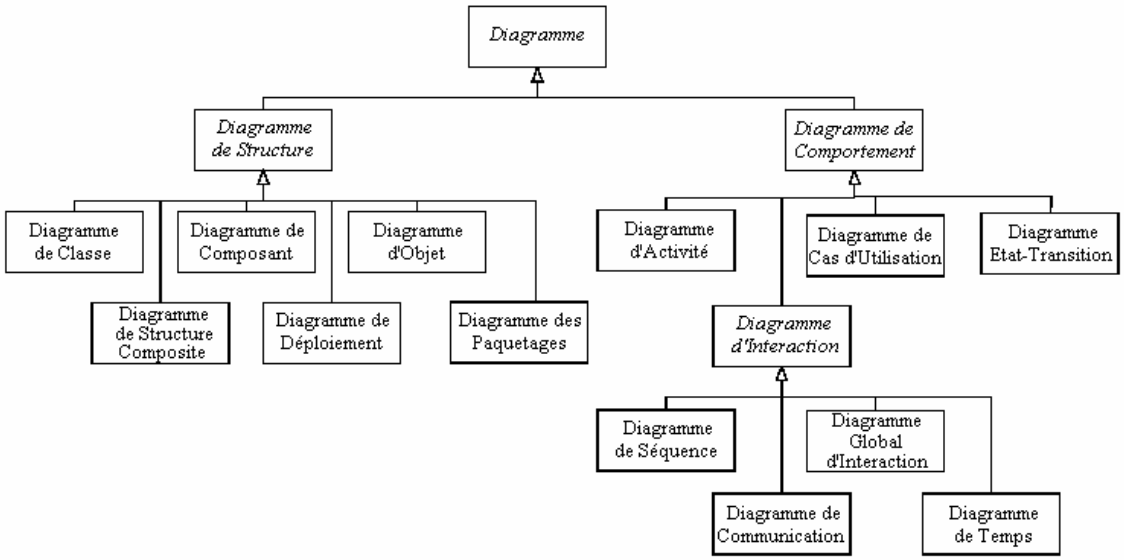


Figure I-54: UML, Hiérarchie des diagrammes

- Diagramme des cas d'utilisation (Use-Cases Diagram): il permet d'identifier les possibilités d'interaction entre le système et les acteurs (intervenants extérieurs au système), c'est-à-dire toutes les fonctionnalités que doit fournir le système.
- Diagramme Etats-Transitions (State Machine Diagram): permet de décrire sous forme de machine à états finis le comportement du système ou de ses composants.
- Diagramme d'activité (Activity Diagram): permet de décrire sous forme de flux ou d'enchaînement d'activités le comportement du système ou de ses composants.
- Diagramme de séquence (Sequence Diagram): représentation séquentielle du déroulement des traitements et des interactions entre les éléments du système et/ou de ses acteurs.
- Diagramme de communication (Communication Diagram): représentation simplifiée d'un diagramme de séquence se concentrant sur les échanges de messages entre les objets.
- Diagramme global d'interaction (Interaction Overview Diagram): permet de décrire les enchaînements possibles entre les scénarios préalablement identifiés sous forme de diagrammes de séquences.
- Diagramme de temps (Timing Diagram): permet de décrire les variations d'une donnée au cours du temps.

Les modèles d'éléments UML sont représentés sur la Figure I-55. La démarche UML est la suivante: le système est d'abord décomposé en un ensemble d'objets, et les relations entre objets sont exprimées simplement par les diagrammes d'objets (notons que les objets d'UML sont hiérarchiques, c'est-à-dire que chacun peut être décomposé en un ensemble de sous-objets jusqu'à ce que les sous-objets atteignent un niveau de détail suffisant). Puis à partir de ces diagrammes, les cas d'utilisation (use-cases) sont définis, chaque cas d'utilisation correspond à une fonctionnalité concrète que le système doit fournir. En conséquence, les diagrammes de séquence, diagrammes de communication sont utilisés pour décrire l'aspect dynamique, ainsi que les diagrammes états-transitions utilisés pour chaque objet actif. Les attributs et les opérations sont également bien définis. Enfin, la notion de classes est considérée sous forme de diagramme de classes et des différents types de relation tels la dépendance, la généralisation, l'association, l'agrégation, la réalisation, et l'utilisation.

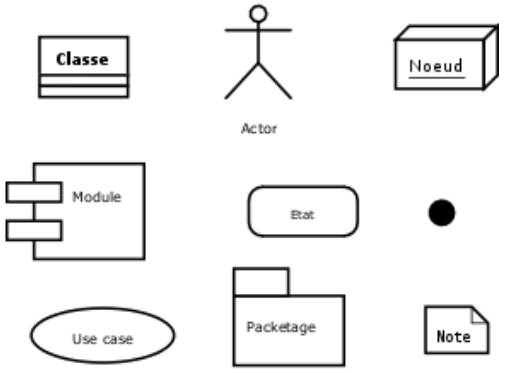


Figure I-55: Les modèles d'éléments dans les diagrammes UML

Le diagramme de classes est considéré comme l'élément central d'UML. Grâce à la notion de classes, UML est une méthode orientée-objet qui intègre de fait l'«encapsulation» et la «réutilisation».

De nos jours, UML est appliqué dans nombreux domaines, surtout dans les domaines de gestion de données, dans les systèmes à grande échelle, et dans les systèmes interactifs. Pour les systèmes réactifs ou les systèmes embarqués temps réel, l'application d'UML est encore nouvelle. Avant la version 2.0, les anciennes versions d'UML ne possèdaient pas suffisamment de sémantiques pour décrire le comportement temporel du système (i.e., des contraintes temporelles de chaque objet, des contraintes temporelles entres objets…). Dans ces versions, le concepteur ne trouvait que la gestion de temps décrite sous forme de l'attente d'un certain délai, le diagramme de séquence utilisé pour représenter les flots d'entrées/sorties, la communication entre objets exprimée sous forme de messages synchrones et/ou asynchrones, et représentée par les canaux. Il ne trouvait pas à partir de ces concepts les sémantiques suffisantes pour décrire des événements de temps (par exemple, horloge temps réel, interruptions…) pour activer/désactiver les objets actifs, la synchronisation/ communication entre objets, notamment les objets concurrents dans le système considéré.

A partir de la version 1.4 d'UML, l'OMG et plusieurs chercheurs tentent de remédier à cette lacune par:

I.5.5.a. Les extensions UML vers les stéréotypes, nommés «capsule», «port», «protocol»

Avec cette approche, la spécification complète de la structure d'un système temps réel est obtenue sous forme d'une combinaison de diagramme de classes et diagramme de collaboration avec les trois constructeurs principaux suivants:

- *Capsules:* correspondent aux concepts des acteurs ROOM (Real-Time Object-Oriented Modeling). Ce sont des objets complexes, physiques, et peut-être architecturaux distribués,

qui interagissent avec leurs environnements par un ou plusieurs canal(aux) de communication basé(s) sur des signaux aux frontières de ces objets, appelés «ports» (signal-based boundary objects).

- *Port:* implémente une interface d'une capsule. Chaque port d'une capsule joue un rôle particulier dans la collaboration entre cette capsule et les autres objets. Donc, afin de capturer la sémantique de toutes ces interactions, les ports sont associés à un protocole qui définit les flots d'information valides entre les ports connectés.

- *Connecteurs*: vues abstraites des canaux de communication qui relient deux ou plusieurs ports. Chaque port dans une connexion doit jouer un rôle mutuellement complémentaire, mais compatible décrit dans un protocole.

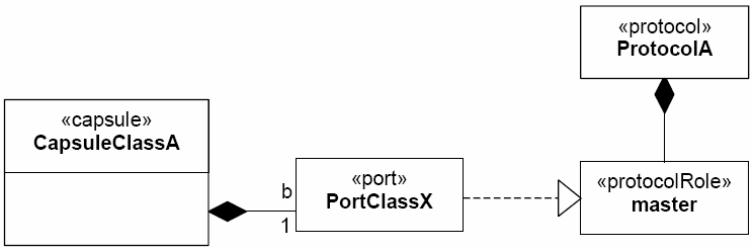


Figure I-56: Capsule – Port – Protocole – Protocole Rôle [SR98]

La Figure I-56 représente un exemple d'un port dont le nom est «b», qui appartient à une classe-capsule, nommée «CapsuleClassA». Ce port réalise un rôle, nommé «master», défini par un classe-protocole, nommée «ProtocoleA».

Cette version est la première version d'UML vers le développement des systèmes temps réel, appelée UML-RT [SR98]. Elle se focalise sur la communication entre objets UML pour décrire le comportement statique d'un système. Pour le comportement dynamique, de même que UML dans la version antérieure, UML-RT propose l'utilisation de Statecharts dans les diagrammes Etats-Transitions, mais avec un stéréotype, nommé «chainState», afin de représenter une connection entre ports continuation spécifique des transitions (voir Figure I-57).

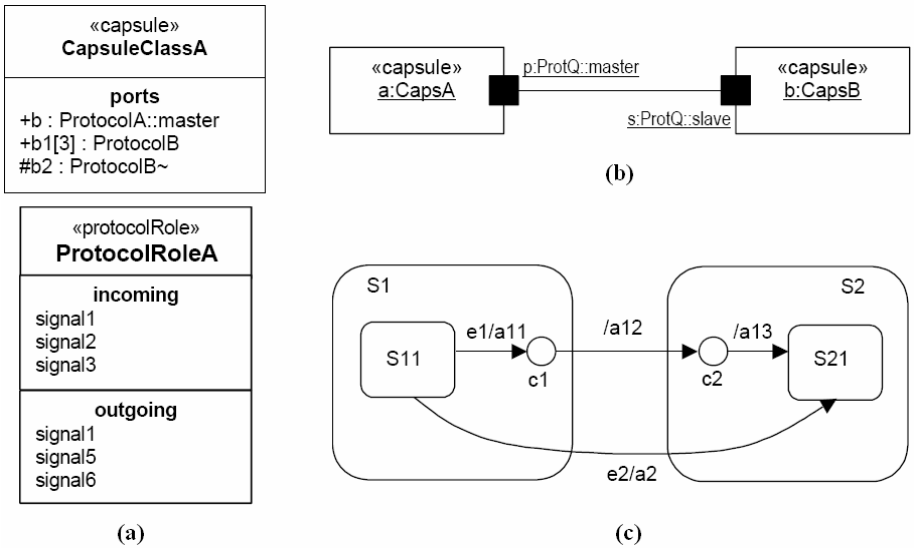


Figure I-57: Notations de Port, de Protocole-Rôle et «chainState»
(a) Notations de Port et Protocole-Rôle dans le diagramme de classes
(b) Notations de Port dans le diagramme de collaboration
(c) chainState

Pour la gestion du temps, il n'y a pas d'extension ou de notation spéciale dans la version UML-RT. Par conséquent, les deux formes de temps suivantes sont utilisées: génération de

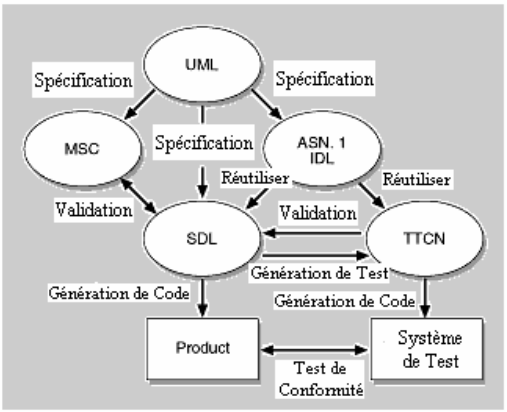
déclenchement à un instant particulier, ou après un certain intervalle à partir d'un point de temps donné.

Afin d'adresser les lacunes de l'aspect temporel d'UML-RT, Küster et Stroop dans [KS01] se focalisent sur le concept de cohérence des diagrammes de séquence et diagrammes de Statechart pour donner une définition générale de la cohérence temporelle entre un ensemble de diagrammes de séquence et les statecharts. Une méthode permettant d'assurer cette cohérence temporelle est aussi présentée en réutilisant les techniques d'analyse temps réel existantes.

Bianco et al. dans [BLM02] proposent une formalisation de l'extension adéquate des UML statecharts avec un ensemble des notations formelles [LQV01] et leurs sémantiques formelles (sous forme de «Timed Statecharts» [KP92]) pour la modélisation des systèmes temps réel.

I.5.5.b. SDL combiné avec UML

Le «Specification and Description Language (SDL)» ou Langage de Description et de Spécification est un standard de l'Union Internationale des Télécommunications (ITU-T, International Telecommunications Union – Telecommunications Standardization Sector) sous la référence Z.100 [Z100]. SDL a pour objectif de décrire de manière non-ambiguë la spécification des applications complexes, dirigées par événements, temps réel, et interactives, comportant plusieurs activités concurrentes qui communiquent sous forme de signaux discrets, et qui se basent sur un modèle EFSM (Extended Finite State Machine). SDL est donc un langage de spécification graphique qui est à la fois formel et orienté-objet. (Le lecteur peut consulter [Ree01] pour un survol de développement de SDL dans ITU-T jusqu'à la fin de l'année 1999 avec le standard Z.109 [Z109] qui définit un profil UML à SDL).



MSC (Mobile Switching Center)
ASN.1 (Abstract Syntax Notation One)
IDL (Interface Description Language)
TTCN (Tree and Tabular Combined Notion)

Figure I-58: Relation entre SDL et les autres langages [Z100]

UML est une collection de concepts et de notations pour la description de plusieurs vues d'un même système (i.e., vue d'objet, vue d'état-transition, vue de cas d'utilisation, vue de collaboration et d'interaction, etc.). Par conséquent, UML possède des sémantiques faibles avec de nombreux points de variations (par exemple, sous forme de «stéréotype»). SDL est un langage (avec des concepts, une grammaire abstraite et des grammaires graphiques/textuelles) se focalisant sur les vues d'Objet et d'Etat-Transition d'un système. Pour ces vues, SDL offre non seulement les sémantiques statiques et dynamiques, mais également la syntaxe concrète (sous forme de graphique/ texte) pour la spécification précise des actions. Le lecteur peut consulter [Ped93] pour de plus amples informations sur les notations et la syntaxe orientée objet de SDL.

L'idée est donc de combiner SDL avec UML. Et Z.109 regroupe les recommandations de l'ITU-T pour cet objectif. Z.109 définit une spécialisation d'un sous-ensemble d'UML qui a une injection («one-to-one mapping») dans un sous-ensemble de SDL (voir Figure I-59). Effectivement, SDL est mappé à partir des diagrammes d'Objets/Classes et diagrammes Etats-Transitions d'UML afin de décrire en détail la spécification de la structure statique des objets, et la spécification de l'aspect comportement.

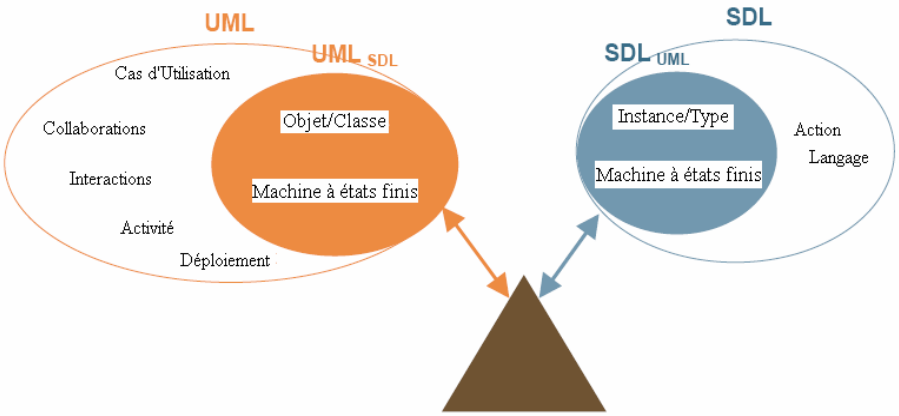


Figure I-59: Modèle d'un outil d'intégration de SDL et UML [Ped00]

De plus, grâce à cette combinaison, l'utilisateur d'UML peut bénéficier des avantages de SDL dans sa relation avec les autres langages (voir Figure I-58) pour la validation du système, notamment la validation vers un outil tiers (par exemple, vers MSC, ou vers un outil basé sur un langage formel tel B, Z, réseaux de Petri…). En fait, ceci est une bonne technique par rapport à l'approche traditionnelle de combinaison d'UML et un langage formel proposée dans plusieurs recherches comme [AR03][LP02][NT04].

I.5.5.c. UML 2.0 et ses apports pour le temps réel

Par rapport aux versions précédentes, UML version 2.0 est divisé en deux parties: l'«*Infrastructure*» [OMG1] qui est conforme aux MOF (Meta-Object Facility), CWM (Common Warehouse Metamodel); et la «*Superstructure*» [OMG2] (i.e., les autres méta-modèles proposés par OMG) qui se fond sur le noyau (Infrastructure) et fournit les fonctionnalités nécessaires pour la construction des modèles (voir Figure I-60). Ceci permet d'identifier d'une part les parts de base du langage qui sont élémentaires pour le développement des modèles, d'autre part les éléments qui sont vraiment utilisés dans les modèles. Néanmoins, UML 2.0 n'est pas d'un langage formel de spécification, mais un modèle informel avec quelques contraintes décrites par l'OCL (Object Constraint Language).

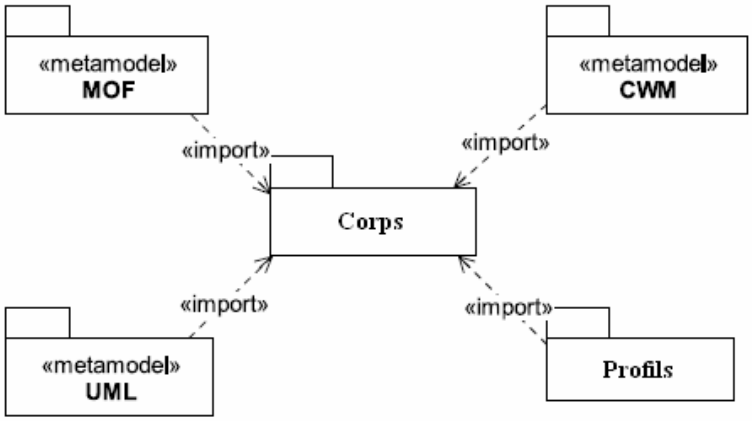


Figure I-60: UML 2.0, Concept principal

L'introduction des profils pour UML, qui autorise l'extension d'UML avec des éléments spécifiques vers un domaine d'application, a déjà été faite dans les versions antérieures d'UML. Cependant, de tels profils ne possèdent pas assez de sémantique, et leur construction et usages sont encore ouverts et ambigus. Dans UML 2.0, le développement et l'application des profils sont améliorés considérablement de la manière suivante: à partir d'un méta-modèle (i.e., UML ou MOF), un méta-modèle dérivé de celui-ci avec de nouvelles informations ajoutées sous forme d'éléments spécifiques, appelés les «stéréotypes», est généré. Un ensemble de ces éléments

dérivés forme un profil (voir Figure I-61 (a)). Donc, si un modèle emploie un profil (voir Figure I-61 (b)), les stéréotypes peuvent être utilisés en tant que les éléments du modèle (voir Figure I-61 (c)). Parmi eux, nous trouvons les deux profils spécifiques aux systèmes temps réel: le «UML Profile for Schedulability, Performance and Time Specification (UML/SPT)» [OMG3], et le «Unified Modeling Language Specification (Action Semantics)» [OMG4]. Le premier comporte des éléments du modèle pour les ressources, les aspects temporels, la concurrence, l'ordonnançabilité, et la performance (par exemple, dispositifs, processeurs, timeout, délais, stimulus, synchronisation, règles d'ordonnancement, et charge). Tandis que le deuxième se concentre sur l'introduction les sémantiques d'actions (i.e., principes d'exécution pour les modèle vis-à-vis des diagrammes de classes et statecharts).

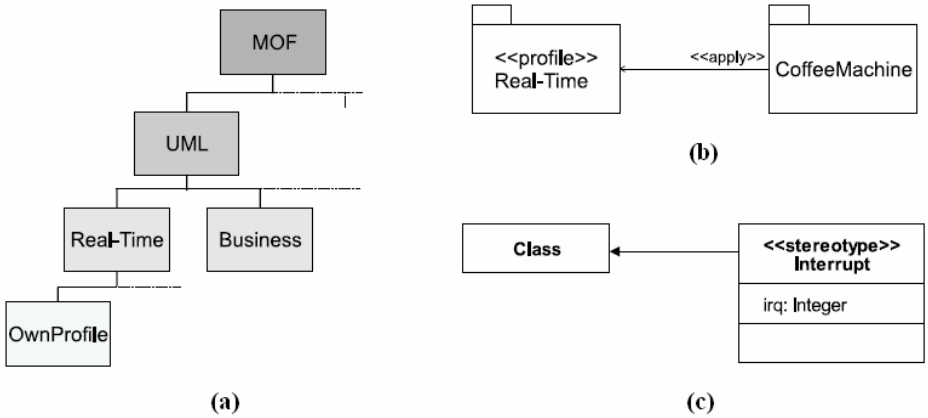


Figure I-61: Profils dans UML 2.0 [Berk03]
(a) UML 2.0, Mécanisme à profiler
(b) UML 2.0, Application de profil
(c) Exemple de l'utilisation d'un stéréotype dans le profil temps réel d'UML 2.0

Pour la description de la structure interne des systèmes, UML 2.0 propose plusieurs nouveaux concepts par rapport à la version 1.4, par exemple la composition hiérarchique des modèles (notons que ce concept existe dans la version UML-RT), les structures de communication, et les composants. Grâce à ces concepts, la structure interne permet de décrire comment un composant est composé, et interagit avec son environnement uniquement au moyen des interfaces (ou très souvent des ports). Contrairement à l'UML-RT, les ports de l'UML 2.0 se servent d'un type de contrat entre les éléments qu'ils connectent. Donc, les ports sont généralement de type interface représentée sous forme de «stéréotypes», nommés «interface», dans les diagrammes de classes. Tous les signaux requis sont bien déclarés dans chaque «interface». Il est important de noter que les signaux transmis par les ports servent d'un seul moyen de connexion entre les éléments, ou entre les éléments et leur environnement dans les diagrammes structurels et/ou comportementaux d'UML 2.0.

Pour la modélisation comportementale, le modèle d'actions d'UML 2.0 est à grain plus fin par rapport au profil «Action Semantics» de la version 1.4 [OMG5] avec plus de vingt types d'actions différents au lieu de sept, surtout avec les actions concernant le déclenchement, la lecture/écriture, etc. En effet les actions représentent la forme la plus petite de l'aspect comportemental dans UML, afin de les traiter et les coordonner, les activités sont introduites. Ce concept n'est pas nouveau, mais les graphes d'activités sont capables de se lier avec les réseaux de Petri à la place d'être une forme spécifique de machine à états finis. En fait dans les diagrammes d'interaction, les interactions peuvent être groupées dans les fragments combinés, qui permettent de modéliser plusieurs possibilités telles les alternatives, les choix, les coupures («breaks»), les boucles, ou les régions critiques (voir Figure I-62 (a)). De plus, UML 2.0 propose un modèle simple pour représenter, spécifier, et observer le déclenchement de temps, les durées, et les points instantanés, etc. d'une manière plus concrète et élémentaire (voir Figure I-62 (b)).

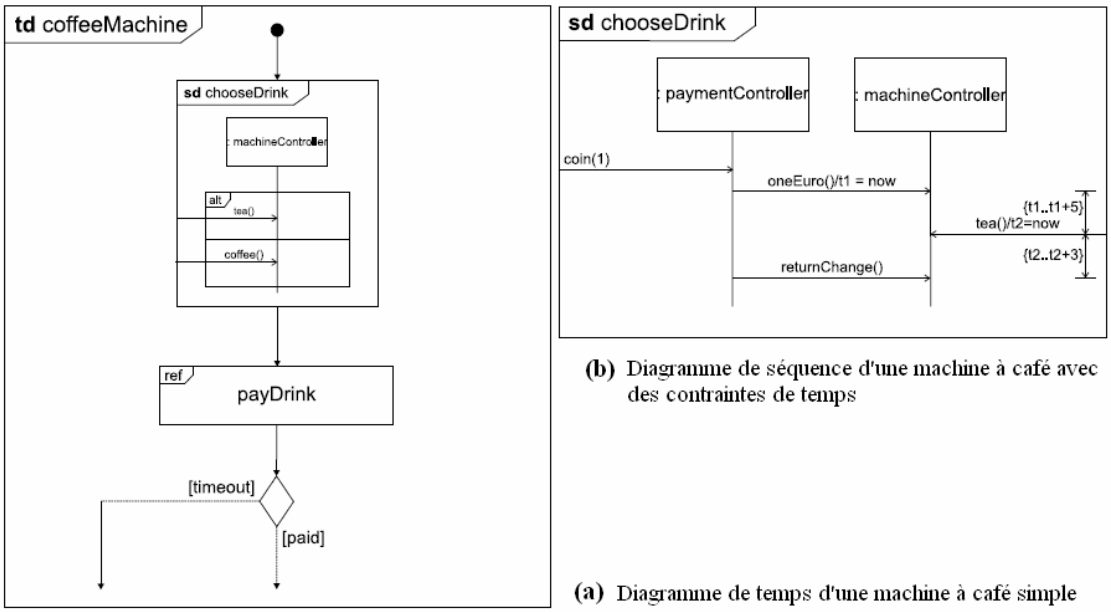


Figure I-62: Modélisation comportementale dans UML 2.0 [Berk03]

Grâce au profil UML/SPT pour le temps réel adopté par OMG en 2002 [OMG3], l'intérêt de l'utilisation des techniques orientées objet, notamment l'UML dans le développement des systèmes temps réel est accru considérablement. L'UML/SPT est actuellement un cadre de modélisation pour les concepts de qualité de service, de ressource, de temps, et de concurrence. Il fournit à l'utilisateur (ou concepteur) non seulement une analyse quantitative prédictive (c'est-à-dire une analyse prédictive de l'ordonnançabilité et de la performance) des modèles UML d'un système considéré, mais également un ensemble des stéréotypes et des valeurs marquées («tagged values») afin de mettre des commentaires sur le modèle UML.

La structure de profil UML/SPT est représentée sur la Figure I-63. Le standard UML/SPT est vraiment très vaste, donc le lecteur qui s'intéresse à ce standard, peut consulter [OMG3]. En pratique, jusqu'à nos jours, l'application de standard UML/SPT et le langage UML dans le développement des systèmes temps réel est encore le sujet de nombreuses recherches telles que Valiente et al. dans [VGC05] tente de déterminer l'ordonnançabilité d'un ensemble de tâches partageant une ressource critique par UML 2.0 et ses profils pour temps réel; Saiedian et Raguraman dans [SRa04] utilisent UML et ses profils basés sur une analyse Rate Monotonic; etc.

Les diagrammes d'UML 2.0 du système de chauffage sont représentés sur les figures ci-dessous. Ils seront utilisés dans la suite pour la comparaison des méthodes orientées objet et des méthodes fonctionnelles dans le développement des systèmes temps réel.

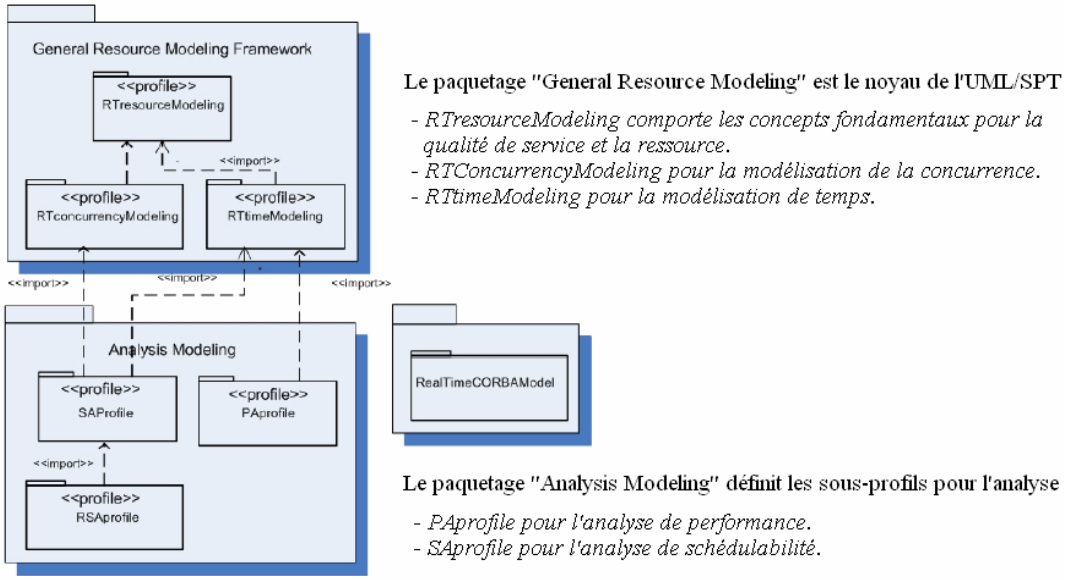


Figure I-63: Structure de profil UML/SPT [GK06]

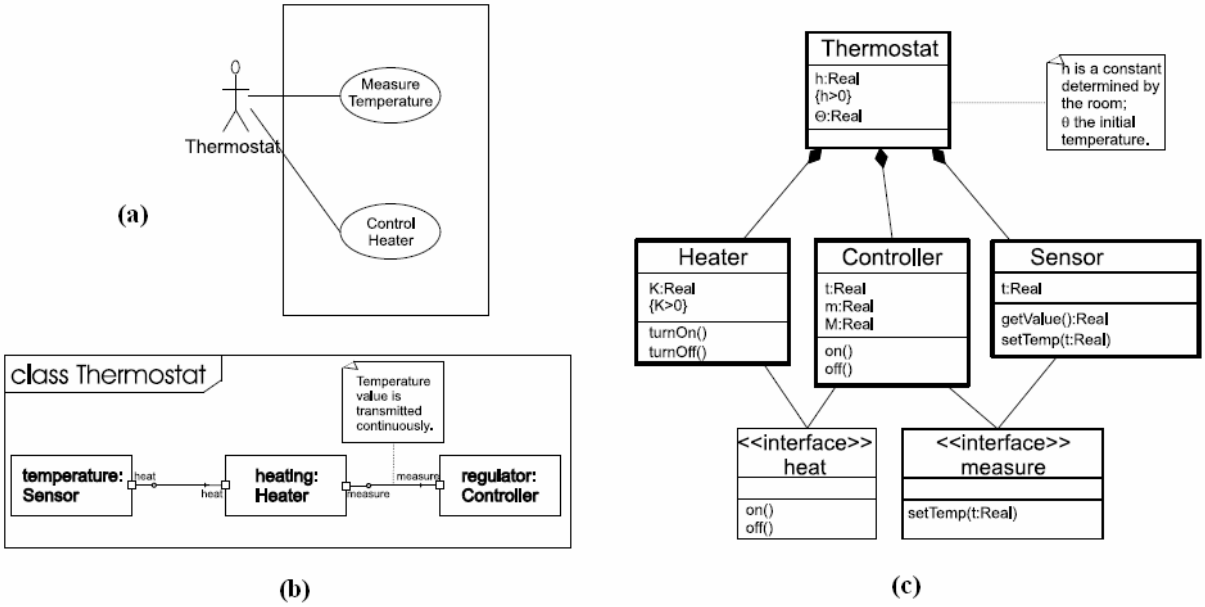


Figure I-64: Digrammes structurels du système de chauffage [Berk03]
(a) Digramme de cas d'utilisation
(b) Diagramme de structure composite
(c) Diagramme de classes

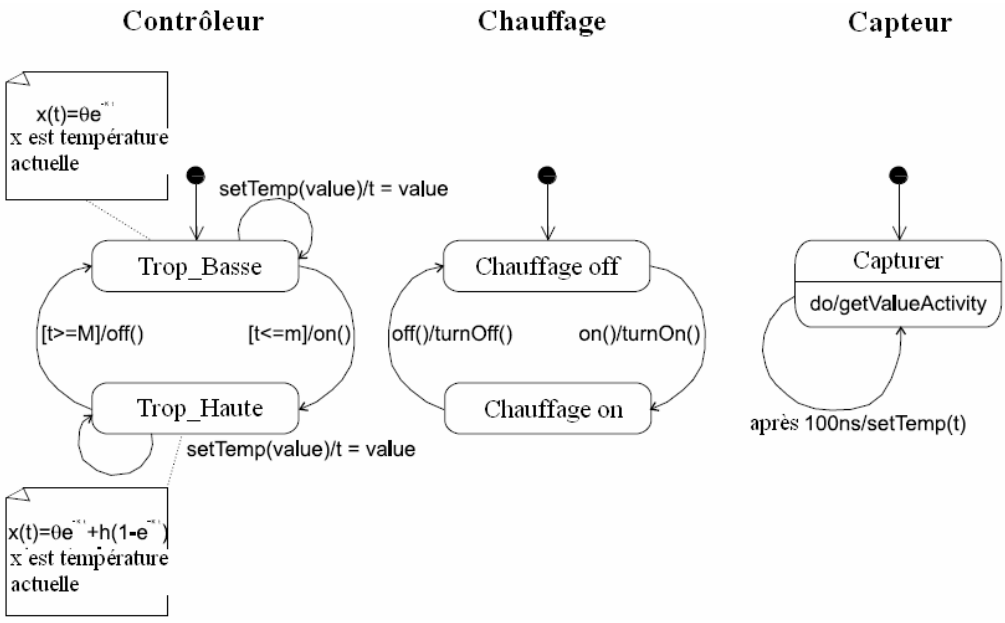


Figure I-65: Diagrammes Etats-Transition du système de chauffage [Berk03]

I.5.5.d. Outils UML commerciaux pour le temps réel

De nos jours, il existe plusieurs outils UML commerciaux basés sur le CASE (Computer Aided Software Engineering) pour le temps réel. Nous pouvons énumérer ici:

- Rhapsody de I-Logix [Ilogix]: inclut la notion de temps associée aux états-transitions, le temps est mesuré lorsque l'état courant est activé. Un système comporte un ensemble des activités-groupes, donc chacun représente un «thread» où dans chaque activité-groupe la concurrence est résolue de manière déterministe. Puis, pour une activité-groupe individuelle, une séquence de code est générée, tandis que les activités-groupes différentes sont soit distribuées, soit ordonnancées par RM.
- ARTISAN Real-Time Studio [Artisan]: étend UML sur la modélisation des systèmes réactifs avec des événements, des contraintes de temps, des tâches concurrentes, et des applications de découpage sur de multiples processeurs.
- Tau Generation de Telelogic [TAU], et ROSE-RT de Rational/IBM [ROSE]: sont basés sur des machines à états finis communiquant par les événements asynchrones. Une notion globale de temps est définie, elle peut être utilisée pour décrire des comportements dépendants du temps en utilisant les temporisateurs, les estampilles temporelles et le temps de garde («guard time», un intervalle de temps laissé inoccupé dans une chaîne de transmission de messages, qui peut être utilisé pour la synchronisation et/ou la compensation des distorsions d'un signal).

I.5.5.e. Remarques sur l'UML-RT

A présent, UML devient un standard très répandu dans le domaine de développement des logiciels. L'objectif d'UML est de créer un environnement orienté-objet pour le développement des systèmes en adressant non seulement la structure et l'architecture (l'aspect statique), mais aussi le comportement (l'aspect dynamique) de ces systèmes. Actuellement, grâce à ses notations, ses diagrammes et ses sémantiques, notamment les deux concepts de l'encapsulation et de la réutilisation, UML permet aux concepteurs de décrire rapidement des systèmes interactifs, en particulier les systèmes à grande taille, de manière plus intuitive et plus claire. Dans le développement des systèmes temps réel ou embarqués, un domaine d'application qui reste encore nouveau pour les premières versions d'UML, plusieurs modifications et extensions sont proposées afin d'adapter mieux UML à ce domaine. Parmi eux, UML 2.0 et le profil pour l'ordonnançabilité, la performance et le temps (UML/SPT) recommandé par l'OMG, comporte

des changements vraiment considérables. En fait, la plupart des faiblesses dans la version 1.4 d'UML sont au moins attaquées avec les différents degrés de succès: la communication entre objets à travers uniquement des ports avec les signaux bien définis par les «stéréotypes», nommés «interfaces», permet de décomposer un système en une structure composite des objets dont chacun peut être décomposé encore en un ensemble des sous-objets sans perdre la cohérence entre les différents niveaux; les modifications de l'UML statecharts offre des possibilités de modéliser l'aspect comportemental de façon plus détaillée avec des objets concurrents associés à plus de vingt types d'action, les nœuds (ou les états) dans les diagrammes d'activité sont soit des nœuds d'objet, soit des nœuds de contrôle, que nous pouvons mettre dans les boucles et/ou les régions afin de raffiner les possibilités de modélisation des activités (il en résulte que les machines à états finis dans UML 2.0 peuvent être considérées sous deux formes différentes: l'une spécifique à la description du comportement intra-objet, l'autre utilisée pour définir les protocoles, un comportement abstrait qui peut être associé aux interfaces ou aux ports). Malgré des améliorations, le modèle de temps fourni dans UML 2.0 est plus simple que celui du profil UML/SPT. Par conséquent, avec le premier modèle de temps, les contraintes temporelles et l'observation des actions correspondantes sont basées principalement sur les expressions de temps définies souvent sous la forme d'une expression d'entier non-négatif. Dans le deuxième modèle, un méta-modèle (qui représente le temps explicite avec des horloges, le temps physique, les valeurs de temps discrètes ou denses…), et les mécanismes de temps («time-related mechanisms») sont bien définis. Donc, grâce au profil UML/SPT, le concepteur peut appliquer ses stéréotypes et ses valeurs marquées associées («associated tagged values») aux éléments d'un modèle UML afin de spécifier les valeurs de temps «RTtime», les mécanismes temporels tels «RTclock», «RTtimer», «RTtimeout»…, ou les contraintes temporelles telles «RTdelay», «RTintervale». De plus, avec le sous-profil «SAprofile», l'analyse quantitative de l'ordonnaçabilité des modèles UML peut être réalisée.

La plupart des recommandations de l'UML 2.0 et les versions précédentes sont définies en langage naturel, il n'existe pas de définition formelle au moins pour le noyau. Tandis que les autres langages de programmation graphique tels MSC (Message Sequence Charts), LSC (Life Sequence Charts), qui correspondent aux diagrammes de séquence d'UML sont basés sur une spécification formelle.

L'interdépendance entre les diagrammes d'UML utilisés pour modéliser un système n'est pas explicitement définie. Cela entraîne des difficultés dans la phase de test et la vérification de la cohérence des spécifications informelles d'UML. En conséquence, les contraintes de temps sont décrites dans un diagramme sans bien préciser la relation de liaison entre des components (classes, objets…), donc seulement la cohérence interne de chaque component peut être analysée (pas le respect de l'aspect fonctionnel).

Pour l'aspect temporel, l'introduction des profils pour la gestion de temps permet à UML d'élargir le domaine des applications vers le développement des systèmes temps réel ou embarqués. Cependant, les sémantiques proposées dans le profil UMP/SPT ne sont pas actuellement complètes. Par conséquent, plusieurs recherches tentent de créer les propres profils UML pour le temps réel, par exemple:

- le profil OMEGA-RT de VERIMAG [GOO06] avec les concepts pertinents des événements représentant un instant de changement d'état, et permettant les expressions des contraintes de durée entre les occurrences des événements (donc, ces contraintes peuvent être décrites sous forme des expressions d'OCL, «Object Constraint Language»),

- le profil TURTLE [ACLS04] avec les sémantiques formelles exprimées au moyen d'un «mapping» vers RT-LOTOS, qui permet une validation temporelle, aussi bien qu'une simulation des modèles UML,

- autre profil UML pour le temps réel basé sur une extension de l'OCL 2.0 [FM02], un méta-modèle pour la spécification des contraintes temporelles dans OCL, donc grâce à ce profil les sémantiques formelles sont données par le «mapping» vers CTL (Computational Tree Logic), qui permet la vérification formelle des modèles UML, etc.

Il est requis que tous les profils UML pour le temps réels en particulier, ou pour les autres domaines d'application en général, doivent être standardisés afin de permettre l'échange de modèle exprimé par les profils UML et/ou les outils UML différents.

De nos jours, UML 2.0 et UML/SPT deviennent le cœur de nombreuses recherches visant à l'application des langages et/ou des méthodes orientés-objet dans le développement des systèmes temps réel. Malgré des succès obtenus dans les autres domaines d'application, UML n'est pas encore un standard complet pour le temps réel. Mais, comme toutes les autres méthodes, il existe toujours à la fois les points forts, et les points faibles. UML est déjà très répandu, grâce aux concepts de classes/objets, UML s'adapte mieux au développement des systèmes à large échelle où l'encapsulation et la réutilisation sont indispensables.

Pour les plus amples informations de l'UML et ses profils pour le temps réel (UML/SPT), consulter www.uml.org

I.5.6. Conclusion sur les méthodes de développement

Nous pouvons classifier les méthodes informelles pour le développement des systèmes temps réel en deux catégories: les méthodes fonctionnelles et les méthodes orientées objet. La première catégorie est représentée typiquement par les méthodes SA-RT, DARTS, Statechart; tandis qu'UML-RT et HOOD-RT sont les deux méthodes typiques pour la deuxième catégorie (Notons que HOOD est une méthode orientée objet qui se concentre sur le mécanisme de l'encapsulation en décomposant un système en des objets, par conséquent dans HOOD, la notion de classes n'existe pas, contrairement à UML. Le lecteur qui s'intéresse à la méthode HOOD-RT, peut consulter [Ros97][BW95][CW96]).

Chacune de ces méthodes, qu'elle soit fonctionnelle ou orientée objet, fournit au concepteur les notations, les sémantiques propres permettant de spécifier / concevoir / tester / valider / implémenter des systèmes temps réel ou embarqués. Cependant, le choix de la méthode ou des méthodes pour le développement d'un système dépend profondément de la nature des méthodes elles-mêmes.

Les méthodes fonctionnelles se focalisent sur le fonctionnement du système. Le système est décomposé en des fonctions, représentées typiquement sous forme des transformations de données, et les relations entre ces fonctions, représentées typiquement par des flots de donnée/contrôle. Puis, c'est le «mapping» des fonctions vers des tâches, donc plusieurs fonctions peuvent être regroupées dans une seule tâche, et les tâches indépendantes peuvent être exécutées concurremment.

Contrairement aux méthodes fonctionnelles, la technique des méthodes orientées objet se concentre principalement sur les notions de classes et d'objets, donc un objet représente un concept, une idée ou toute entité du monde physique tandis que la classe est une structure informatique particulière dans le langage objet, qui décrit la structure interne des données et définit les méthodes s'appliquant aux objets de même classe. Par conséquent, dans la démarche des méthodes orientées objet, il faut décomposer un système en des objets d'abord, et puis au travers des cas d'utilisation les relations entre objets sont déterminées (ainsi que les attributs et les méthodes requises pour chaque objet) où un objet est une structure de données pour répondre à un ensemble des messages (cette structure de données définit son état, tandis que l'ensemble des messages décrit son comportement). Les classes et/ou les méta-classes des objets sont les notions vitales de la modélisation objet. Elles fournissent à la fois les concepts de la modularité et la

structure, qui comporte toutes les caractéristiques spécifiques des méthodes orientées objet: l'encapsulation, la réutilisation, l'abstraction, le polymorphisme…

Nous voyons que les méthodes fonctionnelles permettent au développeur de trouver rapidement un modèle adapté pour une ou plusieurs requêtes sur le fonctionnement d'un système. Il s'agit d'une vue claire de la décomposition d'un système en fonctions et en relations entre ces fonctions, qui permet de faciliter considérablement les tests fonctionnels du système plutôt aux premières phases de développement des logiciels, surtout le développement des systèmes temps réel. Cependant, dans le cas des systèmes distribués ou de grande taille, les sémantiques proposées dans les méthodes fonctionnelles sont insuffisantes. En effet normalement chacun de ces systèmes consiste en un ensemble de sous-systèmes que nous pouvons considérer en tant que système de petit taille interagissant avec les autres sous-systèmes et/ou son environnement afin de répondre à une ou plusieurs requêtes fonctionnelles. Alors dans ce cas-là, avec les méthodes fonctionnelles actuelles, l'architecture locale de chaque sous-système peut être développée aisément, mais pour l'architecture globale du système, il manque les sémantiques adéquates pour exprimer correctement des relations entre sous-systèmes. CODARTS de Gomaa est une tentative pour pallier cette lacune en proposant le concept des «objets concurrents» dans les diagrammes de spécification de COBRA (Concurrent Object-Based Real-time Analysis). Cela permet au concepteur de décomposer un système en des objets avec les sémantiques pour décrire les combinaisons des objets. Mais la nature de CODARTS est une méthode fonctionnelle (c'est-à-dire que CODARTS est basée sur les méthodes fonctionnelles pour exprimer les concepts des objets), donc les sémantiques de la relation entre objets proposées par CODARTS ne sont pas vraiment adaptées, notamment dans la phase de l'implémentation, par rapport à des méthodes orientées-objet (le lecteur qui s'intéresse à la méthode CODARTS, peut consulter [Mil06][MG06]). Par contre, pour le développement des systèmes de petite taille, en particulier des systèmes temps réel ou embarqués, les méthodes fonctionnelles sont bien adaptées. Surtout, les notations et les sémantiques de ces méthodes sont simples et faciles à comprendre et à appliquer (les personnes impliquées sur des projets dans le monde de l'embarqué – informaticiens, électroniciens, électriciens, aérodynamiciens, thermiciens, mécanicien, etc. –n'ont souvent pas une culture informatique les aidant à comprendre l'orienté-objet). Les diagrammes de l'architecture d'un système fondés sur une méthode fonctionnelle peuvent servir directement aux tests de l'aspect fonctionnel (i.e., test unitaire, test d'intégration…), ainsi qu'à la génération des simulations pour le test de comportement du système. Le cycle de développement de logiciel approprié pour les méthodes fonctionnelles est fréquemment le modèle en Cascade, en V, ou en W, mais rarement un modèle en Spirale, en Prototype Rapide, ou un modèle Itératif et Incrémentiel.

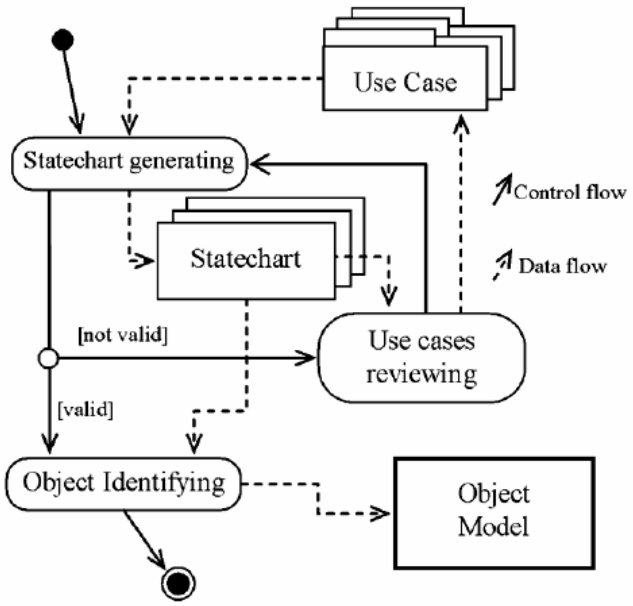


Figure I-66: Schéma de l'identification des objets [KM04]

De l'autre côté, avec les méthodes orientées-objet, le système considéré est décomposé en objets (ou classes d'objets). Cela facilite la réutilisation et les modifications de ces objets pour développer un autre système ou pour satisfaire une autre exigence à part de l'utilisateur. Par conséquent, cette technique s'adapte bien aux développements des systèmes à grande taille, notamment en s'associant avec un cycle de développement en Spirale, en Prototype Rapide, ou un modèle Itératif et Incrémentiel. Cependant si le système développé est un nouveau système (il n'existe pas d'autre similaire), trouver une réponse faisable pour la question "Quels sont les objets nécessaires pour ce système aux premières étapes de développement?" n'est pas du tout une tâche facile. Effectivement, il exige des travaux en plusieurs itérations, et dans certains cas des expériences du concepteur a priori. Comme le fonctionnement du système est déterminé sous forme de cas d'utilisation («use-cases»), les tests fonctionnels ne peuvent être effectués qu'une fois que le modèle des objets du système est bien défini. Kimour et Meslati dans [KM04] proposent une technique de génération d'un modèle des objets pour les systèmes temps réel ou embarqués avec UML de façon inverse: c'est-à-dire, à partir des cas d'utilisation, les diagrammes de statecharts sont utilisés pour adresser les exigences fonctionnelles du système, et puis à partir des statecharts un modèle des objets est construit. Cela permet de vérifier rapidement l'aspect fonctionnel du système par UML ou des méthodes orientées objet, mais pour que cette technique soit utilisable, il faut modifier les sémantiques de statecharts (car les communications dans les diagrammes de statecharts sont synchrones, alors qu'elles seront vraisemblablement asynchrones à l'implémentation). De plus, cette technique focalise sur les modèles des objets et la communication entre objets, et pas sur les modèles des classes et l'association entre classes.

En conclusion, les deux catégories des méthodes de développement de logiciels présentent des avantages et des inconvénients. Donc le choix est dépendant de la nature de chaque catégorie. Pour les systèmes qui peuvent être implémentés chacun sur un exécutif temps réel ou un système centralisé, les méthodes fonctionnelles sont plus simple à prendre en main que les méthodes orientées objet. Au contraire pour les systèmes distribués les méthodes orientées objet s'adaptent mieux que les méthodes fonctionnelles. Et il est important de remarquer qu'il n'existe pas de «méthode idéale». Il faut les utiliser conjointement avec un(e) ou des autres outils et/ou méthodes pour compléter les sémantiques des méthodes elles-mêmes, ainsi que pour satisfaire des exigences pour le développement du système considéré (par exemple, la gestion de temps, les contraintes de temps, la validation temporelle, la génération de code pour l'implémentation, etc.).

Dans la suite de ce manuscrit, les méthodes fonctionnelles sont choisies pour le développement des systèmes temps réel.

I.6. Validation temporelle des systèmes temps réel

La validation temporelle des systèmes temps réel se base principalement sur la théorie de l'ordonnancement. Il convient donc, lorsqu'on choisit une méthode de conception, d'en considérer une qui sera compatible avec les modèles connus d'ordonnancement. A défaut, il faudra adapter les modèles connus au code spécifique qui sera généré.
Cette partie a pour but d'introduire quelques modèles classiques de la théorie de l'ordonnancement temps réel, et des éléments concernant leur validation. Je parlerai des concepts, et non des formules pour lesquelles je renvoie le lecteur aux références citées.

I.6.1. Modèles d'exécution

Le rôle essentiel du système informatique est de gérer l'enchaînement et la concurrence des tâches en optimisant l'occupation du processeur, cette fonction est appelée l'ordonnancement. L'ordonnancement va déterminer les caractéristiques temporelles et être le garant du respect des contraintes de temps imposées à l'exécution de l'application. L'ordonnancement dans les systèmes temps réel désigne le module du noyau temps réel, qui choisit les tâches qui vont être exécutées par le(s) processeur(s) parmi l'ensemble des tâches activables. L'ordonnancement peut être divisé en deux parties :

- *L'*ordonanceur *(scheduler)*: qui est chargé d'appliquer l'algorithme d'ordonnancement en gérant les différentes files ou tables des tâches.
- *Le répartiteur (dispatcher)*: qui réalise l'élection effective de la tâche et le changement de contexte associé.

Il existe deux modèles d'exécution des systèmes de contrôle-commande [CDKM00]: l'exécution synchrone et l'exécution asynchrone.

- Le modèle d'exécution synchrone: la perception de l'occurrence de tout événement par le système est différée du temps d'exécution de la tâche en cours. Autrement dit, les tâches sont considérées non interruptibles ou non préemptibles dans ce modèle. Par conséquent, la capacité du système à appréhender un événement externe est caractérisée par la durée de la tâche la plus longue. Le modèle d'exécution synchrone est associé typiquement aux langages réactifs, lorsque le temps de traitement informatique est très court (peut être négligeable) par rapport à l'occurrence des événements provenant du procédé (on parle de l'hypothèse de la fenêtre de visibilité des événements ou d'instantanéité entre action et réaction).
- Le modèle d'exécution asynchrone: la perception de l'occurrence de tout événement est immédiatement prise en compte par le système pour tenir compte de l'urgence ou de l'importance. De tels systèmes sont multitâches parce que différents rythmes de traitement sont nécessaires.

Nous nous plaçons naturellement dans une approche asynchrone, puisque c'est l'approche multitâche qui est visée par la méthode DARTS, sujet principal de cette thèse.

Dans un système multitâche, des contraintes temporelles, généralement de « bout-en-bout » (la réaction à l'événement qui peut être par exemple la commande d'un actionneur, doit avoir lieu moins de 100 ms après qu'on ait obtenu telle donnée sur un capteur) qui se traduit sur chaque tâche d'une chaîne de traitement par l'existence d'une échéance.

La problématique principale de l'ordonnancement est la vérification du respect des échéances des tâches au cours de toute la vie de l'application. Il y a deux grandes familles d'ordonnancement temps réel : les ordonnancement construisant au préalable (on parle de « hors-ligne ») une séquence à exécuter infiniment pendant la vie du système (on parle d'ordonnancement hors-ligne ou dirigé par le temps) ; et les ordonnancements utilisant un ordonnanceur « en-ligne » qui exécute un algorithme d'ordonnancement. Dans la plupart des cas, cet ordonnanceur se base sur les priorités des tâches (qui peuvent être statiques ou variables) afin de choisir la tâche la plus

prioritaire parmi les tâches prêtes afin qu'elle s'exécute. On parle d'ordonnancement basé sur les priorités ou ordonnancement en-ligne.

Le choix entre l'approche «hors-ligne» et l'approche «en ligne» n'est pas évident. On peut penser que les approches «en-ligne» dynamiques sont plus flexibles que les approches «hors-ligne» (forcément statiques) et permettent notamment une réponse plus rapide aux événements extérieurs (en hors-ligne, seule la scrutation est possible, alors qu'en en-ligne, il est possible de déclencher des tâches sur événement extérieur).

L'approche en-ligne s'adapte aussi bien aux systèmes composés uniquement de tâches strictement périodiques (seul modèle applicable aux approches hors-ligne) qu'aux systèmes mixant tâches strictement périodiques et tâches déclenchées par événement. Au niveau ordonnancement, ont dit que des tâches sont concrètes lorsqu'elles sont parfaitement connues (et notamment les dates d'activation ou de réveil). C'est le cas notamment pour les tâches périodiques au sens de la méthode DARTS ou bien les tâches héritant d'une période car déclenchées par une tâche elle-même périodique. Lorsqu'on ne connaît pas a priori les dates d'activation des tâches, par exemple tâches déclenchées par interruption, on parle de tâches non concrètes.

Dans la suite de ce manuscrit, je m'intéresse plus particulièrement aux systèmes asynchrones composés des tâches préemptibles avec un ordonnancement «en-ligne» avec des tâches concrètes ou non concrètes.

I.6.2. Modélisation des tâches

La plupart du temps, les tâches de base d'un système temps réel ont un comportement périodique. En effet, soit elles sont périodiques, soit elles sont activées par un événement extérieur qui est lui-même périodique ou bien possède une durée minimale entre 2 occurrences successives. Leur pire impact sur le système apparaît alors lorsque les occurrences d'événements sont les plus fréquentes, à savoir périodiques avec comme période la plus petite durée séparant deux occurrences successives de l'événement déclencheur.

Les contraintes structurelles apparaissent bien sur un diagramme DARTS : un accès à un module de données correspond à l'utilisation d'une ressource critique (exclusion mutuelle qui est prise en compte dans certains modèles de tâches), une synchronisation ou une boîte aux lettres implique une contrainte de précédence entre les tâches. Une tâche ne communiquant pas avec d'autres tâches (ni par module de données, ni par boîte aux lettres, ni par synchronisation) est appelée tâche indépendante.

I.6.2.a. Modélisation des tâches périodiques indépendantes

Les tâches périodiques se décomposent en deux catégories : les tâches concrètes et les tâches non concrètes.

Les tâches concrètes correspondent à une implémentation dirigée par le temps : dans celle-ci, les facteurs sporadiques (généralement les interruptions matérielles) sont traités au niveau système et permettent de mettre à jour un modèle interne de représentation du monde extérieur (typiquement, ce sont des variables globales qui représentent l'état des capteurs). Les tâches scrutent alors le modèle de façon strictement périodique, qu'il y ait eu ou non changement entre temps. Bien entendu la réactivité d'un tel système est assez basse car un événement extérieur ne sera pris en compte qu'à la prochaine scrutation de la tâche qui y est sensible. Dans ce cas, les dates d'activation des tâches périodiques sont toutes connues à l'avance. D'un point de vue DARTS, ce système correspond au cas où toutes les tâches matérielles sont déclenchées par l'horloge temps réel (HTR).

Les tâches non concrètes correspondent à une programmation dirigée par les événements. Dans ce cas, les tâches pourront être directement activées par un événement extérieur dont on ne connaît pas a priori les dates d'arrivée (mais on connaît le plus petit délai possible entre deux occurrences

successives du même événement, ainsi considéré comme la période pire cas de la tâche). On parle de tâches non concrètes car on ne connaît ni leur première date d'activation, ni les dates successives, bien qu'on puisse définir un pire cas. Ces tâches correspondent aux tâches matérielles déclenchées par interruption matérielle en DARTS (IT).

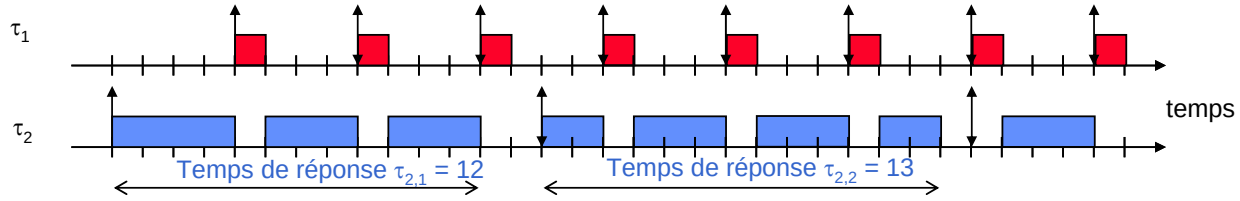


Figure I-67: Chronogramme d'exécution de deux tâches concrètes
τ_1=<r_1=4,C_i=1,T_1=4>, τ_2=<r_2=0,C_2=10,T_2=14>

Notations :

Une tâche est dénotée τ_i, un système de tâches S={τ_1, …,τ_n}.

Dans le cas des tâches concrètes, on définira une première date d'activation (on parle de date de réveil) :

- **r_i:** date de réveil de la tâche τ_i (la première date à laquelle la tâche est activée),

Bien sûr ce paramètre n'est connu à l'avance que pour les tâches concrètes. Dans le cas non concret, on pourra cependant faire une simulation pour des dates d'arrivée particulières en fixant les r_i.

Trois paramètres temporels se retrouvent dans le cas des tâches concrètes et des tâches non concrètes :

- **C_i:** pire durée d'exécution de la tâche τ_i, correspondant au temps processeur requis pour exécuter une occurrence de la tâche (il faut savoir qu'elle est souvent largement surévaluée par rapport à la durée moyenne [WEE07], notamment lorsqu'elle est obtenue de façon statique),
- **D_i:** délai critique ou échéance relative de la tâche (le délai au bout duquel la tâche doit être terminée par rapport à la date de réveil de l'instance en cours), représente sa contrainte de temps,
- **T_i:** période d'activation (la fréquence de renouvellement de la demande d'exécution de la tâche). La tâche est activée toutes les T_i unités de temps à partir de sa date de réveil dans le cas des tâches concrètes. Ou au moins T_i unités de temps après l'activation précédente pour les tâches non concrètes.

Si le délai critique est égal à la période de la tâche (D = T), on parle de tâche à «échéance sur requête» (c'est le cas sur la Figure I-67), puisque sa contrainte de temps est de se terminer avant sa prochaine requête d'activation.

Ces paramètres statiques permettent de calculer les paramètres dynamiques (liés aux instances successives d'une tâche). Ainsi, par exemple, pour une tâche concrète, la date de réveil r_k et l'échéance d_k de la $k^{ième}$ instance sont données par :

$$r_{i,k} = r_{i,0} + kT_i, \text{ avec } r_{i,0}=r_i$$

$$d_{i,k} = r_{i,k} + D_i$$

Un système de tâche S est défini par un ensemble de tâches $\tau_{i,\ i=1..n}$ chacune définie par des paramètres temporels. Ainsi, par exemple un système de tâches non concrètes peut être donné sous la forme $S=\{\tau_i <C_i,D_i,T_i>\}_{i=1..n}$. Ce modèle de tâches est souvent attribué à [LL73] bien que dans le modèle proposé initialement par ces auteurs, les tâches soient à échéance sur requête.

Dans la plupart des applications, ce modèle de tâches périodiques concrètes ou non concrètes est suffisant, cependant, dans certains cas, on ne connaît de délai minimal séparant 2 événements successifs déclenchant une tâche.

I.6.2.b. Modélisation des tâches non périodiques

La terminologie concernant les tâches apériodiques et sporadiques a changé entre la fin des années 90 et nos jours. En effet, si l'on se réfère à la terminologie de [Liu00] que nous utiliserons, il existe deux catégories de tâches non périodiques : les tâches sporadiques et les tâches apériodiques. Dans les deux cas, aucun intervalle minimal de séparation entre deux occurrences successives d'une même tâche n'est connu. En effet, si un intervalle de séparation minimal était connu, alors on pourrait se ramener au cas non concret de tâches périodiques en utilisant cet intervalle comme période. Cela diffère des travaux menés dans les années 90 où l'on distinguait les apériodiques des sporadiques en supposant quand dans un cas un intervalle minimal était connu, et dans l'autre cas non.

Les tâches apériodiques sont caractérisées par une pire durée d'exécution et une date de réveil inconnue a priori. Elles n'ont pas d'échéance et le but de l'ordonnanceur est de minimiser leur temps de réponse après leur arrivée (type meilleur effort), sans mettre en cause l'ordonnançabilité des tâches à contraintes de temps strictes.

Les tâches sporadiques sont elles caractérisées en plus par une échéance. L'ordonnanceur décidera alors en-ligne s'il peut accepter ou non une tâche sporadique lors de son réveil.

Physiquement, les tâches sporadiques et apériodiques sont typiquement déclenchées sur interruption matérielle. Notons qu'une tâche d'alarme logicielle (i.e. une tâche déclenchée sur une condition logique par une autre tâche) peut rarement être considérée comme étant non périodique, puisqu'elle ne peut pas se déclencher plus rapidement que la tâche à l'origine de son déclenchement. Elle hériterait de fait de la période de la tâche déclenchante au pire.

Deux techniques naïves de traitement du trafic non périodique consistent soit à traiter les tâches en arrière-plan, soit à traiter les apériodiques sur interruption matérielle comme si l'on se trouvait dans un système synchrone. Cela ne donne pas de bons résultats puisque dans le premier cas, les tâches non périodiques seraient retardées inutilement (en effet, tout en respectant les contraintes temporelles du trafic périodique, il est souvent possible de repousser l'exécution d'une tâche sans pour autant nuire à son ordonnançabilité), et le taux d'acceptation des sporadiques serait très bas. Dans le second cas qui consiste à traiter les non périodiques dès leur arrivée, le trafic périodique ne pourrait pas être validé.

Les techniques proposées se sont concentrées sur différents types de serveurs dont le but est d'améliorer le temps de réponse du trafic apériodique tout en permettant de faire simplement et très rapidement des tests d'acceptation des sporadiques. Citons le serveur à scrutation [LSS87][SSL89], le serveur ajournable [SLS95], le serveur à échange de priorité [LSS87], le serveur sporadique [SSL89], et le serveur à vol de temps creux [LR92][CC89], et plus récemment les serveurs préservant la bande passant pour les algorithmes dirigés par les échéances [SB96][DLS97].

I.6.2.c. Modélisation de la précédence entre tâches

Les tâches d'un système temps réel sont rarement indépendantes, ainsi dans DARTS, des tâches liées par relation de boîte aux lettres ou de synchronisation doivent pouvoir être modélisées par des tâches temps réel. Cela est généralement fait par des contraintes de précédence.

La précédence entre tâches est mise en œuvre généralement à l'aide de primitives de synchronisation et de communication intégrées dans le code de chacune entre eux. Considérons la synchronisation de la tâche τ_1 par la tâche τ_2 par l'intermédiaire d'un événement, nommé «evt1»,

présentée sur la Figure I-68: l'attente de l'exécution de la partie $\tau_{2\text{-}1}$ de la tâche τ_2 par la partie $\tau_{1\text{-}2}$ de la tâche τ_1.

La première étape à réaliser est la transformation des tâches en tâches atomiques, cela conduit à des tâches qui ne comportent aucune primitive bloquante à l'intérieur du code, à l'exception d'une primitive d'attente (des événements en cas de synchronisation, ou des messages en cas de communication), qui peut se situer au début du code. Ainsi, une règle générale de cette transformation est:

- *Attente* de synchronisation ou de communication en début de tâche.
- *Envoi* d'événement de synchronisation ou de communication en fin de tâche.

Notons que l'ensemble des tâches atomiques obtenues est rigoureusement équivalent au niveau de l'exécution à l'ensemble initial de tâches. La Figure I-69 montre le découpage de des deux tâches τ_1 et τ_2 de la Figure I-68 en quatre tâches atomiques.

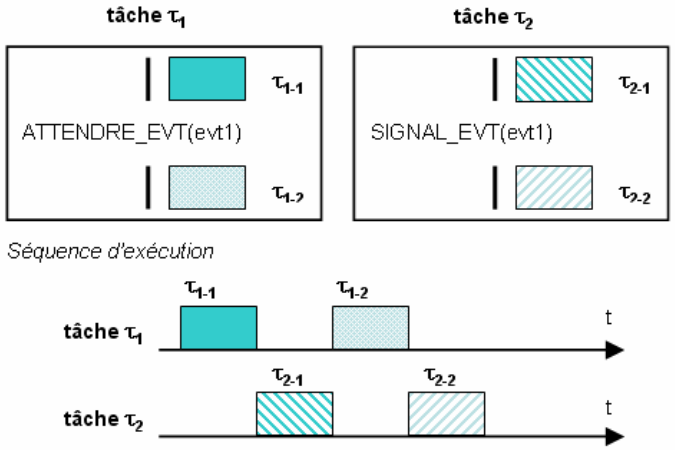


Figure I-68: Relation de précédence d'exécution entre deux tâches

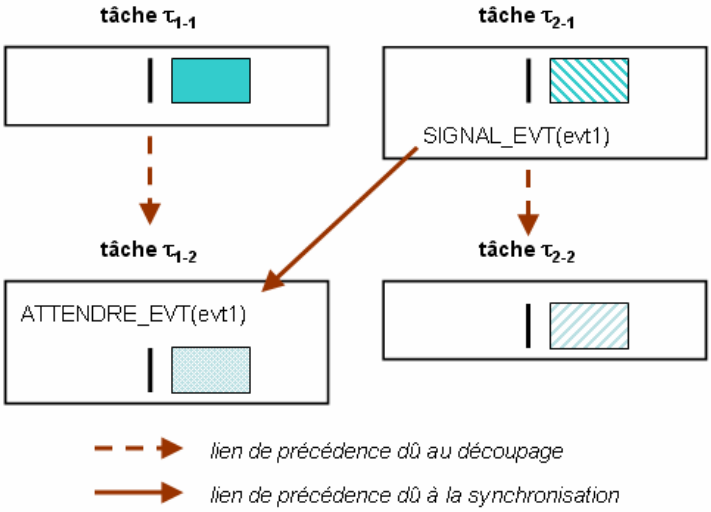


Figure I-69: Découpage des tâches en tâches atomiques

D'un point de vue analyse d'ordonnançabilité, les tâches communicantes sont considérées comme ayant la même période. Cela est tout à fait cohérent avec ce que l'on envisage lors de la conception (par exemple si une tâche périodique précède une tâche par relation de synchronisation par exemple, la tâche en attente de déclenchement hérite naturellement de la période de la tâche

déclenchante). Notons qu'il existe cependant différents travaux menés en ordonnancement pour assouplir cette contrainte [RCK01].

L'idée générale retenue en priorités fixes est de ne pas prendre en compte les contraintes de précédence des tâches ainsi obtenues directement, mais de faire en sorte que les réveils et priorités des tâches fassent que les précédences soient naturellement respectées lors de l'étude d'ordonnançabilité. En priorités fixes, il suffit en effet de faire en sorte qu'une tâche précédée par une autre soit moins prioritaire que celle-ci, et ne puisse pas être activée avant (par le jeu des dates de réveil si l'on est en concret, par l'étude d'un point où toutes les tâches sont activées en même temps en non concret).

Pour le cas des priorités dynamiques, les délais critiques sont modifiés afin que les priorités soient là aussi cohérentes avec les relations de précédence [Bla76].

Par conséquent, les tâches avec relations de précédences sont modélisées à l'aide de tâches indépendantes pour l'étude d'ordonnançabilité.

I.6.2.d. Modélisation du partage de ressources critiques entre tâches

Dans une conception DARTS, l'accent est mis non seulement sur les précédences, mais aussi sur le partage de ressources via les modules de données. Cette partie présente les principaux résultats concernant la prise en compte de ressources critiques en ordonnancement.

Une tâche $\boldsymbol{\tau_i}$ de durée totale $\mathbf{C_i}$ qui utilise une ressource critique **R** possède dans son code une zone protégée, appelée section critique, pendant laquelle elle accède à cette ressource. Cette section critique $\mathbf{Sc_i}$ est protégée par des primitives permettant de garantir l'exclusion mutuelle entre les portions de codes accédant à la ressource critique (comme un sémaphore par exemple).

L'exemple typique que l'on retrouve dans presque toutes les applications est la communication par tableau noir (module de données DARTS) : la ressource critique est la donnée et les opérations en exclusion mutuelle sont la lecture, l'écriture, éventuellement l'initialisation, l'incrémentation etc.

Un problème bien connu en ordonnancement, et rendu célèbre à la suite de la mission Mars Pathfinder [CDKM00], est le problème de l'inversion de priorité. En effet, supposons que deux tâches de priorité différente partagent une ressource, et qu'à un moment la tâche moins prioritaire détienne la ressource critique, alors que la tâche plus prioritaire soit en attente de celle-ci (en état bloquée). Si un tâche de priorité intermédiaire se réveille, elle préempte la tâche moins prioritaire pour s'exécuter, retardant ainsi la tâche la plus prioritaire en attente de la ressource.

[SRL90] propose deux protocoles de gestion de ressources permettant d'éliminer le problème d'inversion de priorité. Le premier, le Protocole à Priorité Héritée est le plus simple à expliquer, mais le plus complexe à mettre en œuvre et le moins efficace. Il s'agit simplement, lorsqu'une tâche plus prioritaire se retrouve en attente de ressource détenue par une tâche moins prioritaire de faire hériter à cette dernière de la priorité de la tâche en attente. Cela résout ainsi le problème de l'inversion de priorité. On trouve ce protocole de gestion de ressources dans l'exécutif VxWorks [Wind95].

Le second protocole, appelé Protocole à Priorité Plafond, est moins intuitif mais plus efficace (dans le sens où le temps d'attente lié aux tâches de priorité inférieur est plus faible). Il s'agit du protocole à priorité héritée avec une règle supplémentaire lors de l'entrée en section critique : chaque ressource est munie d'une priorité plafond qui correspond à la priorité maximale des tâches susceptibles d'utiliser la ressource. Lorsqu'une tâche veut entrer en section critique, elle ne peut le faire que si la ressource est libre, bien entendu, et si sa priorité est strictement supérieur au plafond système (valeur maximale des plafonds des ressources en cours d'utilisation) ou si elle est elle-même responsable de cette valeur plafond. Ce protocole a 2 propriétés intéressantes : il élimine toute possibilité d'interblocage (y compris des interblocages non triviaux tels que le problème classique du dîner des philosophes), et limite la durée de blocage due à une section

critique d'une tâche moins prioritaire à une seule section critique, quel que soit le nombres de ressources utilisées pendant son exécution. Il n'est donc, lorsqu'on utilise un protocole de gestion de ressources, pas nécessaire de recourir à la méthode des classes ordonnées [Hav68].

Ce protocole n'est pas utilisé tel quel dans les exécutifs temps réel, mais sous sa forme immédiate, qui correspond au protocole à super priorité proposé par C. Kaiser plusieurs années auparavant [Kai81][Kai82]. L'idée est tout simplement qu'une tâche hérite de la priorité du plafond de la ressource directement en entrant en section critique. Le fonctionnement est moins intuitif, mais les propriétés sont les mêmes que celles du protocole à priorité plafond (pas d'interblocage, au maximum un seul blocage dû à une seule section critique de tâche de priorité inférieure par exécution de la tâche). Ce protocole est le plus utilisé sur les exécutifs temps réel industriels (Ada, POSIX 1003.1c, OSEK/VDX,…).

[CL90] propose une adaptation du protocole à priorité plafond recalculant les plafonds à chaque terminaison de tâche afin de l'adapter aux algorithmes d'ordonnancement dynamiques dirigés par l'échéance. L'implémentation est coûteuse, et T.P. Baker a proposé un protocole plus simple à implémenter (Protocole d'Allocation de la Pile) dans [Bak91].

Ce qu'il est important de retenir est que quel que soit le protocole de gestion de ressources employé, et l'algorithme d'ordonnancement employé, la pire durée de retard infligé à une tâche par des tâches de priorité inférieures peut être calculée à l'avance, et généralement elle s'exprime, pour une tâche τ_i, par un facteur de blocage B_i. Ce facteur dépend donc des durées de sections critiques des tâches de priorité inférieure, et bien entendu du protocole de gestion de ressources utilisé.

I.6.3. Ordonnancement temps réel

Nous avons vu que dans ce cadre, les contraintes classiques se traduisaient par un découpage en tâches indépendantes pour les tâches avec contraintes de précédence, et en un facteur de blocage B_i permettant de représenter le retard infligé à une tâche à cause de l'accès aux ressources critiques par les tâches de priorité inférieure.

Nous nous concentrons donc ici sur l'ordonnancement de tâches indépendantes. A la fin du cycle de développement d'une application temps réel, lorsque le code est déployé sur la cible, il convient de choisir un algorithme d'ordonnancement, et de s'assurer que les contraintes de temps, traduites par les délais critiques, seront respectées tout au long de la vie de l'application. On parle de validation temporelle.

Une tâche est fiablement ordonnançable par un algorithme d'ordonnancement si toute séquence d'ordonnancement produite par l'algorithme, dans n'importe quelle situation possible (i.e. pour n'importe quelles dates de réveil dans le cas non concret), elle respecte toutes ses échéances. Un système de tâche (une application) est fiablement ordonnancé par un algorithme d'ordonnancement si toutes ses tâches sont fiablement ordonnancées. Une application est ordonnançable s'il existe un algorithme l'ordonnançant fiablement.

Il y a deux grandes catégories d'algorithmes d'ordonnancement : les algorithmes en-ligne, basés sur les priorités, qui élisent sur le processeur la tâche prête de plus forte priorité (dans ce cas les priorités peuvent être fixes ou variables), et les algorithmes hors-ligne, qui calculent une séquence d'ordonnancement préalable qui sera exécutée par un séquenceur.

Dans l'industrie du transport (automobile, avionique civile), sauf exceptions, c'est la méthode hors-ligne qui est choisie à l'heure actuelle, bien que les méthodes en-ligne fournissent une réactivité aux événements plus élevée (en effet, avec une méthode hors-ligne, les événements sont traités par interruption et stockés, il ne seront traités que lors de la prochaine activation périodique de la tâche de traitement).

Cependant, la plupart des exécutifs sur étagère proposent des algorithmes d'ordonnancement en-ligne basés sur des priorités fixes. Récemment, la norme Ada 2005 propose l'utilisation d'algorithmes d'ordonnancement à priorités variables.

L'idée de DARTSVIEW étant de faciliter le couplage Conception-Implémentation-Tests unitaires et d'intégration-Codage croisé-Validation temporelle, nous présentons quelques éléments de base concernant l'ordonnancement en-ligne et hors-ligne, qui sont retrouvés dans des outils, quelques-uns commerciaux, intégrant les principes de base de la RMA [RMA], mais surtout des outils universitaires, intégrant des modèles bien plus avancés. Pour ce manuscrit, nous faisons le choix de ne pas présenter les algorithmes d'ordonnancement classiques en détail (le lecteur peut se référer à différents ouvrages [CKM00][Liu00][but04]), mais de présenter les différentes familles, les types de tests associés (approchés ou exacts), les difficultés qui y sont liées (complexité algorithmique dans certains contextes pour une étude exacte), et les avancées en terme d'aide à la conception.

I.6.3.a. Ordonnancement à priorités fixes

La famille des ordonnancements à priorités fixes est la plus simple à mettre en œuvre : cette politique se retrouve sur tous les exécutifs du commerce, et elle est tellement répandue que la plupart des microprocesseurs intègrent même un dispositif permettant d'aider matériellement le travail du répartiteur lors du changement de contexte.

Politiques d'affectation de priorités

Les algorithmes les plus connus sont RM (*Rate Monotonic*) et DM (*Deadline Monotonic*). On parle d'algorithmes, bien que les algorithmes d'ordonnancement à priorités fixes soient en fait des politiques d'affectation de priorité.

RM [Ser72][LL73] consiste simplement à affecter une priorité inversement proportionnelle à la période des tâches. Cet algorithme est optimal dans la classe des algorithmes à priorités fixes dans le contexte des tâches indépendantes à échéance sur requête, pour les tâches concrètes simultanées, et pour les tâches non concrètes. Cela signifie que soit S un système de tâches de ce contexte, si il existe un algorithme d'ordonnancement à priorités statiques qui ordonnance fiablement S, alors RM ordonnance fiablement S aussi.

DM [LW82] consiste à affecter une priorité inversement proportionnelle au délai critique des tâches. DM est optimal dans la classe des algorithmes à priorités fixes dans le contexte des tâches indépendantes pour les tâches concrètes simultanées et pour les tâches non concrètes.

DM et RM sont en fait exactement équivalents lorsque les tâches sont à échéance sur requête. Par conséquent, ces propriétés d'optimalité très fortes font que DM est l'algorithme le plus utilisé, même lorsque les tâches ne sont pas indépendantes, lorsqu'on a affaire à un système non concret de tâches ou bien lorsque les tâches sont concrètes et simultanées.

Dans le cas où des tâches du système sont concrètes et différées, il existe une façon d'affecter les priorités de façon optimale (pour les tâches indépendantes) : c'est l'affectation de priorités proposée par Audsley [Aud91]. Cette affectation repose cependant sur un algorithme non polynomial.

Tests d'ordonnançabilité

Bien entendu, dans le cadre d'une application temps réel, le choix d'un algorithme d'ordonnancement est couplé à des tests d'ordonnançabilité. Un test répond à la question : « le système de tâches est-il fiablement ordonnancé par l'algorithme d'ordonnancement choisi ? ». Un test est exact ou approché, s'il est approché, une réponse négative ne veut pas nécessairement dire que le système n'est pas fiablement ordonnancé, mais que le système est dans un domaine d'incertitude « oui/non » nécessitant un test d'ordonnançabilité plus complexe afin de répondre.

Dans le cadre de tâches utilisant des ressources critiques, le problème d'ordonnançabilité est NP-difficile [Mok83]. De plus, nous avons vu que sans protocole de gestion de ressources, il se posait un problème d'inversion de priorités. Par conséquent, on utilisera pour les tâches partageant des ressources un protocole de gestion de ressources, qui sera caractérisé par un facteur de blocage maximal B_i. Les tests d'ordonnançabilité concernant ce contexte prennent donc en compte un B_i et sont de fait toujours inexacts (conditions suffisantes) à cause de ca facteur qui ne peut pas toujours intervenir dans la réalité mais sera pris en compte dans les tests.

Les tests d'ordonnançabilité proposés sont donc basés sur des tâches indépendantes, et un B_i sera introduit pour caractériser les blocages potentiels dus aux tâches de priorité inférieure. Nous parlerons cependant de tests exacts (lorsque la réponse donnée pour des tâches indépendantes serait exacte) ou approchés lorsque même pour des tâches indépendantes, le test n'est qu'une condition suffisante d'ordonnançabilité.

Les tests exacts se basent tous sur le calcul du temps de réponse des tâches. Lorsque les tâches sont non concrètes, il convient de caractériser le pire scénario pour chaque tâche. Ce pire scénario s'appelle l'instant critique [LL73][Aud91][BHR93] : il correspond à une activation simultanée de la tâche étudiée avec touts les tâches plus prioritaires qu'elle. Ce n'est pas nécessairement l'instance de la tâche démarrée à l'instant critique qui obtient le pire temps de réponse, mais c'est l'une des instances de cette tâche ayant lieu dans la période d'activité initiée par l'instant critique qui a le pire temps de réponse. Une période d'activité est une période pendant laquelle le processeur a toujours une charge non nulle à traiter pour les tâches de priorité supérieure ou égale au niveau de priorité de la tâche étudiée. Le test exact consiste donc à calculer la longueur de la période d'activité initiée par l'instant critique pour le niveau de priorité étudié. Il repose sur les formules de [JP86][Leh90][LSST91] consistant à calculer par recherche de point fixe la longueur de cette période d'activité.

Lorsque les tâches sont concrètes, et simultanées, l'instant critique a lieu, et le même test est employé. Par contre, lorsque certaines tâches sont différées, l'instant critique peut ne jamais avoir lieu, et le test basé sur l'instant critique est pessimiste. Ainsi, dans l'industrie où l'on trouve des ingénieurs qui ont utilisé par le passé des séquenceurs, il n'est pas rare de trouver des services où l'on décale le réveil des tâches afin de lisser la charge et d'éviter un instant critique (qui se caractérise par un pic de charge). En effet, soient deux tâches de périodes respectives T_i et T_j, et de dates de réveil respectives r_i et r_j, dans un cas strictement périodique (cas assumé lorsqu'on est dans le cas concret), alors il existe $pgcd(T_i,T_j)-1$ possibilités de décaler r_i et r_j de sorte que les eux tâches ne soient jamais réveillées en même temps [Dav02][Goo03]. Dans ce cas, l'analyse exacte est un problème NP-difficile et pour les tâches indépendantes, on est amenés à construire une simulation (i.e. le diagramme de Gantt du système), et n simulations (pour un système de n tâches) lorsque des ressources critiques sont partagées. La durée de simulation étudiée n'est pas polynomiale et est donnée pour les algorithmes à priorités fixes par [LM80][Goo03] et par [GG02] pour tout algorithme conservatif (i.e. ne laissant pas le processeur inoccupé s'il existe au moins une tâche à traiter).

Enfin, les tests approchés, conditions suffisantes d'ordonnançabilité, sont utilisés principalement dans deux contextes :

- lors de tests d'acceptation en-ligne de tâches sporadiques où l'on doit utiliser un algorithme de complexité linéaire au pire, afin de ne pas impliquer un surcoût trop important dû à l'ordonnanceur lui-même.

- lors du dimensionnement d'un système temps réel, qu'il soit manuel ou bien assisté à l'aide de méthodes d'analyse de sensibilité.

Les tests approchés les plus connus concernent RM dans le cadre de tâches à échéance sur requête. Il y a deux catégories de tests : les tests globaux [LL73][DG00][BBB03] donnant une réponse globale pour le système en testant une seule mesure, basée sur la charge, et les bornes sur

les temps de réponse [SH98]. Des avancées récentes, sous l'impulsion de E. Bini, dans l'analyse de sensibilité [BNB03][HG07] (permettant une construction interactive d'espaces de paramètres temporels pour lesquels un système est ordonnançable) sont prometteuses et donneront sans nul doute naissance à des outils performants de dimensionnement de système temps réel dans les prochaines années.

I.6.3.b. Ordonnancement à priorités variables

L'algorithme d'ordonnancement le plus représentatif de cette catégorie est EDF (*Earliest Deadline First*) qui donne la priorité la plus élevée à la tâche la plus urgente. Cet algorithme est optimal pour les tâches indépendantes et est donc celui qui sera à même d'être utilisé le plus lorsque l'industrie se tournera vers cette classe d'algorithmes. [Jack55][Horn74][Der74][Lab74] ont déjà prouvé son optimalité pour les tâches indépendantes.

A l'instar des algorithmes à priorités fixes, on trouve pour EDF deux catégories de tests d'ordonnançabilité : les tests exacts, basés sur la demande processeur, et les tests approchés, conditions suffisantes d'ordonnançabilité.

La demande processeur repose sur la période d'activité complète du système : il s'agit de s'assurer que toutes les échéances présentes dans la période d'activité seront respectées [JS93][BRH90]. Contrairement aux cas des algorithmes à priorités fixes, cette technique ne calcule pas les temps de réponse des tâches, et le pire temps de réponse est complexe à obtenir, car plusieurs cas sont à considérer pour caractériser le pire scénario pour une tâche [Spu96].

II. DARTS: du concept au code

II.1. Utilisation de la méthode DARTS

Suite de la présentation de la méthode DARTS au chapitre I, cette section discute en détail et distinctement les critères pour la décomposition d'un système en un ensemble des tâches DARTS, ou bien la création d'une structure des modules d'encapsulation de données dans le diagramme de l'architecture de ces tâches. Ainsi, une vue générale de l'intégration des tâches et des modules est présentée de manière indépendante de l'implémentation où le comportement statique et le comportement dynamique du système sont tous les deux. Enfin, je présente l'implémentation de ces diagrammes DARTS vers un outil de modélisation et/ou un exécutif temps réel.

II.1.1. Critères de création d'une structure des tâches DARTS

Selon la méthode DARTS, à partir d'une spécification du système, le concepteur décompose le système considéré en tâches concurrentes. Effectivement, il existe quatre catégories de tâches DARTS. Donc, une fonction peut être structurée soit en une tâche d'entrée-sortie (*I/O Task Structuring Criteria*), soit en une tâche interne (*Internal Task Structuring Criteria*), soit en un ensemble des sous-tâches ayant une cohésion (*Task Cohesion Criteria*), soit en une tâche en priorité (*Task Priority Criteria*).

II.1.1.a. A partir d'une spécification du système

La première catégorie nous permet de décrire les tâches qui sont activées en fonction des dispositifs d'entrée-sortie (*I/O Device*).

- Si c'est un dispositif actif ou asynchrone, la tâche est contrainte à s'exécuter à la vitesse du dispositif avec lequel elle interagit. Le dispositif d'entrée-sortie dans ce cas fournit en général des données discrètes, et est déclenché typiquement par un gestionnaire des interruptions de bas niveau, ou dans certains cas par le matériel. Nous appelons cette tâche «*Tâche d'entrée-sortie asynchrone*».
- Si c'est un dispositif passif, une horloge temps réel est utilisée pour activer périodiquement la tâche interagissant avec ce type de dispositif. La fréquence d'exécution de la tâche dans ce cas-là dépend non seulement des caractéristiques de ce dispositif, mais également des caractéristiques de l'environnement externe de l'application. Nous appelons cette tâche «*Tâche d'entrée-sortie périodique*», elle correspond à une tâche de scrutation.
- Si un dispositif d'entrée ou de sortie reçoit plusieurs requêtes provenant de multiples sources, il a besoin d'une tâche pour coordonner ces requêtes. Cette tâche doit mettre les requêtes en ordre afin de maintenir l'intégrité de données, et s'assurer qu'aucune donnée n'est corrompue ou perdue. Elle est analogue au concept de moniteur du langage Ada. Nous appelons telle tâche «*Tâche moniteur de ressources*»

La deuxième catégorie s'adresse à des tâches internes. Il s'agit de *tâches périodiques* pour les traitements des fonctions ou des transformations périodiques; de *tâches héritant du rythme d'autres tâches* pour les transformations qui sont activées par des événements internes ou des messages envoyés par d'autres tâches; de *tâches de contrôle* pour les objets qui sont représentés par des transformations de contrôle, et qui exécutent des diagrammes de l'état-transition; de *tâches utilisateurs* qui jouent le rôle d'interfaces entre le système et les divers dispositifs d'entrée-sorties standards tels que le clavier et l'affichage.

II.1.1.b. Vers l'implémentation sur un exécutif temps réel

Le support d'exécution considéré pour la méthode DARTS est un exécutif temps réel, dont l'unité d'exécution est la tâche. Les outils proposés par DARTS sont donc des outils présents sur la plupart des exécutifs temps réel.

La spécification décrivant le «Quoi?» et amène le plus souvent à une décomposition du «Quoi?» en fonctionnalités/modules/traitements. Avant d'implémenter le système, le concepteur doit donc choisir une architecture en tâches (le «Comment?» ces actions sont-elles activées). La tâche est non seulement l'entité de base du programme temps réel, mais aussi l'élément de base utilisé lors de la validation temporelle (étude d'ordonnançabilité). Différentes contraintes surviennent donc:

- Une tâche a un coût mémoire (une pile propre, sauvegarde de sont état lorsqu'elle est préemptée, utilisation de files d'attente) et processeur (changement de contextes, gestion par le système de l'ensemble des tâches). De plus, plus les tâches sont nombreuses pour une même spécification, plus elles ont des interactions fortes: cela signifie davantage d'utilisation d'outils de communication (d'où plus de temps et de mémoire occupée).
- Le débogage d'un système multitâche est plus complexe que celui d'un système séquentiel, ce qui rend la phase des tests d'intégration d'autant plus délicate que le nombre de tâches est élevé.
- Les techniques de validation temporelle se basent sur des modèles de tâches simples: les tâches obtenues doivent, dans le but de les valider, pouvoir être découpées en tâches en forme normale de sorte qu'il n'y ait aucun cycle de précédence. Ainsi, par exemple, les modèles d'ordonnancement ne prennent pas en compte la possibilité pour deux tâches de pouvoir dialoguer: si une tâche A émet un message vers une tâche B et que la tâche B émet en suite un message vers la tâche A, alors aucun modèle classique de l'ordonnancement ne peut être utilisé. On peut noter que dans la phase de spécification, il est fréquent que les fonctionnalités unitaires aient ce type de comportement, qu'il faudra éradiquer si l'on souhaite pouvoir valider temporellement le système. De plus, on surévalue la durée des tâches «pire cas» (l'expérience montre que si deux tâches indépendantes T_1 et T_2 sont capables de fusionner toutes les deux dans la tâche T', le temps d'exécution «pire cas» de ces deux tâches est beaucoup plus grande que celui de la tâche T').

Ces contraintes montrent que le nombre de tâche doit être restreint pour une meilleure efficacité du système et la volonté de pouvoir le valider temporellement. Il est des cas où l'on ne peut se passer de décomposer en différentes tâches:

- Deux traitements n'ont pas le même rythme, par exemple, le traitement A a lieu lorsqu'une interruption matérielle survient, à des moments inconnus lors de la conception, alors que le traitement B doit s'opérer toutes les 100 millisecondes. Bien entendu, les traitements A et B devront s'exécuter dans des tâches différentes.
- Certains traitements, notamment liés à des dispositifs externes, sont déclenchés par interruption matérielle avec chien de garde (si le dispositif externe est en panne, alors l'interruption n'a pas lieu, mais le traitement doit cependant être déclenché au bout d'un certain temps). Le chien de garde assure une robustesse du système en cas de défaillance du dispositif matériel. Dans ce cas, deux tâches peuvent être utilisées.

Il convient donc de réduire le nombre de tâches au minimum, tout en garantissant le rythme propre des fonctionnalités définies, et la robustesse du système. Il n'y a donc aucune raison de séparer des fonctionnalités élémentaires décrites dans la phase de spécification, sauf dans le cas où celles-ci pourraient avoir plusieurs façons d'être activées (mais dans ce cas on peut noter que la spécification ne serait sans doutes pas allée assez loin dans la décomposition du fonctionnement du système en fonctionnalités élémentaires). La phase de conception DARTS revient donc à regrouper les traitements dans des tâches logicielles.

Deux méthodes peuvent être utilisées: la méthode par raffinement, et la méthode par ajout.

- Regroupement des fonctionnalités en tâches par raffinement

Gomaa part des fonctionnalités élémentaires obtenues lors de la phase de spécification (par exemple le processus de contrôle et les processus fonctionnels primitifs dans la méthode SA-RT)

et les considère initialement comme des tâches. Le concepteur prendra en compte la convergence ou non de différents critères de cohésion afin de juger si des tâches peuvent être fusionnées. Dans ce cas, les traitements contenus seront exécutés séquentiellement à chaque activation de la tâche. Les critères de cohésion proposés sont:

- *Cohésion temporelle*: quand plusieurs transformations sont activées au même moment, ces transformations peuvent être groupées dans une tâche selon le critère de la cohésion temporelle. Cependant il est important de noter que, dans ce cas, l'ordre d'exécution des transformations est arbitraire (déterminé par le concepteur lors de la phase de l'implémentation).
- *Cohésion séquentielle*: cas où il existe un ordre d'exécution séquentiel entre certains traitements: par exemple la première transformation est provoquée par un événement asynchrone ou une horloge temps réel, puis à la fin de son exécution, ce traitement déclenche le traitement suivant. Ainsi, ces fonctionnalités peuvent être rassemblées dans une tâche conformément à la cohésion séquentielle. Notons que la cohésion séquentielle implique la cohésion temporelle.
- *Cohésion de contrôle*: spécifique à la méthode SA-RT, ce critère de cohésion est proche de la cohésion temporelle. Ainsi, lorsque le processus de contrôle déclenche toujours les mêmes traitements sur événement, alors ces traitements peuvent être regroupés.
- *Cohésion fonctionnelle*: si plusieurs fonctions sont liées l'une l'autre, et en sus une seule fonction parmi eux est exécutée à un instant donné, ces fonctions peuvent être groupées dans une tâche selon le critère de cohésion fonctionnelle. En général, la cohésion fonctionnelle est la forme la plus faible de la cohésion des tâches. Elle est utilisé principalement quand elle se soutient les autres formes de cohésion (i.e., cohésion temporelle ou séquentielle).

En appliquant les critères ci-dessus à des transformations de données et de contrôle de SA ou SA-RT, nous pouvons obtenir le diagramme l'architecture de tâches, ce qui permet de donner une vue claire de comportement dynamique du système avec des tâches concurrentes et également de faciliter la définition des interfaces entre tâches. L'architecture de tâches du système de chauffage d'une maison est un exemple.

Dans la première étape, à partir de son diagramme de spécification SA-RT, nous décomposons le chauffage en tâches suivantes (voir Figure II-1): tâches d'entrée-sortie périodiques, tâches d'entrée-sortie apériodiques, tâches périodiques, et tâche de contrôle. Donc, chacune des fonctions d'acquisition de données va correspondre à une tâche d'entrée-sortie périodique distincte (car dès l'activation, chacune entre eux provoque une lecture sur son propre capteur pour la valeur d'input). La tâche «Calcul Différence» est une tâche périodique interne, qui correspond à la transformation «Calcul Différence». «Contrôle Chauffage» est une tâche de contrôle. Elle comporte le diagramme de l'état-transition du système de chauffage. Cette tâche est activée par le matériel en fonction de ses paramètres d'entrée (par exemple, «Disj (ON/OFF)» pour le disjoncteur principal, «Ventil (ON/OFF)» pour le ventilateur, «Vanne (ON/OFF)» pour la circulation de mazout, etc.). Une fois qu'elle est activée, elle reçoit ses entrées d'abord, et puis elle exécute le diagramme état-transition afin de vérifier si un changement d'état a lieu, ce qui peut entraîner des activations/désactivations des transformations en dépendant, ou bien que des expéditions des commandes à des appareils de chauffage (Commande Ventilateur, Commande Vanne, Commande Allumage, et Affichage Etat). Enfin, nous mappons chacun des processus de commande à une tâche séparée selon le critère de tâches d'entrée-sortie. Ce sont des tâches activées par la tâche de contrôle «Contrôle Chauffage».

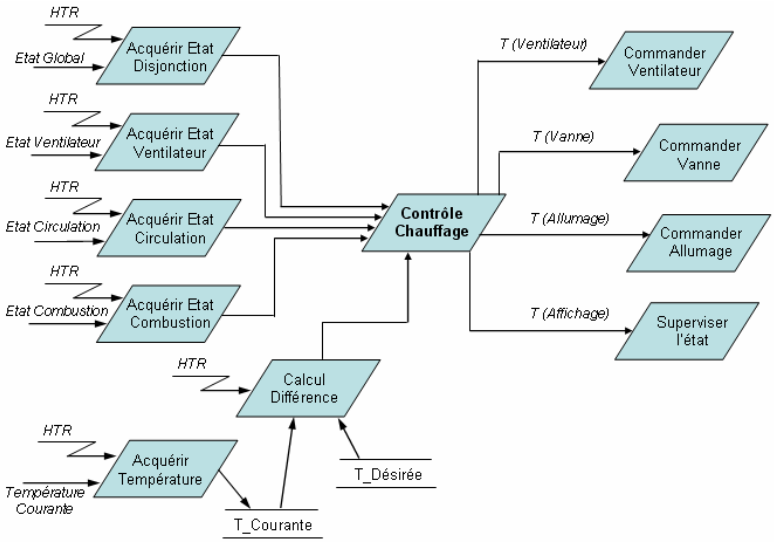


Figure II-1: La forme initiale de l'architecture de tâches DARTS du chauffage

Ensuite, le regroupement des fonctionnalités en tâches par raffinement a lieu. Cela peut être effectué itérativement en appliquant les critères de cohésion jusqu'à ce que le nombre de tâches puisse être réduit au minimum. En effet, la forme au début de l'architecture de tâches DARTS du système de chauffage consiste de 11 tâches. Donc, on peut regrouper les quatre tâches d'acquisition («Acquérir Etat Disjonction», «Acquérir Etat Ventilateur», «Acquérir Etat Circulation», et «Acquérir Etat Combustion») selon la cohésion temporelle si leurs fréquences d'exécution sont les mêmes. La tâche «Calcul Différence» et la tâche «Acquérir Température» peuvent être fusionnées dans une tâche, nommée «Mesurer Température» selon la cohésion séquentielle (dans ce cas-là, en utilisant le critère de tâche d'inversion, le stockage «T_Courante» peut être remplacée par des appels de l'opération «T_Courante» fournie par la tâche «Acquérir Température» à partir de la tâche «Calcul Différence»). Enfin, les trois tâches («Commande Ventilateur», «Commande Vanne», et «Superviser l'Etat») possèdent pour chacune une pair de fonctions liées par cohésion fonctionnelle, par exemple: «Démarrer Ventilateur» et «Arrêter Ventilateur» de la tâche «Commande Ventilateur»; «Ouvrir Vanne» et «Fermer Vanne» de la tâche «Commande Vanne»; «Déclencher Affichage» et «Eteindre Affichage» de la tâche «Superviser l'Etat»). Par conséquent, en fonction de l'événement généré par la tâche de contrôle, une seule fonction est exécutée. Le diagramme de l'architecture de tâches du système de chauffage présenté sur la Figure II-2 est une forme réduite de celui donné dans la Figure II-1.

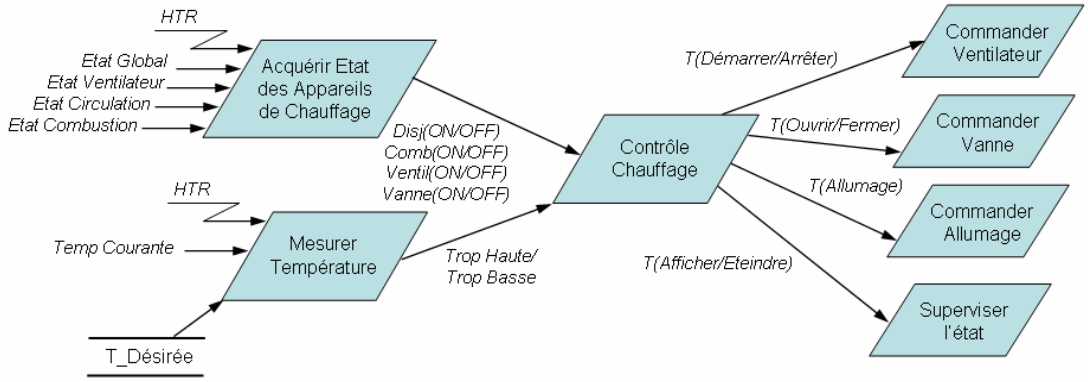


Figure II-2: Une forme réduite de l'architecture de tâches DARTS du chauffage

Mais, on ne s'arrête pas d'ici. Regardons encore la Figure II-2, nous voyons que les 5 tâches («Contrôle Chauffage», «Commander Ventilateur», «Commander Vanne», «Commander Allumage», et «Superviser l'Etat») peuvent être regroupées en une seule tâche, appelée «Contrôle Chauffage» par exemple, d'après le critère de cohésion de contrôle. Et finalement, la forme finale

de l'architecture de tâches du système de chauffage peut consister simplement en 3 tâches comme sur la Figure II-3.

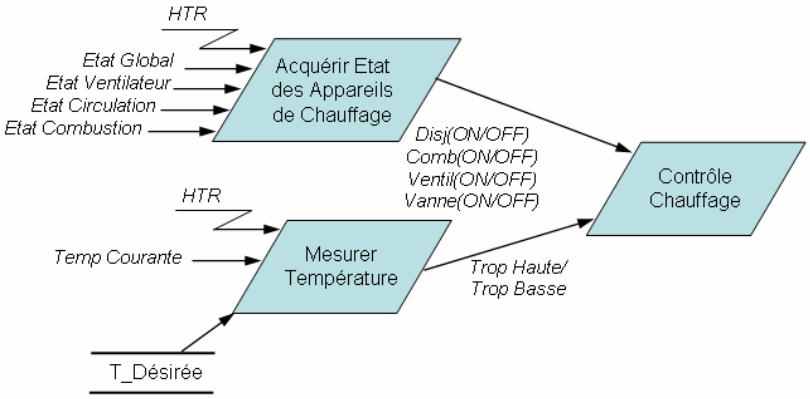


Figure II-3: La forme réduite de l'architecture de tâches DARTS du chauffage

- Regroupement des fonctionnalités en tâches par rythme (ou par ajout)

Le principe de cette approche est d'identifier les rythmes du système : un événement externe déclenchant une interruption matérielle est typiquement un rythme, une scrutation particulière, une commande à appliquer régulièrement traduit un autre rythme, etc. Les fonctionnalités obtenues lors de la spécification sont alors insérées dans les rythmes. Chaque rythme est alors implémenté par une tâche.

Considérons le système de chauffage d'une maison, à partir du diagramme de l'architecture de tâches (voir Figure II-1), les fonctions d'acquisition peuvent être placées dans deux rythmes différents, donc un pour les fonctions d'acquisition des états des appareils du chauffage («Acquérir Etat Disjonction», «Acquérir Etat Ventilateur», «Acquérir Etat Circulation», et «Acquérir Etat Combustion»), un autre pour les deux fonctions de mesure des températures («Acquérir Température», «Calcul Différence»); de même la fonction de contrôle et les quatre fonctions de commande («Commander Ventilateur», «Commander Vanne», «Commander Allumage», «Superviser l'Etat») peuvent être fusionnées dans un rythme. Par conséquent, l'implémentation du système de chauffage nécessite au moins trois rythmes, qui correspondent à trois tâches; et la forme finale de l'architecture de tâches DARTS obtenue est vraiment semblable à celle présentée sur la Figure II-3.

II.1.2. Communication/Synchronisation des tâches

La méthode DARTS propose différents types de communication/synchronisation basés sur les outils offerts par les exécutifs temps réel. Après regroupement des fonctionnalités en des tâches DARTS, il faut donc choisir parmi ces outils comment les communications ou les synchronisations seront implémentées. Les éléments de communication/synchronisation proposés dans la méthode DARTS sont les suivants:

II.1.2.a. Communication de type synchrone faiblement couplée

La communication de type synchrone faiblement couplée représentée typiquement par l'outil BAL (Boîte aux lettres), c'est le passage de messages d'une tâche à une autre où l'expédition et la réception de donnée se produisent de façon à ce que la tâche consommatrice attend un message de la tâche productrice pour activer son exécution, tandis que la tâche productrice envoie un message à la tâche consommatrice, et puis continue son exécution sans l'attente de la tâche consommatrice à l'écoute pour lui envoyer des messages. Les messages sont gérés dans une queue de messages du type FIFO ou bien à priorité.

En outre, ce type de communication peut posséder plusieurs tâches en amont en considérant une opération «OU» entre toutes les communications en provenance des tâches en amont. La Figure

II-4 présente un exemple des communications entre trois tâches productrices et une tâche consommatrice par une BAL à priorité en considérant une opération «OU». Supposons que les messages de la productrice A sont au niveau de priorité la plus haute, les messages de la productrice B sont au niveau de priorité intermédiaire, et les messages de la productrice C sont peu prioritaires, les messages de A sont toujours traités d'abord, tandis que les messages de B sont traités avant les messages de C.

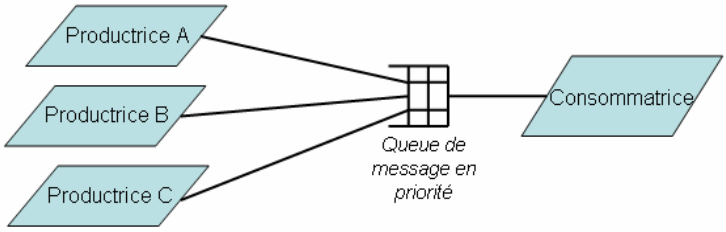


Figure II-4: Communication entre les tâches par BAL à priorité avec entrées multiple de type «OU»

En revanche, il est déconseillé d'avoir plusieurs tâches en aval car cela nuit au déterminisme du programme (on ne sait pas quelle tâche lit le message).

Dans le cas où les communications de type synchrone faiblement couplée possèdent plusieurs tâches en amont en considérant une opération «ET» entre toutes les communications en provenance des tâches en amont, la tâche en aval doit attendre un message généré par chaque tâches en amont. Par conséquent, la fréquence d'activation de la tâche en aval est égale à celle de la tâche la plus lente parmi les tâches en amont. Par exemple, dans le système de chauffage, si les BALs font les interfaces entre la tâche de contrôle et les tâches d'acquisition (voir Figure II-5), la tâche «Contrôle Chauffage» est activée régulièrement à la même fréquence que celle la plus faible des tâches d'acquisition (c'est la tâche «Mesurer Température» avec la fréquence d'activation de 1000 ms). Il en résulte que la tâche «Contrôle Chauffage» lors de l'activation consomme un seul message dans les quatre messages générés par la tâche «Acquérir Etat des Appareils de Chauffage» durant 1000 ms, c'est-à-dire moins de réactivité aux messages des tâches plus rapide, et notamment la possibilité de blocage en écriture de ces tâches au cas où leur BAL est bornée. Ainsi, il est également déconseillé d'avoir plusieurs tâches en amont en considérant une opération «ET» entre toutes les communications en provenance des tâches en amont en cas des communications de type synchrone faiblement couplée.

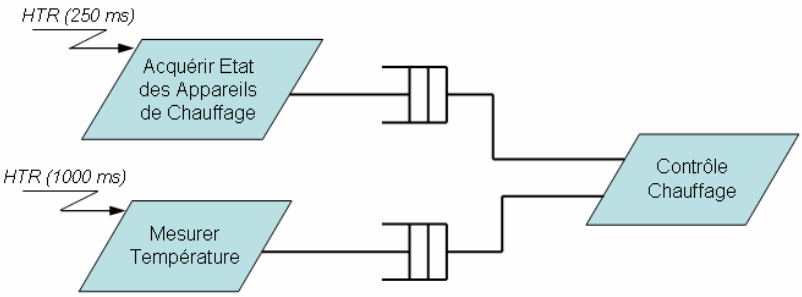


Figure II-5: Communication entre les tâches par BAL avec entrées multiple de type «ET»

II.1.2.b. Communication de type synchrone faiblement couplée à écrasement

Ce sont les communications, dites à écrasement, qui fonctionnent en éliminant la donnée la plus vieille lorsque la BAL est pleine. Il est important de noter que ces communications sont non bloquantes en écriture.

Comme pour les communications par BAL classiques, ces communications peuvent posséder plusieurs tâches en amont en considérant une opération «OU» entre toutes les communications en

provenance des tâches en amont. La Figure II-6 montre une utilisation en « ET » de deux BAL à écrasement.

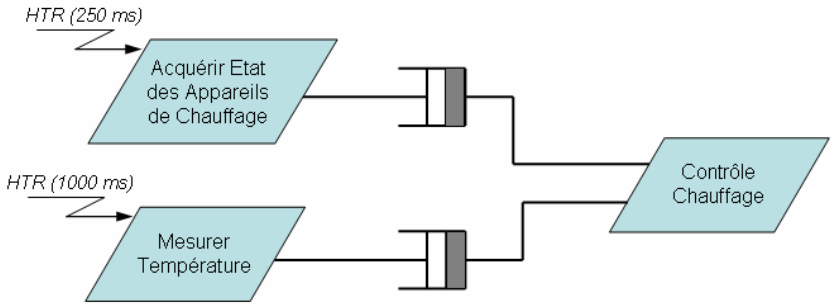


Figure II-6: Communication entre les tâches par BAL à écrasement avec entrées multiple de type «ET»

II.1.2.c. Communication synchrone fortement couplée

Utilisé en Ada83, ce type de communication (le rendez-vous), voir Figure II-7, correspond au passage d'information d'une tâche à une autre où l'expédition d'un message à la tâche consommatrice et l'envoi d'une réponse à la tâche productrice sont synchronisés. La productrice envoie un message à la consommatrice, et puis attend une réponse. Tandis que quand le message est arrivé, la consommatrice reçoit ce message, le traite, et envoie la réponse. Puis, la productrice et la consommatrice, peuvent continuer leur exécution indépendamment.

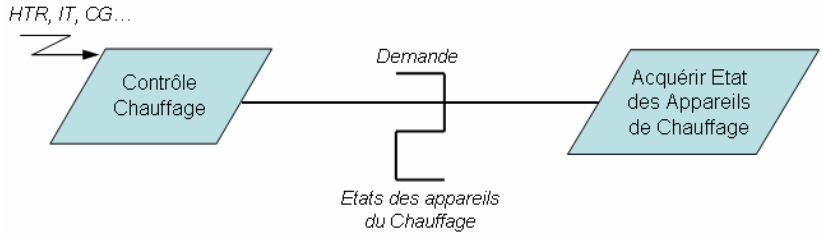


Figure II-7: Communication synchrone fortement couplée dans le chauffage

En pratique ce type de communication a été banni par le profil Ravenscar [Bur99][BDV04], car il entraîne de nombreux changements de contexte et une analyse d'ordonnançabilité délicate. En effet, aucun modèle classique de l'ordonnancement ne peut être utilisé.

II.1.2.d. Synchronisation

La synchronisation est la communication sans données entre tâches où la tâche émettrice génère un événement, et la tâche réceptrice se bloque en attendant cet événement. Lorsque l'événement est signalé, la tâche réceptrice se réveille. Puisque la tâche réceptrice ne peut s'exécuter avant la tâche émettrice, la synchronisation est utilisée pour satisfaire des contraintes de précédence. Dans les exécutifs ou noyaux temps réel, la synchronisation peut être implémentée par le Sémaphore, ou l'Evénement, ou dans certains cas par la communication synchrone avec une donnée « fictive ». La Figure II-8 représente une synchronisation.

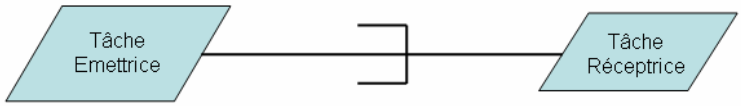


Figure II-8: Modèle de synchronisation entre tâches

Au cas où la synchronisation est réalisée par sémaphore le compte initial vaut 0, ce qui fait que la réceptrice se bloque en tentant de prendre le sémaphore qui est nul. La tâche déclenchante ne s'occupe alors qu'à vendre le sémaphore pour réveiller la tâche en attente de synchronisation.

Notons qu'il existe deux variantes lors de l'implémentation : si l'on utilise un sémaphore binaire, on obtient une synchronisation sans effet de mémoire, alors que si le sémaphore est à compte, la synchronisation est à compte elle aussi.

La Figure II-9 illustre un choix devant lequel le concepteur se retrouve souvent : lorsqu'il souhaite concevoir le fait qu'une tâche soit déclenchée par une autre pour allumer ou éteindre un dispositif par exemple. Dans le cas (a), on choisit un fonctionnement de type interrupteur poussoir (*switch*) : l'état est stocké en interne, déclencher une fois allume, déclencher une autre fois éteint, etc. Ce fonctionnement est risqué car très sujet aux erreurs : imaginons un système dans lequel allumage et extinction se retrouvent inversés suite à une synchronisation déclenchée une fois de trop. Par conséquent, l'utilisation d'une BAL sera conseillée (c).

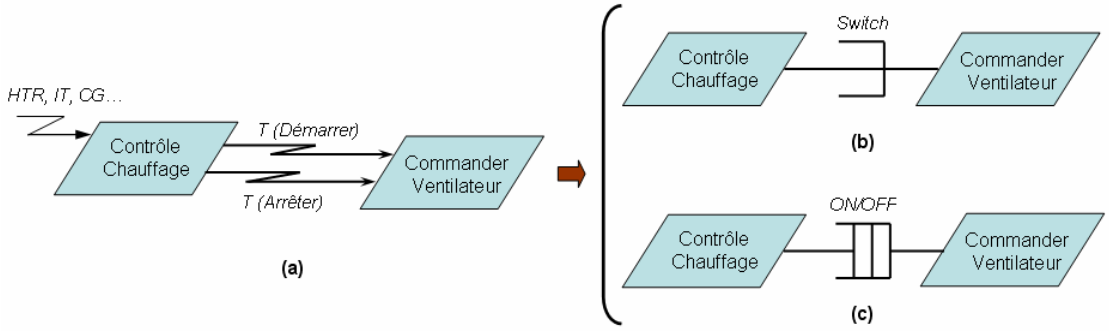


Figure II-9: Synchronisation par événements dans le chauffage
(a) en forme des deux événements séparés
(b) en forme de la communication synchrone faiblement couplée par «switch»
(c) en forme de la communication synchrone faiblement couplée classique

II.1.2.e. Module de données (MDD)

Le module d'encapsulation de données modélise le stockage de données en permettant des accès à une unité de gestion de données en exclusion mutuelle par deux ou plusieurs tâches, et en fournissant des opérations d'accès aux données. Il est important de noter que les accès simultanés à un module sont en exclusion mutuelle afin de garantir l'atomicité des opérations, par exemple une opération ne peut pas interrompre une opération en cours sur le même MDD. Un exemple de module de données implémentant la Figure II-3 est donné sur la Figure II-10.

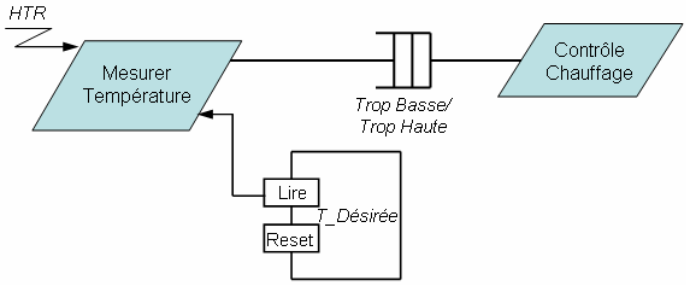


Figure II-10: Module de données dans le chauffage

Le pseudo-code de la tâche «Mesurer Température» dans ce cas-là est ci-après:

```
-------------------------------------
-- Tâche «Mesurer Température»
-- Déclaration de la BAL
-- Déclaration du module de données
-- Déclaration de la gestion de la périodicité
DEBUT
FAIRE TOUJOURS
...
```

```
MDD.LIRE (T_Désirée)
< Calculer la différence des températures,
    Différence_Température := Trop_Basse/Trop_Haute >
ENVOYER Différence_Température à la BAL
...
FAIT
FIN TACHE
-------------------------------------
```

Il est important de noter que les modules de données (MDDs) ne peuvent en aucun cas être utilisés comme des éléments de synchronisations. Par conséquent, les tâches qui accèdent à ces modules sont totalement asynchrones et doivent posséder leur propre mécanisme d'activation.

II.1.2.f. Modélisation de l'activation des tâches

Il existe deux types de tâches: tâches matérielles et tâches logicielles. Dans ces deux cas, l'activation est très différente. Pour les tâches matérielles qui sont déclenchées par des événements ou signaux externes, nous distinguons trois types de signaux:

- *«Horloge temps réel – HTR»*: ce signal provient d'une horloge matérielle interne à l'ordinateur. Il est utilisé pour activer des tâches matérielles périodiques. Les tâches ainsi obtenues sont modélisées par des tâches concrètes strictement périodiques lors de l'étude d'ordonnançabilité.
- *«Interruption – IT»*: ce signal provient du procédé externe. Il doit toujours être considéré comme apériodique du fait de l'asynchronisme du monde extérieur par rapport au cadencement de l'ordinateur. Il sera modélisé par une tâche non concrète sporadiquement périodique ou bien non périodique.
- *«Chien de garde – CG»*: ce signal provient d'une horloge interne utilisée comme un réveil. Un chien de garde est une protection destinée principalement à borner l'état de blocage en attendant un événement et/ou une action d'une tâche dans un délai donné. En termes de signal, il est identique à l'horloge temps réel, mais il se produit de façon apériodique.

Comme je l'ai déjà présenté, les tâches logicielles sont déclenchées par d'autres tâches (soit matérielles, soit logicielles) à travers des mécanismes de synchronisation ou de communication bloquant la réceptrice. Ainsi, le signal d'activation de ces tâches peut être l'un des éléments donnés sur la Figure II-11.

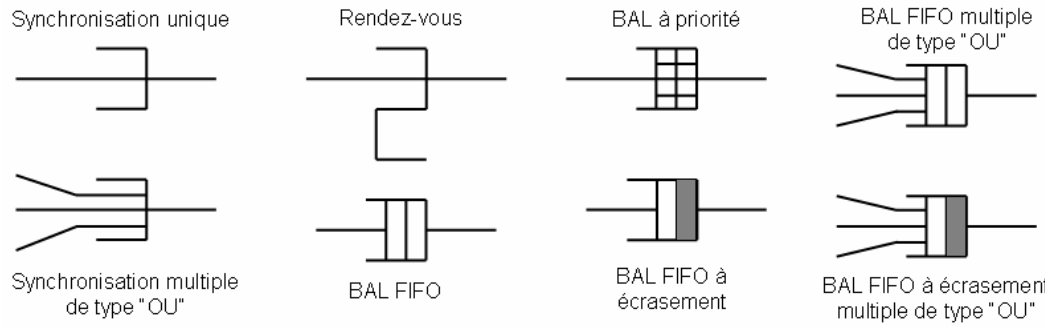


Figure II-11: Représentations des synchronisations et des communications entre tâches

D'un point de vue ordonnancement, en fonction du modèle choisi, une tâche logicielle peut se modéliser par contraintes de précedence.

La Figure II-12 représente une conception DARTS du système de chauffage. Nous voyons très clairement que dans ce diagramme l'activation de la tâche de contrôle est une «union» des déclenchements de la tâche «Acquérir Etat des Appareil de Chauffage» et de ceux de la tâche «Mesurer Température». En conséquence, le rythme auquel la tâche «Contrôle Chauffage» est

exécutée n'est pas connu. De plus il n'est pas possible de déterminer le temps minimal entre deux occurrences successives de cette tâche. Cela entraînera donc des difficultés dans l'étude d'ordonnançabilité du système.

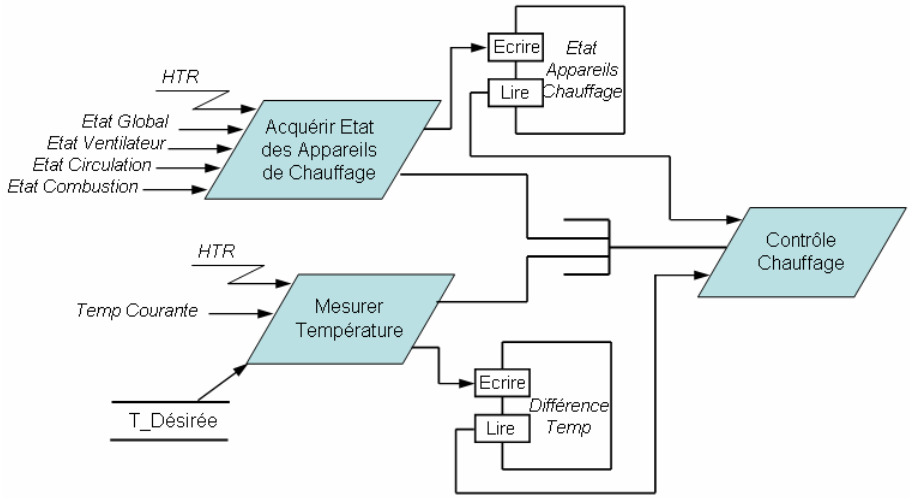


Figure II-12: Spécification préliminaire DARTS du système de chauffage (modèle de synchronisation multiple de type «OU»)

Au contraire, sur la Figure II-13 nous avons choisi de faire hériter la période de la tâche la plus rapide à la tâche « Contrôle Chauffage », dans ce cas, « Contrôle chauffage » pourra être modélisée par une tâche périodique, avec contrainte de précédence.

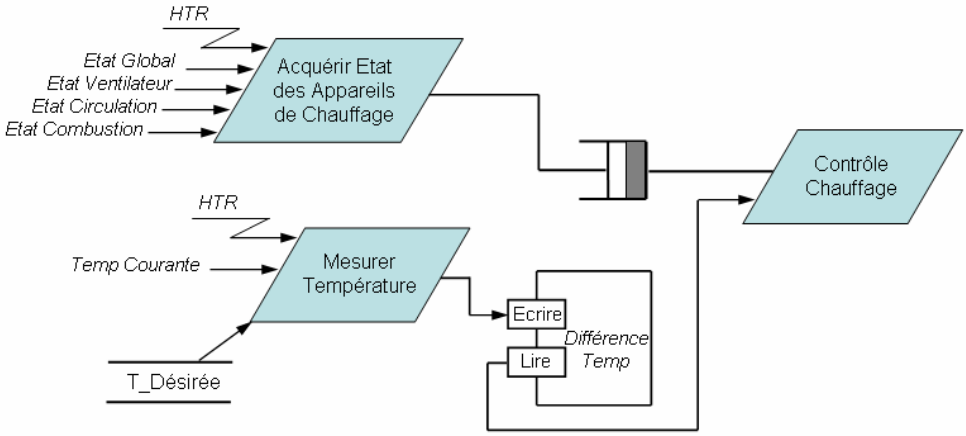


Figure II-13: Spécification préliminaire DARTS du système de chauffage (choix de tâche la plus rapide)

II.1.3. Critères de création d'une structure de modules de données

Il est conseillé d'encapsuler les données et d'y accéder au travers d'interfaces, garantissant le cas échéant l'exclusion mutuelle des accès aux données. Ce faisant, nous avons une vue claire sur la gestion des données ou des ressources partagées, une extension de variables globales que nous trouvons dans certaines méthodes de développement telles JSD, SD, etc. Selon [Par84], la décomposition d'un système en une structure de modules se fait de manière à ce que chacun des aspects du système, qui pourra être amené à changer, doit être une partie secrète cachée dans un module distinct. Chaque module possède une interface abstraite qui décrit la vue externe du module aux utilisateurs, et une structure de données qui peut être accédée directement par une ou plusieurs fonctions définies à l'intérieur de module. Le but principal de décomposition en modules est de réduire le coût total de développement logiciel en autorisant les modules à être construits et révisés indépendamment.

S'inspirant de ces principes, Gomaa dans [Gom93] nous propose des critères de création de différents types de modules de données présentés dans cette partie.

II.1.3.a. Modules d'interface de périphériques (Device Interface Modules, DIM)

Un DIM fournit une interface virtuelle qui cache les caractéristiques des périphériques d'entrée-sortie réels. L'interface virtuelle consiste en un ensemble de fonctions et/ou de procédures d'accès. Donc l'accès à un périphérique d'entrée-sortie n'a lieu qu'à travers cette interface virtuelle.

Sur la Figure II-14, la tâche «Acquérir Etat Ventilateur» appelle l'opération «Echantillonner» d'un DIM représentant physiquement l'accès au capteur.

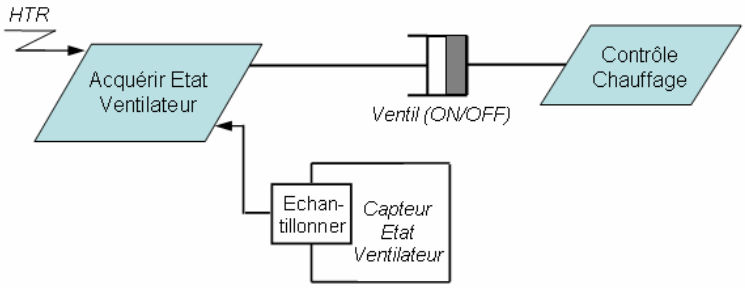


Figure II-14: DIM pour le capteur de l'état du ventilateur

II.1.3.b. Modules d'abstraction de données (Data Abstraction Modules, DAM)

Dans la programmation orientée objet, l'abstraction de données est un mécanisme qui permet de définir un type d'objet en réunissant à la des attributs et des opérations (méthodes) qui permettent de manipuler ces données. Dans la méthode DARTS, en appliquant ce mécanisme, nous pouvons former un module d'abstraction de données pour chaque stockage de données tel que le module d'encapsulation de données «T_Désirée» du système de chauffage (voir Figure II-10).

II.1.3.c. Modules d'état-transition (State Transition Modules, STM)

Il est fréquent d'exprimer le comportement d'une fonction à l'aide d'une machine à états (diagramme états/transitions, Statechart, etc.). Un STM est un module d'abstraction de comportement qui maintient l'état courant du module. En fonction de certaines conditions ou de certains événements d'entrée à un état donné, l'exécution de STM peut entraîner un changement d'état.

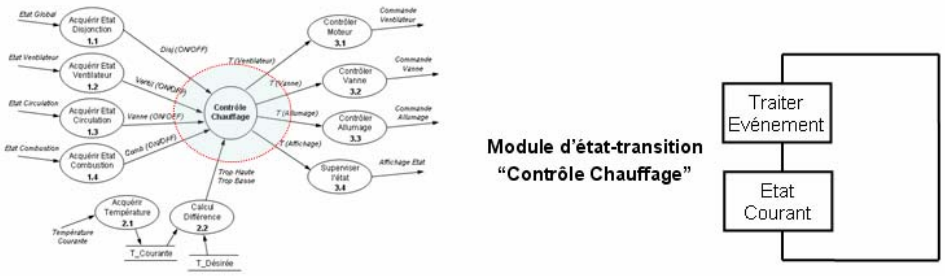


Figure II-15: Module d'état-transition «Contrôle Chauffage»

II.1.3.d. Modules de pilotes (Function Driver Modules)

Les pilotes sont les fonctions de contrôle qui traduisent les commandes d'un logiciel afin de permettre à l'ordinateur de communiquer avec un périphérique. Un module de pilotes est donc également un module d'abstraction de comportement. Il contrôle un ensemble de sorties qui ont une relation réciproque avec un dispositif d'entrée-sortie virtuel (qui est géré par un module d'interface de périphérique).

Pour le système de chauffage, si la vitesse de rotation du ventilateur (i.e., nombre de tours/minute) peut être manipulée d'une manière précise, un module de pilotes pour la gestion des commandes de ventilateur est créé. Son rôle principal est de déterminer les valeurs des sorties pour le contrôle de la vitesse du moteur en tenant compte de l'état d'opération courante du chauffage, et de certaines requêtes (par exemple, «Maintenir Vitesse», «Augmenter Vitesse», «Diminuer Vitesse», «Arrêter», etc.).

Il est important de noter que chaque module correspond à un composant passif qui n'est activé que par des composants actifs (représentés par des tâches dans la phase de structuration en tâches). Les modules nous donnent une vue générale sur l'aspect statique d'un système. Dans la suite, l'intégration des composants passifs et des composants actifs est présentée

II.1.4. Intégration de tâches et de modules

Après la décomposition d'un système en tâches et en modules, la modélisation comportementale du système peut être vue à partir de deux perspectives: la vue statique (modules) et la vue dynamique (tâches). Les éléments actifs utilisent les opérations des composants passifs.

Si un module est utilisé exclusivement par une tâche, ce module peut résider à l'intérieur de cette tâche, tandis que si un module est utilisé par plusieurs tâches, il doit résider à l'extérieur des tâches. De plus dans les cas où un module est accédé par plusieurs tâches, ses opérations doivent être en exclusion mutuelle les unes par rapport aux autres.

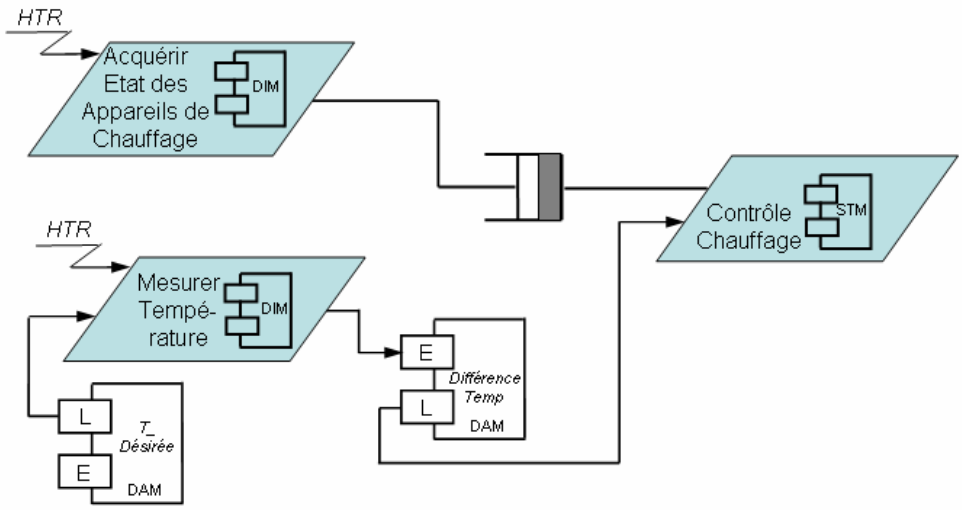


Figure II-16: Diagramme de l'Architecture Logicielle du système de chauffage

La Figure II-16 nous représente la forme réduite du Diagramme de l'Architecture Logicielle du système de chauffage.

II.2. Implémentation DARTS sur LabVIEW

II.2.1. Présentation de LabVIEW

LabVIEW (Laboratory Virtual Instrument Engineering Workbench) est un environnement de développement d'application en Langage G. LabVIEW était initialement destiné à l'acquisition de données, au traitement du signal et au contrôle-commande de procédé. Pour faciliter son utilisation, LabVIEW fournit une boîte à outils et une bibliothèque de fonctions et de contrôles très puissante (Functions Palette/Controls Palette) [Cot01].

On appelle un programme (ou sous-programme) LabVIEW un Instrument Virtuel (VI), parce qu'à l'origine, l'idée était de remplacer les instruments physiques (oscilloscopes, etc.) par des instruments programmés. En fait, un VI est un appareil virtuel ayant des opérations, ce qui nous permet de le faire fonctionner comme s'il s'agissait d'un véritable instrument. Les opérations de

LabVIEW offrent non seulement les différents moyens de traitement et d'analyse des paramètres d'entrée venus d'un utilisateur, mais également diverses façons d'affichage des résultats ou de changement des paramètres opérationnels grâce à la souris ou au clavier de manière à correspondre visuellement aux faces avant des instruments du monde réel.

Un VI est composé de trois parties: une face avant, un diagramme et une icône (voir Figure II-17):

- *La Face Avant*: symbolise l'interface de l'instrument physique. Cette fenêtre est constituée par des objets contenus dans la palette de contrôles. Chaque objet, en fonction de ses attributs est soit une commande lorsqu'il reçoit des entrées, soit un indicateur lorsqu'il affiche des sorties.
- *Le Diagramme*: décrit le fonctionnement interne d'un VI en langage G sous la forme de DFD (Diagramme flots de données). Dans cette fenêtre, chaque objet de la face avant correspond à une boîte étiquetée par son type de donnée associé (LabVIEW est un langage fortement typé). Les nœuds de calcul sont des fonctions contenues dans la palette de fonctions.
- *L'Icône*: symbolise un instrument virtuel (VI) afin de l'utiliser dans un autre VI. C'est le mécanisme d'encapsulation du DFD. Un connecteur est associé à cette icône: chaque partie de ce connecteur est associée à un objet de la face avant. Un VI pourra alors être utilisé comme une fonction dans un autre VI.

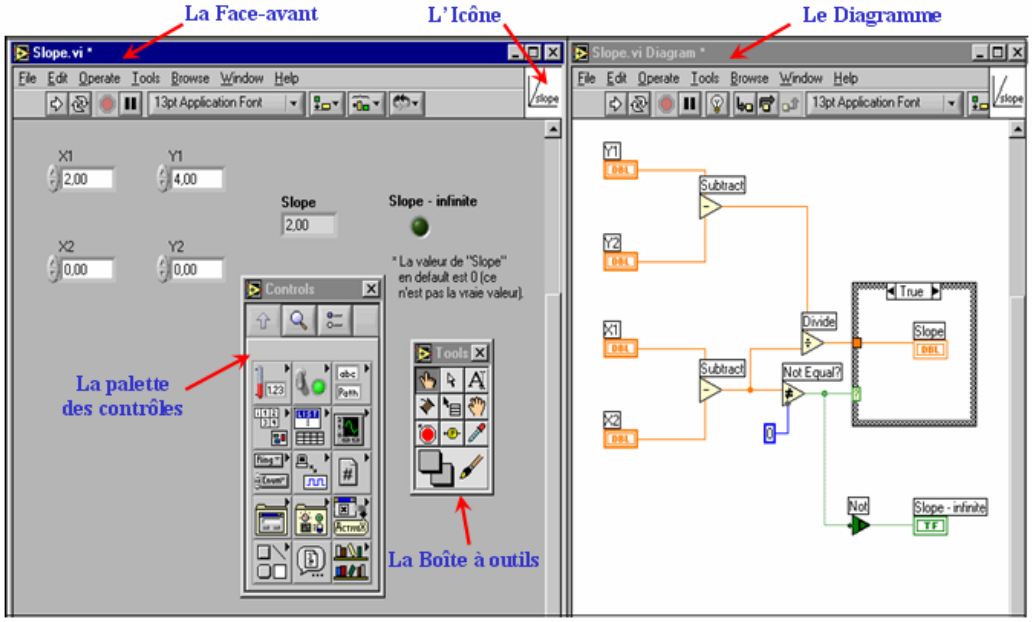


Figure II-17: Trois composants d'un VI: la face avant, le diagramme et l'icône

II.2.1.a. Principe de la programmation graphique flot de données

Le flot de données est une de deux méthodes de représentation graphique d'un programme. Son principe est de décrire les programmes comme étant des nœuds de calculs interconnectés par des arcs spécifiant le flot de données transitant entre les nœuds producteurs de données et les nœuds consommateurs de données [Gev98].

Une façon intuitive permettant d'interpréter ce principe est de suivre la règle d'évolution des jetons suivante (voir Figure II-18):

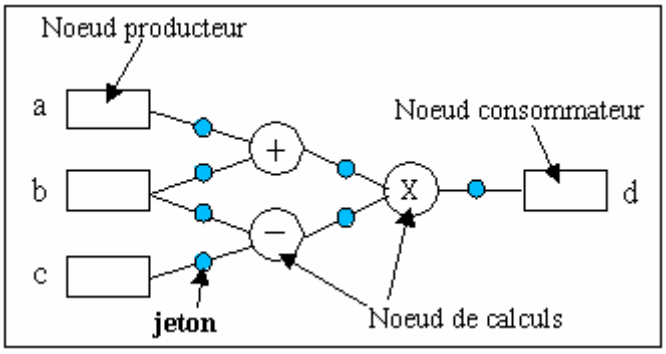


Figure II-18: "d = (a+b) (b-c)" sous la forme d'un DFD

- Au début, chaque nœud producteur produit un jeton sur son arc de sortie (nœuds a, b, c sur la Figure II-18).
- Lorsqu'un nœud de calcul reçoit un jeton à chacune de ses entrées, il peut être exécuté. Les jetons d'entrée sont alors consommés et un jeton est produit sur chacun des arcs de sortie du nœud. Par exemple, sur la Figure II-18, le nœud «+» ne s'exécute qu'après la réception de deux jetons fournis par les deux nœuds producteurs «a» et «b». Ensuite, il envoie un jeton sur son arc de sortie au nœud «×». Il en va de même pour les deux nœuds de calcul «–» et «×». Le nœud «d» est un nœud consommateur qui attend un jeton originaire du nœud «×» pour afficher le résultat de cette expression.

Un concept fondamental de ce principe est que si deux ou plusieurs nœuds possèdent assez de jetons pour pouvoir être exécutables, ils peuvent être exécutés en parallèle (LabVIEW utilise en fait plusieurs tâches pour exécuter les VIs). Donc, les nœuds «+» et «–» dans la Figure II-18 peuvent être exécutés en parallèle. Grâce à cette caractéristique, le langage de programmation graphique LabVIEW est capable d'implémenter aisément des tâches concurrentes, et il est un outil adéquat à associer à des méthodes de développement de logiciels dans le monde de systèmes de contrôle-commande.

II.2.1.b. Les structures de contrôle dans LabVIEW

G est un langage homogène, dynamique, et multidimensionnel [And98].

- *Homogène*: les nœuds G produisent et consomment un seul jeton pour chaque arc dans le diagramme flot de données.
- *Dynamique*: G inclut des structures de contrôle permettant à certaines portions d'un diagramme d'être exécutées continuellement fondées sur les entrées.
- *Multidimensionnel*: G possède des outils complets pour le traitement des tableaux multidimensionnels. Les boucles dans G peuvent être utilisées pour combiner des jetons individuels en des tableaux de jetons, ou pour séparer un tableau d'éléments en des jetons individuels.

Le langage LabVIEW possède les deux structures de base qui permettent d'exprimer tous les algorithmes: la structure de répétition et la structure conditionnelle. Etant donné le parallélisme implicite des programmes flot de données, une structure de séquencement a été ajoutée à ce langage. Ainsi, la structure de répétition est déclinée classiquement en deux types (For/«Pour» et While/«Tant Que») [Cot01].

- Structure Séquence (*Sequence*): (voir Figure II-19) permet de spécifier l'ordre d'exécution d'un un nombre quelconque de sous-diagrammes appelés séquences. Chaque séquence sera numérotée pour que la séquence «0» s'exécute en premier, puis la séquence «1», la séquence «2» et ainsi de suite. Pendant l'exécution, la communication entre la séquence «i» et les séquences d'après (séquence «i+1», …) peut être implémentée facilement grâce à une variable

locale ajoutée sur le bord vertical de la structure dans le cas de séquence empilée, ou grâce à un point de connexion ajouté sur le bord vertical commun entre deux séquences consécutives (par exemple, la séquence «i» et la séquence «i+1») dans le cas de séquence déroulée.

- Structure Condition (*Case*): (voir Figure II-19) permet de conditionner l'exécution d'un ou plusieurs sous-diagrammes, dont un exactement s'exécute quand la structure s'exécute. Chaque sous-diagramme correspond à une condition déterminée par la valeur câblée au terminal du sélecteur de condition. Elle peut être de type booléen, chaîne de caractères, entier ou énumération.

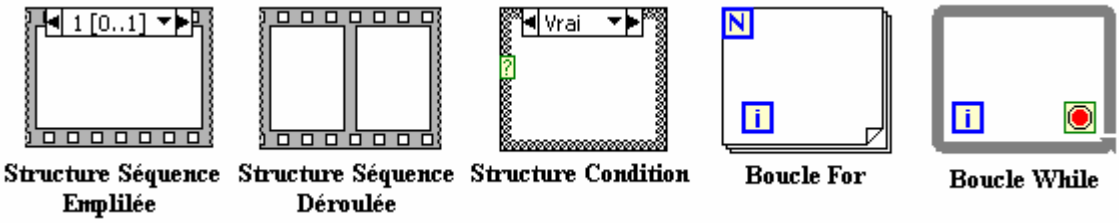


Figure II-19: Structures de contrôle dans LabVIEW

- Boucle Pour (*For*): (voir Figure II-19) permet d'exécuter le sous-programme à l'intérieur de la boucle n fois, où n est la valeur câblée au terminal «N» de décompte de la boucle. Le terminal d'itération «i» fournit le numéro d'itérations actuel de la boucle, qui varie de 0 à n-1.
- Boucle Tant Que (*While*): (voir Figure II-19) permet de répéter le sous-diagramme à l'intérieur de la boucle jusqu'à ce que le terminal conditionnel d'entrée reçoive une valeur booléenne particulière selon la condition de la continuation définie pour cette boucle. Le terminal d'itération «i» fournit le numéro d'itérations actuel de la boucle, qui vaut zéro à la première itération. La boucle Tant Que s'exécute toujours au moins une fois, ce qui fait que c'est plutôt une boucle Repeat/Until (Répéter/Jusqu'à) ou Do/While (Faire/Tant que).

Pour la boucle «Tant Que» et la boucle «Pour», on peut mémoriser des valeurs calculées lors des différentes itérations de la boucle en utilisant des mémoires locales, appelées registres à décalage. Les registres à décalage sont créés sur les bords verticaux de la structure à l'aide de la fonction «Ajouter un registre à décalage». Afin de récupérer la valeur d'une variable calculée à l'itération «k-1» dans l'itération «k», il faut qu'à l'itération «k-1» on câble la valeur calculée de cette variable sur la partie droite de registres à décalage de la boucle, puis à la prochaine itération (l'itération «k») on la récupère sur la partie gauche (voir Figure II-20).

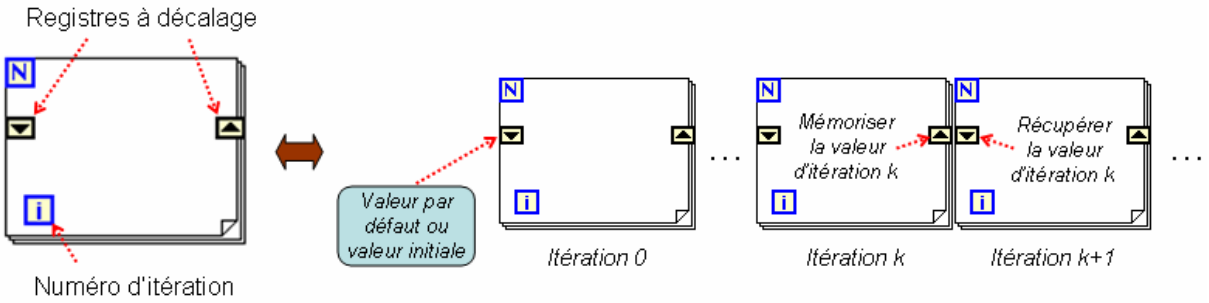


Figure II-20: Mémoire locale, registres à décalage

Il est important de noter qu'au début de l'exécution de la boucle (à l'itération «0»), si l'utilisateur ne fournit pas une valeur par défaut (ou valeur initiale) sur la partie gauche du registre à décalage de la boucle, il existe toujours une valeur par défaut que LabVIEW attribue pour chaque type de données (par exemple, «zéro» pour les entiers, «Faux» pour les booléens, etc.). Dans ce cas, le type de données dans les registres à décalage est déterminé selon la valeur câblée sur la partie droite de ces registres à décalage. Par conséquent, il est conseillé de déclarer une valeur initiale pour les registres à décalage lorsqu'ils sont utilisés dans la boucle «Tant Que» ou la boucle «Pour».

Toutes les icônes et tous les flots de données, placés à l'intérieur du cadre formant la structure de contrôle, sont considérés comme propres à la structure de contrôle et soumis à son évolution (itération, exécution conditionnelle et séquence) [Cot01]. Les entrées ou les sorties de données d'une structure de contrôle n'auront lieu qu'à des points de connexions, appelés tunnels qui sont installés sur les bords de cette structure (c'est-à-dire ses terminaux). Ainsi, chaque structure de contrôle ne s'exécutera pas tant que toutes les données ne sont pas fournies à ses terminaux d'entrée, et elle ne générera pas de données sur ses terminaux de sortie tant qu'elle n'aura pas fini son exécution. La Figure II-21 représente une communication directe entre deux boucles «Faire Tant Que». L'exécution de la boucle à droite est contrôlée par une valeur booléenne générée par la boucle à gauche qui ne s'arrête que lorsque le bouton «Arrêt» est appuyé. Dans ce cas là, la valeur booléenne n'est pas envoyée au terminal d'entrée de la deuxième pour déclencher son exécution tant que l'exécution de la première n'est pas terminée, et la valeur flottante aléatoire n'est pas affichée sur la Face Avant tant que la deuxième n'a pas commencé son exécution.

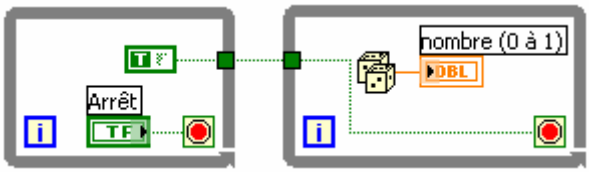


Figure II-21: Echange de données entre structures de contrôle

II.2.1.c. Le mécanisme de non-réentrance et de réentrance

Généralement, quand plusieurs occurrences générées à partir d'un nœud d'encapsulation sont utilisées dans un DFD, ces occurrences ne sont pas exécutées en parallèle, même si cela est possible (l'indépendance de flots de données provoquant chacune de ces occurrences). Donc, elles doivent s'exécuter les unes après les autres. C'est-à-dire que la première occurrence à avoir ses entrées fournies en données s'exécutera. Pendant cette exécution, si une autre occurrence voit également ses entrées fournies en données, alors elle est mise en attente dans une file d'attente (type FIFO, premier entré premier sorti) jusqu'à la fin d'exécution de l'occurrence en cours. Lorsque l'exécution en cours est terminée, l'occurrence mise au début de la file d'attente se réveille et commence son exécution. C'est le principe du mécanisme de non-réentrance, qui est le mécanisme utilisé par défaut sur les VI.

Le mécanisme de réentrance permet aux occurrences créées par un nœud d'encapsulation dans un diagramme flot de données de pouvoir s'exécuter parallèlement si les flots de données causant chacune de ces exécutions sont arrivés. En outre, chaque exécution aura lieu dans un espace propre associé à son occurrence. La Figure II-22 donne un exemple typique de la différence entre les deux mécanismes.

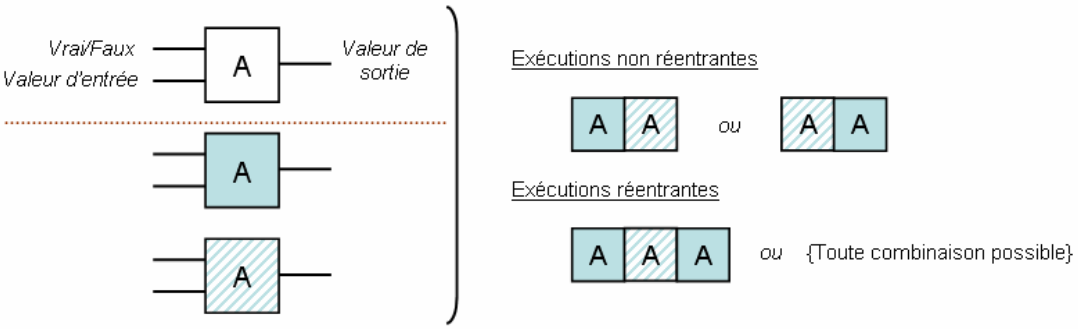


Figure II-22: Non-réentrance et réentrance

Dans cet exemple, un nœud d'encapsulation, nommé «A», est utilisé.

Il existe deux possibilités:

- Si A est un nœud non-réentrant, en fonction de l'ordre d'exécution de ces deux contextes, soit le premier contexte de A s'exécute d'abord et se termine, puis le deuxième s'exécute; soit le deuxième contexte de A s'exécute d'abord et se termine, puis le premier s'exécute.
- Si A est un nœud réentrant, pendant l'exécution du premier contexte (ou du deuxième contexte) de A, le deuxième contexte (ou le premier contexte) peut s'exécute en parallèle, nous ne pouvons pas déterminer l'ordre d'exécution exacte de ces deux contextes, qui dépend de l'ordonnanceur du système d'exploitation.

LabVIEW offre les deux mécanismes: la non-réentrance et la réentrance. Si un VI réentrant est requis, le changement d'un VI existant en mode réentrant est réalisé aisément grâce à la fenêtre «Exécution» dans la fenêtre «Propriétés du VI» du menu «Fichier» (voir Figure II-23). LabVIEW plaquant son exécution sur des tâches système, chaque VI est exécuté dans 1 (ou plusieurs) tâches et a donc son propre contexte d'exécution.

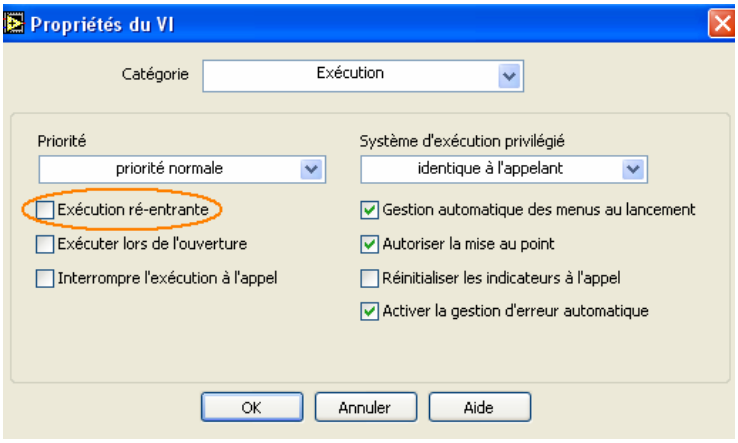


Figure II-23: Fenêtre «Exécution» pour la réentrance

II.2.1.d. Le polymorphisme dans LabVIEW

Le polymorphisme est une caractéristique autorisant un sous-programme à être utilisé avec différents types. En informatique, il y a deux types fondamentaux de polymorphisme: le polymorphisme paramétré et le polymorphisme par sous-typage (dérivation). Le polymorphisme paramétré consiste à définir des fonctions qui peuvent être appliquées à des types paramétrés. C'est-à-dire que dans ce cas-là, le développeur n'a pas à savoir les types précis des paramètres d'entrée sur lesquels ces fonctions vont s'appliquer. Il lui suffit de savoir que ces paramètres provoqueront leurs exécutions. La fonction permettant de concaténer deux listes quelque soit le type de données qu'elles contiennent est un exemple typique d'un tel type de polymorphisme. Alors que le polymorphisme par sous-typage repose sur l'idée de type abstrait, un principe fondamental pour les méthodes orientées objets.

Le concept de polymorphisme dans LabVIEW s'inspire du polymorphisme paramétré. Cependant il n'est pas vraiment semblable. Il est simplement une superposition des VIs ayant les mêmes connecteurs (chacun est utilisé avec un type de donnée déterminé). Le VI «Affichage» sur la Figure II-24 ci-dessous est un exemple. Ce VI regroupe les quatre VIs indépendants correspondant aux quatre types de base, le «Affichage Entier» pour le type Entier, le «Affichage Float» pour des flottants, le «Affichage Booléen» pour le type Booléen et le «Affichage Caractère» pour des caractères. Au début, il attend une entrée appartenant à un de ces types de base sur son arc d'entrée. S'il y en a, le VI polymorphe «Affichage» va le prendre, puis l'afficher à l'écran de l'utilisateur de manière à ce que la réception de l'entrée ne dépende pas de son type de donnée (il s'agit de polymorphisme de types).

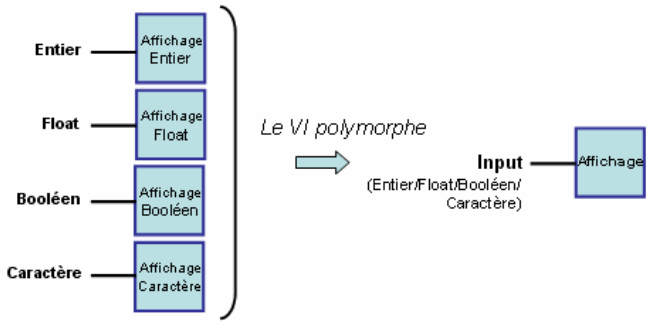


Figure II-24: Polymorphisme dans LabVIEW

Notons que depuis la version 8, LabVIEW offre la notion d'objet et d'héritage, ce qui entraîne l'existence du polymorphisme par sous-typage.

II.2.2. Outils pour le parallélisme dans LabVIEW

Le langage LabVIEW utilise le parallélisme implicite des diagrammes flot de données. Ainsi, les fonctions (les nœuds d'un diagramme LabVIEW) sont habituellement réservées à des programmations de type multitâche, ce qui leur permet de s'exécuter en concurrence d'après la règle des jetons (réf. II.2.1.a.). Pour les systèmes monoprocesseur, une seule fonction (ou un nœud) s'exécute à la fois. Selon un ordonnancement dynamique, à chaque instant l'ordonnanceur attribue le processeur à une tâche en fonction du contexte d'exécution et des autres tâches en attente. On obtient une exécution parallèle virtuelle. Notons que sur un processeur multicoeur on obtient du vrai parallélisme.

Pour simplifier et sans tenir compte des tâches synchrones, le système d'ordonnancement de LabVIEW est un système de partage de temps à tourniquet (Round-Robin), et à priorité (chaque priorité possédant sa propre file d'attente gérée en tourniquet avec un quantum de temps fixé (110 ms)). Un tel système peut reposer sur l'ordonnancement du système d'exploitation (les tâches LabVIEW utilisent des «threads» du système). En outre, LabVIEW offre six niveaux de priorité de bas en haut: Background Priority, Normal, Above Normal, High Priority, Time-Critical Priority, et Subroutine Priority. La priorité, appelée «Priorité Normale» (Normal Priority), est la priorité par défaut. La priorité, appelée «Sous-programme» (Subroutine Priority), est la priorité la plus élevée et en plus le VI ayant cette priorité devient synchrone (non préemptif par les autres VI LabVIEW).

Nous savons que le parallélisme inhérent à LabVIEW permet de programmer naturellement en multitâche: deux boucles indépendantes sont implémentées par deux tâches différentes, et peuvent donc s'exécuter en parallèle. Mais, pour échanger des données entres ces deux tâches de façon directe comme sur la Figure II-21, la deuxième tâche (la boucle à droite) ne s'exécute pas car elle se bloque en attendant les données fournies par la première tâche (la boucle à gauche), n'étant produites que lors de la fin de son exécution.

Pour répondre à des difficultés de synchronisation et de communication entre tâches indépendantes, LabVIEW offre des outils informatiques de communication et de synchronisation: Sémaphore, Boîte aux lettres, Rendez-vous, Evénement, etc. Ces outils se trouvent au niveau de la sous-palette «Synchronisation» située dans la sous-palette «Avancé» de la palette «Fonctions» (voir Figure II-25) et ont été apportés dans la version 5 de LabVIEW à partir des travaux de [Gev98].

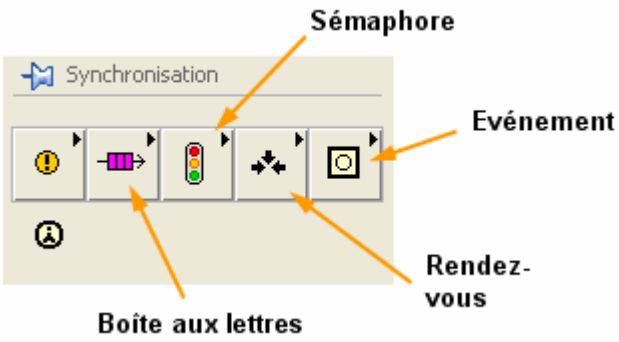


Figure II-25: Outils de synchronisation et de communication dans LabVIEW

II.2.2.a. Le sémaphore dans LabVIEW

Le sémaphore est un outil introduit par Dijkstra en 1965. L'un de ses rôles est de protéger une zone de programme dite critique lorsque le nombre de tâches qui accèdent simultanément à une ressource critique est supérieur à un nombre limité d'instances de cette ressource. Un sémaphore est composé typiquement de deux opérations, appelées «Prendre_Sémaphore P(S)» et «Vendre_Sémaphore V(S)». Quand une tâche veut accéder à une ressource protégée par un sémaphore, il faut que d'abord, elle acquiert une permission d'accès au sémaphore en appelant l'opération P(S). Elle se bloque et est mise dans la file d'attente du sémaphore s'il n'y a aucune instance libre du sémaphore. Sinon, elle prend le sémaphore, et peut continuer son exécution. Puis, à la fin de sa section critique, elle doit libérer le sémaphore par l'opération V(S), ce qui peut avoir pour effet de réveiller une tâche endormie présente dans la file d'attente de ce sémaphore qui obtient alors le sémaphore. Il existe deux types de sémaphore: Sémaphore binaire (ou sémaphore booléen) et Sémaphore non-binaire (ou sémaphore à compte). Dans le premier type, le sémaphore ne peut prendre que les valeurs libre (1) et pris (0).

Dans LabVIEW, les fonctions de gestion de sémaphore se trouvent dans la sous-palette «Sémaphore» de la sous-palette «Synchronisation». Elles servent d'outil de synchronisation des accès à une ressource partagée en exclusion mutuelle entre des VIs s'exécutant en parallèle. Un exemple est représenté sur la Figure II-26: deux boucles infinies indépendantes partagent une variable globale de manière que la boucle en haut y écrit une nouvelle valeur, et la boucle en bas lit la valeur la plus récente de cette variable. Dans ce cas là, afin de s'assurer l'atomicité de ces deux opérations, l'écriture et la lecture sur la variable globale, nous utilisons l'outil sémaphore de LabVIEW. Un sémaphore binaire est créé. Quand une boucle veut accéder à la variable globale, elle doit acquérir le sémaphore. S'il est libre, elle le prend, et effectue son opération sur cette variable (soit l'écriture, soit la lecture). Si pendant le temps d'exécution de la première boucle dans sa section critique, la deuxième boucle tente de prendre le sémaphore, elle se bloque jusqu'à ce que la première libère le sémaphore.

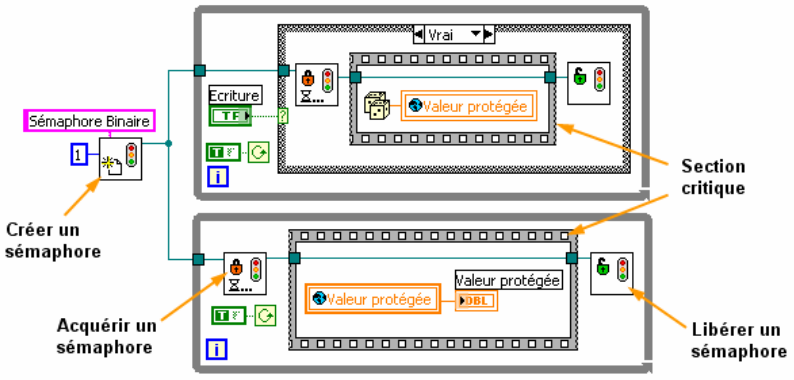


Figure II-26: Accès à une section critique en exclusion mutuelle

L'outil sémaphore dans LabVIEW peut être également utilisé pour synchroniser des tâches. Avec un sémaphore binaire, la synchronisation des tâches se fait de façon suivante: lorsque la valeur du sémaphore est nulle, la tâche qui appelle la fonction «Prendre_Sémaphore» doit attendre jusqu'à ce que l'opération «Rendre_Sémaphore» ait lieu. Une application simple de ce type de sémaphore est donnée sur la Figure II-27 (dans la littérature on parle de sémaphore privé).

Dans cette figure, à gauche une représentation schématique de la synchronisation entre deux tâches est présentée: la tâche «Tâche 2» ne s'exécute qu'après un tir de l'occurrence associé à la tâche «Tâche 1» pour libérer le sémaphore acquis par la première. A droite, une implémentation de cette synchronisation sous LabVIEW version 6i est donnée: deux boucles infinies représentent les deux tâches telles que la boucle en haut implémente la tâche «Tâche 1», et celle en bas implémente la tâche «Tâche 2». Un sémaphore avec une valeur initiale étant mise à 0, nommé «Sémaphore zéro», sert à synchroniser l'exécution de ces deux boucles (il est important de noter que le sémaphore dans LabVIEW version 6i est un sémaphore non-borné: un sémaphore peut avoir une valeur supérieure à sa valeur initiale). Au début, puisque la valeur du sémaphore est nulle, la boucle du bas se bloque en attendant la libération du sémaphore par la boucle du haut. A l'instant où l'on appuie sur le bouton «V (S)», la primitive «Vendre_Sémaphore» est exécutée. En conséquence, la boucle du bas se réveille pour exécuter une itération, avant de se rendormir en attente du sémaphore. Comme la boucle du bas ne peut pas se déclencher avant la boucle du haut, la contrainte de précédence entre ces deux boucles est satisfaite.

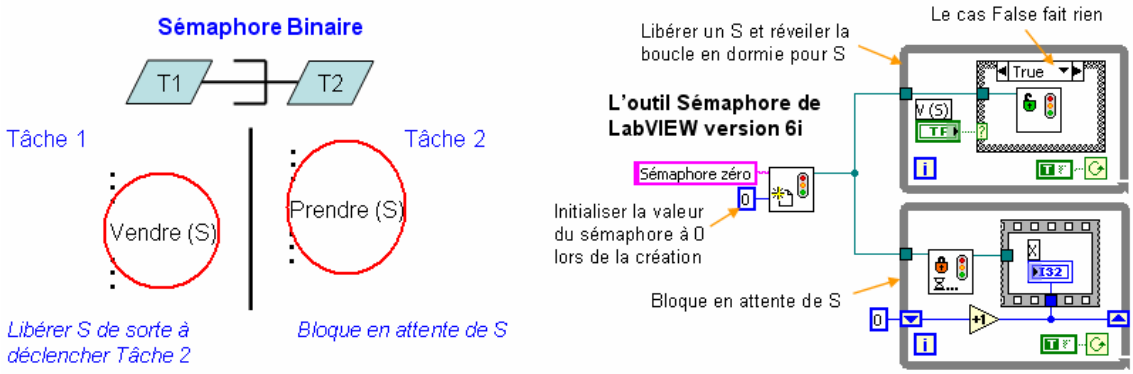


Figure II-27: Synchronisation de deux tâches par l'outil Sémaphore de LabVIEW 6i

Cependant le sémaphore proposé dans la version 7.1 de LabVIEW est un sémaphore spécial dont la valeur est bornée par la valeur initiale. Lors de la libération d'un sémaphore, LabVIEW vérifie que sa valeur n'est pas supérieure à sa valeur initiale. Donc, il ne permet pas d'initialiser le nombre d'accès simultanés autorisés à 0 dès que l'on crée un sémaphore. Notons que ce choix d'implémentation est très discutable et nous a particulièrement supris.

La Figure II-28 représente un cas simple de synchronisation de deux tâches par l'outil Sémaphore de LabVIEW version 7.1 (dans ce cas la valeur initiale du sémaphore est égale à 1). Il est important de remarquer que l'on utilise un «chien de garde», qui est intégré dans la fonction «Acquérir un sémaphore», pour protéger le modèle contre le cas contraire, qui peut entraîner un blocage fatal à l'exécution. Car si la taille du sémaphore est égale à 1 et la boucle du haut test, mais interrompue par la boucle du bas, qui prend le sémaphore, alors la boucle du haut se bloque toujours dans ce cas-là en attendant le sémaphore (le cas «Faux» de la structure conditionnelle à l'intérieur). Donc, grâce à un «timeout», si ce cas a lieu, la boucle du haut attend jusqu'à un certain temps déterminé par le «timeout» (par exemple, 100 ms), et puis continue son exécution sans perturber la synchronisation entre ces deux boucles.

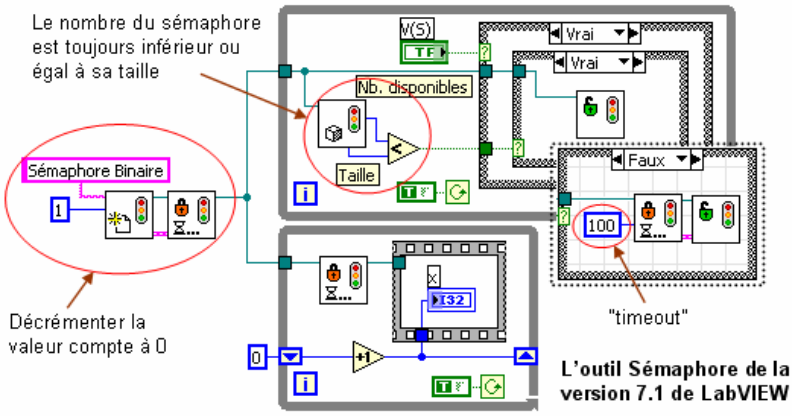


Figure II-28: Synchronisation des tâches par sémaphore de LabVIEW v7.1

Ainsi, il est possible de créer un sémaphore initialisé à n, puis prendre le sémaphore de sorte à réduire sa valeur à 0. Une telle implémentation de sémaphore sera présentée aux paragraphes suivants de ce chapitre.

II.2.2.b. Communication par message dans LabVIEW

La communication est le passage d'information d'une tâche à une autre [Gev98]. Il s'agit de synchroniser des opérations des tâches en échangeant des données. Dans LabVIEW, la communication par message peut être implémentée soit par les variables partagées (Variables globales pour la communication asynchrone), soit par le passage de messages grâce à une file d'attente (Boîte aux lettres), soit en utilisant le concept de VI non-réentrant et les registres à décalage. En fait, il existe beaucoup d'autres façons pour la communication par message à distance telles TCP, UDP, etc., qui débordent des travaux de ce manuscrit.

- ***Les variables globales*** de LabVIEW permettent d'accéder à des données et de les transmettre dans plusieurs VIs. Une variable globale est un nœud d'encapsulation des données pouvant être utilisé dans plusieurs sous-diagrammes ou dans plusieurs boucles (un outil de communication asynchrone dans LabVIEW). En fonction de la direction des données, une variable globale peut être configurée en écriture ou en lecture. Remarquez que le parallélisme dans LabVIEW autorise des boucles indépendantes à s'exécuter en concurrence. Les deux opérations, l'écriture et la lecture, sur une variable globale doivent être effectuées en exclusion mutuelle. L'utilisation conjointe de la variable globale et de l'outil Sémaphore peut résoudre ce problème (voir Figure II-26). Cependant lorsque nous lisons depuis une variable globale, LabVIEW crée une copie des données stockées dans cette variable globale. Pour des grands tableaux ou chaînes, le temps et la mémoire nécessaire pour manipuler les variables globales peuvent être considérables. En outre, si la variable globale est lue à plusieurs endroits dans l'application, plusieurs zones de mémoire tampon sont créées, ce qui ralentit les performances.

- ***Communication par VI non-réentrant***

  Le concept de VI non-réentrant et les registres à décalage peuvent être utilisés conjointement afin d'échanger des données entre plusieurs VIs. Il s'agit d'un outil de communication asynchrone. Mais par rapport à la communication par des variables globales, grâce au mécanisme de non-réentrance, ce type de communication permet de gérer des accès simultanés aux données partagées en exclusion mutuelle d'une manière efficace. La Figure II-29 représente un modèle du lecteur/écrivain simple implémenté par le concept de la non-réentrance de LabVIEW: une boucle «Faire Tant Que» qui ne s'exécute qu'une seule fois avec les registres à décalage est utilisée (on l'utilise en fait uniquement en tant que support syntaxique du registre à décalage); l'idée est la suivante : on n'initialise pas le registre, par conséquent, la valeur inscrite la dernière fois a été conservée (lors du premier appel c'est la

valeur par défaut du type qui est contenue). A chaque fois que ce VI est appelé, on inscrit dans le registre la valeur courante. Elle est donc disponible au prochain appel.

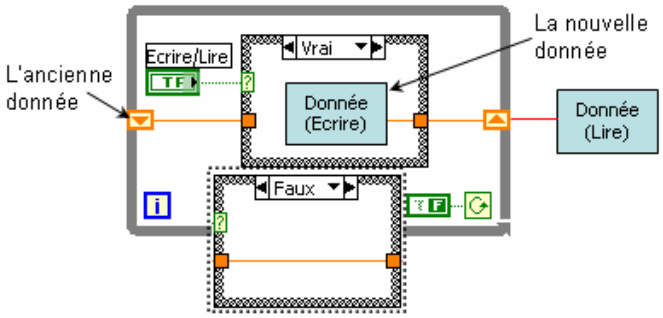


Figure II-29: Communication par VI non-réentrant

- ***Boîte aux lettres (BAL) dans LabVIEW***

Une boîte aux lettres (BAL) est une zone d'échange dans laquelle une tâche, dite «productrice» dépose des données et une autre tâche, dite «consommatrice» retire ces données dans l'ordre d'arrivée (ou suivant une priorité des messages). Du point de vue de l'utilisateur, nous trouvons les outils de gestion de la boîte aux lettres dans LabVIEW au niveau de la sous-palette «File d'attente» de la sous-palette «Synchronisation» (voir Figure II-30).

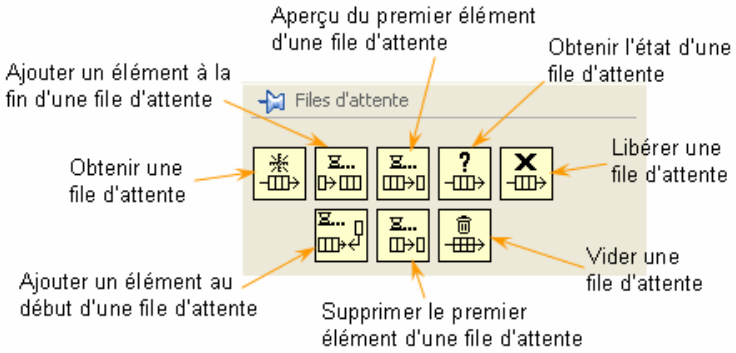


Figure II-30: Boîte aux lettres dans LabVIEW version 7.1

La lecture des données dans la file d'attente, dont la taille est programmable, est protégée en accès. Une lecture n'est possible que s'il y a au moins un élément inséré dans la fille d'attente, et l'écriture n'est possible que s'il y a une place de libre dans la fille d'attente. La boîte aux lettres de LabVIEW par défaut est bornée et bloquante en écriture (lorsque la BAL est pleine) et en lecture (lorsque la BAL est vide). La Figure II-31 illustre un exemple de la communication entre deux boucles par l'outil BAL. Une BAL de taille 1 est produite grâce à la fonction «Obtenir une file d'attente». Puisqu'il n'y a pas de donnée dans la BAL au début, la boucle du bas est mise en attente jusqu'à ce qu'une donnée soit déposée par la boucle du haut. Mais la boucle du haut n'a pas besoin d'attendre que celle du bas soit à l'écoute pour lui envoyer des données.

Notons que dans la version de LabVIEW 6i, pour envoyer un message à la file d'attente, il faut qu'on la convertisse d'abord en chaîne de caractères grâce à la fonction «Aplatir en chaîne» dans la sous-palette «Manipulation de données» de la sous-palette «Avancé». Puis, afin de récupérer le message initial à partir de la chaîne de caractères lors de la lecture, on effectue une opération inverse en appelant la fonction «Redresser à partir d'une chaîne». Mais, s'il y a incohérence dans les types de données, « Redresser » ne fonctionne pas correctement, et génère la valeur par défaut du type. En utilisant les trois VIs polymorphes réentrants, un pour la file d'attente et deux pour l'émettrice et la réceptrice, nous pouvons faire indépendant le transfert de messages d'une file d'attente par rapport aux types de

donnée. Cependant le polymorphisme de LabVIEW nécessite une superposition des VIs ayant les mêmes connecteurs (réf. II.2.1.d.). Ce modèle est donc trop dépendant du type.

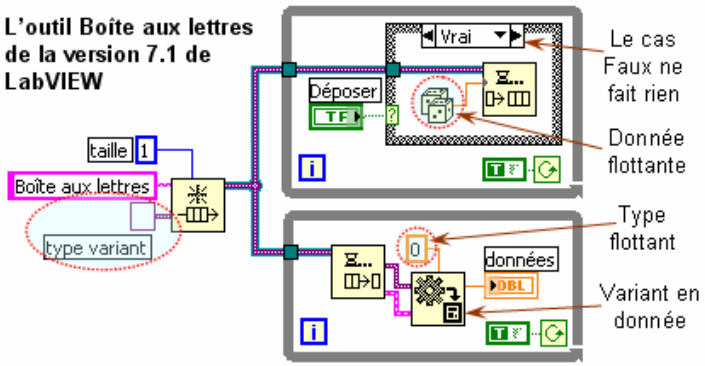


Figure II-31: Communication par File d'attente de LabVIEW version 7.1

La communication par BAL sous LabVIEW 7.1 est plus simple que celle de l'ancienne version. Avec le type variant, n'importe quel type de message peut être envoyé à la BAL lors de l'écriture. Lors de la lecture, afin de récupérer le message du bon type, il faut câbler le type de donnée à la fonction «Variant en donnée», qui se trouve dans la sous-palette «Variant» de la sous-palette «Manipulation de données» (voir Figure II-31). Si ce type n'est pas cohérent, une erreur est levée (LabVIEW vérifie donc la cohérence du type de l'élément reçu par rapport au type attendu). La compatibilité des types de données est toujours assurée (contrairement au cas des versions précédentes).

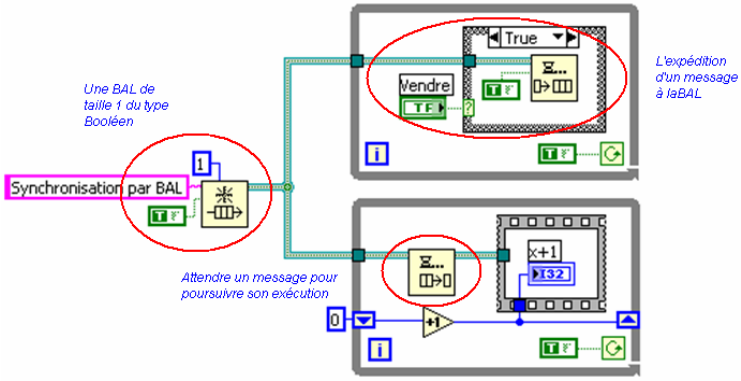


Figure II-32: Synchronisation par BAL de LabVIEW version 7.1

Lorsque la file d'attente est vide, la boucle «Consommatrice» ne s'exécute qu'après l'arrivée d'un message généré par la boucle «Productrice», ce qui implique une contrainte de précédence entre ces deux boucles. La synchronisation des tâches par sémaphore (réf. II.2.2.a.) peut être remplacée par l'outil «Boîte aux lettres». Dans ce cas, on utilise une BAL, de taille 1 avec un message du type booléen servant de signal de déclenchement. Donc, la libération du sémaphore est remplacée par l'expédition du message à la BAL (voir Figure II-32). S'il y a plusieurs boucles «Consommatrice» en attente, elles sont gérées par une file d'attente du type FIFO.

Comme la vitesse d'exécution des boucles «Productrice» et «Consommatrice» est réglable de façon indépendante grâce aux gestions de temps qui seront présentées dans les paragraphes suivants de ce chapitre, il y a deux cas possibles: soit la «productrice» va trop vite et attend la «consommatrice» dès que la BAL est pleine, soit la «productrice» va trop lentement et la «consommatrice» attend. LabVIEW n'offrant que les boîtes aux lettres sans écrasement, il est intéressant de savoir comment on peut implémenter des boîtes aux lettres avec écrasement. La Figure II-33 représente une implémentation de la communication par BAL avec écrasement de taille 1 dans LabVIEW version 7.1. Cette BAL est non bloquante en écriture (l'ancien

message sera écrasé par le nouveau lorsque la BAL est pleine). De même que pour la BAL classique, nous utilisons le type variant. En écriture, la BAL est vidée, puis le nouveau message est ajouté. A la lecture, si la BAL est vide, la boucle «consommatrice» se bloque jusqu'à l'arrivée d'un nouveau message.

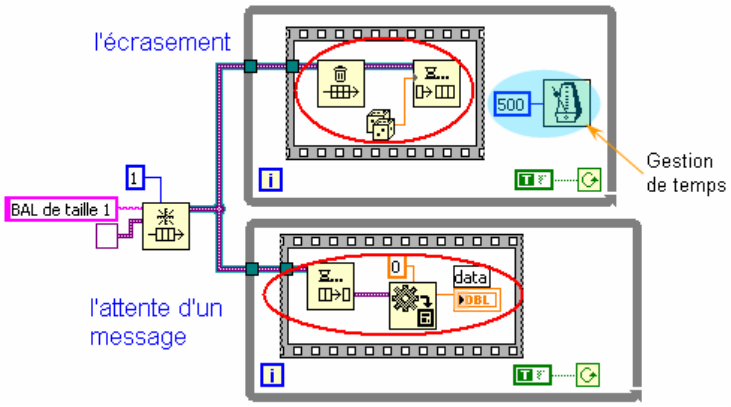


Figure II-33: Communication par BAL à écrasement dans LabVIEW version 7.1

II.2.2.c. Rendez-vous et Evénement dans LabVIEW

LabVIEW propose des outils de gestion de Rendez-vous et d'Evénement. Ces outils sont situés dans la sous-palette «Rendez-vous» et la sous-palette «Occurrence» de la sous-palette «Synchronisation» (voir Figure II-34).

En Ada, un rendez-vous est une communication synchrone asymétrique des tâches avec passage de données, où une tâche qui demande le rendez-vous attend qu'une autre tâche accepte le rendez-vous. Lorsque le rendez-vous est établi, la tâche qui a accepté le rendez-vous exécute le code et la tâche demandeur attend que ce code soit terminé avant de continuer seule. Dans LabVIEW, un rendez-vous est simplement un objet permettant à plusieurs tâches d'attendre en un point donné dans leur exécution comme une barrière dans la norme POSIX. Ainsi, si trois tâches sont liées par un rendez-vous, ces tâches doivent s'attendre en un point avant de reprendre le cours de leur exécution. Si une tâche arrive au point de rendez-vous avant les deux autres, elle se bloque jusqu'à ce que les deux dernières soient arrivées au point de rendez-vous.

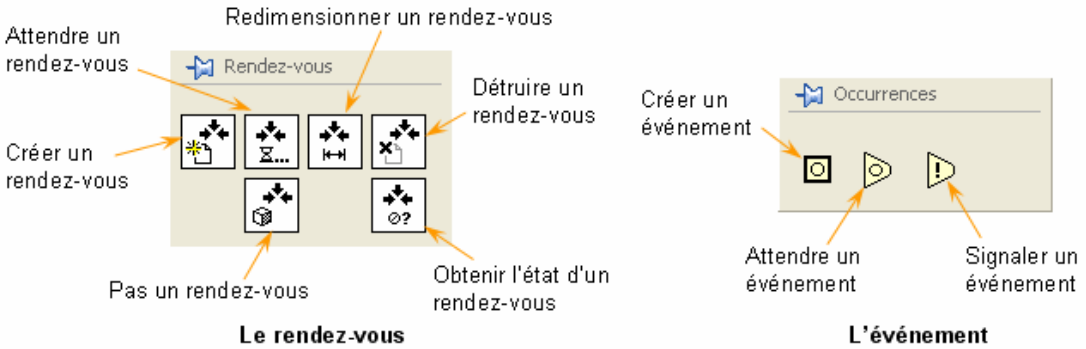


Figure II-34: Gestion de Rendez-vous et Evénement dans LabVIEW

L'événement permet la synchronisation par signaux. Contrairement à la communication par boîte aux lettres, il n'y a pas d'échange de données lorsque des tâches communiquent par événement. Donc, à certains moments de la vie de l'application, des tâches peuvent émettre ou attendre des événements. Les primitives de gestion d'événement sont «Créer un événement», «Attendre un événement», et «Signaler un événement» (voir Figure II-34). Cependant la communication par événement présente quelques inconvénients: soient deux tâches τ_1 et τ_2, telles que τ_1 génère un événement et que τ_2 attende cet événement. Pour les événements non mémorisés, le problème se

pose lorsque l'instruction «Signaler un événement» de τ_1 est exécutée avant l'instruction «Attendre un événement» de τ_2; τ_2 doit alors rester en attente jusqu'à ce qu'un nouvel événement soit signalé (voir Figure II-35). Par conséquent, l'ordonnancement des tâches influe sur leur comportement car si τ_1 s'exécute avant τ_2, τ_2 devra attendre le prochain événement signalé (dans le cas contraire, τ_2 pourra s'exécuter sur l'événement signalé). Pour les événements mémorisés, ce problème devient beaucoup plus compliqué [Gev98].

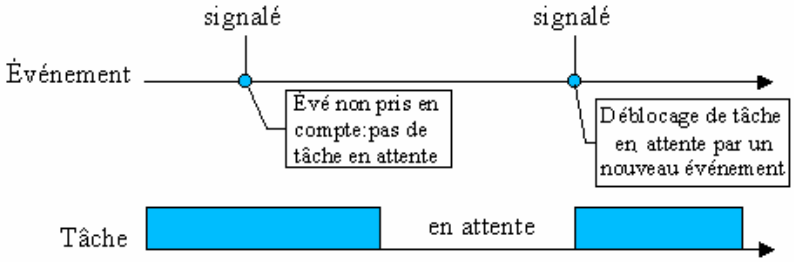


Figure II-35: Evénement non mémorisé: cas où l'événement a lieu avant la mise en attente

Comme l'ordonnancement influe sur le comportement des tâches utilisant les événements, ils sont peu utilisés dans les langages asynchrones bien que très utilisés dans les langages synchrones. Dans LabVIEW, pour synchroniser l'opération des tâches, nous choisirons donc l'outil «Sémaphore» ou l'outil «File d'attente» au lieu de l'événement.

II.2.2.d. La gestion du temps

Deux VIs permettant de gérer le temps sont fréquemment utilisées dans LabVIEW. Ce sont «Attendre (ms)» et «Attendre un multiple de ms».

- *Attendre un multiple de ms*: permet d'attendre que le prochain multiple de la durée spécifiée se soit écoulé. C'est-à-dire que quand cette fonction est appelée, elle va finir lorsque la valeur du temps noyau sera un multiple de la valeur du paramètre d'entrée «n millisecondes» de cette fonction.
- *Attendre (ms)*: permet d'attendre pendant au moins la durée spécifiée en millisecondes. Par exemple, «Attendre (ms)» est déclenchée à 12 ms, et la valeur du paramètre d'entrée «attente (ms)» est égale à 10 ms. Son exécution sera terminée au plus tôt à 22 ms.

La Figure II-36 montre les différences de comportement entre «Attendre (ms)» et «Attendre un multiple de ms»: trois boucles infinies représentent trois tâches avec des temps d'exécution différents. Dans le cas (a), la période de la boucle est au moins la somme du temps d'attente de la fonction «Attendre (ms)» et la durée d'exécution de la tâche A ($\Rightarrow$ Période(a) $\geq$ 1000 + 250 = 1250 ms). Néanmoins, dans (b), comme l'attente et l'action ont lieu en parallèle, la période est égale au moins au maximum du temps de la fonction «Attendre (ms)» et de la durée-exécution de A ($\Rightarrow$ Période(b) $\geq$ max (1000 ms, 250 ms)). Enfin, la période de (c) est égale à la valeur du paramètre d'entrée de la fonction «Attendre un multiple de ms» ($\Rightarrow$ P(c) = 1000 ms).

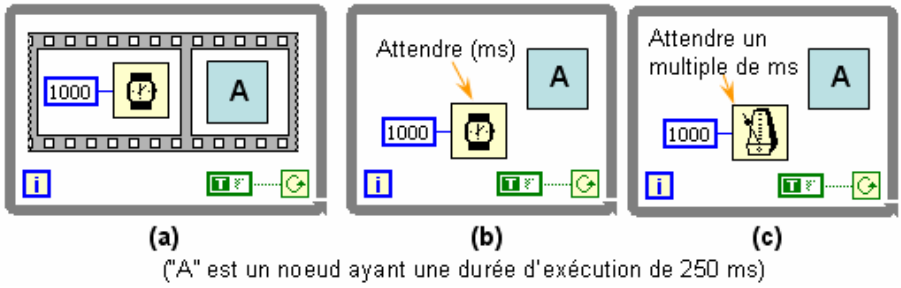


Figure II-36: Utilisation de «Attendre (ms)» et «Attendre un multiple de ms»

En pratique, un ou plusieurs circuits programmables spécialisés dans la gestion du temps sont utilisés. Ils peuvent être programmés de sorte à:

```
----------------------------------------
-- Une tâche périodique
-- Déclaration de la tâche
DEBUT
FAIRE TOUJOURS
-- des actions
Attendre une durée égale à une période
FAIT
FIN TACHE
----------------------------------------
```

Soit t l'instant où la tâche est activée, il lui faut un temps non nul pour effectuer les actions, d'autant que d'autres tâches peuvent utiliser le processeur et la retarder. L'instruction d'attente d'un délai a donc lieu à une date ($t + \varepsilon_1$), la tâche n'est pas réveillée qu'au plus tôt à la date (t + période + ε_1). Il lui faut ensuite un temps ε_2 avant d'arriver à l'instruction d'attente, ce qui décale son prochain réveil au plus tôt à la date (t + période + ε_1 + ε_2). Et ainsi de suite à chaque période. Ce phénomène s'appelle la «dérive des horloges». Par conséquent, une tâche périodique se doit de reposer sur un réveil par date, et non par délai. Ainsi, dans LabVIEW pour synchroniser l'exécution périodique de plusieurs boucles tournant en parallèle, la fonction «Attendre un multiple de ms» est utilisée.

A partir de LabVIEW 8, on trouve des boucles cadencées sur une horloge que l'on programme. Ces boucles sont extrêmement puissantes et permettent non seulement de gérer la cadence mais aussi de surveiller les échéances de la boucle, de surveiller finement le temps, ou de choisir le processeur et la priorité pour la boucle (voir Figure II-37).

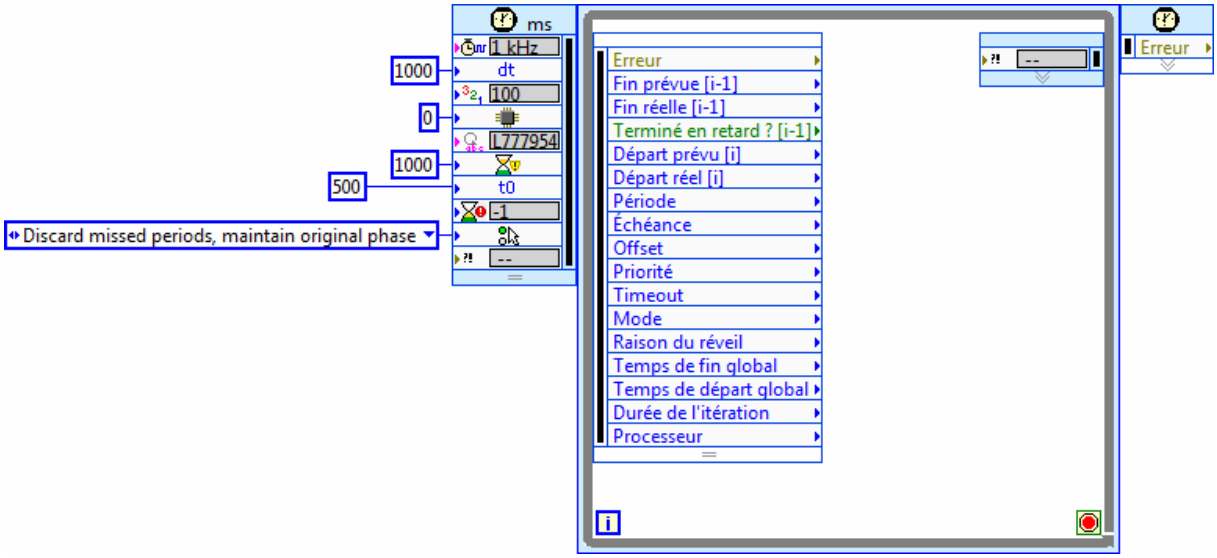


Figure II-37: Boucle cadencée à partir de LabVIEW 8

II.2.3. LabVIEW, un outil d'implémentation des concepts DARTS

Les concepts DARTS décrits dans le chapitre I (réf. I.5.4.) et aux paragraphes en dessus peuvent être implémentés en choisissant un langage de programmation approprié. En fait, les normes temps réel basées sur C et Ada sont capables de le faire. Avec Ada, en associant ce langage à DARTS, Gomaa propose une nouvelle méthode, appelée ADARTS [Gom93, chapitre 17]. En

utilisant le langage C, nous pouvons implémenter les éléments DARTS d'une manière conforme à des normes temps réel basées sur ce langage (par exemple, norme POSIX, norme OSEK/VDX, norme ARINC, bibliothèque propriétaire VxWorks, etc.). Mais, étant les langages de programmation textuels, ils ont une complexité de mise en œuvre plus importante que LabVIEW notamment pour réaliser les tests, ou faire une simulation. En outre, il faut définir sur chacun de ces langages des paquetages, des fonctions, ou des classes qui servent de bibliothèque des primitives prédéfinies.

LabVIEW est un langage de programmation graphique conçu pour le contrôle, l'analyse et la présentation de données. Par rapport à des langages de programmation textuels, LabVIEW permet non seulement d'implémenter les concepts DARTS, mais également de construire une bibliothèque reprenant les notations de la méthode DARTS. En conséquence, le programmeur n'a pas à écrire une seule ligne de code informatique: une fenêtre remplace l'éditeur de texte, des icônes remplacent les sous-programmes, des lignes et des flèches contrôlent le traitement des données. Des applications peuvent être construites avec des interfaces utilisateurs très interactives et des fonctionnalités complexes.

A la suite des principes de programmation graphique, ou bien des fonctionnalités graphiques offertes par LabVIEW, dans les paragraphes suivants je présenterai le modèle d'implémentation des concepts DARTS dans LabVIEW. Notons que la version de LabVIEW utilisée est LabVIEW version 7.1.

II.2.3.a. Implémentation des tâches DARTS

Dans DARTS, une tâche est représentée par un parallélogramme. Elle peut être soit une tâche matérielle, soit une tâche logicielle.

- *Implémentation des tâches matérielles*: (voir Figure II-38) une tâche matérielle peut être déclinée en tâche périodique ou apériodique. En cas de tâche périodique activée par une horloge temps réel, on l'implémente dans LabVIEW par une boucle «Tant Que» s'exécutant en mode infini. L'activation périodique de la boucle est manipulée par la primitive de gestion de temps «Attendre un multiple de ms» avec une valeur du type U32 (nombre entier non signé 32 bits) spécifiant la période de la tâche en millisecondes.

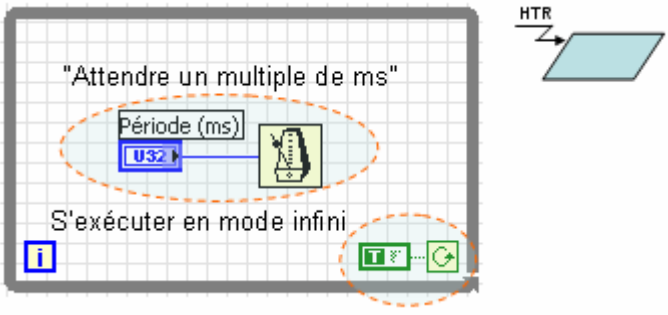


Figure II-38: Modèle de tâche périodique dans LabVIEW

Pour les tâches périodiques à contraintes strictes, chaque occurrence de ces tâches s'associe à un délai critique, il s'agit d'une échéance que le système doit impérativement respecter. Dans ce cas afin de garantir le respect des échéances, un chien de garde peut être utilisé. Un modèle simple de chien de garde dans LabVIEW est présenté sur la Figure II-39. Ce modèle se base sur la fonction «Compteur d'impulsion d'horloge (ms)» de la palette «Temps et Dialogue» de LabVIEW. Donc, si le temps de réponse de la tâche à chaque occurrence dépasse son délai critique, l'horloge de surveillance déclenche immédiatement une alarme et/ou le remplacement d'une unité fonctionnelle par une autre.

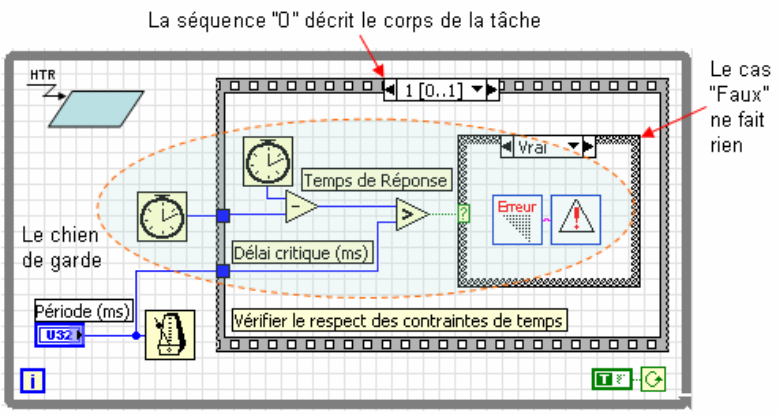


Figure II-39: Modèle de tâche périodique à contrainte stricte et à échéance sur requête dans LabVIEW

Ces deux cas de boucles périodiques ont été avantageusement remplacés à partir de LabVIEW 8 par la boucle cadencée (voir Figure II-37).

Pour les tâches matérielles apériodiques activées par des événements internes ou externes provenant du procédé, elles peuvent être implémentées dans LabVIEW de manière analogique les tâches matérielles périodiques, sauf le déclencheur: la gestion de temps par horloge est remplacée par l'attente de l'arrivée d'un déclenchement (voir Figure II-40). Par conséquent, à chaque itération la tâche doit attendre un événement généré par l'opérateur, ou à partir d'un dispositif (i.e., une carte d'acquisition programmable) en vue de déclencher son exécution.

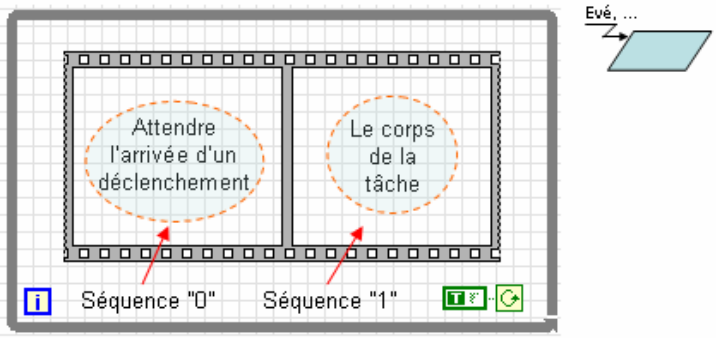


Figure II-40: Modèle de tâche matérielle apériodique dans LabVIEW

- *Implémentation de tâches logicielles*: (voir Figure II-41) contrairement aux tâches matérielles, les tâches logicielles sont activée par une autre tâche (soit matérielle, soit logicielle). Ainsi, l'implémentation de ces tâches dans LabVIEW peut être réalisée par la boucle «Tant Que» s'exécutant en mode infini en l'associant à l'un des outils de synchronisation.

  En effet, une boucle infinie qui consiste d'une structure de séquence: la première séquence (la séquence «0») contient l'attente d'événement (message ou synchronisation), et son corps est implémenté dans la deuxième séquence (la séquence «1»).

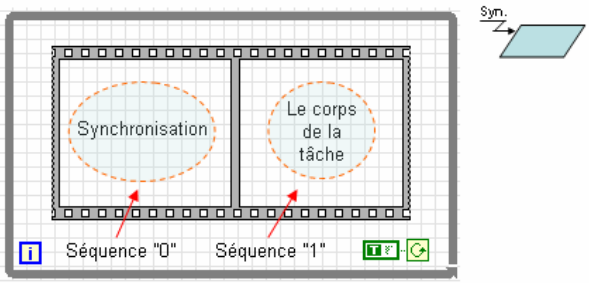


Figure II-41: Modèle de tâche logicielle dans LabVIEW

II.2.3.b. Implémentation de la synchronisation

Dans DARTS, la synchronisation peut être réalisée soit par l'envoi et la réception d'événements, soit par un outil de synchronisation. Dans LabVIEW, à cause des caractéristiques spécifiques du langage de programmation graphique flot de données, la synchronisation ne peut qu'être implémentée par les fonctions avancées dans la sous-palette «Sémaphore» de la palette «Synchronisation». Dans DARTSVIEW nous prévoirons 3 éléments pour la synchronisation : un modèle de création de synchronisation implémentée par sémaphore, un modèle de tâche déclenchante (émettrice), et un modèle de tâche déclenchée (réceptrice) Il est important de noter que le sémaphore est utilisé ici en tant qu'un outil de synchronisation.

- *Le VI de tâche «Emettrice»*: (voir Figure II-42) dans ce cas, c'est la tâche qui utilise le sémaphore en mode «Vendre Sémaphore». Une fois que cette tâche est activée, selon certaines conditions elle envoie un tir de l'occurrence à l'opération «V (S)» pour libérer une instance de sémaphore. Ceci est réalisé simplement en insérant à l'intérieur de la boucle infinie une structure de condition où la fonction «Libérer un sémaphore» se trouve dans le cas «Vrai». Si l'occurrence ne tire pas (le cas «Faux»), la tâche continue sa propre exécution jusqu'à la terminaison. Et puis, elle attend un autre déclenchement afin de s'exécuter de nouveau.

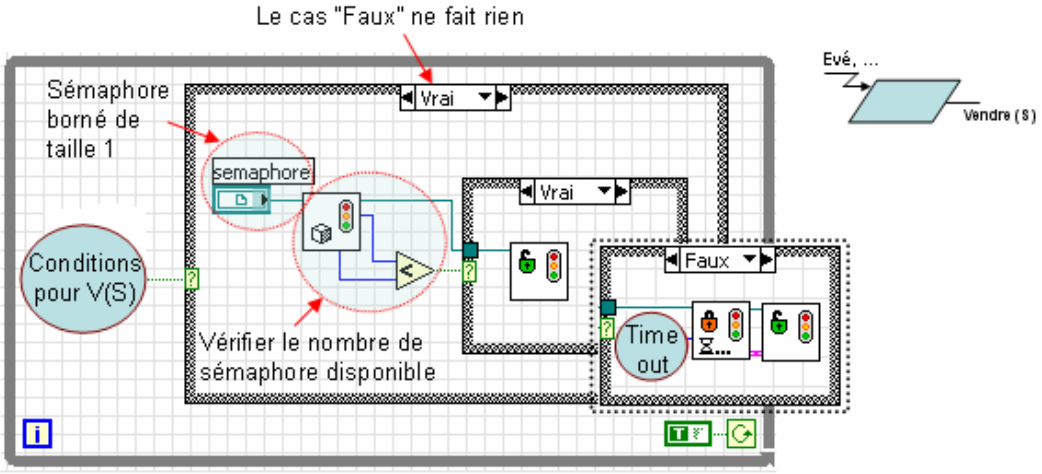


Figure II-42: Modèle de tâche «Emettrice» - Synchronisation dans LabVIEW

Comme je l'ai déjà expliqué, le sémaphore dans LabVIEW version 7.1 est un sémaphore avec une valeur initiale bornée. Il faut alors vérifier d'abord le nombre des instances disponibles (grâce à la fonction «Obtenir l'état d'un sémaphore») afin de s'assurer que ce nombre est toujours inférieur ou égal à sa taille. De plus, un «timeout» est utilisé pour borner l'attente de cette tâche au cas où une erreur à l'exécution se produit (réf. II.2.2.a.).

- *Le VI de tâche «Réceptrice»*: (voir Figure II-43) c'est une tâche logicielle. A chaque exécution, cette tâche est en attente du sémaphore. Lorsqu'une instance de sémaphore est libérée par la tâche «Emettrice», la tâche «Réceptrice» est réveillée et commence son exécution, qui est décrite dans la seconde partie de la séquence. Puisque la tâche «Réceptrice» ne s'exécute qu'après la tâche «Emettrice», la contrainte de précédence entre ces deux tâches est bien établie.

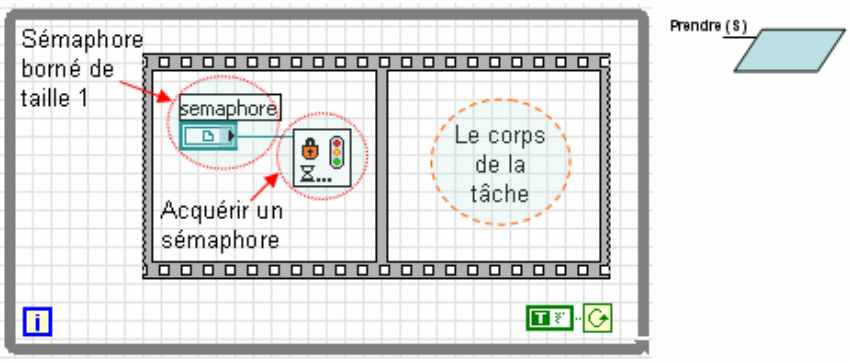


Figure II-43: Modèle de tâche «Réceptrice» - Synchronisation dans LabVIEW

- *Le VI pour le sémaphore*: (voir Figure II-44) le sémaphore utilisé dans ce VI est un sémaphore zéro de taille 1, car sa valeur compte est initialisée à 0 lors de la création du sémaphore. Alors, pour un sémaphore de taille quelconque en général, on décrémente la valeur compte n (étant égale à sa taille au début) à 0 en appelant n fois consécutives la fonction «Acquérir un sémaphore». Enfin, un connecteur de deux terminaux est construit pour relier les deux références de ce VI, appelées «Libérer Sémaphore» et «Acquérir Sémaphore», aux références appropriées du VI de la tâche «Emettrice» et du VI de la tâche «Réceptrice».

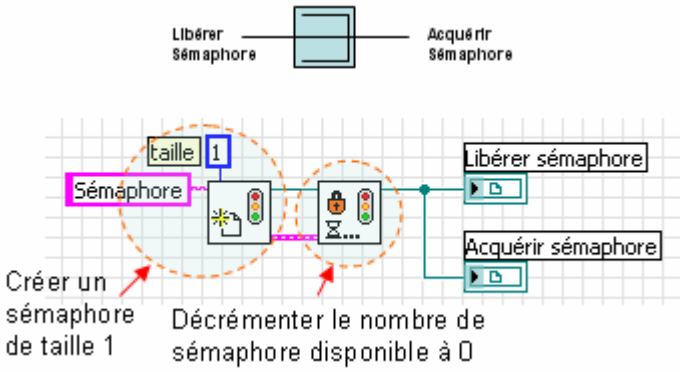


Figure II-44: Modèle de sémaphore – Synchronisation dans LabVIEW

II.2.3.c. Implémentation de la communication synchrone faiblement couplée

La communication synchrone faiblement couplée est représentée typiquement par la boîte aux lettres (BAL) classique. Ce type de communication est bloquant en écriture (lorsque la BAL est pleine) et en lecture, donc la tâche «Consommatrice» se bloque en attendant un message de la tâche «Productrice», tandis que la tâche «Productrice» peut se bloquer pour déposer un message lorsque la BAL est pleine. Trois éléments DARTS qui constituent une telle communication sont la tâche «Productrice», la tâche «Consommatrice» et la BAL. Dans LabVIEW, ils sont implémentés par trois VIs de façon suivante:

- *Le VI de tâche «Productrice»*: (voir Figure II-45) tâche «Productrice» peut être déclinée en tâche matérielle ou en tâche logicielle. Son rôle est d'expédier un message dans la boîte aux lettres quand elle est activée. Grâce au type Variant défini pour les messages d'une file d'attente de LabVIEW version 7.1, l'envoi d'un message peut être effectué indépendamment du type du message en reliant ce message au connecteur «Elément» de la fonction «Ajouter un élément à la fin d'une file d'attente» de la sous-palette «Files d'attente» (contrairement à la version 6.1: il fallait convertir le message en type chaîne de caractères avant de l'envoyer dans la file d'attente). Le modèle représenté sur la Figure II-45 est une implémentation simple de tâche «Productrice» dans LabVIEW en tant qu'une tâche matérielle périodique activée par une horloge.

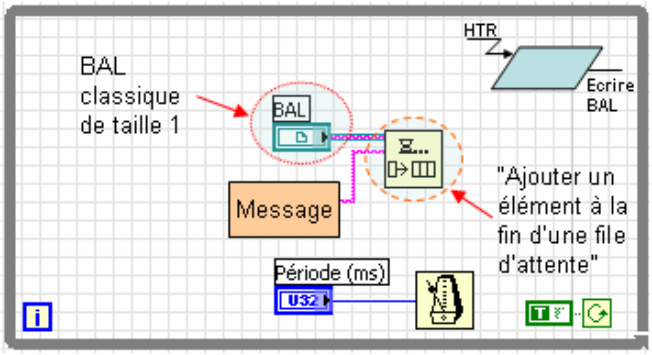


Figure II-45: Modèle de tâche «Productrice» périodique – Communication synchrone faiblement couplée dans LabVIEW

Le modèle de tâche «Productrice» apériodique est donné sur la Figure II-46. Dans ce modèle, la tâche «Productrice» au début de chaque itération se bloque en attente d'un événement (soit une interruption provenant d'un périphérique, soit un chien de garde) afin d'activer et de commencer son exécution (il s'agit de l'envoi d'un message dans BAL).

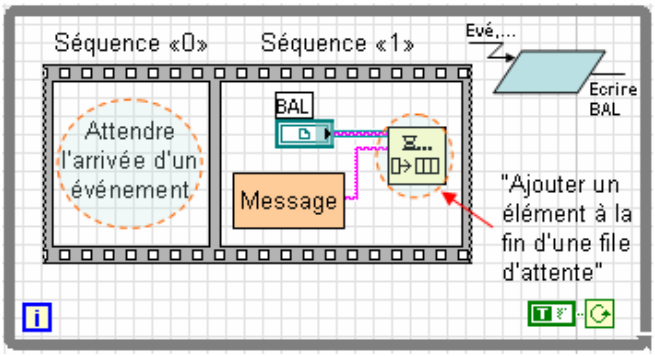


Figure II-46: Modèle de tâche «Productrice» apériodique – Communication synchrone faiblement couplée dans LabVIEW

Puisque la BAL est bloquante en écriture, le VI de tâche «Productrice» doit être utilisé avec prudence, notamment au cas où la vitesse d'expédition des messages dans BAL est supérieure à celle de réception des messages. Ainsi, dans certains cas, il est mieux de mettre l'expédition des messages à l'intérieur d'une structure de condition afin de diminuer le taux de l'envoi de messages dans BAL: par conséquent, à chaque activation de la tâche «Productrice» seulement les messages qui satisfont la condition sont envoyés à la BAL (voir Figure II-47).

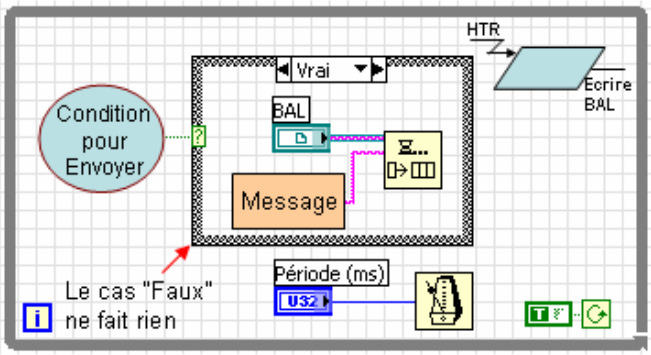


Figure II-47: Conditionner l'envoi d'un message dans le modèle de tâche «Productrice» - Communication synchrone faiblement couplée dans LabVIEW

- *Le VI de tâche «Consommatrice»*: (voir Figure II-48) tâche «Consommatrice» ne peut être qu'une tâche logicielle. Elle est activée par la tâche «Productrice» lorsqu'un nouveau message envoyé par celle-ci puisque la consommatrice est en attente sur la fonction «Supprimer le

premier élément d'une file d'attente» (la séquence «0»). Et puis, la tâche «Consommatrice» retire le message, le convertit en type de données correct, et l'utilise pour la suite de son exécution (la séquence «1»).

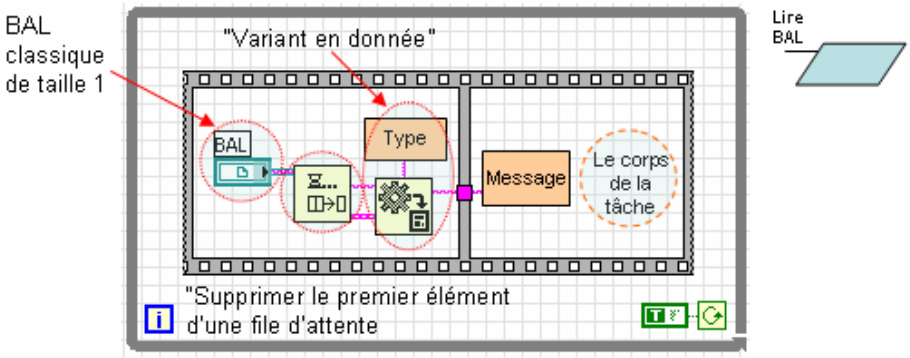


Figure II-48: Modèle de tâche «Consommatrice» - Communication synchrone faiblement couplée dans LabVIEW

Notons que pour récupérer un message du bon type à partir d'une BAL dans LabVIEW, il faut fournir le type de données sur le connecteur "Type" du VI "Variant en donnée". Ce type doit être cohérent avec le type de message stocké dans BAL, sinon une erreur s'est produite. Autrement dit, contrairement aux versions précédentes, LabVIEW 7.1 vérifie la cohérence du type de données de l'élément reçu par rapport au type de données attendu.

- *Le VI de la BAL classique*: (voir Figure II-48) une BAL classique dans ce modèle est une BAL de taille 1 créée par la fonction «Obtenir une file d'attente» de la sous-palette «File d'attente» de la palette «Synchronisation». Une valeur constante du type Variant est câblée sur le connecteur «Type de l'élément» de cette fonction, ce qui permet d'envoyer un message d'un type quelconque à la BAL. Enfin, une icône avec deux terminaux, nommés «Envoyer BAL» et «Recevoir BAL», est construite pour passer les références de la BAL aux deux tâches «Productrice» et «Consommatrice», qui l'utilisent en tant qu'un outil de communication synchrone faiblement couplée. Ainsi, le terminal «Envoyer BAL» relie la BAL au connecteur «Ecrire BAL» de la tâche «Productrice», tandis que le terminal «Recevoir BAL» relie la BAL au connecteur «Lire BAL» de la tâche «Consommatrice».

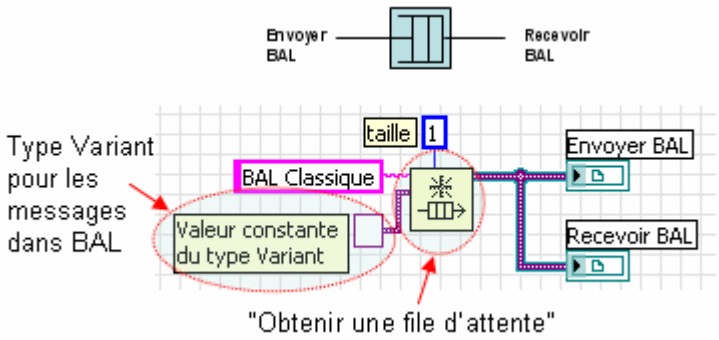


Figure II-49: Modèle de BAL Classique – Communication synchrone faiblement couplée dans LabVIEW

II.2.3.d. Implémentation de la communication synchrone faiblement couplée à écrasement

L'un des inconvénients de la boîte aux lettres classique (BAL) est que la tâche «Productrice» peut être suspendue lorsque la BAL est pleine. Une BAL_Ecr permet à la tâche «Productrice» d'écraser le message le plus ancien par celui le plus récent au cas où la file d'attente est pleine et un nouveau message est envoyé (on parle d'écriture non bloquante).

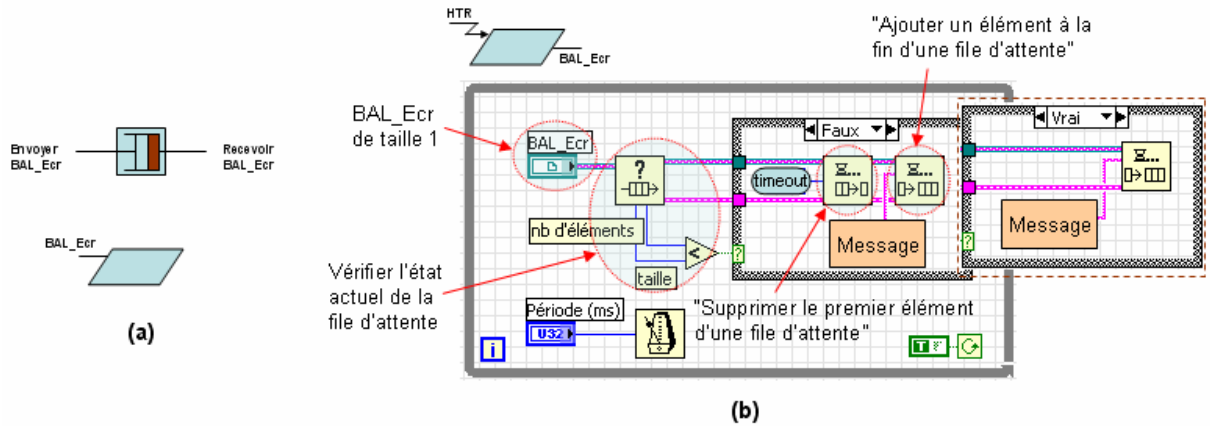


Figure II-50: Implémentation de la communication asynchrone faiblement couplée
(a) Notations de la BAL_Ecr et de la tâche «Consommatrice»
(b) Modèle de tâche «Productrice» en tant qu'une tâche périodique

Pour le VI de la tâche «Productrice», le mécanisme de l'écrasement peut être modélisé dans LabVIEW de la façon suivante (voir Figure II-50 (b)): une structure de Condition sert à vérifier l'état actuel de la file d'attente des messages à chaque fois que la tâche «Productrice» envoie un message; si la file d'attente des messages est pleine (le cas «Faux»), il faut supprimer le premier message (le plus ancien message) d'abord, et puis ajouter le plus récent message à la fin de la file d'attente. Un «timeout» est utilisé pour borner le temps d'attente de la tâche «Productrice» au cas où une erreur à l'exécution s'est produite, qui peut entraîner un blocage fatal de l'exécution de cette tâche (si la taille est égale à 1 et le test détecte un message existant dans la file d'attente des messages, mais à cet instant là, la tâche «Productrice» est interrompue par la tâche «Consommatrice, qui prend tout de suite ce message, alors la tâche «Productrice» reprend son exécution et se bloque toujours sur la fonction «Supprimer le premier élément d'une file d'attente»). Effectivement, lorsque la taille de la BAL_Ecr est égale à 1, l'utilisation de la fonction «Vider une file d'attente» au lieu de la fonction «Supprimer le premier élément d'une file d'attente» peut protéger le problème de blocage fatal ci-dessus. Cependant, le modèle de tâche «Productrice» présenté sur la Figure II-50 (b) est le modèle général pour une BAL_Ecr de taille quelconque. Il montre qu'il est difficile d'implémenter une BAL de taille n (avec n > 1) sans le concept de variable conditionnelle présent dans POSIX. Et l'emploi de «timeout» de LabVIEW est une solution pour cette faiblesse.

De plus, dans certains cas si l'expédition des messages de la tâche «Productrice» a lieu très souvent, la tâche «Consommatrice» activée par cette tâche devient alors très chargée. Donc, en attribuant une pseudo-période, l'activation de la tâche «Consommatrice» peut être dirigée de manière périodique sans perturber le comportement de la tâche «Productrice». La Figure II-51 représente un modèle de la tâche «Consommatrice» périodique dans la communication asynchrone faiblement couplée. La fréquence d'activation de la tâche dans ce cas est déterminée par le temps minimal entre deux occurrences consécutives de la tâche (la séquence «0»). Une fois qu'elle est activée, elle retire un message dans la BAL_Ecr et continue son exécution (les séquences «1» et «2»).

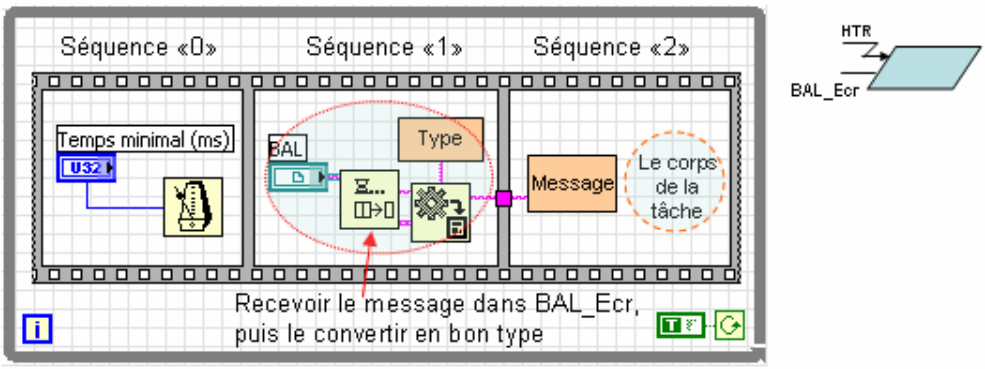


Figure II-51: Modèle de tâche «Consommatrice» périodique – Communication asynchrone faiblement couplée

II.2.3.e. Implémentation de la synchronisation par la communication synchrone

Dans la communication synchrone faiblement couplée présentée auparavant, la tâche «Consommatrice» ne peut être exécutée qu'avant que la tâche «Productrice» envoie un message. Il s'agissait clairement d'une contrainte de précédence entre ces deux tâches où la tâche «Consommatrice» joue le rôle de tâche «Réceptrice», et la tâche «Productrice» joue le rôle de tâche «Emettrice». Le message sert de déclenchement à la tâche «Réceptrice».

Dans la pratique, dans certains cas en fonction de langages de programmation ou de normes temps réel, la synchronisation peut être réalisée sous la forme de la communication synchrone faiblement couplée. L'implémentation de tel modèle est donnée sur la Figure II-52 ci-dessous.

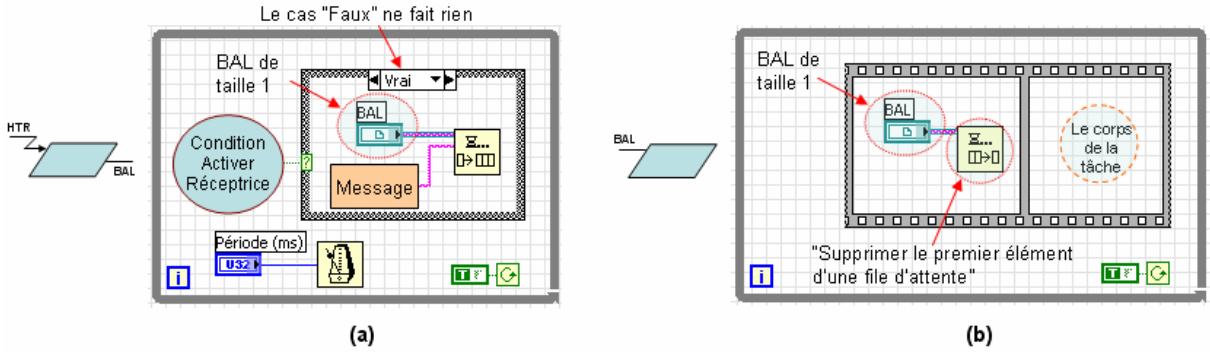


Figure II-52: Modèle de la synchronisation par la communication synchrone
(a) Modèle de tâche «Emettrice» - Synchronisation par BAL
(b) Modèle de tâche «Réceptrice» - Synchronisation par BAL

II.2.3.f. Implémentation de la communication synchrone fortement couplée

Rappelons que la communication synchrone fortement couplée, représentée typiquement par le Rendez-vous (réf. II.1.2.c.), est le passage de l'information de façon synchrone entre deux tâches. Donc, une tâche fait appeler le rendez-vous en y envoyant un message d'abord, et puis se bloque en y attendant une réponse pour poursuivre son exécution. Une autre tâche au début est suspendue jusqu'à un appel du rendez-vous, et puis l'accepte, et renvoie une réponse à la fin de son exécution.

Le «rendez-vous» proposé dans la palette «Synchronisation» de LabVIEW est un outil qui permet de synchroniser deux ou plusieurs tâches à des points spécifiques de leur exécution. Ce n'est pas de mécanisme de la communication synchrone fortement couplée. En effet, ce type de communication n'existe pas dans la plupart des langages de programmation et/ou normes temps réel, à part dans Ada. Cependant, le mécanisme de rendez-vous peut être implémenté en utilisant conjointement deux boîtes aux lettres de taille 1: une BAL pour les messages et une autre BAL pour les réponses. Par conséquent, cela peut être réalisé dans LabVIEW par trois VIs distincts

représentant respectivement la tâche appelant le rendez-vous, la tâche appelée par le rendez-vous, et le rendez-vous lui-même. Un tel modèle est illustré sur la figure ci-après:

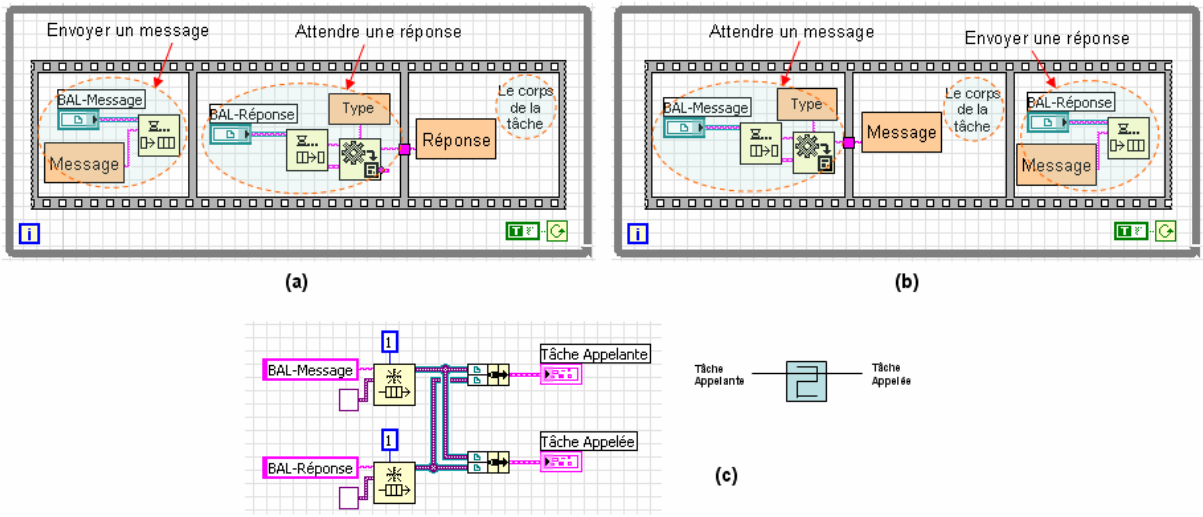


Figure II-53: Modèle de communication synchrone fortement couplée dans LabVIEW
(a) Modèle de tâche appelante (b) Modèle de tâche appelée (c) Modèle de Rendez-vous

II.2.3.g. Implémentation des modules de données

Un module de données (Hiding Information Module) est un objet passif qui possède une interface abstraite décrivant sa vue externe aux utilisateurs, et une structure de données pouvant être accédée directement par une ou plusieurs fonctions définies dans son interface. La méthode DARTS propose les critères pour structurer un système en un ensemble de modules de données (réf. II.1.3.). Dans cette section, je présente comment implémenter chacun de ces critères dans LabVIEW, un langage de programmation graphique flot de données.

- ***Implémentation de module d'interface de périphérique (DIM)***

Un DIM fournit une interface virtuelle qui cache les caractéristiques des périphériques d'entrée-sortie réel. Ce type de module est souvent placé à l'intérieur de la tâche qui joue le rôle de son objet actif. Pour implémenter le module d'interface de périphérique dans LabVIEW, une structure de Condition est insérée dans une boucle «Tant Que» s'exécutant une seule fois (principe de VI non-réentrant). Cette condition est utilisée de façon à ce que chaque cas de la structure conditionnelle corresponde à un appel d'une opération spécifique parmi un ensemble des opérations prédéfinies du module (voir Figure II-54).

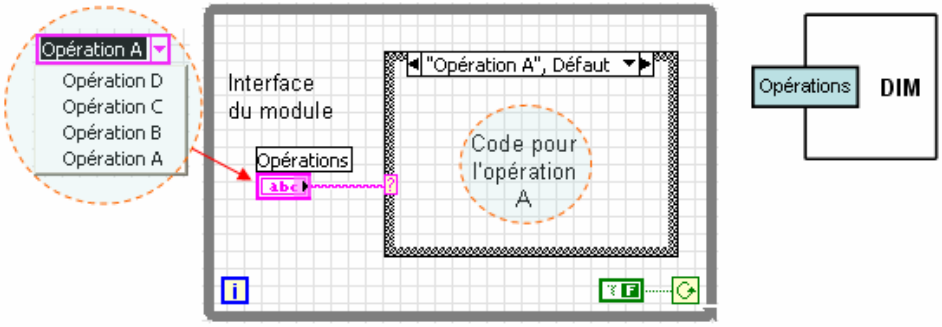


Figure II-54: Modèle d'un DIM de quatre opérations dans LabVIEW

Pour effectuer la transmission des paramètres d'entrée-sortie des opérations du module dans LabVIEW, le mécanisme de non-réentrance (réf. II.2.2.b.) couplé aux registres à décalage est utilisé. Par exemple, la transmission d'un paramètre d'entrée-sortie pour l'opération A d'un DIM est implémentée par le VI suivant:

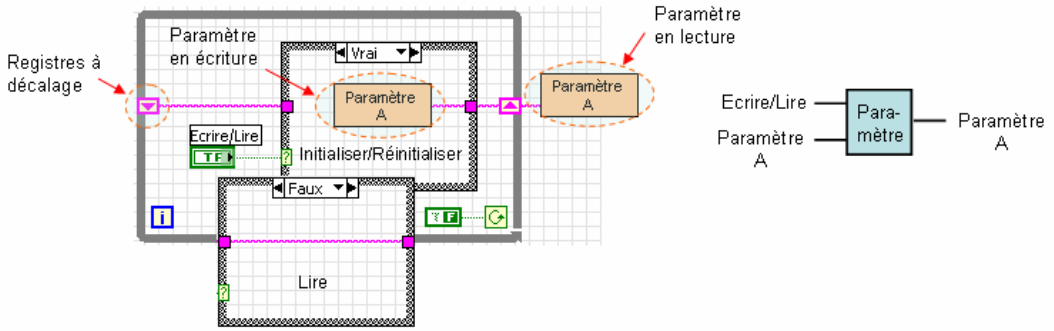


Figure II-55: Simulation de la transmission des paramètres d'entrée-sortie

A l'exécution de ce VI (voir Figure II-55), si la valeur de «Ecrire/Lire» est Vrai, l'initialisation ou la réinitialisation du paramètre «A» a lieu; sinon, «A» est lu.

- ***Implémentation des modules d'abstraction de données (DAM)***

Dans DARTS, le module d'abstraction de données est utilisé comme un outil d'encapsulation de données. Il fournit des opérations (en forme des procédures et/ou des fonctions) permettant à des tâches d'accéder aux données encapsulées. Un exemple typique de ce module est le module de données de deux opérations, appelées «Ecrire» et «Lire»: la tâche «Ecriture» appelle l'opération «Ecrire» pour la mise à jour de la donnée encapsulée, alors que la tâche «Lecture» appelle l'opération «Lire» pour l'aperçu de cette donnée (au début, la donnée encapsulée est initialisée par une valeur par défaut).

La Figure II-56 représente le modèle de la tâche «Ecriture» et le modèle de la tâche «Lecture» en tant que deux tâches périodiques dans LabVIEW. Grâce à l'outil «File d'attente» de LabVIEW, l'enregistrement et la récupération de données sont effectués par ces deux tâches de la manière suivante: à l'écriture, il faut d'abord vider la file d'attente, puis ajouter la nouvelle donnée (voir Figure II-56 (a)); tandis qu'à la lecture, la donnée encapsulée est lue sans la retirer (donc, plusieurs lectures successives sans écriture obtiennent les mêmes résultats) (voir Figure II-56 (b)). Enfin, un sémaphore binaire est utilisé pour assurer l'exclusion mutuelle.

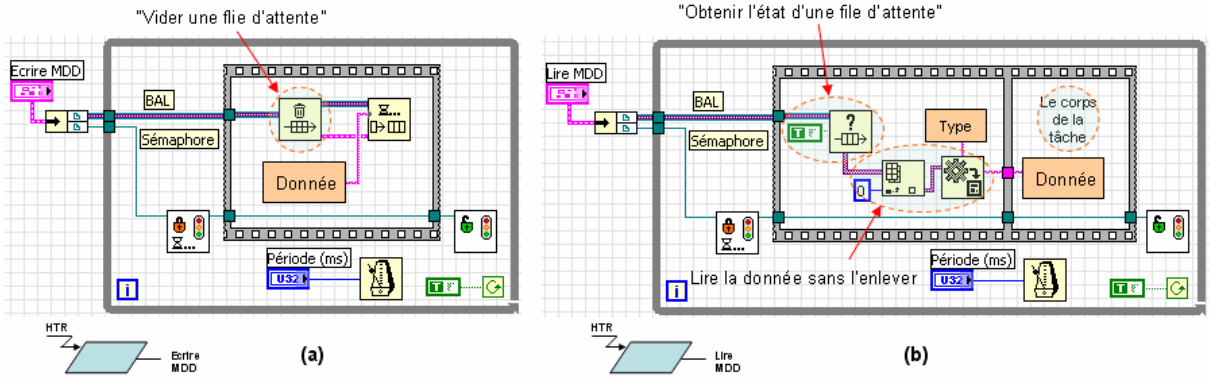


Figure II-56: Modèles de tâches dans le Module de Données
(a) Modèle de la tâche «Ecriture» (b) Modèle de la tâche «Lecture»

Grâce au type «Variant», l'écriture peut être réalisée avec un type de données quelconque, mais en lecture, il faut fournir le type de données original au connecteur «Type» de la fonction «Variant en donnée» afin de récupérer la donnée du bon type. Une erreur se produit si ce type n'est pas correct.

Dans la Figure II-57, le VI du module de données est vu comme une association d'une boîte aux lettres de taille 1 et d'un sémaphore binaire; une icône de deux terminaux représentant deux opérations «Ecrire» et «Lire». Une constante est ajoutée à la BAL pour la valeur par défaut du

module (il est important de noter que son type de données doit être identique à celui de donnée encapsulée dans le module).

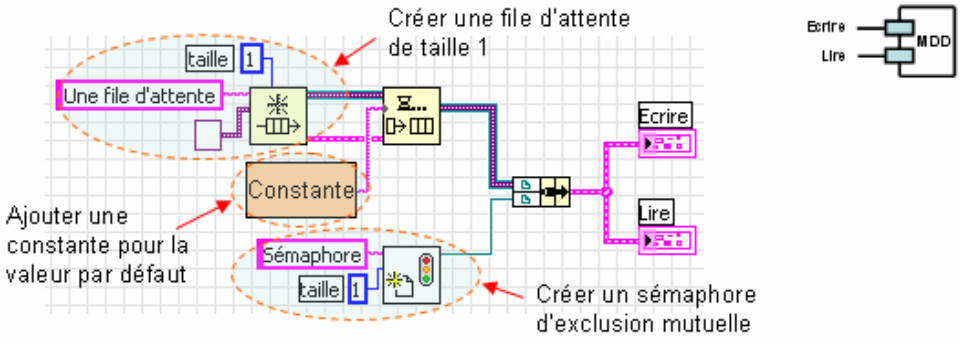


Figure II-57: Modèle de Module de Données dans LabVIEW

L'implémentation d'une MDD de type lecteur/écrivain autorisant les lectures réentrantes est plus complexe. Dans la pratique, il n'est pas possible de faire face à un nombre infini de lecteurs, donc on pourrait aussi implémenter cela avec le pseudo-code suivant :

```
-----------------------------------------------------------------

/* Module de Données: Ecriture/Lecture Multiple */

Sémaphore_MDD : Sémaphore_A_Compte(65535)

Opérations :

1/ Lire (Donnée_output)
      /* Fonction : renvoyer la donnée encapsulée dans le MDD. Plusieurs
      lectures peuvent accéder en même temps à cette donnée*/
      Attendre (Sémaphore_MDD)
         Donnée_output = Donnée_encapsulée
      Signaler (Sémaphore_MDD)

2/ Ecrire (Donnée_input)
      /* Procédure : mettre à jour la donnée encapsulée dans le MDD. Si
      plusieurs écritures veulent mettre à jour cette donnée à la fois,
         elles doivent être exécutées en exclusion mutuelle. */
      Attendre (Sémaphore_MDD,65535) /* prend toutes les occurrences*/
         Donnée_encapsulée = Donnée_input
      Signaler (Sémaphore_MDD,65535)

-----------------------------------------------------------------
```

Malheureusement LabVIEW ne permet pas de prendre plusieurs instances d'un sémaphore à la fois, ce qui empêche une implémentation efficace de cette solution. Voici le pseudo-code à implémenter en LabVIEW si l'on souhaite un nombre potentiellement non borné de lecteurs :

```
   ---------------------------------------------------------------------------

   /* Module de Données: Ecriture/Lecture Multiple */

   Opérations :

   1/ Lire (Donnée_output)
      /* Fonction : renvoyer la donnée encapsulée dans le MDD. Plusieurs
         lectures peuvent accéder en même temps à cette donnée*/
      Attendre (Sémaphore_Lecture)
         Augmenter nombre_de_lectures
         Si nombre_de_lectures = 1, alors
            Attendre (Sémaphore_Ecriture)
      Signaler (Sémaphore_Lecture)
         Donnée_output = Donnée_encapsulée
      Attendre (Sémaphore_Lecture)
         Diminuer nombre_de_lectures
         Si nombre_de_lectures = 0, alors
            Signaler (Sémaphore_Ecriture)
      Signaler (Sémaphore_Lecture)

   2/ Ecrire (Donnée_input)
      /* Procédure : mettre à jour la donnée encapsulée dans le MDD. Si
```

```
      plusieurs écritures veulent mettre à jour cette donnée à la fois,
      elles doivent être exécutées en exclusion mutuelle. */
   Attendre (Sémaphore_Ecriture)
      Donnée_encapsulée = Donnée_input
   Signaler (Sémaphore_Ecriture)
```

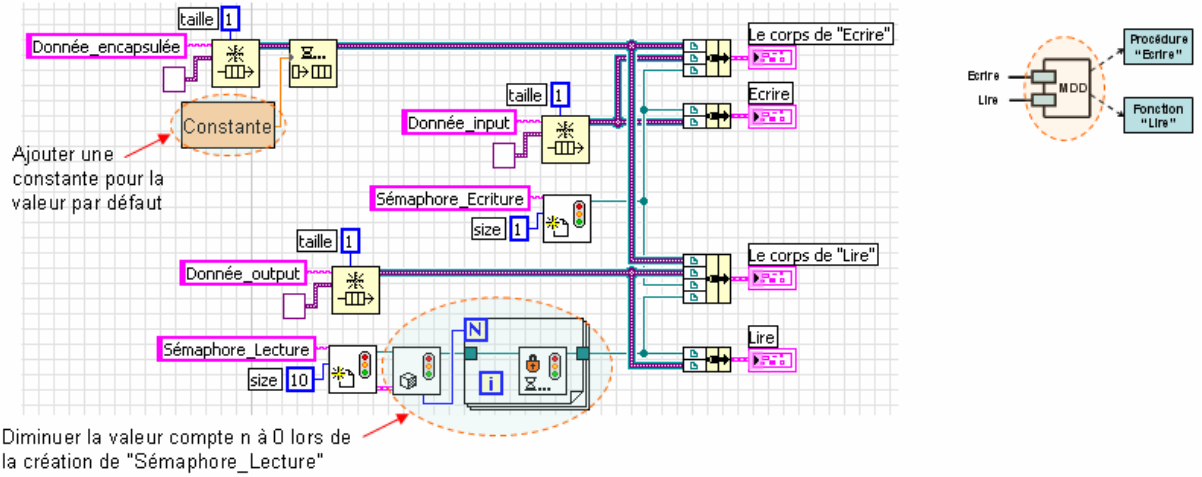


Figure II-58: Partie déclarative du Module de «Lecteur/Ecrivain» multiple dans LabVIEW

La Figure II-58 présente la création des éléments de communication et synchonisation nécessaires pour cette implémentation. Ces éléments sont regroupés dans un cluster (type enregistrement). Ce sont ces éléments qui sont utilisés par les tâches effectuant une lecture ou une écriture.

- ***Implémentation des modules d'état-transition (STM)***

Un STM est un module d'abstraction de comportement qui s'applique à chaque objet de contrôle dans les diagrammes de spécifications d'un système. Chaque STM est un objet passif activé par un objet de contrôle. Une fois qu'il est exécuté, en fonction de certains événements associés à un état donné, un changement d'état a lieu comme un «automate».

La notion d'automate est une notion mathématique abstraite correspondant au terme courant de «machine». Un automate est un mécanisme (ou une procédure) logique permettant d'aboutir en un nombre de pas fini ou non fini à un état final à partir d'un état initial.

Dans DARTS, un automate est représenté par un diagramme d'état-transition tel que sur la Figure II-59.

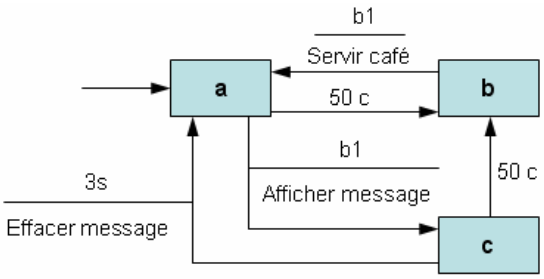


Figure II-59: Un exemple d'automate

L'automate donné ici a trois états: «a», «b», et «c» modélisant les services d'une machine à café. Le changement d'état est effectué à l'aide des paramètres de contrôle. Par exemple, à partir de l'état initial «a», si un montant de 50 centimes a été ajouté, l'état «b» est atteint et on peut servir un café après appui sur le bouton «b1». Sinon, l'état «c» est atteint avec un message (i.e., «Veuillez introduire 50 c»), et après 3 secondes si aucune action n'a lieu, il y a un retour à l'état initial.

```
/* Automate: Machine à café */
Etat = "a"
Loop
   Case Etat,
      Quand "a" ⇒ si 50c alors Etat = "b";
                  Sinon, si b1, alors temps = heure_système;
                                      Afficher_message;
                                      Etat = "c";
      Quand "b" ⇒ si b1, alors servir_café; Etat = "a";
      Quand "c" ⇒ si 50c, alors Etat = "b";
                   Sinon, si heure_système - temps ≥ 3s, alors
                           Effacer_message; Etat = "a";
   Fin Case
Fin Loop
```

Dans LabVIEW, pour implémenter le fonctionnement de cet automate une structure de contrôle conditionnelle est utilisée de manière à ce que chaque cas de cette structure corresponde à un état. A chaque itération, le programme teste si une transition d'état s'est produite, ou si l'état courant est maintenu. Cela se traduit en LabVIEW par une imbrication de structures de Condition (voir Figure II-60).

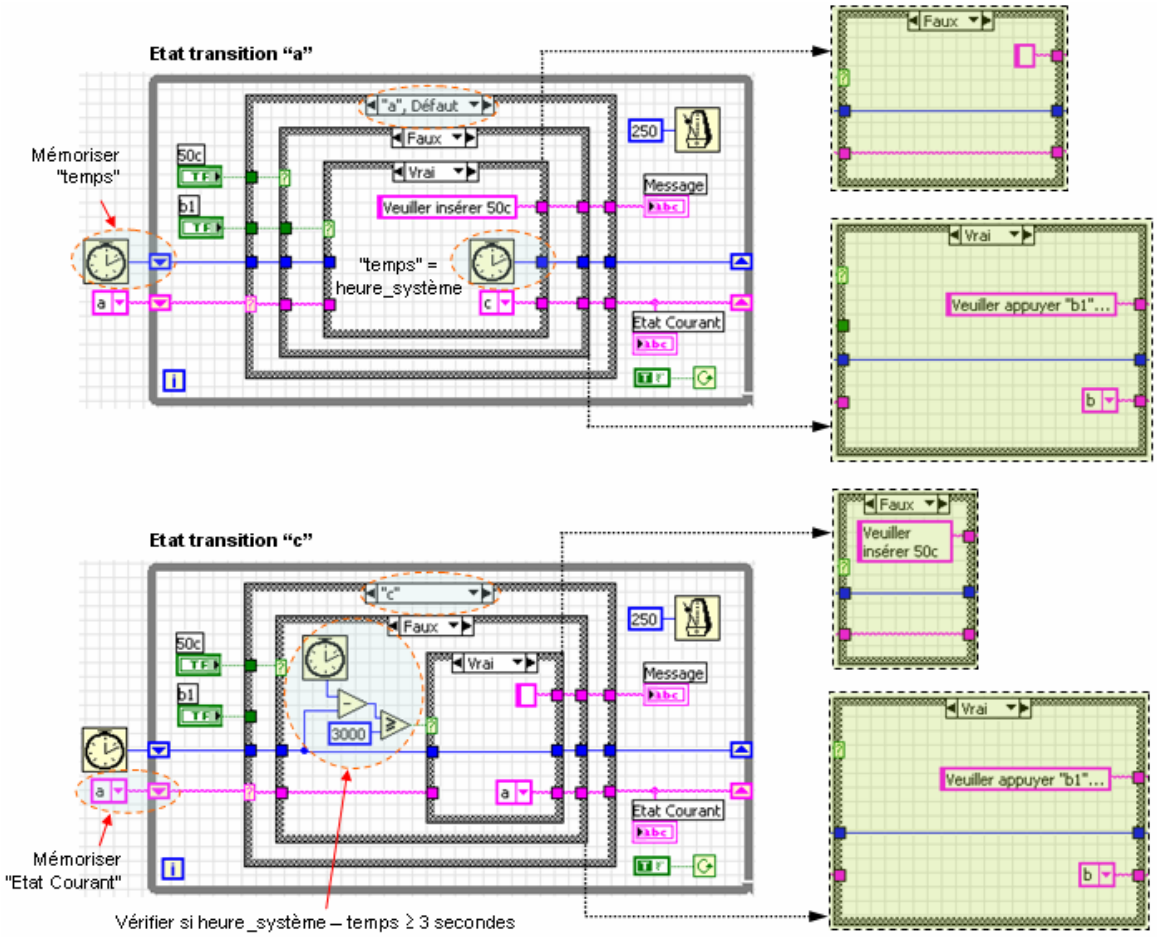


Figure II-60: Automate de machine à café dans LabVIEW

Un modèle de tâche de contrôle avec le modèle du module d'état-transition en général dans LabVIEW est montré sur la figure ci-dessous.

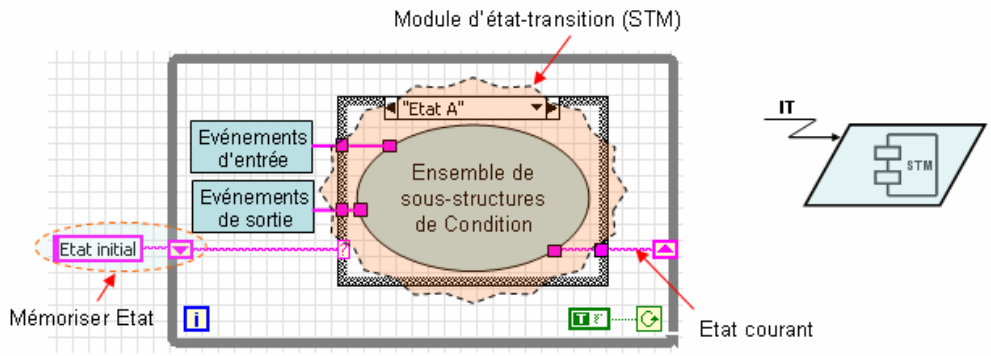


Figure II-61: Modèle de tâche de contrôle dans LabVIEW

Remarquer que l'utilisation des registres à décalage dans la boucle «Tant Que» de la tâche de contrôle a pour but de mémoriser l'état transition actuel du STM. Les registres à décalage sont initialisés au début de l'exécution de cette tâche, et sont mis à jour par le STM une fois qu'une transition d'état a lieu.

II.2.4. Rôle de LabVIEW dans le développement d'un logiciel

Le langage LabVIEW permet de développer rapidement des applications de contrôle/ commande ou temps réel, mais il n'est pas nécessairement adapté au temps réel strict, ni aux systèmes embarqués (pour lesquels les concepteurs préfèrent souvent Ada, C-POSIX, C-VxWorks,…). Le fait que l'ordonnancement des «threads» LabVIEW repose sur un système d'exploitation le plus souvent non temps réel rend difficilement validable temporellement une application LabVIEW s'exécutant sur un système d'exploitation généraliste. Il existe cependant des cibles LabVIEW Real-Time puissantes mais limitées à quelques architectures matérielles. Quel est alors le rôle de LabVIEW dans le développement d'un logiciel?

Pour les systèmes temps réel, LabVIEW n'est pas dédié à la validation temporelle. Il existe cependant des outils de validation temporelle, commerciaux ou universitaires spécialisés. Donc, le rôle de LabVIEW dans notre démarche est de servir de plate-forme de développement permettant la génération de code à destination d'outils de validation temporelle, ceci est montré sur la figure ci-après:

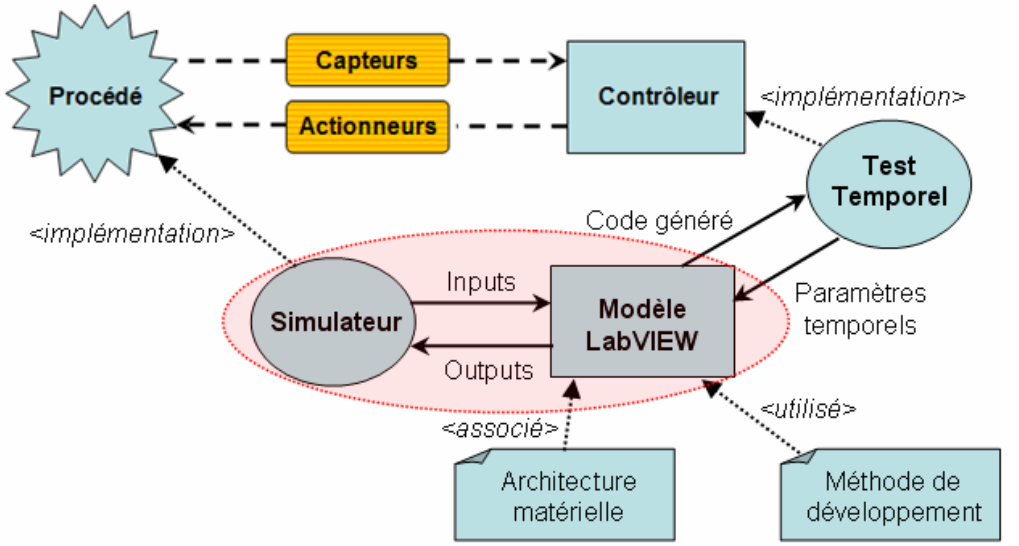


Figure II-62: Le rôle de LabVIEW dans le développement d'un logiciel

La génération de code part de DARTSVIEW vers une norme de programmation impérative pour le temps réel ainsi que vers des outils de validation temporelle. Ce sont des outils et normes mouvants, nous avons donc dans ce manuscrit un instantané des normes de programmation

choisies comme langages cibles pour la génération de code. Nous présentons au chapitre 4 la façon dont nous avons procédé pour rendre DARTSVIEW aussi ouvert que possible aux modifications des normes et outils.

II.3. Implémentation DARTS sur langages de programmation impératifs

Il y a nombre considérable d'exécutifs temps réel sur le marché, gratuits ou commerciaux. La plupart des exécutifs utilisent un développement croisé, et fournissent un certain nombre d'outils de traçage d'application, d'évaluation de performance et de débogage. VxWorks, OSEKturbo, RTEMS, RTLinux en sont des exemples.

Normalement, chaque exécutif supporte un ou plusieurs langages de programmation multitâches permettant aux concepteurs de programmer et de générer le code exécutable sur cet exécutif. Donc, au minimum, l'assembleur est supporté, la plupart des exécutifs sont accompagnés de compilateurs pour le langage C, voire le langage C++, ou encore le langage Ada. Notons que le langage C n'offre pas de support natif pour le multitâche, il existe différentes normes (par exemple, POSIX, OSEK/VDX, etc.) ou bibliothèques propriétaires (VxWorks, RTEMS, etc.) permettant une programmation multitâche.

Cette section montre donc comment on peut implémenter une conception DARTS dans les normes et les bibliothèques propriétaires.

II.3.1. Implémentation DARTS sur la norme POSIX

L'interface multitâche (*pthread*) définie par la norme POSIX se trouve dans les amendements 1003.1c (intégrée dans 1003.1 depuis 2001) et 1003.1j. L'apport principal de l'amendement 1003.1j est l'ajout de la possibilité de spécifier l'utilisation de différentes horloges (notamment CLOCK_MONOTONIC).

II.3.1.a. Implémentation des tâches

Une tâche POSIX (*pthread*) se lance dynamiquement grâce à fonction `pthread_create`. La syntaxe de cette fonction est:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
      void *(*start_routine)(void*), void *arg);
```

Le paramètre `thread` est un paramètre de sortie, qui contiendra l'identificateur de la tâche nouvellement créée.

Le paramètre `start_routine` est un pointeur vers la fonction implémentant la tâche.

Le dernier paramètre, `arg`, est le paramètre éventuel passé à cette fonction. Ce paramètre est le plus complexe, en effet, on y trouve des paramètres de configuration d'une tâche comme:

- le fait qu'elle soit détachée ou pouvant être attachée (par défaut);
- la taille et l'adresse de sa pile;
- son type d'ordonnancement, sa priorité, et le fait qu'elle soit ordonnancée localement (à l'intérieur du processus père) ou de façon globale.

Enfin, la valeur de retour est 0 si la tâche a bien été créée, sinon un code d'erreur.

Un exemple simple de la création d'une instance de la tâche f (observer l'utilisation du paramètre pour passer un entier) dans la programmation multitâche POSIX est donné ci-après:

```
-----------------------------------------------------------------
#include <pthread.h> /* Librairie POSIX threads */
#include <stdio.h> /* Entrées/Sorties (printf) */

void f(int no) {
    ...
```

```
}

void main() {
   pthread_t tache1; /* identificateur de tâche */
   pthread_create(&tache1, 0, f, (void *)1);
   ...
   pthread_join(tache1,NULL); /* Le programme principal s'attache à la
tâche 1. Il ne se termine que lorsque cette tâche se termine */
}
-----------------------------------------------------------------------
```

Noter qu'en passant la valeur nulle au paramètre d'attribut, les attributs par défaut sont choisis. Toutes les fonctions manipulant les attributs de tâches commencent par `pthread_attr` suivi du nom de l'attribut à modifier. Par exemple,

```
-----------------------------------------------------------------------
pthread_attr_t attributs;
/* Attribut utilisé lors de la création des tâches */

pthread_attr_init(&attributs);
/* Initialisation des attributs aux valeurs par défaut */

pthread_attr_setscope(&attributs, PTHREAD_SCOPE_SYSTEM);
/* L'ordonnancement des tâches créées avec cet attribut s'effectue au
niveau global */

-----------------------------------------------------------------------
```

II.3.1.b. Modules de données

Les modules de données DARTS peuvent être implémentés à l'aide d'un sémaphore d'exclusion mutuelle (*mutex*). Le sémaphore est déclaré de façon globale pour le rendre visible par toutes les tâches concernées. Il est créé et initialisé avant le lancement des tâches.

L'implémentation du module de données, appelé «T_Désirée», accédé par deux tâches, T1 et T2, dans POSIX est :

```
-----------------------------------------------------------------------
#include <pthread.h>

float T_Desiree; /* Déclaration du module de données T_Désirée */
pthread_mutex_t s_T_Desiree; /* Mutex protégeant T_Désirée */

void T1() {
/* Tâche 1 lisant le module de données */
   float Temperature;
   while(1) { /* Faire toujours */
      /* Exclusion mutuelle sur le module de données */
      pthread_mutex_lock(&s_T_Desiree);
      /* Lecture du MDD */
      Temperature=T_Desire;
      pthread_mutex_unlock(&s_T_Desiree;
   }
}

void T2() {
/* Tâche 2 modifiant le module de données */
   float Nouvelle_Temp;
   while(1) { /* Faire toujours */
      /* Lecture de capteur en dehors de la section critique */
      Nouvelle_Temp=Lire_Capteur_Temperature();
      /* Exclusion mutuelle sur le module de données */
      pthread_mutex_lock(&s_T_Desiree);
      /* Modification du MDD */
      T_Desire=Nouvelle_Temp;
      pthread_mutex_unlock(&s_T_Desiree);
   }
```

```
}

void main() {
   pthread_t t1,t; /* Identification des tâches */
   pthread_mutex_init(&s_T_Desiree,NULL); /* Initialisation du mutex de
T_Desiree */
   /* Initialisation de T_Desiree */
   T_Desiree=Lire_Capteur_Temperature();
   /* Lancement des tâches */
   pthread_create(&t1,NULL,T1,NULL);
   pthread_create(&t2,NULL,T2,NULL);
   /* Attente de terminaison */
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
}
-----------------------------------------------------------------------
```

Pour pouvoir effectuer une étude d'ordonnançabilité, il est possible de définir des priorités pour les tâches, de définir une priorité plafond pour le sémaphore (la plus forte des priorités des tâches l'utilisant), et de lui affecter le protocole à priorité plafond.

II.3.1.c. Boîtes aux lettres

POSIX ne définit aucun outil de communication par message spécifique aux tâches. L'amendement 1003.1b, orienté processus, définit quant à lui les boîtes aux lettres nommées permettant à deux processus de communiquer. Par contre, en utilisant les mutex et les variables conditionnelles, une communication par message au niveau tâches uniquement peut être réalisée. De plus, l'utilisation de ce couple d'outils permet en effet de définir des moniteurs de Hoare, à partir desquels tout outil de communication et/ou synchronisation peut être créé.

- ***Boîte aux lettres de taille 1 avec écrasement***

Ce type de BAL, nommé `bal_ecr`, est une boîte aux lettres de taille 1 avec écrasement. La spécification de ce type abstrait, et l'implémentation de deux primitives, appelées `bal_ecr_envoyer` et `bal_ecr_recevoir`, sont présentées ci-dessous:

```
-----------------------------------------------------------------------
/* Boîte aux lettres avec écrasement de taille 1 */
Typedef struct s_bal_ecr {
   char * buf; /* Zone dans laquelle le message est stocké, mémoire
allouée DYNAMIQUEMENT par bal_ecr_init */
   unsigned char vide; /* Booléen permettant de savoir si la BAL est
vide ou pleine */
   pthread_mutex_t mutex; /* Sémaphore d'exclusion mutuelle
garantissant qu'une seule tâche manipule la structure */
   pthread_cond_t pas_vide; /* Variable conditionnelle utilisée pour
signaler une ajout d'élément susceptible de réveiller une tâche en
attente */
   unsigned taille_element; /* Taille d'un élément de BAL */
} *bal_ecr;

bal_ecr bal_ecr_init (const unsigned taille_element);
/* Créer et initialiser une BAL à écrasement vide pouvant contenir un
élement de taille_element octets.
Renvoie: bal_ecr si succès, 0 sinon */

int bal_ecr_recevoir(bal_ecr bal, char *buf);
/* Attend un message et reçoit un message de la BAL, nommée bal
Primitive bloquante
Renvoie: taille_element */

int bal_ecr_envoyer(bal_ecr bal, const char *buf);
/* Envoie un message dans la BAL, nommée bal, en écrasant si besoin le
message présent.
```

```
Primitive non bloquante
Renvoie: taille_element */

void bal_ecr_delete(bal_ecr bal);
/* Supprime la BAL, nommée bal */
                    ---------------------------------

...
int bal_ecr_envoyer(bal_ecr bal, const char *buf) {
   pthread_mutex_lock(&(*bal).mutex);
   /* Exclusion mutuelle */
   memcpy((*bal).buf,buf,(*bal).taille_element; /* Le message écrase un
éventuel message présent dans la bal */
   (*bal).vide=0; /* La BAL contient un message */
   pthread_mutex_unlock(&(*bal).mutex);
   /* Fin de l'exclusion mutuelle */
   pthread_cond_signal(&(*bal).pas_vide; /* On reveille l'éventuel
thread en attente d'un message */
   return (*bal).taille_elemen;
}

int bal_ecr_recevoir(bal_ecr bal, char *buf) {
   pthread_mutex_lock(&(*bal).mutex);
   /* Exclusion mutuelle */
   while ((*bal).vide) {
   /* Tant que la boîte aux lettres est vide */
     pthread_cond_wait(&(*bal).pas_vide,&(*bal).mutex); /* On attend
que la bal contienne un message */
   }
   memcpy(buf,(*bal).buf,(*bal).taille_element); /* Copie du message
dans buf */
   (*bal).vide=1; /* La bal est maintenant vide */
   pthread_mutex_unlock(&(*bal).mutex);
   /* Fin de l'exclusion mutuelle */
   return (*bal).taille_element;
}
...
-----------------------------------------------------------------------
```

♦ ***Boîte aux lettres de taille 1 sans écrasement***

Pour les boîtes aux lettres de taille 1 sans écrasement, l'idée d'implémentation est similaire à celle utilisée pour les boîtes aux lettres de taille 1 avec écrasement. La différence est que dans ce cas, les tâches émettrices peuvent être bloquées. Par conséquent, on utilise deux variables conditionnelle (au lieu d'une pour BAL avec écrasement).

♦ ***Autres boîtes aux lettres***

Les boîtes aux lettres dans ce cas sont les boîtes aux lettre de taille n avec et sans écrasement. La file de messages est implémentée par un tableau circulaire. La prise en compte d'une priorité de messages peut s'effectuer de deux façons: soit le nombre de priorités est élevé (dans ce cas la gestion de la file d'attente consiste en un tri par insertion des messages), soit il n'y a que deux niveaux de priorité (normal et urgent).

Le lecteur peut consulter [CG05] pour l'implémentation détaillée des boîtes aux lettres de taille n avec et sans écrasement.

II.3.1.d. Synchronisation

Les amendements 1003.1c et 1003.1j ne proposent pas l'outil de synchronisation au niveau tâches. Les sémaphores à compte sont définis dans la partie 1003.1b orientée processus. Cependant, il est possible, à la création d'un sémaphore à compte 1003.1b, de forcer de celui-ci à être local à un processus.

Par exemple, la synchronisation entre la tâche T1 et la tâche T2 à l'aide de sémaphore à compte, nommé S1, dans la programmation multitâche POSIX est présentée ci-après:

```
-----------------------------------------------------------------------
#include <pthread.h> /* Librairie POSIX threads */
#include <semaphore.h> /* Primitives de gestion de sémaphore */

sem_t S1; /* Sémaphore de synchronisation */

void T1() {
/* Tâche déclenchant une synchronisation */
   while (1) { /* Faire tujours */
      ...
      sem_post(&S1); /* Déclenchement de la synchronisaiton */
      ...
   }
}

void T2() {
/* Tâche en attente sur la synchronisation */
   while (1) { /* Faire toujours */
      sem_wait(&S1); /* Attendre de synchronisaiton */
      ...
   }
}

void main() {
   pthread_t t1,t2; /* Identificateurs de tâches */
   sem_init(&S1,0,0); /* Sémaphore non partagé par des processus
initialisé à 0 */
   /* Lancement des tâches */
   pthread_create(&t1,NULL,T1,NULL);
   pthread_create(&t2,NULL,T2,NULL);
   /* Attente de terminaison */
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
}
-----------------------------------------------------------------------
```

II.3.1.e. Tâches périodiques

Pour implémenter une tâche périodique en POSIX, certains systèmes proposent des primitives spécifiques non POSIX permettant de gérer la périodicité stricte des tâches. Par exemple, RTLinux fournit une fonction `pthread_make_periodic_np` permettant de rendre une tâche périodique. Cependant, la non-portabilité de cette fonction est gênante.

La solution présentée dans [CG05] consiste à utiliser les sémaphores de l'amendement 1003.1, car ces sémaphores intègrent une primitive d'attente bornée dans le temps par une date absolue. Ainsi, l'implémentation d'une tâche périodique, nommée `tache_periodique`, avec une période de 500 millisecondes en POSIX est donnée ci-après :

```
-----------------------------------------------------------------------
#include <pthread.h> /* Librairie POSIX threads */
#include <semaphore.h> /* Primitives de gestion de sémaphore */

void Periodique(long periode_us) {
/* periode_us: période en microsecondes */
   struct timespec horloge; /* Constituée d'un champ en secodes, et
d'un autre en nanosecondes */
   sem_t Reveil;
   sem_init(&Reveil,0,0); /* Création d'un sémaphore non partagé entre
processus initialisé à 0 */
   clock_gettime(CLOCK_REALTIME,&horloge); /* L'heure courante de
l'horloge au démarrage de la tâche */
   while (1) { /* Faire toujours */
```

```
      ...
      ajouter_microsecondes(&horloge,periode_us); /* Calcul de la date
du prochain réveil */
   sem_timedwait(&Reveil,&horloge); /* Le sémaphore étant toujours nul,
cette instruction a pour effet d'endormir la tâche jusqu'à sa prochaine
date de réveil */
   }
}

void main() {
   pthread_t tache_periodique; /* Identificateur de tâche */
   pthread_create(&tache_periodique,NULL,Periodique,(void*)500000); /*
Lancement de la tâche avec une période de 500 millisecondes */
   pthread_join(tache_periodique,NULL); /* Attente de terminaison */
}
-----------------------------------------------------------------------
```

En fait, il existe une autre solution qui consiste à utiliser la fonction `pthread_cond_timedwait` (attente bornée sur une variable conditionnelle), car cette fonction utilise une date absolue de *timeout*. L'idée est de créer une variable conditionnelle jamais signalée dans chaque tâche périodique: l'attente de la prochaine date d'activation s'effectue par le biais du *timeout* de `pthread_cond_timedwait` (réf. [CG05], chapitre 6, page 298). Dans les deux cas, l'idée est d'attendre une certaine date (et non pas un certain temps) pour éviter la dérive des horloges.

II.3.1.f. Tâches réveillées par interruption

POSIX ne définit pas d'interface normalisée pour la gestion des interruptions. Cela signifie que la programmation d'une routine de traitement d'interruption est liée au système sous-jacent. Si le traitement de l'interruption doit être effectué par une tâche, alors on utilise la technique du traitement différé (DSR, *Differed Service Routine*): un code très court, appelé ISR, est lié à l'interruption traitée. Son action consiste à signaler par synchronisation à la tâche de traitement l'occurrence de l'interruption.

Ainsi, le mécanisme de réveil d'une tâche matérielle se basant sur la synchronisation sur sémaphore s'implémente de la façon suivante :

```
-----------------------------------------------------------------------
#include <pthread.h> /* Librairie POSIX threads */
#include <semaphore.h> /* Primitives de gestion de sémaphore */

sem_t declenche_dsr; /* Sémaphore de synchronisation entre l'ISR et la
tâche de traitement */

void mon_isr(/* Paramètres spécifiques au système */) {
/* Fonction appelée sur l'occurrence de l'interruption traitée */
/* Appels spécifiques permettant d'acsuiescer l'interruption */
   sem_post(&declenche_dsr); /* Déclenche la tâche de traitement */
}

void Tache_Materielle() {
/* Tâche déclenchée sur interruption */
   while (1) { /* Faire toujours */
      sem_wait(&declenche_dsr); /* Attente d'interrruption */
   /* Traitement d'interruption */
      ...
   }
}

void main() {
   pthread_t tache_materielle; /* Identificateur de tâche */
   sem_init(declenche_dsr,0,0); /* Initialisation du sémaphore de
synchronisation ISR-]-DSR */
/* Appel d'une primitive système branchant l'ISR à l'interruption
traitée */
```

```
/* Démasquage de l'interruption traitée */
   pthread_create(&tache_materielle,NULL,Tache_Materielle,NULL);
   pthread_join(tache_materielle,NULL); /* Attente de terminaison */
}
-----------------------------------------------------------------------
```

II.3.2. Implémentation DARTS sur exécutif VxWorks

VxWorks est l'exécutif temps réel sur étagère le plus répandu sur le marché. Dans ses versions 5.x, il implémente une partie des interfaces POSIX 1003.1b et 1003.1c. Afin de donner un autre exemple de programmation d'applications de contrôle-commande en langage C, la programmation présentée dans cette section est basée sur l'interface propriétaire de VxWorks.

II.3.2.a. Implémentation des tâches

Une tâche (*task*) est représentée par une fonction pouvant posséder au plus dix paramètres de type `int`. Chaque tâche se lance dynamiquement grâce à la fonction `taskSpawn` ou bien à la combinaison des deux fonctions, `taskInit` et `taskResume`. La syntaxe de cette fonction est:

```
int taskSpawn(char *nom, int priority, int options, int stackSize,
FUNCPTR entryPoint, int arg1, …, int arg10);
```

La valeur de retour de cette fonction est l'identificateur de la tâche créée, `nom` est le nom alphanumérique de la tâche, `priority` est sa priorité (par rapport à POSIX, l'ordre est inversé, et 0 est la priorité la plus forte), `stackSize` est la taille de sa pile, `entryPoint` est la fonction implémentant la tâche, et les paramètres `arg1` à `arg10` sont les paramètres passés à cette fonction. Le paramètre `options` permet de spécifier différentes particularités au système.

Une implémentation d'une tâche de programme multitâche VxWorks est donnée ci-après. C'est une tâche, nommée `tache1`, de priorité 90, et implémentée par la fonction `f`.

```
-----------------------------------------------------------------------
#include <taskLib.h> /* Librairie tâches */
#include <stdio.h> /* Entrées/Sorties (printf) */

void f(int no) {
   ...
}

void LancerTaches() {
   taskSpawn("tache1",90,0,20000,f,1,0,0,0,0,0,0,0,0,0); /* Lancement
de la tâche 1, de priorité 90, avec 20000 octets de pile, implémentée
par la fonction f, avec le premier paramètre valant 1 */
}
-----------------------------------------------------------------------
```

L'une des particularités des exécutifs embarqués est que le programmeur ne définit pas de programme principal: lorsque le code est compilé et chargé sur la cible, la fonction `LancerTaches` peut être lancée à l'aide d'un `shell`.

II.3.2.b. Modules de données

De même que la norme POSIX, les modules de données DARTS peuvent être implémentés dans VxWorks à l'aide d'un sémaphore d'exclusion mutuelle de façon suivante:

```
-----------------------------------------------------------------------
#include <taskLib.h> /* Librairie tâches */
#include <semLib.h> /* Module définissant les sémaphores d'exclusion
mutuelle */

float T_Desiree; /* Déclaration du module de données T_Désirée */
SEM_ID s_T_Desiree; /* Mutex protégeant T_Désirée */

void T1() {
/* Tâche 1 lisant le module de données */
```

```
   float Temperature;
   while (1) { /* Faire toujours */
      /* Exclusion mutuelle sur le module de données */
      semTake(s_T_Desiree,WAIT_FOREVER);
      /* Lecteur du T_Desirée */
      Temperature=T_Desiree;
      semGive(s_T_Desiree);
      ...
   }
}

void T2() {
/* Tâche 2 modifiant le module de données */
   float Nouvelle_Temp;
   while (1) { /* Faire toujours */
      /* Lecture de capteur en dehors de la section critique */
      Nouvelle_Temp=Lire_Capteur_Temperature();
      /* Exclusion mutuelle sur le module de données */
      semTake(s_T_Desiree,WAIT_FOREVER);
      /* Modification de MDD */
      T_Desiree=Nouvelle_Temp;
      semGive(s_T_Desiree);
      ...
   }
}

void LancerTaches() {
   s_T_Desiree=semCreate(SEM_Q_PRIORITY|SEM_INVERSION_SAFE); /*
Initialisation du mutex de T_Désirée: La file d'attente des tâches est
gérée par niveaux de priorité, et le protocole à priorité héritée est
utilisé */
   /* Initialisation de T_Désirée */
   T_Desiree=Lire_Capteur_Temperature();
   /* Lancement des tâches */
   taskSpawn("T1",90,0,20000,T1,0,0,0,0,0,0,0,0,0,0);
   taskSpawn("T2",90,0,20000,T2,0,0,0,0,0,0,0,0,0,0);
}
-------------------------------------------------------------------------
```

Notons que VxWorks ne propose pas le protocole à priorité plafond mais uniquement le protocole à priorité héritée.

II.3.2.c. Boîtes aux lettres

VxWorks définit des boîtes aux lettres bornées sans écrasement, avec une gestion des priorités de message à deux niveaux (normal, urgent). Ainsi, l'implémentation de la communication entre la tâche T1 et la tâche T2 par boîte aux lettres en VxWorks est donnée ci-après:

```
-------------------------------------------------------------------------
#include <taskLib.h> /* Librairie tâches */
#include <msgQLib.h> /* Module définissant les boîtes aux lettres */

MSG_Q_ID Bal; /* Déclaration de la boîte aux lettres */

void T1() {
/* Tâche 1 en attente de message sur la boîte aux lettres */
   float Temperature;
   while (1) { /* Faire toujours */
      msgQReceive(Bal,(char*)&Temperature,sizeof(Temperature),
                  WAIT_FOREVER);
      ...
   }
}

void T2() {
/* Tâche 2 envoyant des messages */
```

```
   float Nouvelle_Temp;
   while (1) { /* Faire toujours */
      /* Lecture de capteur en dehors de la section critique */
      Nouvelle_Temp=Lire_Capteur_Temperature();
      msgQSend(Bal,(char*)&Nouvelle_Temp,sizeof(Nouvelle_Temp),
                  WAIT_FOREVER,MSG_PRI_NORMAL); /* Envoi du message en
priorité normale, si la boîte est pleine, cette instruction est
bloquante */
      ...
   }
}

void LancerTaches() {
   Bal=msgQCreate(1,sizeof(float),MSG_QçPRIORITY); /* Initialisation de
la boîte aux lettre Bal: tampon de 1 message. Les tâches en attente
sont gérées par niveau priorité */
   /* Lancement des tâches */
   taskSpawn("T1",90,0,20000,T1,0,0,0,0,0,0,0,0,0,0);
   taskSpawn("T2",90,0,20000,T2,0,0,0,0,0,0,0,0,0,0);
}
-----------------------------------------------------------------------
```

II.3.2.d. Synchronisation

Comme en POSIX, la synchronisation peut être implémentée à l'aide d'un sémaphore à compte. Ainsi, la synchronisation entre la tâche T1 et la tâche T2 à l'aide de sémaphore à compte, nommé S1, dans la programmation multitâche VxWorks est présentée ci-après:

```
-----------------------------------------------------------------------
#include <taskLib.h> /* Librairie tâches */
#include <semLib.h> /* Module définissant les sémaphores à compte */

SEM_ID S1; /* Déclaration du sémaphore de synchronisation */

void T1() {
/* Tâche déclenchant une synchronisation */
   while (1) { /* Faire tujours */
      ...
      semGive(S1); /* Déclenchement de la synchronisaiton */
      ...
   }
}

void T2() {
/* Tâche en attente sur la synchronisation */
   while (1) { /* Faire toujours */
      semTake(S1); /* Attendre de synchronisaiton */
      ...
   }
}

void LancerTaches() {
   S1=semCCreate(SEM_Q_FIFO,0); /* Initialisation du sémaphore de
synchronisation à 0, la file d'attente des tâches est FIFO pure */
   /* Lancement des tâches */
   taskSpawn("T1",90,0,20000,T1,0,0,0,0,0,0,0,0,0,0);
   taskSpawn("T2",90,0,20000,T2,0,0,0,0,0,0,0,0,0,0);
}
-----------------------------------------------------------------------
```

Remarquons que les sémaphores à compte ne se différencient, au niveau du programme, des sémaphores d'exclusion mutuelle qu'à la création (utilisation de `semCCreate` pour les premiers, et de `semMCreate` pour les seconds).

II.3.2.e. Tâches périodiques

Etant donné que VxWorks est un exécutif dirigé par le temps, le système prend la main à chaque *tick*. Le nombre de *ticks* par seconde est donné par la fonction `sysClkRateGet()`, et peut être modifié à l'aide de `sysClkRateSet`. Un intervalle de temps est donc forcément multiple de 1/`sysClkRateGet()`.

L'inconvénient majeur de l'interface de programmation native de VxWorks est qu'il n'existe aucune primitive permettant d'attendre jusqu'à une certaine date (en *ticks*). Par conséquent, le seul moyen de déclencher périodiquement une tâche consiste à attendre un certain nombre de *ticks*, ce qui entraîne une dérive des horloges. Ainsi, l'implémentation d'une tâche périodique, nommée `tache_periodique`, avec une période de 500 millisecondes en POSIX est donnée ci-après:

```
-----------------------------------------------------------------------
#include <taskLib.h> /* Librairie tâches */

void Periodique(int periode_ticks) {
/* Tâche périodique (avec dérive des horloges) */
/* periode_ticks est sa période désirée en ticks */
   while (1) { /* Faire toujours */
      ...
      taskDelay(periode_ticks); /* Attente d'au moins periode_ticks
ticks */
   }
}

void LancerTaches() {
   /* Lancement des tâches */
   taskSpawn("tache_periodique",90,0,20000,Periodique,
                  sysClkRateGet()/2,0,0,0,0,0,0,0,0,0); /* Période de 500
millisecondes */
}
-----------------------------------------------------------------------
```

Notons que la période est arrondie au *tick* inférieur, donc dans cet exemple, supposons que la fréquence des *ticks* est de 300 *ticks* par seconde (un *tick* dure 3,33… millisecondes), la période de la tâche `tache_periodique` est de 150 *ticks*, soit 500 millisecondes.

II.3.2.f. Tâches réveillées par interruption

La gestion des interruptions se base comme POSIX sur le concept d'ISR (fonction appelée lors de l'implémentation) s'exécutant dans un contexte spécifique aux interruptions. Ainsi, le mécanisme de réveil d'une la tâche matérielle s'implémente en utilisant une DSR, déclenchée à l'aide d'une synchronisation sur sémaphore par l'ISR.

```
-----------------------------------------------------------------------
#include <taskLib.h> /* Librairie tâches */
#include <semLib.h> /* Module définissant les sémaphores à compte */

SEM_ID declencheT1; /* Sémaphore de synchronisation */

void traiteIntr(int dummy) {
/* ISR déclenchant la tâche de traitement par synchronisation */
   semGive(declencheT1);
}

void TacheMaterielle() {
/* Tâche de traitement de l'interruption */
   while (1) { /* Faire toujours */
      semTake(declencheT1,WAIT_FOREVER); /* Attente de synchronisation
*/
      ...
   }
}
```

```
void LancerTaches() {
   declencheT1=semCCreate(SEM_Q_FIFO,0); /* Initialisation du semaphore
de synchronisation */
   intConnect(INUM_TO_IVEC(7),traiteIntr,0); /* La function traiteIntr
est appelée sur l'interruption IRQ7 */
   taskSpawn("TacheMaterielle",90,0,20000,TacheMaterielle,
                0,0,0,0,0,0,0,0,0,0);
}
-----------------------------------------------------------------------
```

II.3.3. Implémentation DARTS sur Ada

L'idée du langage Ada naît au milieu des années 70 sous l'impulsion du département de la défense américain qui souhaite un langage unique pour les système embarqués. Ce langage porte le nom d'Ada en mémoire du premier programmeur au monde: Augusta Ada Byron. Ada est un langage impératif à typage fort, compilé, modulaire, à compilation séparée. Il permet le traitement des exceptions, et la généricité.

En fait, le langage Ada lui-même est une norme, supportant différentes implémentation. Les implémentations de la norme Ada reposent sur un exécutif ou noyau, pouvant être lui-même conforme à POSIX ou encore OSEK.

II.3.3.a. Les tâches Ada

La possibilité de créer des tâches en langage Ada est native, car Ada est multitâche: en langage Ada, une tâche est un type nommé `task type` que l'on peut instancier. Lorsqu'une tâche est instanciée de façon globale (variable de paquetage), elle est lancée automatiquement au démarrage du programme principal. Lorsqu'une tâche est instanciée dynamiquement (tâche déclarée dans un bloc, ou allouée dynamiquement par `new`), elle est lancée au moment de l'instanciation.

Les tâches sont toujours exécutées automatiquement en mode attaché dans le programme principal, c'est-à-dire que le programme principal ne se termine pas s'il existe une tâche en fonctionnement. La déclaration d'un type tâche s'effectue en Ada en deux parties: une spécification de type (`task type`), et un corps de type (`task body`). Une tâche de Ada est créée soit en instanciant un type de tâche, soit en définissant une instance de tâche directement de type anonyme. Il est important de noter qu'un type de tâche est un type «limité», c'est-à-dire que toute affectation ou comparaison de deux tâches est impossible.

Ainsi, le code ci-après définit un type de tâche et l'instancie:

```
-----------------------------------------------------------------------
with System; use System;

procedure une_tache_typee is
   -- Spécification de type tâche
   task type T_Tache is
     pragma Priority(Defaut_Priority);
   end T_Tache;

   -- Le corps de type tâche
   task body T_Tache is
     ...
   begin
     ...
   end T_Tache;

   MaTache : T_Tache; -- Instanciation
begin
   null;
end une_tache_typee; -- ne se termine que lorsque la tâche est
                     -- terminée
```

```
-----------------------------------------------------------------------
```

Le code ci-dessous est équivalent, mais définit une tâche de type anonyme:

```
-----------------------------------------------------------------------
with System; use System;

procedure une_tache_anonyme is
   -- Spécification de type tâche anonyme
   task MaTache is
      pragma Priority(Defaut_Priority);
   end MaTache;

   -- Le corps de type tâche anonyme
   task body MaTache is
      ...
   begin
      ...
   end MaTache;
   -- La tâche MaTache a été instanciée en même temps qu'elle a été
   -- déclarée
begin
   null;
end une_tache_typee; -- ne se termine que lorsque la tâche est
                     -- terminée
-----------------------------------------------------------------------
```

Dans Ada, quand plusieurs tâches s'exécutent en même temps, l'ordre de l'exécution actuel peut être déterminé par la priorité attribuée à chaque tâche: donc, parmi les tâches en attente, la tâche ayant la priorité la plus élevée s'exécute. Une tâche, nommée `Tache_A` par exemple, peut avoir une priorité initiale de 10 en utilisant un `pragma System.Priority(10)` dans sa spécification. Le système d'ordonnancement de Ada, avant Ada 2005, est basé sur des algorithmes d'ordonnancement à priorités fixes et des algorithmes à priorité dynamique en Ada 2005. Les tâches sont ordonnancées en FIFO selon leur priorité, c'est-à-dire qu'il y a une file d'attente gérée en FIFO pour chaque niveau de priorité grâce à la directive pragma Task_Dispatching_`Policy (FIFO_Within_Priorities)`.

De plus, on peut passer une priorité à une tâche en paramètre lors de son instanciation par la technique des discriminants. Le type d'un discriminant est forcément un type discret (entier, caractère, type énuméré, etc.). Ainsi, le code ci-après crée un type de tâche ayant une priorité passée en paramètre:

```
-----------------------------------------------------------------------
with System; use System;

procedure une_tache_typee_discriminant is
   -- Spécification de type tâche
   task type T_Tache(Priority: Integer) is
   -- Le discriminant permet de passer des paramètres scalaires lors
   -- de l'instanciation
      pragma Priority(Priority); -- Définit la priorité de la tâche
   end T_Tache;

   -- Le corps de type tâche
   task body T_Tache is
      ...
   begin
      ...
   end T_Tache;

   MaTache : T_Tache(10); -- Instanciation
begin
   null;
end une_tache_typee; -- ne se termine que lorsque la tâche est
                     -- terminée
```

```
-----------------------------------------------------------------------
```

Notons qu'il est possible de changer la priorité d'une tâche au cours de son exécution grâce à la fonction `Set_Priority`.

II.3.3.b. Le moniteur Ada

Depuis Ada95, le moniteur de haut niveau (objet protégé) a été introduit. Le moniteur Ada permet d'implémenter tout type de synchronisation et de communication, synchrone et asynchrone (même puissance d'expression que les variables conditionnelles dans POSIX).

La spécification d'un objet protégé contient une partie privée, correspondant aux variables internes au moniteur, et un ensemble de primitives. Il y a trois types de primitives:

- *procédures (procedure)*: sont des primitives non réentrantes par rapport aux autres primitives de l'objet protégé. Donc, une procédure protégée permet des accès exclusifs en écriture/lecture aux variables internes au moniteur. De plus, elle est non bloquante, c'est-à-dire qu'elle n'intègre pas la possibilité d'attendre un événement du moniteur.
- *fonctions (function)*: permettent des accès concurrents en lecture seule sur l'objet protégé: elles ne peuvent pas modifier les variables internes de l'objet. Cependant, les fonctions sont réentrantes les unes par rapport aux autres. Les fonctions permettent la distinction lecteur/écrivain sur un moniteur.
- *procédures gardées (entry)*: sont des procédures dont l'accès est conditionné (gardé). Une `entry` est gardée par une expression booléenne, appelée une barrière, à l'intérieur de la partie d'implémentation de l'objet protégé. Si cette barrière est évaluée à Faux lorsqu'un appel sur cette `entry` est produit, la tâche faisant cet appel est bloquée jusqu'à ce que cette barrière soit évaluée à Vrai.

La norme Ada propose le protocole à priorité plafond immédiat pour éviter l'inversion de priorité lors d'accès concurrents à un objet protégé. Par conséquent, un objet protégé peut être muni d'une priorité: sa priorité plafond, c'est-à-dire la priorité maximale des tâches susceptibles de l'utiliser (par défaut, la priorité plafond d'un objet protégé est `priority'last`). Pour plus amples informations concernant objets protégés, le lecteur pourra consulter [Bur98].

II.3.3.c. Modules de données

Les modules de données DARTS peuvent être implémentés à l'aide de l'encapsulation du module de données dans un objet protégé. L'exclusion mutuelle est alors garantie par la non-réentrance de primitives d'accès à un objet protégé. Ainsi, l'implémentation du module de données, appelé «T_Désirée», accédé par deux tâches, T1 et T2, en Ada est :

```
-----------------------------------------------------------------------
with Procede; use Procede;
-- Module contenant des fonctions d'accès aux capteurs/actionneurs

procedure Main is
   -- Procédure montrant un exemple de module de données

   -- Spécification du Module de données
   protected T_Desiree is
      procedure Ecrire(Temp: Float);
      -- Modifie la valeur contenue
      function Lire return Float;
      -- Renvoie la valeur contenue
      private -- Variable interne
            Temperature: Float := (Lire_Capteur_Temperature);
            -- Initialisation de la température
   end T_Desiree;

   -- Implémentation du module de données
```

```
   protected body T_Desiree is
      procedure Ecrire(Temp: Float) is
      begin
            Temperature:= Temp;
      end Ecrire;
      function Lire return Float is
      begin
            return Temperature;
      end Lire;
   end T_Desiree;

   task T1; -- Spécification de la tâche modifiant module de donnée
   task body T1 is -- corps de la tâche
      Nouvelle_Temp: Float;
   begin
      -- Code d'initialisation des capteurs
      ...
      loop
            ...
            -- Lecture de capteur
            Nouvelle_Temp:= Lire_Capteur_Temperature;
            -- Ecrire dans le module de données
            T_Desiree.Ecrire(Nouvelle_Temp);
            ...
      end loop;
   end T1;

   task T2; -- Spécification de la tâche lisant module de données
   task body T2 is -- corps de la tâche
      Temp: Float;
   begin
      loop
            -- Lecture dans le module de données
            Temp:= T_Desiree.Lire;
            ...
      end loop;
   end T2;
begin -- Les tâches sont lancées
   null; -- Le programme principal ne fait rien
end Main; -- Attente de la terminaison des tâches
-----------------------------------------------------------------------
```

Dans cette implémentation, il n'est pas possible d'effectuer une initialisation du capteur avant de les lire. Par conséquent, avec ce code on ne peut pas garantir qu'une écriture sur le MDD a forcément lieu avant une lecture. Si cela est gênant (pas de valeur initiale neutre possible) alors on pourra utiliser une procédure gardée (`entry`) `Lire`, au lieu de la fonction `Lire` dont la garde sera ouverte après une écriture Pour le code d'une telle implémentation, lecteur pourra consulter [CG05].

II.3.3.d. Boîtes aux lettres

L'implémentation des boîtes aux lettres par objet protégé est assez similaire à l'implémentation proposée pour POSIX.

- ***Boîte aux lettres de taille 1 avec écrasement***

Ce type de BAL est simple à implémenter à partir d'un objet protégé. L'idée est que l'envoi de message est non bloquant (emploi d'une procédure), et que l'attente de message est bloquante (emploi d'une procédure gardée). Ainsi, l'implémentation de la communication entre la tâche T1 et la tâche T2 par boîte aux lettres de taille 1 avec écrasement en Ada est donnée ci-après:

```
-----------------------------------------------------------------------
procedure Main is
   -- Spécification de BAL de taille 1 avec écrasement
```

```
   protected BAL_Ecr_1 is
      procedure Envoyer(Temp: Float); -- Envoi d'un message dans BAL
      entry Recevoir(Temp: out Float); -- Attente d'un message

      private
            vide: Boolean := True; -- Etat de la boîte
            Temperature: Float;
   end BAL_Ecr_1;

   protected body BAL_Ecr_1 is
      procedure Envoyer(Temp: Float) is
      begin
            Temperature:= Temp;
            -- Dans ce cas, la Temperature reçoit Temp, même si la
            -- boîte était non vide (l'écrasement)
            vide:= False; -- Il y a un message non lu
      end Envoyer;

      entry Recevoir(Temp: out Float) when not vide is
      -- Il n'est possible de recevoir un message que s'il existe un
      -- message non lu
      begin
            Temp:= Temperature;
            vide:= True;
      end Recevoir;
   end BAL_Ecr_1;

   task T1; -- Spécification de la tâche envoyant un message
   task body T1 is -- Le corps de la tâche 1
      Nouvelle_Temp: Float;
   begin
      loop
            ...
            BAL_Ecr_1.Envoyer(Nouvelle_Temp);
            ...
      end loop;
   end T1;

   task T2; -- Spécification de la tâche attendant un message
   task body T2 is -- Le corps de la tâche 2
      Temp_Recu: Float;
   begin
      loop
            BAL_Ecr_1.Recevoir(Temp_Recu);
            ...
      end loop;
   end T2;
begin -- Les tâches sont lancées
   null; -- Le programme principal ne fait rien
end Main; -- Attente de la terminaison des tâches
-------------------------------------------------------------------
```

- ***Boîte aux lettres de taille 1 sans écrasement***

Par rapport à l'implémentation de la boîte aux lettres à écrasement, la différence au niveau implémentation réside uniquement dans le fait que l'envoi est une procédure gardée (`entry`). Ainsi, le code modifié de l'objet protégé est donné ci-après:

```
-------------------------------------------------------------------
protected BAL_1 is
   enry Envoyer(Temp: Float); -- Envoi d'un message dans BAL
   entry Recevoir(Temp: out Float); -- Attente d'un message

   private
      vide: Boolean := True; -- Etat de la boîte
      Temperature: Float;
```

```
   end BAL_1;

protected body BAL_1 is
   entry Envoyer(Temp: Float) when vide is
   begin
      Temperature:= Temp;
      -- Il n'est possible d'envoyer un message que si la boîte est
      -- vide (sans écrasement)
      vide:= False; -- Il y a un message non lu
   end Envoyer;

   entry Recevoir(Temp: out Float) when not vide is
   -- Il n'est possible de recevoir un message que s'il existe un
   -- message non lu
   begin
      Temp:= Temperature;
      vide:= True;
   end Recevoir;
end BAL_1;
-------------------------------------------------------------------------
```

II.3.3.e. Synchronisation

L'implémentation de la synchronisation est identique à l'implémentation du sémaphore à compte dont le code, placé dans un paquetage Ada, est présenté ci-dessous:

```
-------------------------------------------------------------------------
package Synchronisation is
   protected type T_Synchronisation is
      -- Implémentation d'une synchronisation à compte
      entry Wait; -- Attente de synchronisation
      procedure Signal; -- Déclenchement de synchronisation
   private
      valeur: Natural:= 0; -- Nombre de synchronisation non prise en
                           -- compte
   end T_Synchronisation;
end Synchronisation;

package body Synchronisation is
   protected body T_Synchronisation is
      entry Wait when valeur > 0 is
      -- Tant qu'il n'y a pas de déclenchement non pris en compte,
      -- les tâches appelant cette entrée sont bloquées et mises en
      -- attente dans la file d'attente de l'entrée
      begin
            Valeur:= Valeur - 1;
      end Wait;
      procedure Signal is
      begin
            Valeur:= Valeur + 1;
      end Signal;
   end T_Synchronisation;
end Synchronisation;
-------------------------------------------------------------------------
```

L'utilisation d'une synchronisation sur l'implémentation de la synchronisation entre deux tâches, T1 et T2, est de la forme suivante:

```
-------------------------------------------------------------------------
with Synchronisation; use Synchronisation;

procedure Main is
   S1: T_Synchronisation; -- Instanciation d'une synchronisation
   task T1;
   task body T1 is
   -- Tâche déclenchant T2 par synchronisation
   begin
```

```
      loop
            ...
            S1.Signal;
            ...
      end loop;
   end T1;
   task T2;
   task body T2 is
   -- Tâche en attente de synchronisation
   begin
      loop
            S1.Wait;
            ...
      end loop;
   end T2;
begin -- Les tâches sont lancées
   null; -- Le programme principal ne fait rien
end Main; -- Attente de la terminaison des tâches
-----------------------------------------------------------------------
```

II.3.3.f. Tâches périodiques

Depuis Ada95 la primitive d'attente de date (`delay until`) est utilisée. Ainsi, afin d'éviter un dérive des horloges, l'implémentation d'une tâche périodique, nommée `tache_periodique`, avec une période de 500 millisecondes en Ada est donnée ci-après:

```
-----------------------------------------------------------------------
with Ada.Real_Time; use Ada.Real_Time;

procedure Main is
   task type T_Periodique(Periode_ms: Integer);
   -- Type de tâche périodique avec période en millisecondes
   task body T_Periodique is
      Periode: constant Time_Span := Milliseconds(Periode_ms);
      -- Le type Time_Span défini dans Ada.Real_Time, petmet de
      -- représenter une durée
      Prochaine_Activation: Time := Clock + Periode;
      -- Le type Time défini dans Ada.Real_Time, représente une date
      -- Date de prochaine activation, initialisée à l'heure courante
      -- (retour de la fonction Clock) + la période
   begin
      loop
            ...
            delay until Prochaine_Activation;
            -- Attente de la prochaine date d'activation
            Prochaine_Activation:= Prochaine_Activation + Periode;
            -- Calcul de la prochaine date de réveil
      end loop;
   end T_Periodique;

   tache_periodique: T_Periodique(500);
   -- Instanciation d'une tâche de période 500 ms
begin -- Les tâches sont lancées
   null; -- Le programme principal ne fait rien
end Main; -- Attente de la terminaison des tâches
-----------------------------------------------------------------------
```

II.3.3.g. Tâches réveillées par interruption

Ada prévoit directement une gestion d'interruption de type ISR déclenchant une DSR. Une ISR doit forcément se trouver dans un objet protégé situé dans un paquetage. Le protocole à priorité plafond immédiat doit être utilisé, et la priorité plafond de l'objet protégé contenant l'ISR doit être une priorité d'interruption. Le lecteur qui s'intéresse à la gestion d'interruption de type ISR en Ada, peut consulter [Bur98].

III. Architecture DARTSVIEW

Dans le chapitre précédent, j'ai déjà montré comment implémenter les concepts de la méthode DARTS par LabVIEW, un langage de programmation graphique basé sur le langage G. En assemblant tous ces VIs une nouvelle palette, appelée «DARTSVIEW», est construite. C'est la bibliothèque des notations DARTS implémentées dans la programmation multitâche LabVIEW.

III.1. Présentation de la palette DARTSVIEW

La palette DARTSVIEW représentée dans ce manuscrit est réalisée sous la version 7.1 de LabVIEW. Elle comporte quatre sous-palettes et deux VIs. Les sous-palettes sont nommées respectivement «Tâche Matérielle», «Tâche Logicielle», «Outil Communication», et «Générer Code». Les deux VI sont appelés «Modèle DARTSVIEW du Système» et «Informations Enregistrées» (voir Figure III-1).

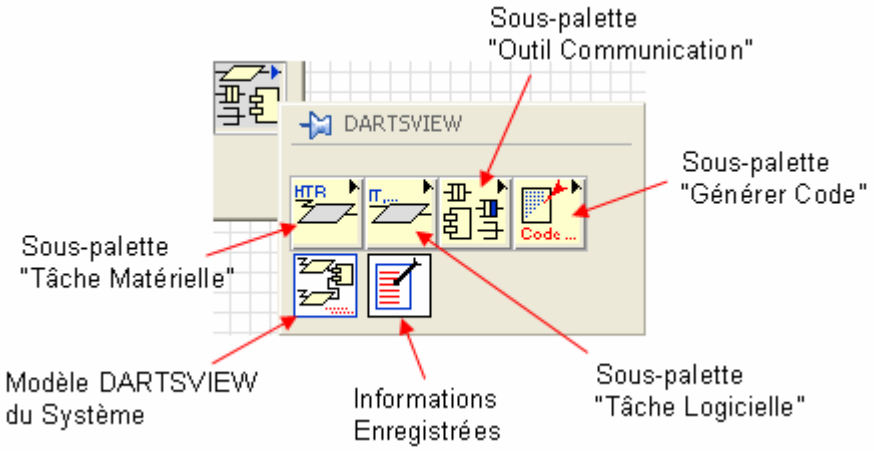


Figure III-1: Palette DARTSVIEW

Les icônes utilisées pour des VI de la palette DARTSVIEW sont conformes aux représentations présentées dans le Tableau 1.

III.1.1. Sous-palette «Tâche Matérielle»

«Tâche Matérielle» (voir Figure III-2) est la sous-palette des tâches matérielles soit périodiques déclenchées par une horloge temps réel, soit périodiques activées par une interruption externe ou un chien-de-garde. Une tâche matérielle peut communiquer avec d'autres tâches au moyen des outils de communication/synchronisation

Dans la Figure III-2, les modèles de tâches matérielles de gauche à droite et de haut en bas sont les suivants:

Entrée (IN)		Sortie (OUT)	
HTR	Représenter une tâche matérielle périodique activée par une horloge temps réel.	Sem.	Représenter une tâche qui produit un déclenchement en forme d'un sémaphore sur son arc de sortie afin de réveiller la tâche qui est en attente.
Evé,...	Représenter une tâche matérielle apériodique activée par un événement, une interruption, ou un CG.	BAL	Représenter une tâche qui envoie un message dans une BAL sur son arc de sortie à l'exécution.
Sem.	Représenter une tâche logicielle activée par une autre tâche à travers d'un sémaphore	BAL_Ecr	Représenter une tâche qui envoie un message dans une BAL à écrasement sur son arc de sortie à l'exécution
BAL	Représenter une tâche logicielle activée par une autre tâche à travers d'une BAL classique.	MDD	Représenter une tâche qui envoie sur son arc de sorite un appel de l'opération "Ecrire" d'un MDD pour lancer la mise à jour de sa donnée encapsulée
BAL_Ecr	Représenter une tâche logicielle activée par une autre tâche à travers d'une BAL avec écrasement.	MDD	Représenter une tâche qui envoie sur son arc d'entrée un appel de l'opération "Lire" d'un MDD pour acquérir sa donnée la plus récente
HTR	Représenter une tâche périodique activée par une horloge temps réel, elle reçoit un paramètre d'entrée au moyen d'un outil de communicationà chaque iération (i.e., BAL_Ecr, MDD).		
	Représenter une tâche qui reçoit deux ou plusieurs paramètres d'entrée de même type lorsqu'elle est activée.		

Tableau 1: Représentations des VIs dans la palette DARTSVIEW

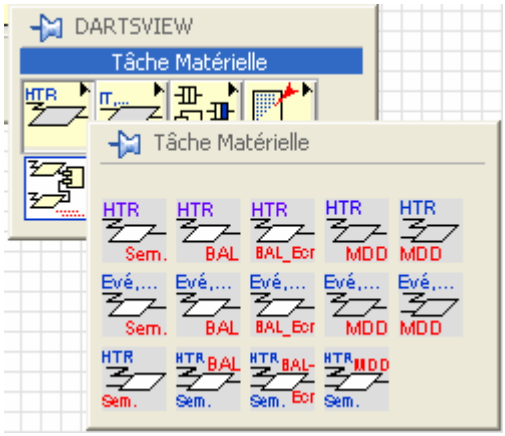


Figure III-2: Sous-palette «Tâche Matérielle»

- «HTR – (out) Sémaphore»: (voir Figure III-3) est un modèle de tâche matérielle périodique, pouvant déclencher une tâche en attente de synchronisation à l'aide d'un sémaphore.

  Si le temps de réponse dépasse l'échéance de la tâche, une erreur est déclenchée lance avec un traitement des erreurs associé (la séquence «1»). Par défaut, la tâche est à échéance sur requête.

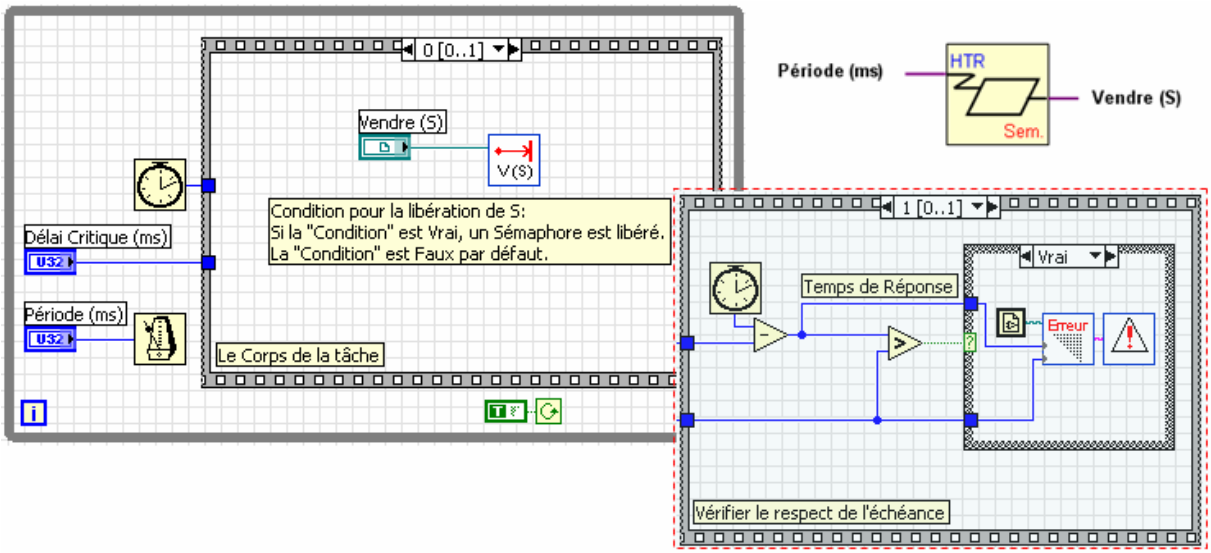


Figure III-3: Diagramme de la tâche «HTR – (out) Sémaphore»

«HTR – (out) BAL»: est un modèle de tâche matérielle périodique, pouvant envoyer un message dans une BAL.

«HTR – (out) BAL_Ecr»: modèle de tâche matérielle périodique, envoyant un message dans une BAL à écrasement.

Le diagramme du modèle de la tâche «HRT – (out) BAL_Ecr» est de même que celui du modèle de tâche «HRT – (out) BAL» présenté sur la Figure III-3). La seule différence est que le VI «Ecrire Outil Commu» est exécutée en mode «Ecrire (BAL_Ecr)». En fait, ce VI est un VI polymorphe que peut s'exécuter soit en mode «Ecrire (BAL)», soit en mode «Ecrire (BAL_Ecr)».

«HTR – (out) MDD»: modèle de tâche matérielle périodique modifiant un MDD.

«HTR – (in) MDD»: modèle de tâche matérielle périodique lisant un MDD.

«Evénement – (out) Sémaphore»: (voir Figure III-4) représente des tâches matérielles apériodiques activées par des signaux «Interruption – IT» en provenance du procédé, ou des signaux «Chien de garde – CG» qui proviennent des horloges internes utilisées comme des réveils (séquence «0») déclenchant une autre tâche par synchronisation.

«Evénement – (out) BAL»: identique au précédent sauf que la tâche écrit dans une BAL.

«Evénement – (out) BAL_Ecr»: identique au précédent sauf que la tpache écrit dans une BAL à écrasement.

«Evénement – (out) MDD»: identique au précédent sauf que la tâche écrit dans un MDD.

«Événement – (in) MDD»: identique au précédent sauf que la tâche lit dans un MDD.

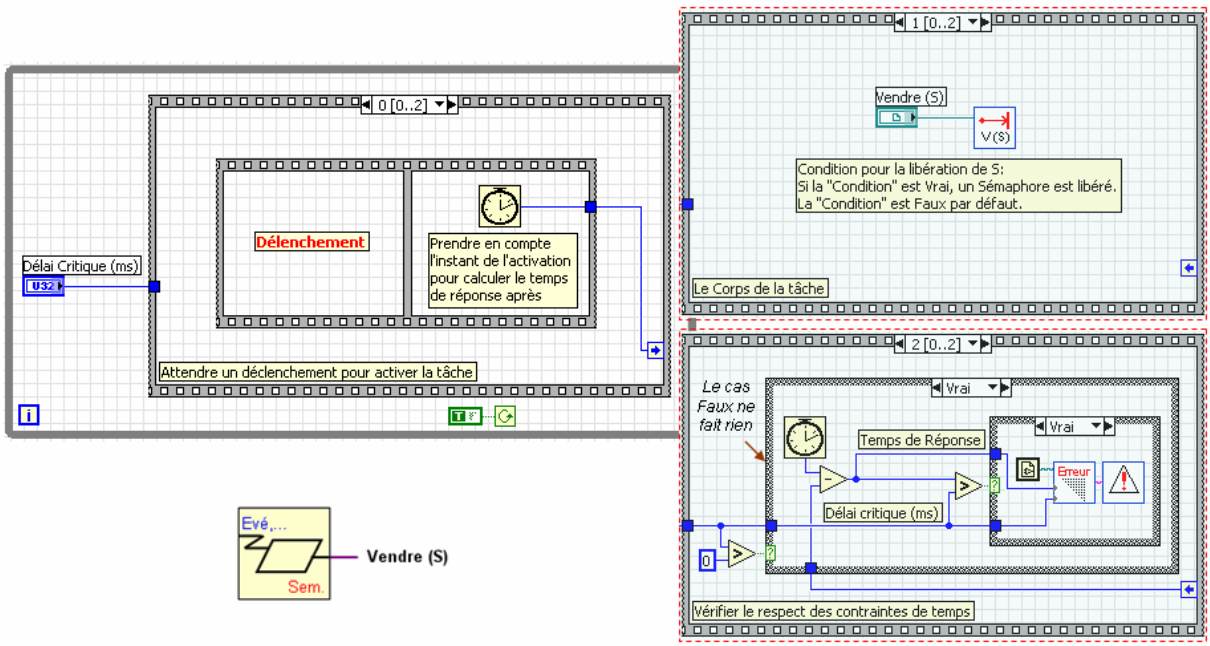


Figure III-4: Diagramme de la tâche «Evénement – (out) Sémaphore»

- «HTR – (in) Sémaphore»: (voir Figure III-5) modèle de tâche périodique activable/désactivable par sémaphore. La libération de son sémaphore déclenche son fonctionnement périodique, et une second libération l'arrête, jusqu'à une nouvelle libération qui l'active, etc. Ce fonctionnement correspond à un fonctionnement de type E/D en SA/RT.

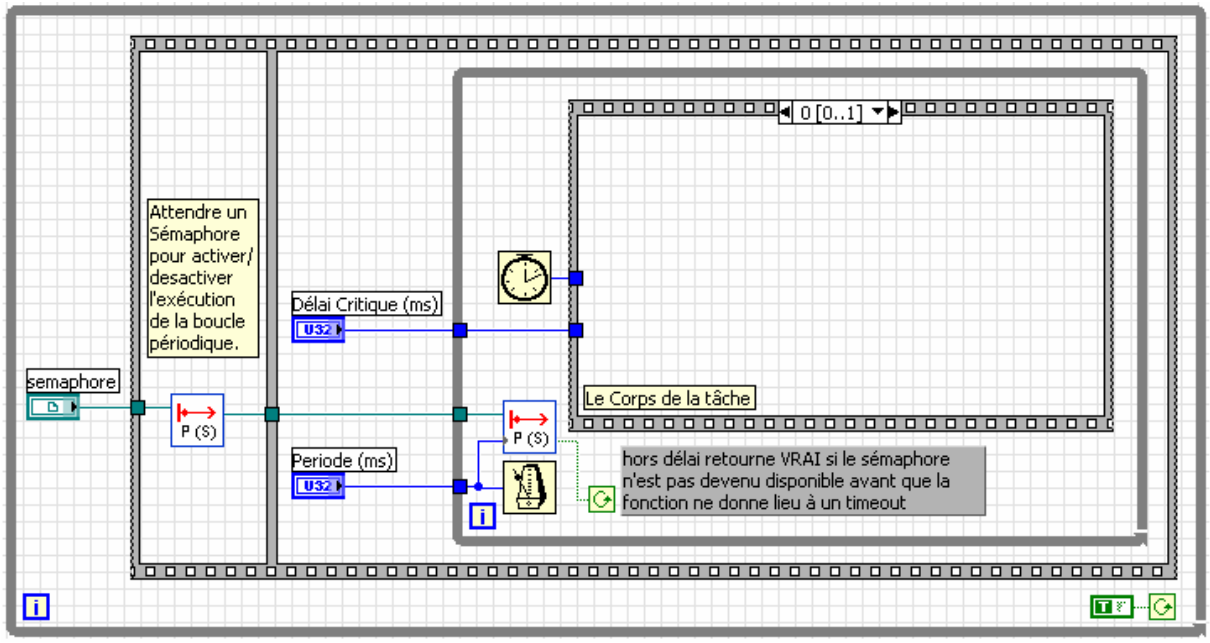


Figure III-5: Digramme de la tâche «HTR – (in) Sémaphore»

- Il en va de même pour les VI suivant qui utilisent respectivement une BAL en sortie, une BAL à écrasement en sortie, une synchronisation en sortie : « HTR – (in) Sémaphore – (out) BAL», «HTR - (in) Sémaphore – (out) BAL_Ecr», «Tâche périodique – (in) Sémaphore – (out) MDD».

Il est important de remarquer que les quatre derniers modèles n'existent pas dans la méthode DARTS. Ils sont des éléments de la méthode CODARTS (une extension de la méthode DARTS orientée objets concurrents).

III.1.2. Sous-palette «Tâche Logicielle»

La sous-palette «Tâche Logicielle» de la palette DARTSVIEW comporte les modèles des tâches logicielles DARTS, qui sont activées par des autres tâches (soit matérielles, soit logicielles) au moyen des outils de la communication ou de la synchronisation. Les modèles des tâches logicielles dans cette sous-palette représentés sur la Figure III-6 de gauche à droite sont:

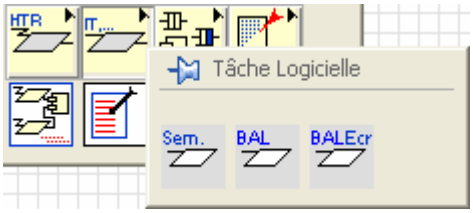


Figure III-6: Sous-palette «Tâche Logicielle»

- «Tâche – (in) Sémaphore»: (voir Figure III-7) modèle de tâche logicielle déclenchée par sémaphore.

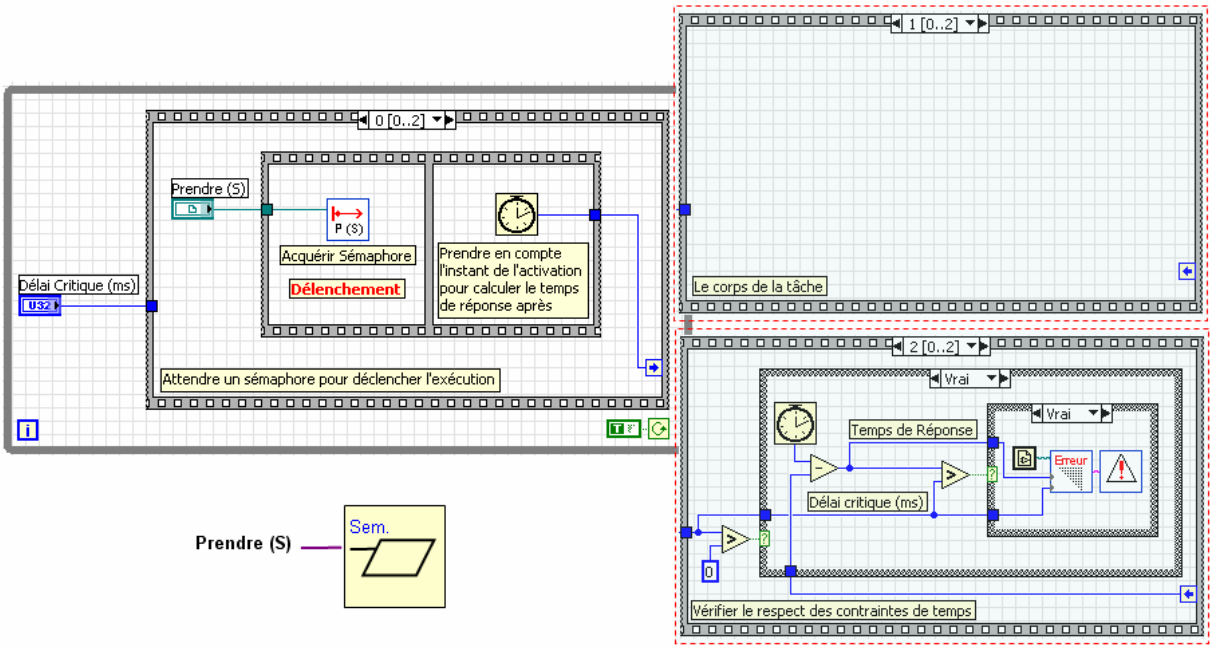


Figure III-7: Diagramme de la tâche «Tâche – (in) Sémaphore»

- «Tâche – (in) BAL»: modèle de tâche logicielle déclenchée par arrivée d'un message dans une BAL.

- «Tâche – (in) BAL_Ecr»: modèle de tâche logicielle en attente de message dans une BAL à écrasement.

III.1.3. Sous-palette «Outil Communication»

La sous-palette «Outil Communication» (voir Figure III-8) comporte des outils LabVIEW permettant d'implémenter la communication et la synchronisation entre tâches, qui se trouvent souvent dans les diagrammes DARTS. Comme j'en ai déjà expliqué dans le chapitre précédent, ce sont la synchronisation par Sémaphore, la communication synchrone faiblement couplée par BAL classique, la communication asynchrone par BAL avec écrasement (BAL_Ecr), et la communication par le module d'encapsulation de données (MDD).

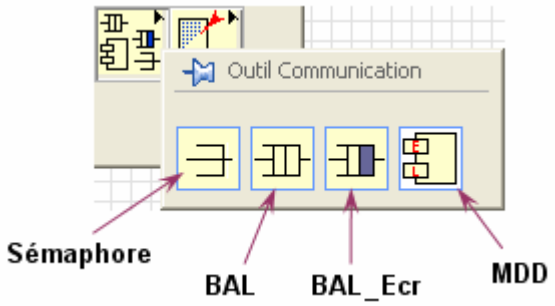


Figure III-8: Sous-palette «Outil Communication»

- Synchronisation par Sémaphore: (voir Figure III-9) c'est le sémaphore privé utilisé pour la synchronisation. Afin de synchroniser l'exécution des tâches, un sémaphore de taille 1 est créé par la fonction «Créer un sémaphore» de manière à ce que sa valeur de compte soit diminuée jusqu'à 0 en appelant n fois de la fonction «Acquérir un sémaphore» (dans ce cas, n = 1) dès sa création. Il s'agit d'un sémaphore zéro. Ainsi, la tâche «Emettrice» et le tâche «Réceptrice» se connectent à ce VI à travers respectivement le connecteur «Libérer Sémaphore» et le connecteur «Acquérir Sémaphore».

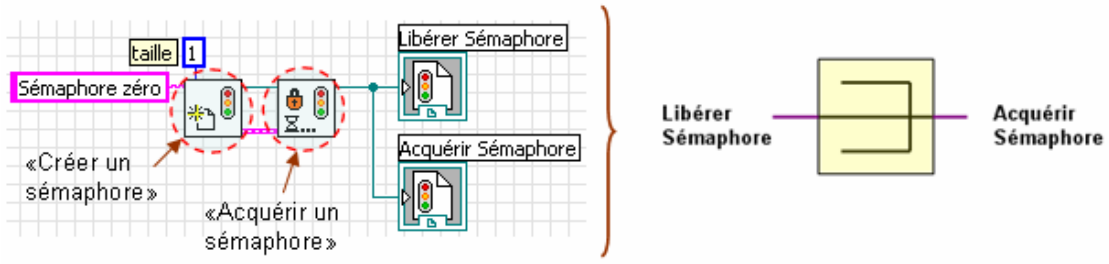


Figure III-9: Diagramme du VI de la synchronisation par Sémaphore LabVIEW

- Communication synchrone par BAL classique: (voir Figure III-10) crée une BAL qui sera utilisée par des tâches (une ou plusieurs émettrices, une réceptrice).

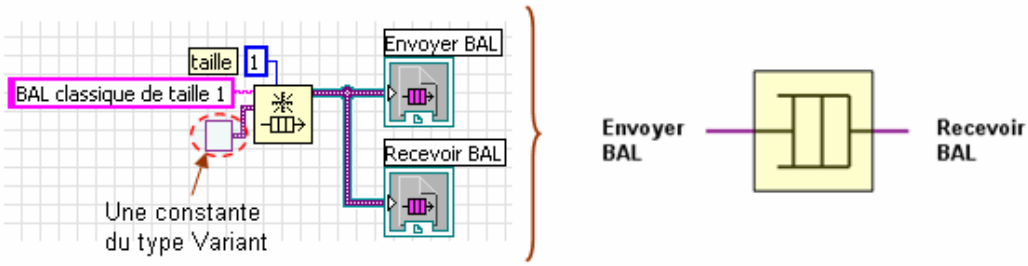


Figure III-10: Diagramme du VI de la communication synchrone par BAL LabVIEW

- Communication synchrone par BAL avec écrasement (BAL_Ecr): (voir Figure III-11) similaire à la BAL, mais sera gérée comme un BAL à écrasement.

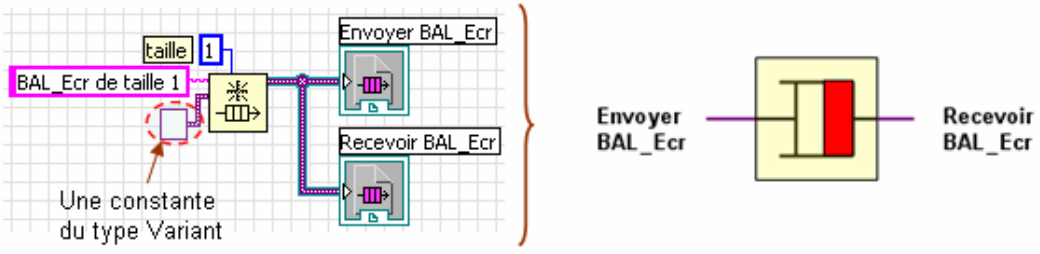


Figure III-11: Diagramme du VI de la communication synchrone par BAL_Ecr

- Communication asynchrone par Module d'encapsulation de données (MDD): (voir Figure III-12) le MDD dans cette sous-palette est un exemple typique de Module de Donnée du type «Lecteur/Ecrivain» en exclusion mutuelle. Il comporte de deux opérations, appelées «Ecrire MDD» et «Lire MDD», et une variable interne de type «Variant». L'atomicité de ces deux opérations est garantie par un sémaphore d'exclusion mutuelle.

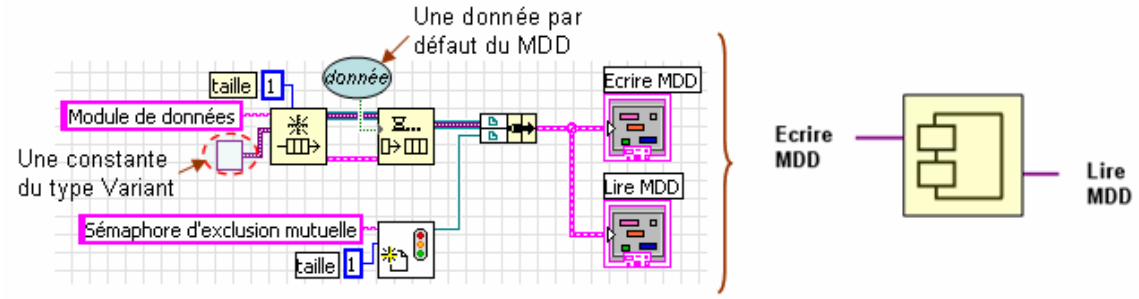


Figure III-12: Diagramme de la communication asynchrone par MDD

III.1.4. Sous-palette «Générer Code»

La sous-palette «Générer Code» (voir Figure III-13) contient des VI qui permet de générer le code à partir du modèle DARTSVIEW vers un langage de programmation impératif ou un outil de validation temporelle.

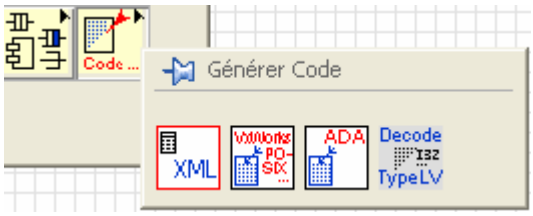


Figure III-13: Sous-palette «Générer Code»

Les modèles pour la génération de code présentés dans cette sous-palette de gauche à droite sont «Générer Fichier XML», «Générer Code Norme Langage C», «Générer Code Ada», et «Interpréter Descripteur de Type LV». Leurs diagrammes sont donnés en détail dans la partie de la génération de code à partir d'un modèle DARTSVIEW aux paragraphes suivants de ce chapitre.

III.1.5. «Modèle DARTSVIEW du Système»

Le VI «Modèle DARTSVIEW du Système.vit» (voir Figure III-14) est un modèle de VI qui permet de construire un diagramme DARTS en utilisant les modèles de tâches et les outils de communication/synchronisation présentés auparavant, en vue de pouvoir générer du code.

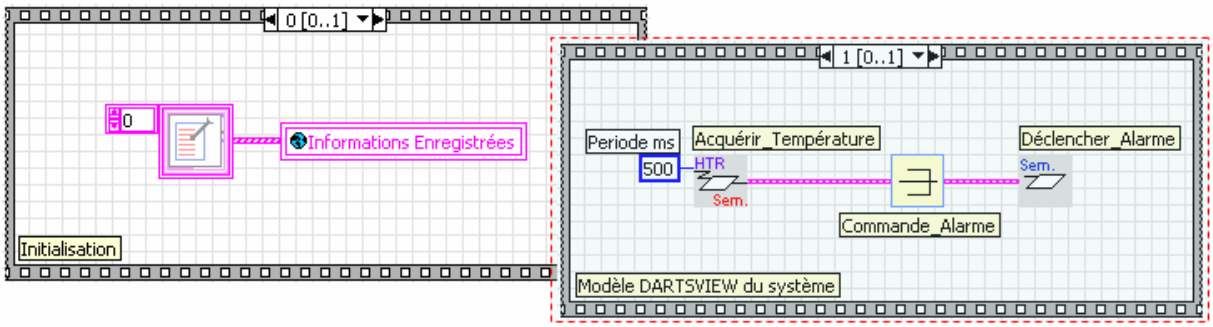


Figure III-14: Diagramme du VI Template «Modèle DARTSVIEW du Système»

Ce VI comporte une structure de Séquence dont la première séquence sert à l'initialisation d'une variable globale, nommée «Informations Enregistrées», qui sera utilisée pour la génération de code, tandis que la deuxième séquence contient le programme implémenté en DARTSVIEW.

III.1.6. «Informations Enregistrées»

Ce VI initialise une variable globale qui sera utilisée par chaque élément DARTSVIEW pour s'y déclarer. Cette variable contiendra les informations sur les tâches, et éléments de communication/synchronisation de l'application, ce qui permettra dans l'étape ultérieure de générer du code. Cette initialisation est exécutée une fois lorsque l'application DARTSVIEW du système est lancée. De même que la sous-palette «Générer Code», ce VI sera présenté en détail aux paragraphes suivants.

III.2. Palette DARTSVIEW et le premier V du cycle de vie en W

Dans le cycle de développement en W, le premier V correspond au cycle de développement d'un logiciel en V classique avec les différentes phases bien ordonnées qui sont nécessaires à la mise en œuvre d'une application (voir Figure III-15). Ainsi, dans le cadre des applications temps réel, ou des applications de contrôle/commande multitâches, à partir de la spécification par exemple SA-RT (Structured Analysis For Real-Time Systems), en choisissant la méthode DARTS, une architecture multitâche peut être générée dans la phase de conception («Conception Préliminaire» et «Conception Détaillée»).

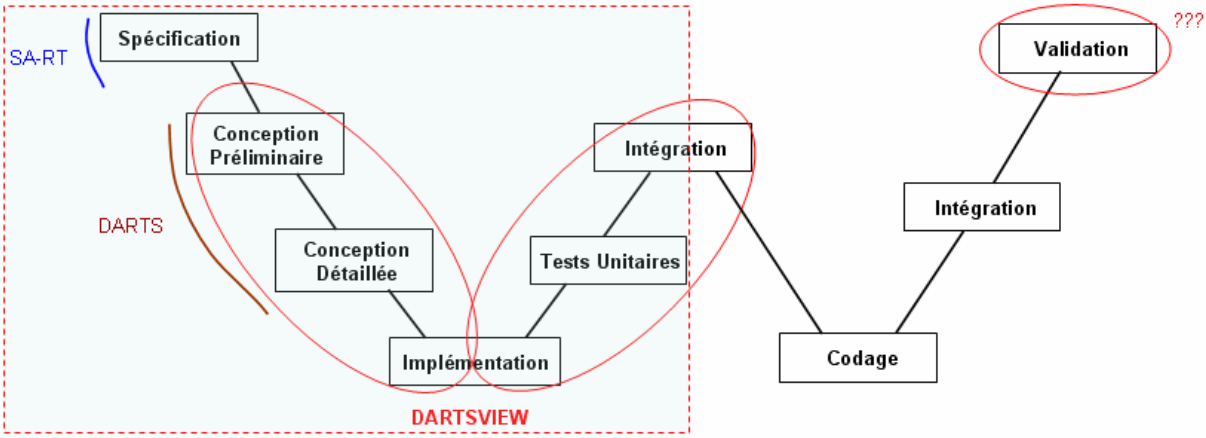


Figure III-15: Palette DARTSVIEW et le premier V du cycle de vie en W

Les éléments prédéfinis de la palette DARTSVIEW permettent au concepteur d'implémenter directement en LabVIEW les diagrammes d'Architecture DARTS dans la phase d'implémentation, et puis de faciliter les tests unitaires et les tests d'intégration. Dans le premier V du cycle de vie en W, la palette DARTSVIEW sert donc à tester fonctionnellement le système, notamment en le couplant à un simulateur du procédé (écrit typiquement an MATLAB/SIMULINK). L'application s'exécute sur un microprocesseur puissant de type station de travail, ce qui dans le cas général lève la plupart des contraintes temporelles. On teste donc :

- les composants indépendants du système répondent chacun à leurs spécifications et fonctionnent correctement en toutes circonstances;
- les relations d'utilisation entre les composants, le fonctionnement des tâches qui s'appuient sur ces composants, des mécanismes de communication et de synchronisation entre tâches, etc.

Cependant, si l'objectif de l'application est de s'exécuter sur un système embarqué avec des contraintes temporelles strictes, le langage de programmation cible dans ce cas sera vraisemblable un langage impératif (tel que «C-based», Ada), par ailleurs, une validation temporelle doit être réalisée à la fin du cycle de vie. Par conséquent, le second V consiste à implanter l'architecture dont le comportement parallèle a été déjà testé sur une station de travail, sur un microprocesseur ou microcontrôleur pouvant être embarqué. Je parle de codage croisé et d'intégration avec le noyau pour lesquels la génération de code à partir d'un modèle DARTSVIEW joue un rôle très important.

Suite à la présentation de l'implémentation des applications multitâches par DARTSVIEW dans le chapitre II, les paragraphes suivants présentent les aspects avancés que DARTSVIEW fournit pour la génération de code: d'abord, j'introduis succinctement le langage XML, puis je parle des outils XML dans la version 7.1 de LabVIEW et explique comment XML permet d'ajouter simplement différents langages d'implémentation orientés temps réel ou validation, ce qui justifie le rôle de DARTSVIEW dans le second V du cycle de vie en W.

Afin de terminer cette section, il est important de noter que les modèles prédéfinis dans la palette DARTSVIEW sont les modèles des éléments utilisés fréquemment dans la plupart des diagrammes DARTS. Il s'agissait des cas les plus généraux auxquels nous appliquons les idées principales de l'implémentation des notions d'un diagramme DARTS au profit des modèles proposés dans la palette DARTSVIEW. En réalité, il y a de nombreuses de possibilités qui peuvent advenir dans la relation entre tâches. Il n'est donc pas possible de réaliser tous ces cas. En conséquence, les utilisateurs dans certains cas peuvent s'inspirer des principes que je présente dans ce manuscrit, construire et/ou modifier le modèle DARTSVIEW, s'il est déjà existant, pour qu'il s'adapte mieux à leurs applications.

III.3. Introduction à XML

XML (eXtensible Markup Language) est un langage permettant de mettre en forme des documents grâce à des balises qui décrivent le contenu plutôt que la présentation (contrairement à HTML).

XML = HTML – {balises de présentation} + DTD

Un document XML consiste en une structure arborescente et peut être utilisable dans tous les domaines, par toute application pourvue d'un «*parser*». Depuis le 1998, les spécifications XML 1.0 ont été reconnues comme recommandations par le W3C (World Wide Web Consortium) [W3C].

Généralement un fichier XML est structuré en trois parties: un prologue, une DTD, et un arbre des éléments. Le XML prologue comporte des paramètres présentant la version de la norme XML, le jeu de caractères, le mode d'utilisation de DTD, etc. (voir Figure III-16).

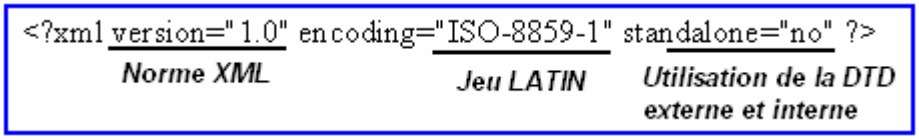


Figure III-16: Un XML prologue

La Figure III-16 représente un prologue utilisé couramment dans les fichiers XML. Le paramètre, appelé «*standalone*», détermine que le document XML a permission de faire référence aux ressources externes ou non (donc, dans ce cas le document XML a l'autorisation de se référencer à une ressource externe, qui est en général un fichier XML-DTD, une Déclaration de Type de Document dont je parle aux paragraphes suivants).

III.3.1. Fichier XML bien formé

Un fichier XML doit être soit bien formé (*well-formed*), soit valide. Un fichier XML est dit «bien formé» si celui-ci ne respecte que les règles de la grammaire XML, c'est-à-dire qu'il est syntaxiquement correct. Les syntaxes de documents XML que je présente ci-après sont ceux de base (pour les détails, le lecteur peut consulter [W3C] [LB99]):

- Soit il n'existe pas de la déclaration de DTD dans les documents XML, ou s'il y en a, le «parser» la traite sans vérifier leur respect de DTD.

- XML est un langage sensible à la casse. Il fait distinction entre les lettres majuscules et les lettres minuscules. Par conséquent, <Tâche> et <tâche> sont interprétées en tant que deux balises différentes.
- Chaque élément XML est défini par une paire de balises ou bien par une balise unique. La balise de fin doit être conforme à la balise de début. Les éléments XML sont les marquages dans lesquels les textes (PCDATA, Parsed Character Data) sont enfermés.
- Les éléments peuvent être emboîtés, mais il faut assurer que chacun n'est pas recouvert par un autre. Et il est important de noter qu'il existe un seul élément racine dans un fichier XML.
- Un ou plusieurs attributs peuvent être ajoutés dans la balise de début de chaque élément. Ils contiennent des informations additionnelles sur l'élément. Un attribut comporte un nom et une valeur, donc sa valeur doit être mise entre guillemets. La syntaxe de déclaration des attributs d'un élément XML est la suivante:

  ```
  <NomElement NomAttribut1 = "valeur" NomAttribut2 = "valeur" […]>
  ```

  Contrairement à l'élément, l'attribut contient seulement des données caractères (CDATA).
- Les commentaires peuvent être insérés n'import où dans un document à l'extérieur des marquages selon la syntaxe suivante:

  ```
  < ! -- c'est un commentaire -- !>
  ```
- XML fournit des outils supplémentaires qui permettent aux développeurs de pouvoir manipuler plus finement leurs documents et leur traitement. L'instruction de traitement (*Processing Instruction*, PI) est un tel outil. Par exemple, nous voyons souvent dans un fichier HTML les PIs ci-dessous:

  ```
  <Phrase>C&apos;est <?activer gras?> un bonheur!
                     <?désactiver gras?></ Phrase>
  ```

  Donc «`<?activer gras?>`» et «`<?désactiver gras?>`» sont les deux PIs utilisées pour mettre les mots «un bonheur» en gras.

Toutes les syntaxes représentées ci-dessus sont les syntaxes élémentaires que les documents XML doivent satisfaire pour être bien-formés. Remarquons que ces règles ne sont pas difficiles, même pour les gens qui ne connaissent pas XML, ils peuvent les apprendre aisément afin de créer eux-mêmes des fichiers XML «bien-formé». Le fichier XML «contacts.xml» ci-après est un exemple:

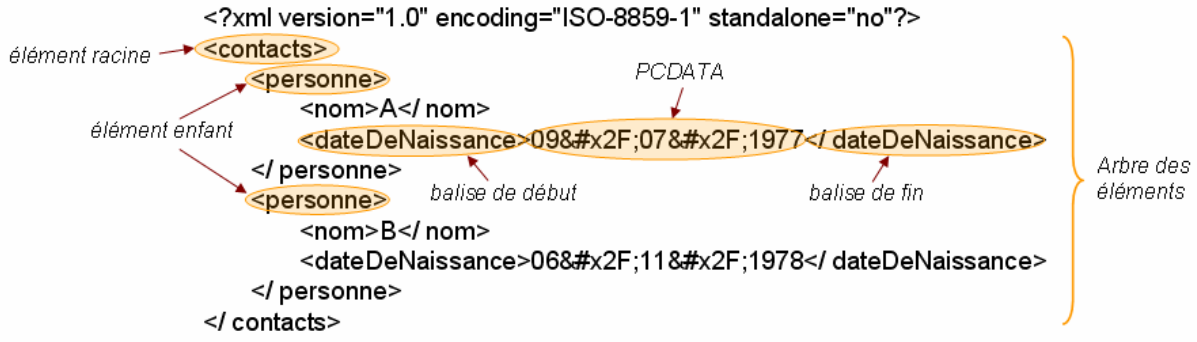


Figure III-17: «contacts.xml», un fichier XML bien-formé

III.3.2. Validation des fichiers XML

XML est un langage hiérarchique. Les éléments XML peuvent être structurés en un arbre des éléments, ce qui permet de faciliter et de faire rapidement des accès à un ou plusieurs élément(s) dans cet arbre. La Figure III-18 montre l'arbre des éléments du fichier XML «contacts.xml» présenté ci-dessus. A partir de l'élément racine «contacts», il existe deux éléments d'enfant ayant

les mêmes marquages, «personne», dont chacun comporte ensuite les deux éléments du type PCDATA (*Parsed Character Data*), nommés respectivement «nom» et «dateDeNaissance». Grâce_à l'outil de référencement d'une structure arborescente d'éléments d'un fichier XML tel que le «XPointer» [LB99], le texte «A» de l'élément «nom» du premier élément «personne» peut être référencé aisément par la syntaxe suivante:

```
root().child(1,personne).child(1,nom)                    (*)
```

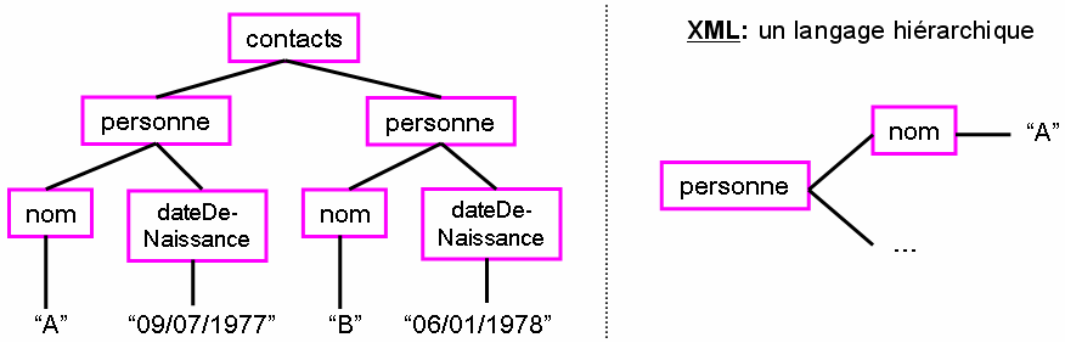


Figure III-18: Hiérarchie dans XML

La DTD (Déclaration de Type de Document) permet de vérifier automatiquement qui vérifie la conformité d'un document XML. On appelle cette vérification la validation.

La validation s'effectue toujours avant l'entrée d'un document XML dans un processus. Un document XML est dit «valide» s'il contient et se conforme à une DTD, qui est structurée en un ensemble de déclaration permettant de définir le type des éléments, leur liste d'attributs, des entités, des notations,… Une DTD doit être déclarée dans un document XML entre le prologue et l'élément racine. Elle peut être soit interne, soit externe, ou interne/externe. Mais il faut assurer qu'une seule DTD est définie pour chaque document XML.

```
< !DOCTYPE NomElémentRacine [... déclarations ...]>
```

Au cas où la DTD peut comporter un sous-ensemble (ou une partie) externe, il s'agit d'un document XML dont le paramètre «*standalone*» du prologue prend la valeur de «*no*», la syntaxe s'appliquant à une telle DTD est :

```
< !DOCTYPE NomElémentRacine SYSTEM "URIpourSousEnsembleExterne"
[... déclarations ...]>
```

L'objectif de ce chapitre ne cerne pas en détail de XML-DTD, donc dans les paragraphes suivants je présenterai un nombre limité des possibilités de déclaration, qui sont utilisées dans les fichiers XML-DTD liés à DARTSVIEW et la génération de code.

L'annexe A présente quelques bases sur XML pour le lecteur intéressé.

III.4. Palettes XML de LabVIEW

LabVIEW est un langage de programmation graphique fortement typé. Pour chaque type de données LabVIEW, il existe une balise prédéfinie correspondant à la syntaxe XML lorsque l'on effectue un échange de données entre LabVIEW et XML. Dans LabVIEW, nous trouvons deux palettes spécifiques pour les traitements des documents XML, ce sont la sous-palette «XML» de la palette «Manipulation de données» et la sous-palette «XML» fournie dans le Toolkit «Internet» à partir de la version 7.0 de LabVIEW.

III.4.1. Palette «XML» dans la palette «Manipulation de données»

La palette «XML» dans la palette «Manipulation de données» (voir Figure III-19**Error! Reference source not found.**) comporte les VIs et les fonctions permettant de manipuler des données LabVIEW au format XML: le VI «Coder une chaîne XML» convertit des caractères

spéciaux en syntaxe de textes XML conformément au jeu de caractère Latin-1; le VI «Décoder une chaîne XML» reconvertit la syntaxe XML pour des caractères spéciaux en ces caractères spéciaux; ou la fonction «Aplatir en XML» convertit des données de n'importe quel type en une chaîne XML selon le schéma XML de LabVIEW; etc.. Avec cette palette «XML», LabVIEW convertit des données en des fichiers XML conformément à un schéma établi et fixé. Nous ne pouvons pas actuellement créer des schémas personnalisés, et nous ne pouvons pas contrôler la manière dont LabVIEW gère les tags de chaque partie de donnée. Par conséquent les outils XML présents par défaut ne nous seront pas utiles.

III.4.2. Sous-palette «XML» dans le Toolkit Internet de LabVIEW

Les outils XML proposés dans cette palette sont plus flexibles (voir Figure III-19). Ce sont des VI, des nœuds de méthode, des nœuds de propriétés qui nous permettent des traitements des fichiers XML de façon plus précise et circonstanciée : par exemple, pour ouvrir ou sauvegarder un fichier XML, on utilise le VI «XML Load Document», ou le VI «XML Save Document»; alors que pour chercher une balise par son nom, on utilise le nœud de méthode «Get Element By Tag Name», etc.

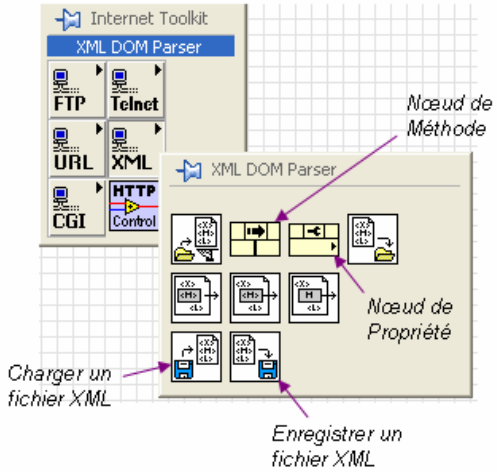


Figure III-19: Palette «XML» dans le Toolkit Internet de LabVIEW

Un «*parser*» est intégré au VI «Charger un fichier XML» (ou «*XML Load Document*») afin de vérifier le respect des règles de la grammaire XML d'un fichier à l'ouverture. La validation d'un fichier XML est effectuée par une DTD (soit interne, soit externe) grâce aux paramètres «*Validate on Load*» et «*Load External DTD*» de ce VI, donc «*Validate on Load*» est mis à «*Auto/Always*» et «*Load External DTD*» est mis à «*True*» en cas de DTD externe. Si le fichier XML n'est pas bien-formé, ou n'est pas valide, une erreur est générée.

Le VI «Enregistrer un fichier XML» (ou «*XML Save Document*») permet d'enregistrer un document XML à partir d'un modèle DOM (*Document Object Model*), créé par soit le VI «Charger un fichier XML», soit le VI «Ouvrir XML». Grâce à des méthodes et des propriétés proposées par respectivement le nœud de méthode et le nœud de propriété de cette palette, le traitement d'un fichier XML peut être effectué d'une manière indépendante des types de données LabVIEW (c'est-à-dire avec des fichiers XML avec des tags d'utilisateurs qui décrivent le contenu et sont conformes aux recommandations XML-DTD 1.0).

La Figure III-20 représente un exemple de l'utilisation des outils de la palette «XML» dans le Toolkit Intenet: le fichier «contacts.xml» est lu et validé par le VI «Charger un fichier XML» (puisque le paramètre «*Load External DTD*» est mis à Vrai, une DTD externe, nommée «contacts.dtd», peut être utilisée pour la validation de ce fichier). Une erreur se lance si ce fichier n'est pas valide, qui correspond au cas «Vrai» de la boucle conditionnelle. Au contraire, dans le cas «Faux», nous voyons qu'il y a deux nœuds de méthode, donc l'un, appelé «*Get XML*», pour lire le fichier «contacts.xml», et l'autre, appelé «*Get Elements By Tag Name Array*», pour

chercher tous les éléments ayant le tag, nommé «nom». Puis la récupération de textes dans le contenu de ces éléments est effectuée aisément par la fonction «Obtenir le Texte d'un Tag XML».

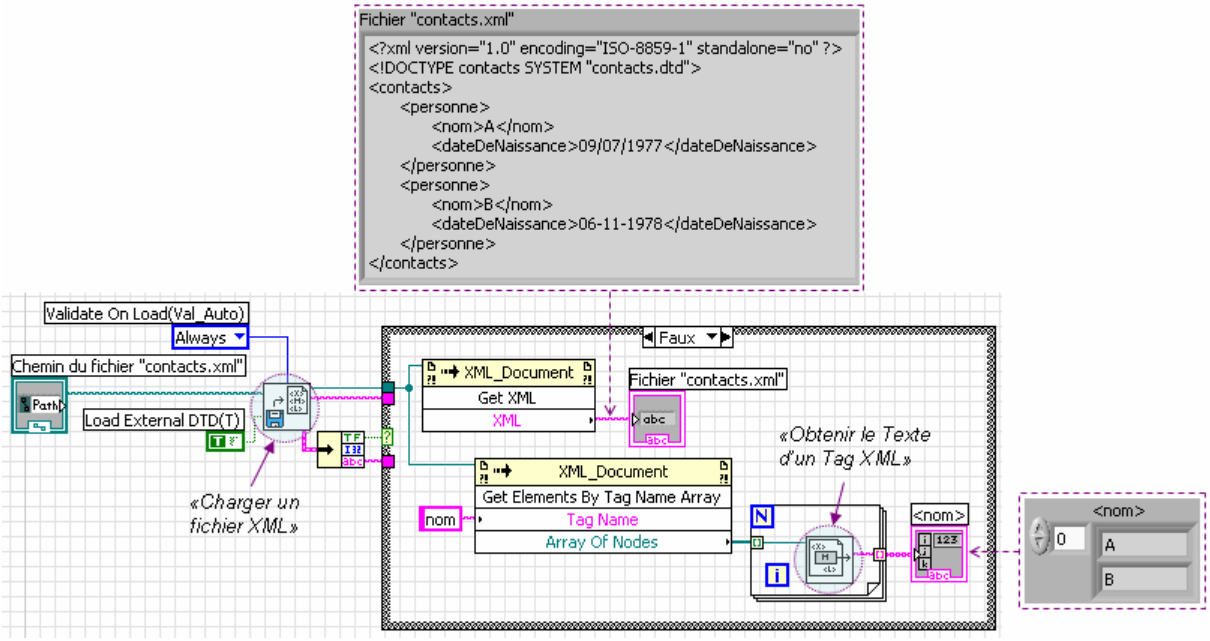


Figure III-20: Fichier «contacts.xml» et la palette «XML» dans le Toolkit Internet

III.4.3. Palettes XML de LabVIEW: Désavantages

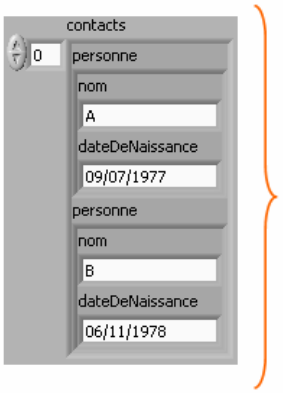
LabVIEW est un langage fortement typé, tandis que XML est une norme indépendante du langages de programmation. La validation de documents XML par DTD en dessus présente une lacune considérable de types de données pour le contenu des éléments XML. En fait, le contenu d'un élément XML est effectivement soit un groupe d'un ou plusieurs sous-éléments/ sous-groupes de données caractères (#PCDATA), soit un élément vide (EMPTY), ou un élément qui comporte des textes et des éléments d'enfants dont les contenus sont déjà bien déclarés. Les types de données LabVIEW (par exemple, I8, I32, DBL, Cluster, Array, etc.) ne sont pas évidemment définis dans les syntaxes de DTD.

Donc, afin de générer des fichiers XML-DTD avec des balises non-conformes au format de XML-LabVIEW pour échanger des informations entre LabVIEW et un outil tiers, l'idée est d'utiliser un attribut permettant de représenter le type de données LabVIEW de chaque élément XML généré. Par exemple, dans LabVIEW une valeur «a» du type entier va correspondre à un élément XML de même nom tel celui-ci-dessous:

```
...
<!ELEMENT a (#PCDATA)>
<!ATTLIST a type (I8|I32|DBL|Cluster|Array|…) #REQUIRED>
...
<a type="I32">8</a>
...
```

«type» est un attribut d'énumération qui comporte une liste de tous les types de données LabVIEW. C'est un attribut obligatoire (#REQUIRED), donc chaque élément instancié de «a» doit fournir explicitement une valeur pour cet attribut parmi ses valeurs prédéfinies. La vérification de la cohérence de types aura lieu lors de la lecture des documents XML en associant un autre outil tel que la palette «Manipulation de données» de LabVIEW. Par exemple, à partir de la liste de types de données de tous les éléments dans un fichier XML, l'utilisateur fournit le type de données qu'il souhaite récupérer, au connecteur «type» de la fonction «Redresser à partir d'une chaîne». Il est important de noter que ce type doit être correct, sinon une erreur est lancée. Et en écriture, afin de générer tel fichier XML, nous utilisons la fonction «Aplatir en chaîne», qui

nous permet de convertir une donnée quelconque en une chaîne de donnée aplatie de valeur binaire, et une chaîne de type correspond à un descripteur de type décrivant le type de données de cette donnée.




```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<contacts type="Array">
    <personne type="Cluster">
        <nom type="String">A</nom>
        <dateDeNaissance type="String">09&#x2F;07&#x2F;1977</dateDeNaissance>
    </personne>
    <personne type="Array">
        <nom type="String">B</nom>
        <dateDeNaissance type="String">06&#x2F;11&#x2F;1978</dateDeNaissance>
    </personne>
</contacts>
```

Figure III-21: «contacts.xml», un fichier XML avec des balises décrivant le contenu

La Figure III-21 présente le fichier «contacts.xml» avec des balises qui décrivent le contenu, généré dans LabVIEW à partir d'un tableau, nommé «`contacts`», de cluster, nommé «`personne`» (donc, chaque cluster comporte les deux éléments de type chaîne de caractères, nommés «`nom`» et «`dateDeNaissance`» respectivement). Par rapport au fichier «contacts.xml» sur la Figure III-17, ce fichier est ressemblant, la différence est qu'un attribut, nommé «`type`», est ajouté à chaque élément pour désigner le type de données des éléments LabVIEW. Ainsi, pour valider ce fichier par DTD, le fichier «contacts.dtd» dans la **Figure 0-1** peut être modifié de façon suivante:

```
<!ELEMENT contacts (personne)+>
<!ELEMENT personne (nom, dateDeNaissance)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT dateDeNaissance (#PCDATA)>
<!ENTITY % Type_LabVIEW "I8|I16|I32|String|Array|Cluster|…">
<!ATTLIST contacts
          type (%Type_LabVIEW;) #REQUIRED>
<!ATTLIST personne
          type (%Type_LabVIEW;) #REQUIRED>
<!ATTLIST nom
          type (%Type_LabVIEW;) #REQUIRED>
<!ATTLIST dateDeNaissance
          type (%Type_LabVIEW;) #REQUIRED>
```

«`Type_LabVIEW`» est une paramètre-entité interne, qui représente un ensemble des types de données dans LabVIEW. Cette entité est utilisée pour déclarer la liste des valeurs acceptables de l'attribut énuméré, nommé «`type`», de chaque élément dans le document XML généré. Il est important de noter que le fait de convertir un descripteur du type en type de données LabVIEW correspondant se base sur la règle suivante: le descripteur du type est un tableau 1D de nombres entiers, dont le premier nombre détermine la dimension du tableau, le second nombre identifie le type de données LabVIEW (en convertissant ce nombre en un nombre binaire de 16 bits d'abord, et puis en considérant les 8 bits de poids faible, le type de données LabVIEW est déterminé par la valeur décimale de ces 8 bits. Par exemple, le nombre entier du type DBL est 24586, qui est correspondant à un nombre binaire de 16 bits suivant: 01100000 00001010, et la valeur décimale de ses 8 bits de poids faibles est 10), le reste du descripteur peut être séparé en deux parties, donc l'une pour les types composants si le type LabVIEW interprété en dessus est soit le type «Tableau», soit le type «Cluster» (cette partie est vide en cas des types de base tels I8, I16, DBL, etc.), l'autre pour le nom d'élément LabVIEW. La figure ci-dessous montre un exemple, il s'agit d'une constance du type «Tableau», nommé «Table», de «DBL», nommé «Flottant». Nous voyons très clairement sur cette figure comment les types et les noms exacts de chaque élément LabVIEW peuvent être récupérés.

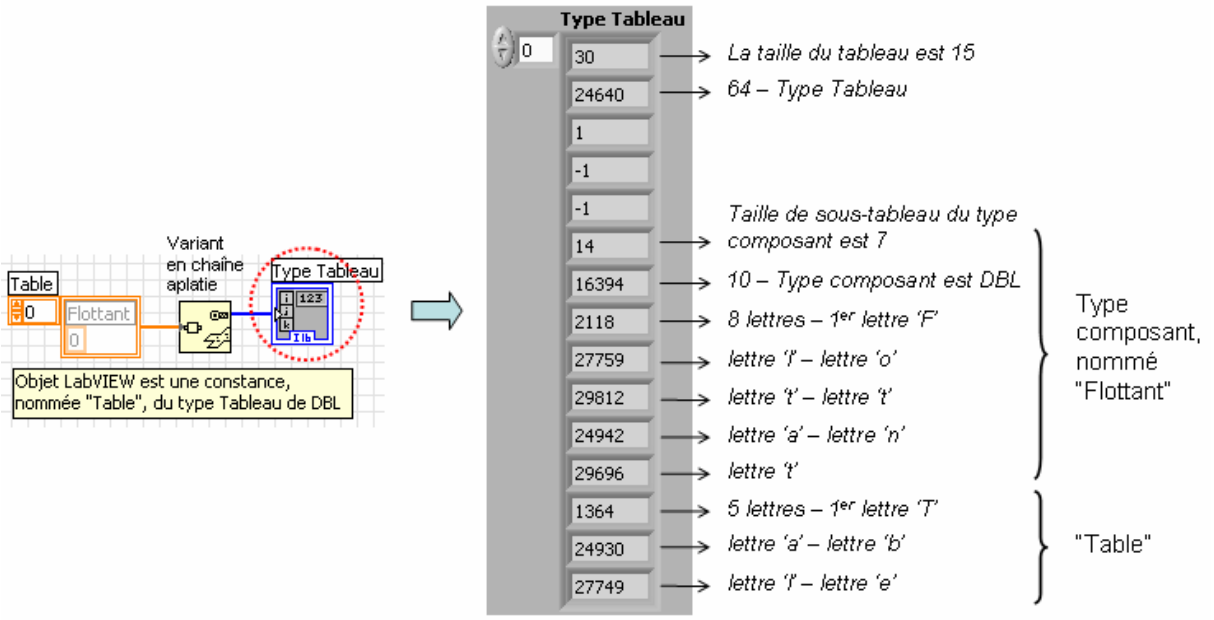


Figure III-22: Interpréter le descripteur de type d'un Tableau de DBL

M'appuyant sur cette idée, j'ai créé un VI, nommé «Interpreter Descripteur du Type», qui permet de faciliter la transformation d'un descripteur du type d'un élément LabVIEW en noms et types de données corrects. Cependant, à cause de la récursivité des types composants en cas des types Tableau, ou Cluster (par exemple, Tableau de Cluster de Tableau, etc.), l'utilisation de ce VI pour ces deux types se limite à certains cas. Donc, d'une manière précise les types de données LabVIEW énumérés ci-après peuvent être interprétés par le VI «Interpreter Descripteur du Type»:

- types de base (par exemple, I8, I16, I32, U8, U16, U32, DBL, SGL, Booléen, Chaîne),
- type Tableau 1D de type de base,
- type Tableau 1D de Cluster de type de base,
- type Cluster de types de base,
- type Cluster de types de base et/ou d'un ou plusieurs Tableau(x) 1D de type de base,
- type Cluster de types de base et/ou d'un ou plusieurs Tableau(x) 1D de Cluster de types de base.

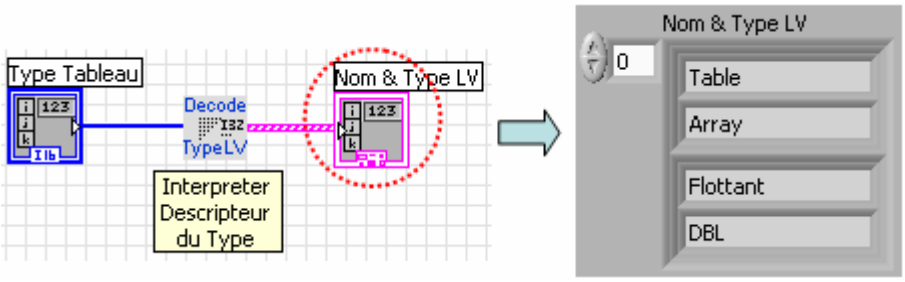


Figure III-23: Nom & Type LV avec le VI «Interpreter Descripteur du Type»

La Figure III-23 représente le résultat obtenu en appliquant le VI «Interpreter Descripteur du Type»: le descripteur de type en entrée dans ce cas est le descripteur de type de l'élément «`Table`» généré par la fonction «Variant en chaîne aplatie» (voir Figure III-22); à la sortie, un tableau, nommé «`Nom&Type LV`», est produit. Il comporte deux Clusters dont chacun comporte deux chaînes de caractères (la première chaîne désigne le nom, et la deuxième indique le type de données LabVIEW). L'interprétation de ce tableau donne: un Array, nommé «`Table`», de type DBL, nommé «`Flottant`».

Le VI «Interpreter Descripteur du Type» se trouve dans la sous-palette «Générer Code» de la palette DARTSVIEW. Ce VI est utilisé dans les VI de génération de code afin de générer le fichier XML d'un modèle des tâches DARTSVIEW, ou bien de vérifier la cohérence de types de données dès la transformation d'un message reçu à partir d'un outil de communication en un bon type LabVIEW.

Un autre point faible des palettes XML de LabVIEW porte sur la propriété hiérarchique de XML. XML est un langage hiérarchique, cette propriété permet de faire rapidement des accès aux éléments définis dans un arbre d'éléments XML avec les syntaxes de XPointers telles (*). Malheureusement, ces fonctionnalités ne sont pas fournies par les outils XML de LabVIEW (ni dans la sous-palette «XML» de la palette «Manipulation de données», ni dans la sous-palette «XML» de Toolkit Internet). Ainsi, lorsqu'on utilise un nœud de méthode dans le Toolkit Internet de LabVIEW pour rechercher une balise par son nom (par exemple, la balise «`personne`» du fichier «contacts.xml»), toutes les balises ayant ce nom dans ce fichier sont trouvées et affichées, mais à partir de chacune d'entre elles, on ne peut pas accéder à leurs balises d'enfant (s'il y en a). Donc, les syntaxes de XPointers telles (*) ne sont pas appliquées dans les palettes XML de LabVIEW.

En utilisant conjointement un attribut du type «ID» (*Identification*) ce problème peut être résolu. Pourtant le modèle de recherche devient compliqué et lourd, car il faut associer les fonctions dans les palettes XML de LabVIEW avec les fonctions permettant des traitements des chaînes de caractères dans la palette de fonctions, appelée «Chaîne». Un exemple d'un tel modèle est donné ci-après:

```
...
<!ELEMENT personne (nom, dateDeNaissance)>
<!ATTLIST personne
                numero ID #REQUIRED>
<!ELEMENT nom (#PCDATA)>
<!ATTLIST nom
          iden IDREF #REQUIRED>
...
<personne numero="P01">
   <nom iden="P01">A</nom>
...
</personne>
...
```

Le nom «A» est obtenu si la valeur de son attribut «`iden`» est le même que celle du premier élément «`personne`». Cela requière du temps en plus pour comparer la valeur de l'attribut «`numero`» de la balise «`personne`» et la valeur de l'attribut «`iden`» de chacune des balises «`nom`» trouvées (voir Figure III-24).

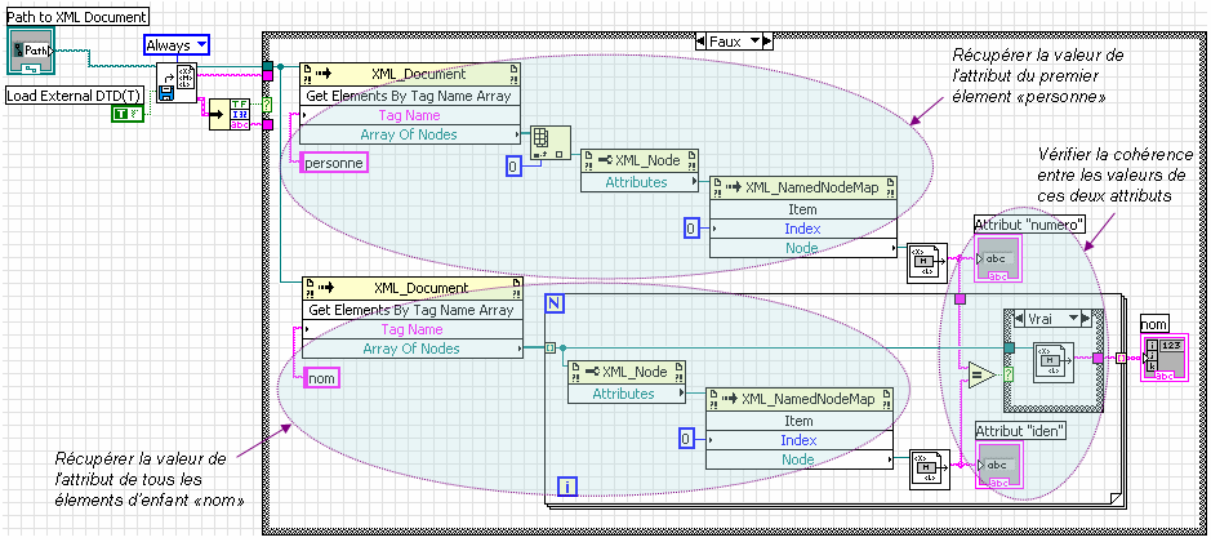


Figure III-24: Récupérer l'élément «nom» du premier élément «personne» dans LabVIEW

En conclusion, les palettes XML de LabVIEW sont utilisées dans la suite pour générer des fichiers XML facilitant le codage croisé entre les modèles DARTSVIEW et des autres outils de validation temporelle dans le second V du cycle de vie en W.

III.5. Palette DARTSVIEW et le second V du cycle de vie en W

La validation est un ensemble des activités de vérification dans le but de confirmer que le logiciel développé répond aux besoins exprimés dans la phase de spécification. La phase de validation temporelle est alors nécessaire dans le cadre des applications temps réel à contraintes strictes. La validation temporelle d'un système temps réel consiste en souvent une analyse de l'ordonnancement basée sur un modèle temporel des tâches [CDKM02].

Il existe des outils de validation temporelle, commerciaux ou universitaires, spécialisés dans tel ou tel type d'ordonnancement. Par exemple des outils reposent sur l'analyse des temps de réponse (*Response Time Analysis*) comme MAST [MAST01], ou des autres reposent sur la construction de séquance d'ordonnancement hors-ligne tels que PeNSMARTS [Gro99]. DARTSVIEW ne peut pas regrouper tous les outils de validation. Dans cette section, nous voyons comment DARTSVIEW permet de faciliter la génération de code dans un langage cible, ou bien que de générer une grande partie de code permettant la validation temporelle de l'application par un outil tiers dans le cadre du 2nd V du cycle en W (voir Figure III-25).

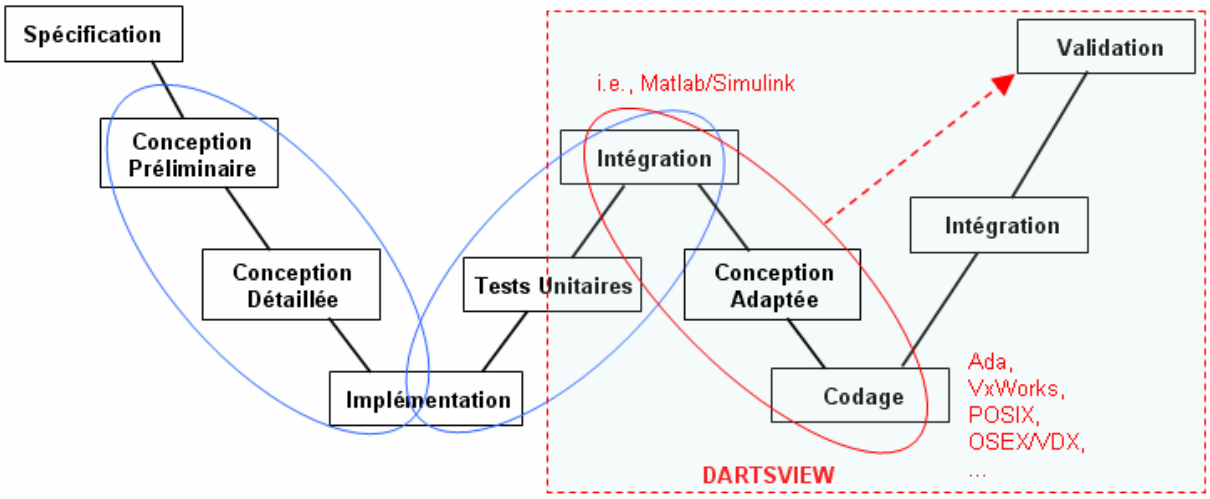


Figure III-25: Palette DARTSVIEW et le second V du cycle de vie en W

III.5.1. Schéma de la génération de code

Le langage LabVIEW permet de développer très rapidement des applications de contrôle/ commande ou temps réel, mail il n'est pas nécessairement adapté au temps réel strict, ni aux systèmes embarqués (Ada, C, …). Donc, l'idée est de générer du code Ada ou C ciblant un exécutif temps réel. Etant donnée la diversité des bibliothèques orientées temps réel (Ada, C-POSIX, VxWorks, OSEK/VDX, ARINC 653, etc.), il convient de trouver une façon flexible d'ajouter de nouveaux langages générés par DARTSVIEW. Il en va de même lors de la génération de code en vue de valider l'application par un outil tiers. Le choix effectué a été d'utiliser les fonctionnalités XML offertes par LabVIEW.

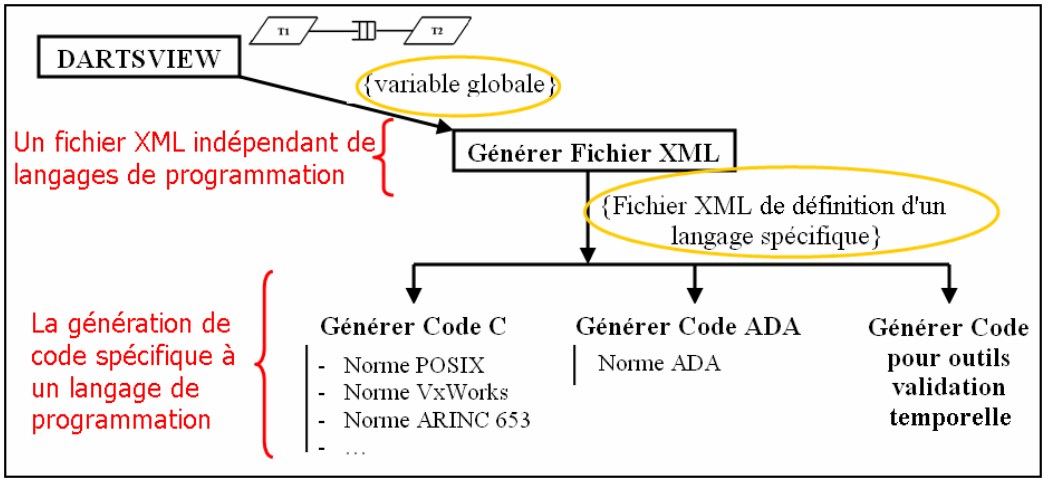


Figure III-26: Schéma de la génération de code

Le schéma de génération de code est représenté sur la Figure III-26. A partir d'un modèle DARTSVIEW du système considéré, une variable globale est utilisée. Elle est intégrée à chaque VI de DARTSVIEW, et sert à stocker des informations des tâches, des outils de communication et de synchronisation utilisés dans ce modèle. Ensuite, à partir de cette variable globale, un VI génère un fichier XML représentant l'application, c'est un fichier indépendant des langages de programmation ou des normes temps réel. Pour générer le code du système conformément à une norme temps réel ou à un langage de programmation particulier pouvant être exécuté sur un exécutif temps réel, on associe un autre fichier XML au fichier XML du modèle. Ce fichier XML ne consiste qu'en des balises de définition propres à la norme temps réel ou au langage spécifique que l'on choisit (par exemple, norme C-POSIX, norme OSEK/VDX, norme VxWorks, norme ARINC 653 basées sur le langage C; norme Ada de Ada; ou code pour un outil tiers de la validation temporelle,…).

III.5.2. Variable globale dans le schéma de la génération de code

Pour enregistrer des informations du modèle DARTSVIEW, une variable globale, nommée «Informations Enregistrées», est introduite. Il est requis que cette variable doive être intégrée à l'intérieur de chaque VI dans la palette DARTSVIEW, elle doive s'exécuter une seule fois lorsque le modèle DARTSVIEW du système est lancé. De plus, elle doit comporter des éléments qui sont capables de stocker toutes les informations nécessaires du modèle DARTSVIEW pour le but de génération de code dans la phase de codage croisé du second V du cycle de vie en W. Le modèle contenu dans la variable globale est le suivant :

- Pour le modèle des tâches:
  - *nom de tâche*;
  - *types de déclenchement*: par horloge temps réel, par interruption, par chien-de-garde, ou par outils de communication/synchronisation;

- *paramètres temporels*: période (en cas des tâches périodiques), délai critique (par défaut, les tâches périodiques sont à l'échéance sur requête), temps de réponse et/ou temps d'exécution au pire cas;
- *priorité*: normalement, la priorité d'une tâche est déterminée selon ses paramètres temporels par une affectation des priorités. Cela est réalisé dans le noyau des exécutifs lors de l'implémentation sur cible;
- *outils de communication/synchronisation utilisés*: c'est le cas où les tâches ne sont pas indépendantes. Chaque tâche doit alors satisfaire une contrainte de précédence avec autres tâches à travers des outils de communication/synchronisation de manière à ce que soit elle génère un déclenchement pour activer une autre tâche qui se bloque en attente, soit elle attend un déclenchement généré par une autre tâche pour lancer son exécution. Par conséquente, les modes d'utilisation des outils de communication/ synchronisation dans ce cas est soit «Signaler/Attendre» pour la synchronisation, ou «Envoyer/Recevoir» de messages pour la communication;
- *type de données des messages*: en cas de la communication par messages, il faut assurer la cohérence des types de données entre les messages envoyés par les tâches «Productrices» et ceux reçus par les tâches «Consommatrices». Ce paramètre permet de vérifier les transactions des messages du bon type de données entre ces tâches.

- Pour le modèle des outils de synchronisation:
  - *nom des outils d'implémentation de la synchronisation*;
  - *nombre de déclenchements non pris en compte*: en cas de la synchronisation n/1 à compte.
- Pour le modèle des outils de communication:
  - *nom des outils d'implémentation de la communication;*
  - *taille du tampon (file de messages);*
  - *type de données des messages*: en cas de la communication par messages, il faut assurer la cohérence des types de données entre les messages envoyés par les tâches «Productrices» et ceux reçus par les tâches «Consommatrices». Ce paramètre permet de vérifier la transformation des messages du bon type de données dans la tâche «Consommatrice».

III.5.2.a. Modèle de la variable globale dans le schéma de la génération de code

La Figure III-27 représente le modèle de la variable globale, nommé «Informations Enregistrées». C'est un tableau, donc chaque élément tableau correspond à un élément dans le modèle DARTSVIEW du système. Actuellement, un élément de ce tableau est un cluster comportant des composants suivants:

- *«Type d'Elément»*: indique le type de l'élément du modèle DARTSVIEW correspondant à cet élément. Il est soit un type «Tâche», soit un type «BAL»/«BAL_Ecr»/« MDD» (outil de la communication), soit un type «Semaphore» (outil de la synchronisation).
- *«Nom d'Elément»*: indique le nom de l'élément du modèle DARTSVIEW correspondant à cet élément.
- *«Paramètres Temporels»*: enregistre les paramètres temporels du modèle des tâches DARTSVIEW. Actuellement, ce paramètre est un cluster de trois composants: la «Période (ms)» pour les tâches périodiques (donc, en cas des tâches apériodiques, ce composant est égale à 0, valeur par défaut); le «Délai Critique (ms)» pour des tâches à contraintes strictes (par défaut, ce composant prend la valeur 0 en cas des tâches apériodiques, et prend la valeur de la période en cas des tâches périodiques); le «Temps d'Exécution (ms)» indique la durée d'exécution maximale de la tâche (ce composant n'est pas utilisé dans le modèle

DARTSVIEW durant la premier V du cycle de vie en W, il sert aux tests d'ordonnancement lors de la validation temporelle des codes générés à partir du modèle DARTSVIEW durant le second V).

- *«Type de donnée (BAL ou MDD)»*: indique le type de données des messages dans la communication par messages entre tâches. Ce composant stocke le descripteur du type de message. Donc, lors de génération de code, il faut le convertir en types LabVIEW grâce au VI «Interpreter Descripteur du Type LV» présenté auparavant (réf. III.4.3.).
- *«Outils de Commu/Synchro utilisés»*: indique les outils de communication et de synchronisation qu'une tâche utilise pour communiquer et/ou synchroniser une ou des autres tâches durant son exécution. C'est un tableau qui correspond à la liste des outils de communication/synchronisation. Chaque élément de ce tableau comporte le «Nom d'Outil»; le «Type d'Outil» (soit BAL/BAL_Ecr/MDD, soit Semaphore); le «Type de donnée» pour la communication par messages; et le «Mode E/L ou P/V» (c'est le mode d'opération de chaque outil. Donc s'il est un outil de communication, le mode est Envoyer/Lire de message. Au contraire pour un outil de synchronisation, le mode est Prendre/Vendre de sémaphore. Dans ce cas, un booléen est utilisé, une valeur Vrai correspond au mode E ou P, tandis qu'une valeur Faux correspond au mode L ou V).

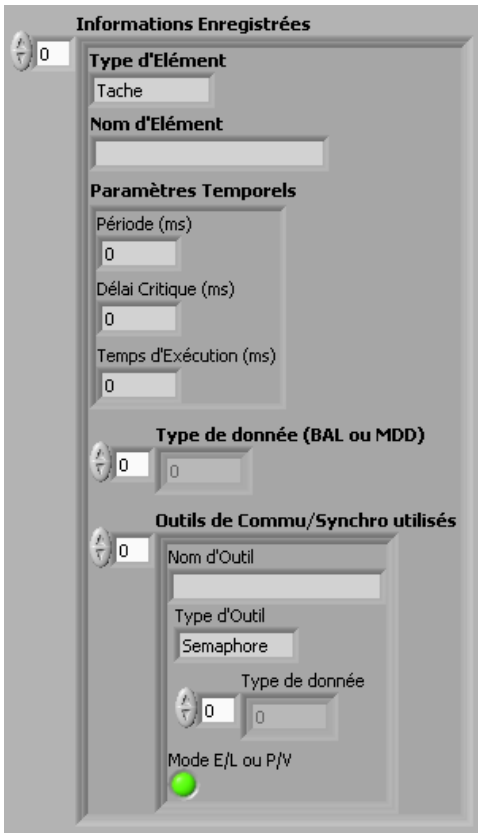


Figure III-27: Modèle de la variable globale

Remarque : la valeur de «Période (ms)» correspond à une tâche déclenchée par horloge temps réel si cette valeur est strictement positive, sinon, la tâche est déclenchée par un outil de communication/synchronisation dans la liste des «Outils de Commu/Synchro utilisés».

III.5.2.b. Modification du modèle des tâches DARTSVIEW

Les modifications du modèle des tâches périodiques DARTSVIEW sont présentées sur la figure ci-après:

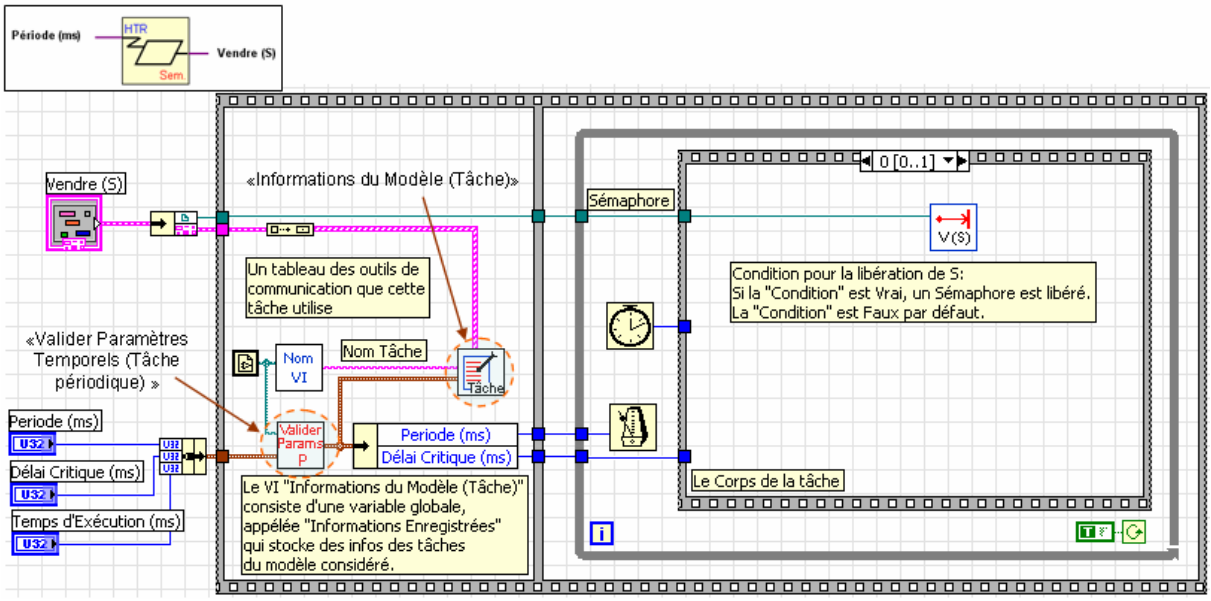


Figure III-28: Modifications du modèle des tâches périodiques DARTSVIEW

Par rapport du modèle de tâche «HRT – (out) Sémaphore» de la sous-palette «Tâche Matérielle» présenté sur la Figure III-3, le diagramme de tâche «HRT – (out) Sémaphore» dans la Figure III-28 possède une structure de séquence, dont la première correspond à la partie d'enregistrement des informations du modèle ans la variable globale; alors que la deuxième séquence (la séquence à droite) correspond à l'implémentation de la Figure III-3.

Pour cet exemple :

- le «Type d'Elément» est le type «Tache»;
- le «Nom d'Elément» est le nom du VI du modèle de tâche fourni par le VI «Nom VI»;
- pour les «Paramètres Temporels» la période est un paramètre obligatoire qui doit être fourni lors d'utilisation de ce modèle. Tandis que les deux autres, le délai critique et le temps d'exécution, sont optionnels. Par défaut, pour le modèle des tâches périodiques DARTSVIEW, si l'utilisateur ne fournit pas la valeur pour le délai critique, la tâche est l'échéance sur requête, c'est-à-dire D = T. Cela est réalisé grâce au VI «Valider Paramètres Temporels (Tâche périodique)». De plus, ce VI permet aussi de tester la validité des paramètres temporels d'entrée (il faut que ces paramètres satisfassent la contrainte suivante, $T \geq D \geq C$; sinon, une erreur est lancée).

Des modifications similaires sont appliquées à tous les modèles de tâche afin de se « déclarer » dans la variable globale.

III.5.2.c. Modification du modèle des éléments synchronisation/communication

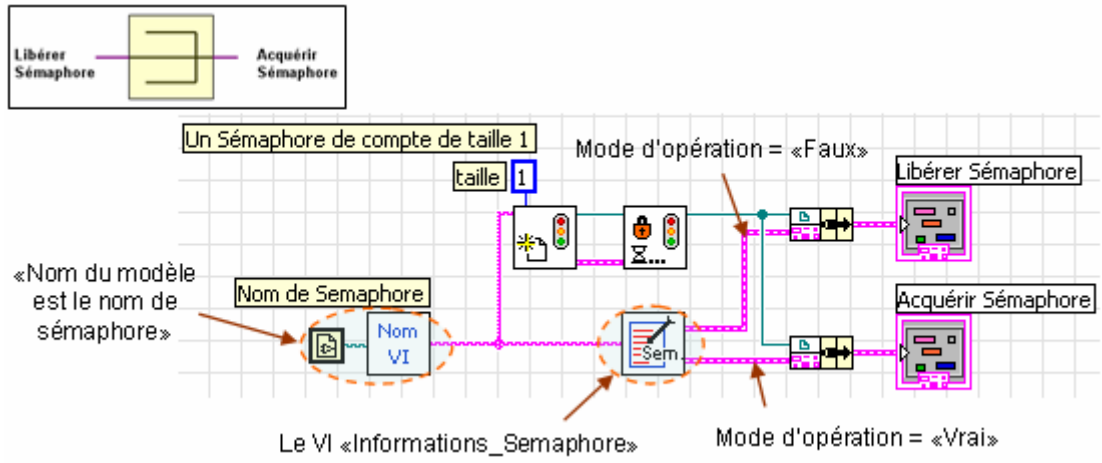


Figure III-29: Modifications du modèle de la synchronisation DARTSVIEW

La Figure III-29 représente le diagramme modifié du modèle de la synchronisation. De même que les modifications pour les modèles des tâches, la variable globale se trouve dans le VI «Informations_Semaphore», le nom de sémaphore dans ce cas est le nom du modèle DARTSVIEW de cette synchronisation. Pour les deux connecteurs, «Acquérir Sémaphore» et «Libérer Sémaphore», un flot de données contenant des informations du sémaphore est ajouté (noter que le mode d'opération de sémaphore est «Vrai» en cas de demande de sémaphore, alors qu'il est «Faux» en cas de libération de sémaphore). Par conséquent, lorsqu'une tâche est câblée sur un des deux connecteurs, elle obtient des informations utiles pour la liste des «Outils de Commu/Synchro » qu'elle utilise.

Le même type de modification est opéré sur les autres éléments de syncrhonisation et communication.

III.5.2.d. Variable globale et le VI «Informations Enregistrées»

Le VI «Informations Enregistrées» de la palette DARTSVIEW (réf. III.1.6.) ne comporte que le modèle de la variable globale. Par conséquent, une fois que le modèle DARTSVIEW d'un système est exécuté, au moment du lancement des tâches, la variable globale contient dans une table la description de tous les éléments DARTS du système.

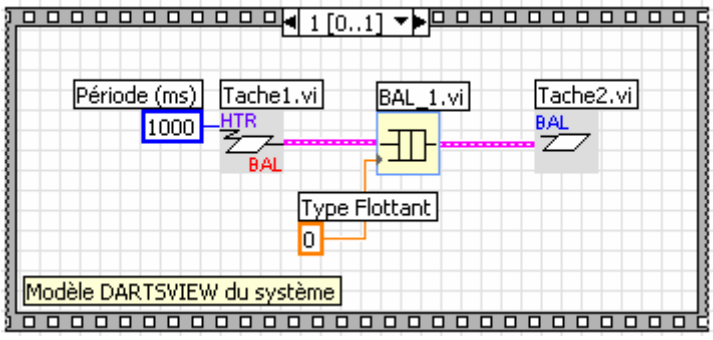


Figure III-30: Un modèle DARTSVIEW de la communication par BAL

Dans cet exemple, deux tâches, nommées «Tache1» et «Tache2», communiquent par l'intermédiaire d'une BAL de taille 1 de type Flottant, nommée «BAL_1», de manière à ce que la «Tache1» est une tâche périodique avec une période de 1000 ms, et la «Tache2» est une tâche logicielle activée par la «Tache1». Le diagramme DARTSVIEW de ce modèle est présenté sur la Figure III-30.

Lorsque le modèle DARTSVIEW du système est exécuté, grâce au VI «Informations Enregistrées», l'information du modèle peut être récupérée aisément (voir Figure III-31).

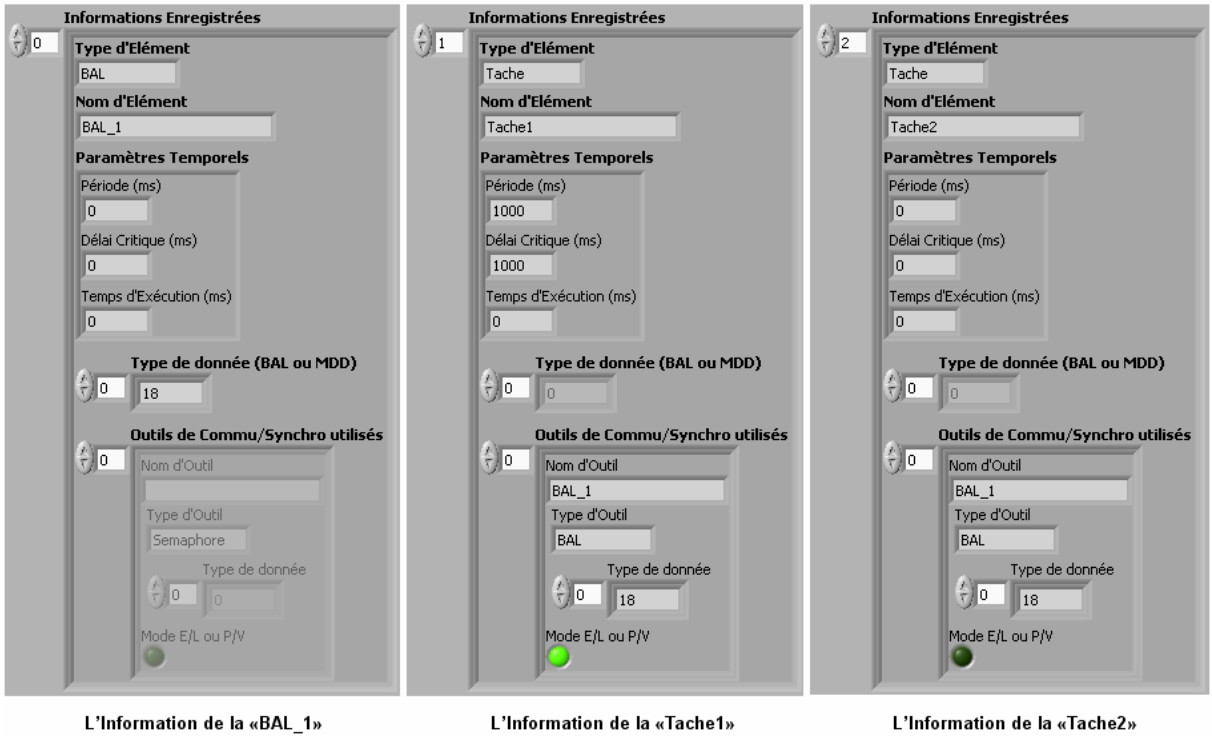


Figure III-31: Récupérations des informations du modèle grâce au VI «Informations Enregistrées»

Les diagrammes DARTSVIEW des «Tache1» et «Tache2» sont présentés ci-après:

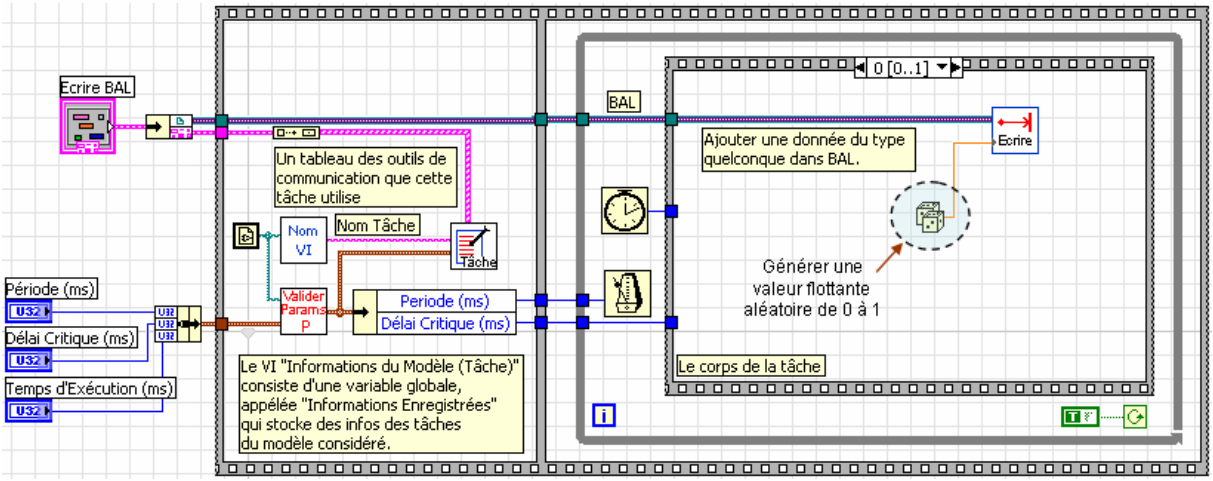


Figure III-32: Diagramme DARTSVIEW de la «Tache1»

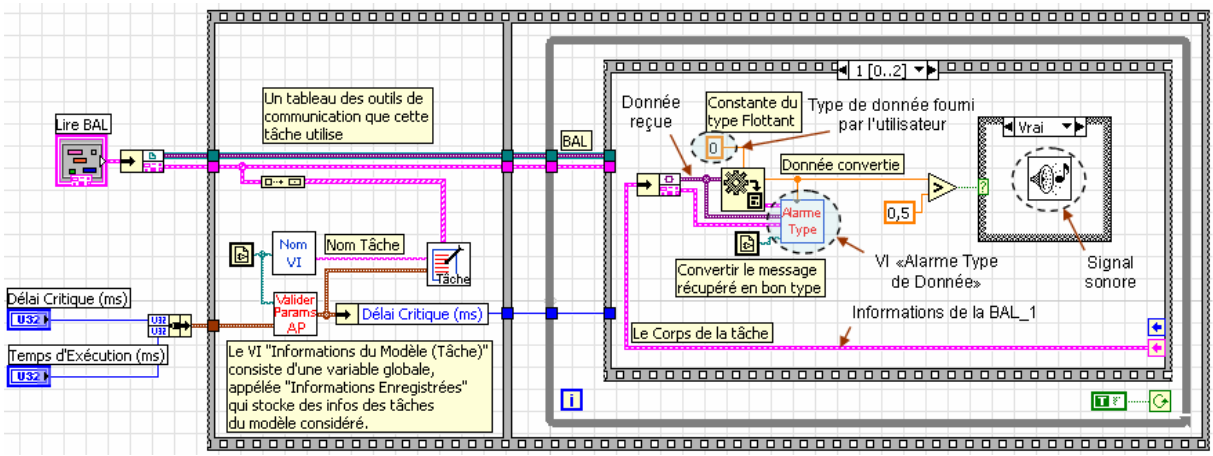


Figure III-33: Diagramme DARTSVIEW de la «Tache2»

III.5.3. Fichier XML de description du modèle DARTSVIEW

Dans cette étape, à partir des informations récupérées dans la variable globale, un fichier XML est généré. Il s'agit d'un fichier XML indépendant de langages de programmation, car il ne contient que des informations des éléments existant dans le diagramme DARTSVIEW d'un système.

III.5.3.a. La DTD de fichier XML de description du modèle DARTSVIEW

Voici la DTD proposée pour la description d'un diagramme DARTSVIEW en XML :

```
-----------------------------------------------------------------------
-- Fichier «Modele_DTD.dtd»
 1| <!ELEMENT Modele ((BAL|MDD|SEMAPHORE)*,(Tache)*)>
 2| <!ELEMENT BAL (Donnee)>
 3| <!ATTLIST BAL
 4|      type (bal|bal_ecr) #REQUIRED
 5|      nom ID #REQUIRED>
 6| <!ELEMENT MDD (Donnee)>
 7| <!ATTLIST MDD
 8|      nom ID #REQUIRED>
 9| <!ELEMENT SEMAPHORE EMPTY>
10| <!ATTLIST SEMAPHORE
11|      nom ID #REQUIRED>
12| <!ELEMENT Donnee (typedonnee)+>
13| <!ATTLIST Donnee
14|      nomtype ID #REQUIRED>
15| <!ELEMENT typedonnee EMPTY>
16| <!ATTLIST typedonnee
17|      iden IDREF #REQUIRED
18|      type (I8|I16|I32|U8|U16|U32|DBL|EXT|SGL|String|
19|            Boolean|Array|Cluster) #REQUIRED>
20| <!ELEMENT Tache (BALU|MDDU|SEMU)*>
21| <!ATTLIST Tache
22|      nomTache ID #REQUIRED
23|      periode CDATA #IMPLIED
24|      delai_critique CDATA #IMPLIED
25|      temps_execution CDATA #IMPLIED>
26| <!ELEMENT BALU EMPTY>
27| <!ATTLIST BALU
28|      iden IDREF #REQUIRED
29|      nomBAL IDREF #REQUIRED
30|      typeDonnee IDREF #REQUIRED
31|      typeBAL (bal|bal_ecr) "bal"
32|      modeBAL (Envoyer|Recevoir) "Envoyer">
33| <!ELEMENT MDDU EMPTY>
34| <!ATTLIST MDDU
```

```
35|      iden IDREF #REQUIRED
36|      nomMDD IDREF #REQUIRED
37|      typeDonnee IDREF #REQUIRED
38|      modeMDD (Ecrire|Lire) "Ecrire">
39| <!ELEMENT SEMU EMPTY>
40| <!ATTLIST SEMU
41|      iden IDREF #REQUIRED
42|      nomSEM IDREF #REQUIRED
43|      modeSEM (Prendre|Vendre) "Prendre">
```

Dans le fichier «Modele_DTD.dtd», la ligne 1 définit l'élément racine et ses éléments composants. Donc, `<Modele>` est l'élément racine. Il est soit un élément vide, soit un élément composite qui comporte des outils de communication/synchronisation, et puis des tâches. Un outil de communication/synchronisation est soit une `BAL`, soit un `MDD`, ou un `SEMAPHORE`.

Les lignes de 2 à 19 décrivent en détail les outils de communication/synchronisation. Ainsi, une `BAL` ou un `MDD` contiennent `Donnee`, qui contient au moins un `typedonnee`; tandis qu'un `SEMAPHORE` est un élément vide. Chaque outil de communication/synchronisation possède un attribut exigé (`#REQUIRED`), appelé «`nom`», du type `ID` qui est le nom de cet outil dans le modèle DARTSVIEW du système. L'outil `BAL` possède un attribut exigé en plus, appelé «`type`», car une BAL peut être soit une BAL classique (`bal`), soit une BAL avec écrasement (`bal_ecr`). Enfin, l'élément «`typedonnee`» est un élément vide qui possède deux attributs obligatoires: l'un du type `IDREF`, appelé «`iden`», comportant le nom d'un des outils de communication/synchronisation bien définis dans le fichier XML du modèle DARTSVIEW; l'autre du type d'énumération, appelé «`type`», consistant en des types de données LabVIEW.

L'élément «Tache» est défini dans le reste de la DTD (à partir de la ligne 20 jusqu'à la ligne 43). Donc, un élément «`Tache`» soit un élément vide, soit un élément qui comporte au moins un outil de communication/synchronisation (dans cette DTD, «`BALU`», «`MDDU`», «`SEMU`» représentent respectivement l'outil `BAL`, l'outil `MDD`, l'outil `SEMAPHORE` qu'une tâche utilise). Chaque élément «Tache» possède quatre attributs, appelés «nomTache» « periode» «delai_critique» et «temps_execution», donc les trois derniers sont les attributs optionnels, alors que le premier est un attribut obligatoire de type ID pour assurer qu'une seule tâche ayant ce «nomTache» existe dans un fichier XML de description du modèle DARTSVIEW. Ensuite, chaque élément représentant un outil de communication/synchronisation utilisé par une tâche, est défini en tant qu'un élément vide, dont les informations sont stockées dans ses attributs. Par exemple, pour chaque élément «`BALU`», les attributs «`ident`» «`nomBAL`» «`typeDonnee`» sont trois attributs exigés (car ils référencent les noms de «`Tache`», de «`BAL`», et de «`typeDonnee`» respectivement, qui sont bien définis auparavant dans le fichier XML du modèle DARTSVIEW); «`typeBAL`» et «`modeBAL`» sont deux attributs énumérés (car une BAL est soit «`bal`», soit «`bal_ecr`» qui peut être exécutée en mode «`Envoyer`» ou «`Recevoir`» des messages); les valeurs par défaut de ces deux attributs sont «`bal`» et «`Envoyer`». Il en va de même pour l'élément «`MDDU`» et l'élément «`SEMU`». Notons qu'un MDD peut s'exécuter soit en mode «Ecrire un message», soit en mode «Lire un message», l'attribut «`modeMDD`» est une énumération de deux valeurs, «`Ecrire`» (par défaut) et «`Lire`»; tandis qu'un sémaphore peut être exécuté en mode «Acquérir un sémaphore», ou en mode «Libérer un sémaphore», l'attribut «`modeSEM`» est une énumération de deux valeurs, «`Prendre`» (par défaut) et «`Vendre`».

En conclusion, les informations récupérées à partir de la variable globale peuvent être représentées de façon précise et conforme à la grammaire bien décrite dans cette DTD. Par conséquent, dans la suite le fichier «Modele_DTD.dtd» est utilisé pour valider des fichiers XML de description du modèle DARTSVIEW.

III.5.3.b. Génération de fichier de description du modèle DARTSVIEW

Le VI «Generer Fichier XML» dans la sous-palette «Générer Code» (voir Figure III-34) de la palette DARTSVIEW permet de générer le fichier XML de description du modèle DARTSVIEW à partir des informations stockées dans la variable globale.

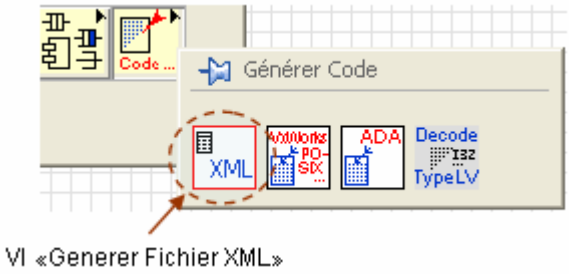


Figure III-34: VI «Generer Fichier XML» dans la sous-palette «Générer Code»

Le diagramme du VI «Generer Fichier XML» est présenté sur la Figure III-35. Dans ce diagramme, la DTD des fichiers XML du modèle DARTSVIEW est déclarée de façon externe par le chemin du fichier «Modele_DTD.dtd» (par défaut, ce fichier est dans le même répertoire du VI «Generer Fichier XML»); les informations du modèle DARTSVIEW dans la variable globale sont séparées en deux parties: l'une pour la description des outils de communication/synchronisation, qui est traitée par le VI «BAL-MDD Infos.vi», et l'autre pour la description des tâches, qui est traitée par le VI «Tache Infos.vi».

Une fois que les informations du modèle DARTSVIEW d'un système sont bien enregistrées dans la variable globale, en exécutant le VI «Generer Fichier XML», le fichier XML de description du modèle peut être généré. Il est important de noter que la génération de ce fichier XML a lieu de façon transparente, l'utilisateur ne fait que déterminer le chemin pour sauvegarder ce fichier. Dans la section suivante, en utilisant conjointement le fichier XML du modèle DARTSVIEW d'un système avec un autre fichier XML, on peut générer le code dans différents langages de programmation, ou pour différents outils de validation.

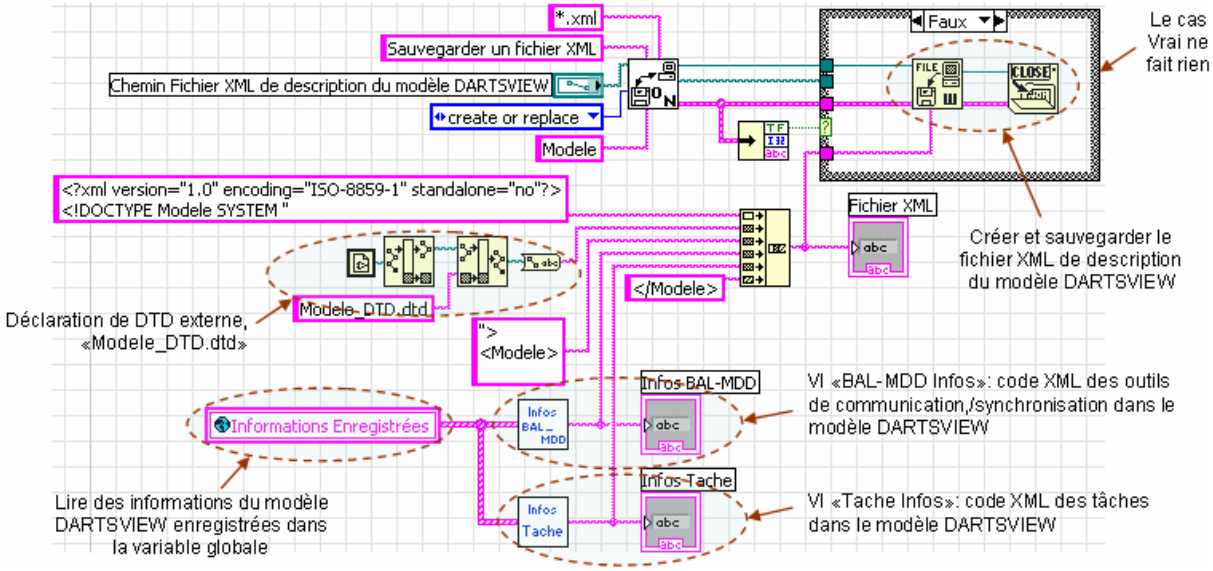


Figure III-35: Diagramme du VI «Generer Fichier XML» de la sous-palette «Générer Code»

En conclusion de cette section, le code XML du modèle DARTSVIEW dans la Figure III-30 est donné ci-après, ce code est généré par le VI «Generer Fichier XML» à partir des informations du modèle dans la variable globale (voir Figure III-31).

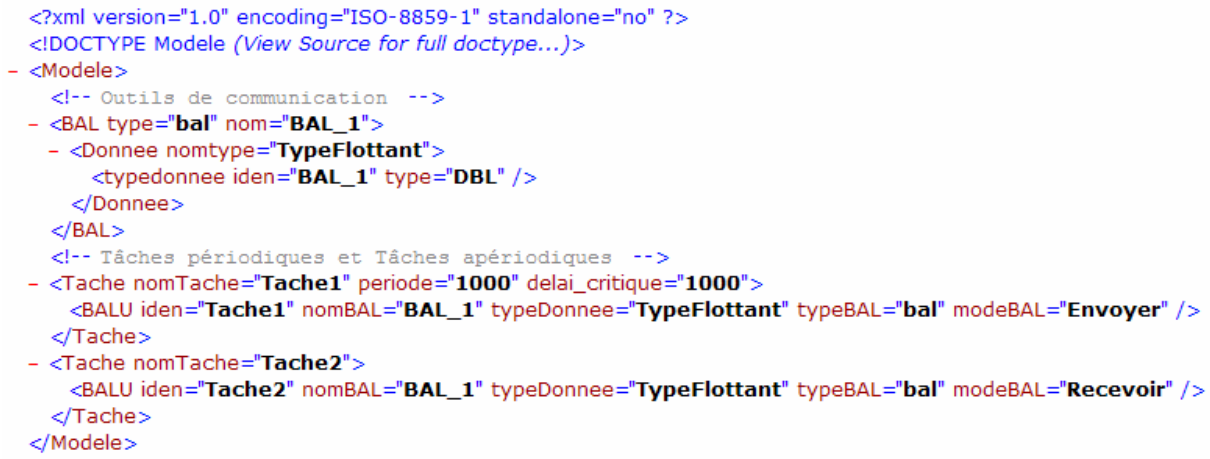


Figure III-36: Ouvrir le fichier XML du modèle DARTSVIEW avec Internet Explorer 7

III.5.4. Génération de code d'un langage particulier

Dans cette étape, en associant un autre fichier XML au fichier XML du système, on peut générer le code du système pour un langage cible particulier, qui peut s'exécuter sur un exécutif temps réel ou un outil de validation.

Dans le schéma de la génération de code (voir Figure III-26), un tel fichier XML est appelé le fichier XML de définition d'un langage spécifique, car il consiste en des balises spécifiques à un langage de programmation (par exemple, C, Ada,…). A cause de différences entre des normes temps réel basées sur le langage C, et ceux basées sur d'autres langages (Ada, langages synchrones,…), il est difficile d'interpréter cette étape en un seul VI. En fait, un VI s'occupe de générer le code pour toutes les normes du langage C (i.e., norme POSIX, norme VxWorks, norme ARINC 653,…), un autre VI pour la norme Ada, et un autre pour le code d'un outil de validation. Notons que le but est de pouvoir permettre à un utilisateur de DARTSVIEW d'ajouter le langage cible spécifique pour ses propres besoins, de façon relativement simple. Ainsi, un stagiaire de Master 1 a eu comme projet d'ajouter la norme OSEK/VDX en définissant le fichier XML nécessaire. Cela était un peu particulier car OSK/VDX utilise non seulement le langage C (le fichier XML nécessaire a été décrit très rapidement par le stagiaire) mais aussi le langage OIL.

III.5.4.a. Fichier XML-DTD pour les normes basées sur le langage C

Suite à la présentation des principales normes temps réel du langage C aux chapitre I (réf. I.4.6.) et chapitre II (réf. II.3.1., II.3.2.), je propose une DTD pour les normes temps réels basées sur le langage C comme la suivante:

```
-----------------------------------------------------------------------
-- Fichier «Norme_C_DTD.dtd»
 1| <!-- DTD pour les normes temps-r&#xe9;el bas&#xe9;es sur le
         langage C -->
 2| <!ELEMENT NORME_C (En_Tete,SEMAPHORE,BAL,MDD,Gestion_Temps,
                       TACHE,Main)>
 3| <!ELEMENT En_Tete (#PCDATA)>
 4| <!ELEMENT SEMAPHORE (Decla_Sem,Init_Sem,Vendre_Sem,Prendre_Sem)>
 5| <!ELEMENT Decla_Sem (#PCDATA)>
 6| <!ELEMENT Init_Sem (#PCDATA)>
 7| <!ELEMENT Vendre_Sem (#PCDATA)>
 8| <!ELEMENT Prendre_Sem (#PCDATA)>
 9| <!ELEMENT BAL (Bal,Bal_Ecr)>
10| <!ELEMENT Bal (Decla_Bal,Init_Bal,Envoyer_Bal,Recevoir_Bal)>
11| <!ELEMENT Decla_Bal (#PCDATA)>
12| <!ELEMENT Init_Bal (#PCDATA)>
13| <!ELEMENT Envoyer_Bal (#PCDATA)>
14| <!ELEMENT Recevoir_Bal (#PCDATA)>
```

```
15| <!ELEMENT Bal_Ecr ((Decla_Bal_Ecr,Init_Bal_Ecr,Envoyer_Bal_Ecr,
                        Recevoir_Bal_Ecr)|EMPTY)*>
16| <!ELEMENT Decla_Bal_Ecr (#PCDATA)>
17| <!ELEMENT Init_Bal_Ecr (#PCDATA)>
18| <!ELEMENT Envoyer_Bal_Ecr (#PCDATA)>
19| <!ELEMENT Recevoir_Bal_Ecr (#PCDATA)>
20| <!ELEMENT MDD (Decla_Mdd,Init_Mdd,Ecrire_Mdd,Lire_Mdd)>
21| <!ELEMENT Decla_Mdd (#PCDATA)>
22| <!ELEMENT Init_Mdd (#PCDATA)>
23| <!ELEMENT Ecrire_Mdd (#PCDATA)>
24| <!ELEMENT Lire_Mdd (#PCDATA)>
25| <!ELEMENT Gestion_Temps (#PCDATA|EMPTY)*>
26| <!ELEMENT TACHE (DeclaTacheP,DeclaTacheAP,Creer_Tache,
                     Mode_Attache,Iden_Tache)>
27| <!ELEMENT DeclaTacheP (#PCDATA)>
28| <!ELEMENT DeclaTacheAP (#PCDATA)>
29| <!ELEMENT Creer_Tache (#PCDATA)>
30| <!ELEMENT Mode_Attache (#PCDATA|EMPTY)*>
31| <!ELEMENT Iden_Tache (#PCDATA)>
32| <!ELEMENT Main (#PCDATA)>
```

Selon cette DTD, la grammaire des fichiers XML spécifiques aux normes temps réel du langage C est définie de manière à ce que `<NORME_C>` est l'élément racine qui consiste de sept éléments enfants:

- `<En_Tete>`: (ligne 3) un élément qui ne contient que les textes (#PCDATA), correspond à la partie de déclaration des bibliothèques, des fonctions spécifiques aux utilisateurs, et des autres définitions (par exemple, des variables externes, des constantes,…);
- `<SEMAPHORE>`: (lignes 4-8) contient quatre éléments, appelés `<Decla_Sem>` `<Init_Sem>` `<Vendre_Sem>` `<Prendre_Sem>` respectivement, qui comporte le code d'implémentation de la synchronisation par l'outil Sémaphore;
- `<BAL>`: (lignes 9-19) comporte le code d'implémentation de la communication par l'outil BAL. Puisqu'une BAL peut être une BAL classique, ou une BAL avec écrasement, l'élément `<BAL>` est déclaré par deux éléments d'enfant représentant ces deux types de BAL (`<Bal>` et `<Bal_Ecr>`). Il est important de noter que la BAL avec écrasement n'existe pas toujours dans toutes les normes temps réel basée sur le langage C (par exemple, la norme VxWorks ne définit pas ce type de BAL), par conséquent `<Bal_Ecr>` peut être un élément vide (voir ligne 15). De même que `<SEMAPHORE>`, chaque type d'élément `<BAL>` consiste en quatre éléments enfants propres à ce type, donc pour l'élément `<Bal>`, les éléments enfants sont `<Decla_Bal>` `<Init_Bal>` `<Envoyer_Bal>` `<Recevoir_Bal>`; alors que pour l'élément `<Bal_Ecr>`, ses éléments enfants correspondants sont `<Decla_Bal_Ecr>` `<Init_Bal_Ecr>` `<Envoyer_Bal_Ecr>` `<Recevoir_Bal_Ecr>` respectivement;
- `<MDD>`: (lignes 20-24) il est similaire à l'élément `<BAL>`, cet élément comporte le code d'implémentation de la communication par module de données. Ainsi, un `<MDD>` est défini par quatre éléments suivants, `<Decla_Mdd>` `<Init_Mdd>` `<Ecrire_Mdd>` `<Lire_Mdd>`;
- `<Gestion_Temps>`: (ligne 25) cet élément contient la partie du code qui s'occupe de la gestion de temps dans les normes temps réel du langage C. Comme j'en ai déjà parlé auparavant, il existe deux modèles de la gestion de temps: modèle dirigé par le temps et modèle dirigé par événement. Avec le modèle dirigé par le temps, le système prend la main à chaque *tick*, par conséquent, il n'y a pas de fonction spécifique permettant d'attendre jusqu'à une certaine date (en *ticks*). Ainsi, l'élément `<Gestions_Tems>` devient vide en cas des normes du langage C dirigée par le temps;
- `<TACHE>`: (lignes 26-31) une tâche peut être périodique ou apériodique, donc `<DeclaTacheP>` et `<DeclaTacheAP>` correspondent aux déclarations des tâches

périodiques et des tâches apériodiques. De plus, il faut qu'on crée une tâche (<Créer_Tache>), et puis l'identifier (<Iden_Tache>) avant de la lancer dans la procédure *main*. Enfin, dans certaines normes, une tâche peut être exécutée en mode attaché au lieu d'en mode détaché, donc l'élément <Mode_Attache> peut être vide, ou contenir le code permettant d'attacher une tâche;

- <Main>: (ligne 32) comporte le code de la principale procédure, *void main()*, qu'on trouve souvent dans les programmes C.

En se basant sur cette DTD, le fichier XML pour la génération de code spécifique à la norme POSIX, appelé «POSIX.xml», est donné ci-après:

-- **Fichier «POSIX.xml»**

```
 1  <?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
 2  <!DOCTYPE NORME_C SYSTEM "Norme_C_DTD.dtd">
 3  <!-- La norme POSIX -->
 4  <NORME_C>
 5  <En_Tete>
 6  #include &lt;pthread.h&gt;
 7  #include &lt;semaphore.h&gt;
 8  #include &lt;BALs.h&gt;
 9
10  /* Fonctions sp&#xe9;cifiques aux utilisateurs */
11
12  /* D&#xe9;finition de constantes */
13
14  </En_Tete>
15  <SEMAPHORE>
16  <Decla_Sem>
17  /* La synchronisation par Semaphore est remplac&#xe9;e par BAL, 18  avec un
message du type Bool&#xe9;en */
19  /* Type utilisé et la déclaration de la BAL: [nomSEM] */
20  typedef struct {
21  Boolean message;
22  } t_[nomSEM];
23  bal [nomSEM]; /*BAL de taille 1 sans &#xe9;crasement */
24  </Decla_Sem>
25  <Init_Sem>[nomSEM]=bal_init(sizeof(t_[nomSEM]));</Init_Sem>
26  <Vendre_Sem>/* Envoi un message */
27  bal_envoyer([nomSEM],(char*)&amp;[variable]);
28  </Vendre_Sem>
29  <Prendre_Sem>/* Attente de message */
30  bal_recevoir([nomSEM],(char*)&amp;[variable]);
31  </Prendre_Sem>
32  </SEMAPHORE>
33  <BAL>
34  <Bal>
35  <Decla_Bal>/*BAL de taille 1 sans &#xe9;crasement */
36  bal [nomBAL];
37  </Decla_Bal>
38  <Init_Bal>[nomBAL]=bal_init(sizeof([typedonnee]));</Init_Bal>
39  <Envoyer_Bal>/* Envoi un message */
40  bal_envoyer([nomBAL],(char*)&amp;[variable]);
41  </Envoyer_Bal>
42  <Recevoir_Bal>/* Attente de message */
43  bal_recevoir([nomBAL],(char*)&amp;[variable]);
44  </Recevoir_Bal>
45  </Bal>
46  <Bal_Ecr>
47  <Decla_Bal_Ecr>/*BAL de taille 1 &#xe0; &#xe9;crasement */
48  bal_ecr [nomBAL];
49  </Decla_Bal_Ecr>
50  <Init_Bal_Ecr>
51  [nomBAL]=bal_ecr_init(sizeof([typedonnee]));
52  </Init_Bal_Ecr>
53  <Envoyer_Bal_Ecr>/* Envoi un message */
54  bal_ecr_envoyer([nomBAL],(char*)&amp;[variable]);
55  </Envoyer_Bal_Ecr>
```

```
<Recevoir_Bal_Ecr>/* Attente de message */
bal_ecr_revevoir([nomBAL],(char*)&amp;[variable]);
</Recevoir_Bal_Ecr>
</Bal_Ecr>
</BAL>
<MDD>
<Decla_Mdd>
[typedonnee] [nomMDD]; /* D&#xe9;claration du MDD */
pthread_mutex_t s_[nomMDD]; /* Mutex du MDD */
</Decla_Mdd>
<Init_Mdd>pthread_mutex_init(&amp;s_[nomMDD],NULL);</Init_Mdd>
<Ecrire_Mdd>
pthread_mutex_lock(&amp;s_[nomMDD]);
[nomMDD]=[variable];
pthread_mutex_unlock(&amp;s_[nomMDD]);
</Ecrire_Mdd>
<Lire_Mdd>
pthread_mutex_lock(&amp;s_[nomMDD]);
[variable]=[nomMDD];
pthread_mutex_unlock(&amp;s_[nomMDD]);
</Lire_Mdd>
</MDD>
<Gestion_Temps>
/* Modifie la structure time afin d'y ajouter us microsecondes*/
void ajouter_microsecondes(struc timespec *time,long us) {
/* Ajout des microsecondes au champs en nanosecondes */
(*time).tv.nsec+=(us%1000000)*1000;
/* Ajout des secondes enti&#xe8;res au champs en secondes */
(*time).tv.sec+=(us/1000000)+((*time).tv.nsec/1000000000);
/* Si il y avait d&#xe9;bordement des nanosecondes sur les
secondes, il a &#xe9;t&#xe9; ajout&#xe9; aux secondes dans
l'instruction pr&#xe9;c&#xe9;dente */
(*time).tv.nsec%=1000000000;
}
</Gestion_Temps>
<TACHE>
<DeclaTacheP>/* T&#xe2;che p&#xe9;riodique d'acquisition */
void [nomTache]() {
struct timespec horloge; /* Date du prochain r&#xe9;veil */
sem_t Reveil; /* S&#xe9;maphore utilis&#xe9; pour le r&#xe9;veil
p&#xe9;riodique */

/* Variable locale de la t&#xe2;che: [nomTache] */
[variables]
/* end variable locale: [nomTache] */

sem_init(Reveil,0,0);
clock_gettime(CLOCK_REALTIME,horloge); /* Heure courante de
l'horloge
au d&#xe9;marrage de la t&#xe2;che */
while (1) {
/* Corps de la t&#xe2;che: [nomTache] */

/* L'instruction de recevoir un message de la t&#xe2;che:
[nomTache] */
[instructions_1]

/* Partie d'utilisateur de la t&#xe2;che: [nomTache] */
/* ----------------code----------------- */
/* end partie d'utilisateur: [nomTache] */

/* L'instruction d'envoyer un message de la t&#xe2;che:
[nomTache] */
[instructions_2]

/* P&#xe9;riodicit&#xe9; */
/* Calcul de la prochaine date de r&#xe9;veil */
ajouter_microsecondes(&amp;horloge,[periode]*1000);
/* Attente de la prochaine date de r&#xe9;veil */
sem_timedwait(&amp;Reveil,&amp;horloge);
}
}
</DeclaTacheP>
<DeclaTacheAP>/* T&#xe2;che ap&#xe9;riodique */
```

```
128 void [nomTache]() {
129 /* Variable locale de la t&#xe2;che: [nomTache] */
130 [variables]
131 /* end variable locale: [nomTache] */
132
133 while (1) {
134 /* Corps de la t&#xe2;che: [nomTache] */
135 /* Attendre de d&#xe9;clenchement pour activer la t&#xe2;che:
136 [nomTache] */
137 [instructions_1]
138
139 /* Partie d'utilisateurs de la t&#xe2;che: [nomTache] */
140 /* ----------------code----------------- */
141 /* end partie d'utilisateur: [nomTache] */
142
143 /* L'instruction d'envoyer un message de la t&#xe2;che:
144 [nomTache] */
145 [instructions_2]
146 }
147 }
148 </DeclaTacheAP>
149 <Creer_Tache>
150 /* Priorit&#xe9; de la prochaine t&#xe2;che &#xe0; cr&#xe9;er*/
151 schedparam.sched_priority=[priority];
152 /* La priorit&#xe9; est plac&#xe9;e dans les attributes */
153 pthread_attr_setschedparam(&amp;attributs,&amp;schedparam);
154 pthread-create(&amp;thread_[nomTache],&amp;attributs,
155                    [nomTache],NULL);
156 </Creer_Tache>
157 <!-- Supposer que [nom_thread]=thread_[nomTache] -->
158 <Mode_Attache>
159 pthread_join(thread_[nomTache],NULL);
160 </Mode_Attache>
161 <!-- Supposer que [nom_thread]=thread_[nomTache] -->
162 <Iden_Tache>
163 /* Identificateur de t&#xe2;che [nomTache] */
164 pthread_t thread_[nomTache];
165 </Iden_Tache>
166 <!-- Supposer que [nom_thread]=thread_[nomTache] -->
167 </TACHE>
168 <Main><!-- [iden_tache_global] -->
169 void main() {
170 [iden_tache_local]
171 struct sched_param schedparam; /* Paramètre servant à affecter
172 une priorit&#xe9; aux t&#xe2;che */
173 pthread_attr_t attributs; /* Attributs utilis&#xe9;s lors de la
174 cr&#xe9;ation des t&#xe2;ches */
175
176 /* Initialisation du mat&#xe9;riel (capteurs, actionneurs) */
177
178 /* Initialisation des outils de communication/synchronisation*/
179 [init_outils_com/syn]
180
181 pthread_attr_init(&amp;attributs); /* Initialisation des
182 attributs aux valeurs par d&#xe9;faut */
183 pthread_attr_setinheritsched(&amp;attributs,
184                                  PTHREAD_EXPLICIT_SCHED); /* Force
185 la prise en compte des param&#xe8;tres d'ordonnancement de
186 l'attribut: sans cela, ils seraient h&#xe9;rit&#xe9;s du
187 programme principal */
188 [creer_tache]
189 [mode_attache]
190 }
191 </Main>
192 </NORME_C>
```

De même, le fichier XML pour la génération de code spécifique à la norme VxWorks, appelé «VxWorks.xml», est le suivant:

-- **Fichier «VxWorks.xml»**

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE NORME_C SYSTEM "Norme_C_DTD.dtd">
<!-- La norme VxWorks -->
<NORME_C>
<En_Tete>
#include &lt;taskLib.h&gt;
#include &lt;msgQLib.h&gt;
#include &lt;semCLib.h&gt;
#include &lt;semMLib.h&gt;

/* Fonctions sp&#xe9;cifiques aux utilisateurs */

/* D&#xe9;finition de constantes */

</En_Tete>
<SEMAPHORE>
<Decla_Sem>
/* La synchronisation par Semaphore est remplac&#xe9;e par BAL, avec un message du type Bool&#xe9;en */
/* Type utilisé et la déclaration de la BAL: [nomSEM] */
typedef struct {
Boolean message;
} t_[nomSEM];
MSQ_Q_ID [nomSEM]; /* BAL born&#xe9;e sans &#xe9;crasement */
</Decla_Sem>
<Init_Sem>
[nomSEM]=msgQCreate(1,sizeof(t_[nomSEM]),MSG_Q_PRIORITY);
</Init_Sem>
<Vendre_Sem>
msgQSend([nomSEM],(char*)&amp;[variable],sizeof([variable]),
          WAIT_FOREVER,MSG_PRI_NORMAL);
</Vendre_Sem>
<Prendre_Sem>
msgQReceive([nomSEM],(char*)&amp;[variable],sizeof([variable]),
             WAIT_FOREVER);
</Prendre_Sem>
</SEMAPHORE>
<BAL>
<Bal>
<Decla_Bal>
MSQ_Q_ID [nomBAL]; /* BAL born&#xe9;e sans &#xe9;crasement */
</Decla_Bal>
<Init_Bal>
[nomBAL]=msgQCreate(1,sizeof([typedonnee]),MSG_Q_PRIORITY);
</Init_Bal>
<Envoyer_Bal>
msgQSend([nomBAL],(char*)&amp;[variable],sizeof([variable]),
          WAIT_FOREVER,MSG_PRI_NORMAL);
</Envoyer_Bal>
<Recevoir_Bal>
msgQReceive([nomBAL],(char*)&amp;[variable],sizeof([variable]),
             WAIT_FOREVER);
</Recevoir_Bal>
</Bal>
<!-- La BAL_Ecr n'existe pas dans la norme VxWorks. Elle est remplacée par l'outil BAL sans &#xe9;crasement -->
<Bal_Ecr />
</BAL>
<MDD>
<Decla_Mdd>
[typedonnee] [nomMDD]; /* D&#xe9;claration du MDD */
SEM_ID s_[nomMDD]; /* Mutex du MDD */
</Decla_Mdd>
<Init_Mdd>
s_[nomMDD]=semMCreate(SEM_Q_PRIORITY|SEM_INVERSION_SAFE);
```

```
</Init_Mdd>
<Ecrire_Mdd>
semTake(s_[nomMDD],WAIT_FOREVER);
[nomMDD]=[variable];
semGive(s_[nomMDD]);
</Ecrire_Mdd>
<Lire_Mdd>
semTake(s_[nomMDD],WAIT_FOREVER);
[variable]=[nomMDD];
semGive(s_[nomMDD]);
</Lire_Mdd>
</MDD>
<Gestion_Temps />
<TACHE>
<DeclaTacheP>/* T&#xe2;che p&#xe9;riodique d'acquisition */
void [nomTache]() {
/* Variable locale de la t&#xe2;che: [nomTache] */
[variables]
/* end variable locale: [nomTache] */

while (1) {
/* Corps de la t&#xe2;che: [nomTache] */

/* L'instruction de recevoir un message de la t&#xe2;che: [nomTache] */
[instructions_1]

/* Partie d'utilisateur de la t&#xe2;che: [nomTache] */
/* ----------------code----------------- */
/* end partie d'utilisateur: [nomTache] */

/* L'instruction d'envoyer ou de recevoir un message de la t&#xe2;che: 97|
[nomTache] */
[instructions_2]

/* P&#xe9;riodicit&#xe9; */
/* P&#xe9;riode de [periode] avec d&#xe9;rive des horloges */
taskDelay(sysClkRateGet()*[periode]/1000);
}
}
</DeclaTacheP>
<DeclaTacheAP>/* T&#xe2;che ap&#xe9;riodique */
void [nomTache]() {
/* Variable locale de la t&#xe2;che: [nomTache] */
[variables]
/* end variable locale: [nomTache] */

while (1) {
/* Corps de la t&#xe2;che: [nomTache] */
/* Attendre de d&#xe9;clenchement pour activer la t&#xe2;che:
[nomTache]*/
[instructions_1]

/* Partie d'utilisateurs de la t&#xe2;che: [nomTache] */
/* ----------------code----------------- */
/* end partie d'utilisateur: [nomTache] */

/* L'instruction d'envoyer ou de recevoir un message de la t&#xe2;che:
[instructions_2]
}
}
</DeclaTacheAP>
<Creer_Tache>
thread_[nomTache]=taskSpawn(&quot;[nomTache]&quot;,[priority],0,20000,
                            [nomTache],0,0,0,0,0,0,0,0,0,0);
</Creer_Tache>
```

```
131| <Mode_Attache />
132| <!-- Les taches s'executent en mode detache -->
133| <Iden_Tache>
134| TASK_ID thread_[nomTache]; /* Identificateur de t&#xe2; [nomTache] */
135| </Iden_Tache>
136| <!-- Supposer que [nom_thread]=thread_[nomTache] -->
137| </TACHE>
138| <Main>
139| [iden_tache_global]
140| void LancerTaches() {
141| <!-- [ident_tache_local] -->
142| /* Initialisation mat&#xe9;rielle du syst&#xe8;me (capteurs,
143| actionneurs, etc) */
144|
145| /* Initialisation des outils de communication et synchronisation */
146| [init_outils_com/syn]
147|
148| [creer_tache]
149| [mode_attache]
150| }
151| </Main>
152| </NORME_C>
```

Nous voyons donc que la DTD donnée dans le fichier «Norme_C_DTD.dtd» peut être utilisée pour valider des fichiers XML spécifiques à différentes normes temps réel du langage C. Grâce à la grammaire définie par cette DTD, l'utilisateur peut construire aisément son propre fichier XML conforme à la norme de langage C qu'il utilise.

En pratique, pour être exécutable sur un exécutif temps réel, les informations spécifiques à cet exécutif, ou bien qu'aux utilisateurs doivent être bien déclarées dans le code final généré. En utilisant des marquages en forme des commentaires du langage C, les parties de code dépendant du matériel ou d'utilisateur peut être ajoutées facilement soit dans les fichiers XML de définition de la norme temps réel, soit au plus tard dans le code final généré. Par exemple, dans le fichier «POSIX.xml», dans la balise `<En_Tete>` le commentaire «`/* Fonctions spécifiques aux utilisateurs */`» indique la déclaration des fonctions spécifiques aux utilisateurs, donc elle peut être donnée directement dans ce fichier. Tandis que le code dans le corps de chaque tâche entre le commentaire «`/* Partie d'utilisateurs de la tâche: [nomTache] */`» (ligne 111 ou 139) et le commentaire «`/* end partie d'utilisateur: [nomTache] */`» (ligne 113 ou 141), ne peut être déclaré que de façon indirecte dans le code final s'il est nécessaire.

Il est important de noter que les parties entre crochets ([…]) dans les fichiers XML des normes POSIX et VxWorks sont remplacées par des informations correspondantes récupérées à partir du fichier XML de description du modèle DARTSVIEW durant la génération de code. De plus, certains marquages entre crochets parmi eux tels «`[instructions_1]`», «`[instructions_2]`», ou «`[variables]`» (lignes 99, 109, 117, 130, 137, 145 dans «POSIX.xml»; ou lignes 83, 90, 98, 109, 116, 123 dans «VxWorks.xml») sont utilisés pour faciliter la gestion de génération de code d'une tâche, car ils permettent de grouper un ensemble des codes: par exemple «`[instructions_1]`» représente le code pour toutes les attentes de synchronisation ou de communication lors d'exécution d'une tâche; tandis que «`[instructions_2]`» représente le code pour toutes les arrivées de synchronisation ou de communication; ou bien que «`[variables]`» pour le code des variables utilisées dans des outils de communication.

Enfin, afin de terminer cette section, je parle de code d'implémentation des déclenchements des tâches apériodiques que j'ai proposé dans les fichiers XML des normes temps réel du langage C. Puisqu'une tâche apériodique peut être une tâche matérielle, ou une tâche logicielle, donc en

fonction de chaque tâche (matérielle ou logicielle), son déclenchement peut être interprété de façon suivante: en cas des tâches logicielles, le code de l'«`[instructions_1]`» joue le rôle de déclenchement de ces tâches; alors que pour des tâches matérielles, le code de leur déclenchement doit être rédigé par l'utilisateur après la génération de code. Cette partie de code se trouve entre le commentaire «`/* Partie d'utilisateurs de la tâche: [nomTache] */`» et le commentaire «`/* end partie d'utilisateur: [nomTache] */`» dans le corps de chaque tâche.

III.5.4.b. Introduction au VI de génération de code pour les normes du langage C

Le VI «Generer Code Norme Langage C» dans la sous-palette «Générer Code» (voir Figure III-37) de la palette DARTSVIEW permet de générer le code d'un système conformément à une norme temps réel du langage C à partir d'un fichier XML de description de son modèle DARTSVIEW.

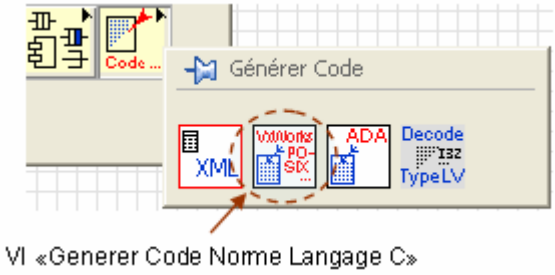


Figure III-37: VI «Generer Code Norme Langage C» dans la sous-palette «Générer Code»

Le diagramme du VI «Generer Code Norme Langage C» est présenté sur la Figure III-38. Dans ce diagramme, le choix des normes temps réel du langage C est réalisé par un objet «Bouton radio» de LabVIEW, donc chaque choix de cet objet correspond à une norme avec le chemin de son propre fichier XML (il en résulte que les fichiers XML spécifiques à des normes du langage C et le fichier XML-DTD pour la validation des fichiers des normes doivent être dans le répertoire du VI «Generer Code Norme Langage C»); le chemin du fichier XML de description du modèle DARTSVIEW doit être précisé pour récupérer des informations du modèle. Si tous les chemins sont corrects, la génération de code des normes du langage C a lieu de manière à ce que le code final soit un regroupement des cinq parties de code générées par cinq VI, qui correspondent chacune à une partie dans la DTD des normes du langage C: ainsi, le VI «Code En_Tete» pour le code d'en-tête; le VI «Code Declaration Outils Comm-Synch» pour le code de déclaration de type de données des outils de communication/synchronisation, et de ces outils eux-mêmes; le VI «Code Gestion Temps» pour le code des fonctions et/ou primitives permettant la gestion de temps des tâches périodiques; le VI «Code Taches» pour le code d'implémentation des tâches; et le VI «Code main()» pour le code de la procédure «*void main()*».

Puisque le code est généré à partir des informations du modèle DARTSVIEW, il est évident qu'un changement ou une modification d'un des éléments dans le modèle DARTSVIEW entraîne immédiatement un changement de la partie de code de cet élément, mais n'entraîne aucun impact sur le code des autres éléments, notamment le code concernant le matériel (i.e., capteurs, actionneurs, etc.), ou le code utilisateur dans le cops de chaque tâche. Par conséquent, il faut que l'utilisateur réécrive tous le code spécifique une fois que le modèle DARTSVIEW du système est changé. Je propose une technique permettant de diminuer considérablement les travaux requis pour modifier le code en cas de changement du modèle DARTSVIEW: s'il y a une modification dans le modèle DARTSVIEW d'un système après la génération de code, grâce aux marquages prédéfinis en forme des commentaires dans l'ancien fichier de code, les informations utilisateur sont récupérées, et réinsérées dans le nouveau fichier de code généré.

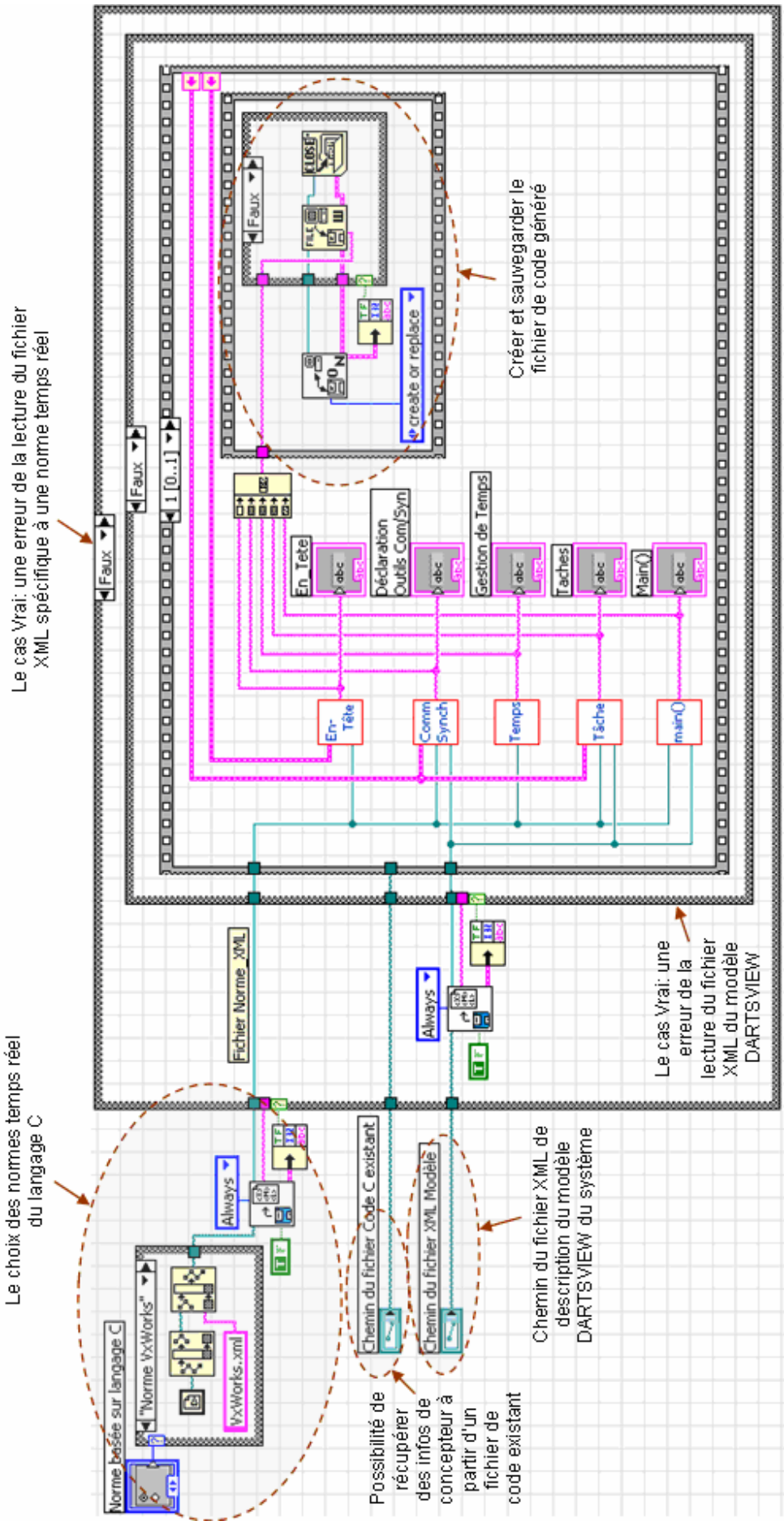


Figure III-38: Diagramme du VI «Generer Code Norme Langage C»

Le code de la norme POSIX du modèle DARTSVIEW de l'exemple présenté sur la Figure III-30 est donné ci-après, ce code est généré par le VI «Generer Code Norme Langage C» en associant avec le fichier XML de ce modèle généré par le VI «Generer Fichier XML» en dessus (réf. III.5.3.b.):

```
-----------------------------------------------------------------------

#include <pthread.h>
#include <semaphore.h>
#include <BALs.h>

/* Fonctions spécifiques aux utilisateurs */

/* Définition de constantes */

/* Définition des outils de communication/synchronisation */
/* Type utilisé et la déclaration de la BAL: BAL_1 */
typedef struct {
/* type utilisé dans BAL_1 est:  DBL  */
} TypeFlottant;
bal BAL_1; /*BAL de taille 1 sans écrasement */

/* Gestion de temps dans */
void ajouter_microsecondes(struc timespec *time,long us) {
/* Modifie la structure time afin d'y ajouter us microsecondes */
      (*time).tv.nsec+=(us%1000000)*1000; /* Ajout des microsecondes au
champs en nanosecondes */
      (*time).tv.sec+=(us/1000000)+((*time).tv.nsec/1000000000); /*
Ajout des secondes entières au champs en secondes */
      (*time).tv.nsec%=1000000000; /* Si il y avait débordement des
nanosecondes sur les secondes, il a été ajouté aux secondes dans
l'instruction précédente */
}

void Tache1() {
/* Tâche périodique d'acquisition */
struct timespec horloge; /* Date du prochain réveil */
sem_t Reveil; /* Sémaphore utilisé pour le réveil périodique */

/* Variable locale de la tâche: Tache1 */
TypeFlottant var_BAL_1;
/* end variable locale: Tache1 */

sem_init(Reveil,0,0);
clock_gettime(CLOCK_REALTIME,horloge); /* Heure courante de l'horloge
au démarrage de la tâche */
while (1) {
/* Corps de la tâche: Tache1 */

/* L'instruction de recevoir un message de la tâche: Tache1 */


/* Partie d'utilisateur de la tâche: Tache1 */
/* ----------------code----------------- */
/* end partie d'utilisateur: Tache1 */

/* L'instruction d'envoyer un message de la tâche: Tache1 */
bal_envoyer(BAL_1,(char*)&var_BAL_1); /* Envoi un message */

/* Périodicité */
ajouter_microsecondes(&horloge,1000*1000); /* Calcul de la prochaine
date de réveil */
sem_timedwait(&Reveil,&horloge); /* Attente de la prochaine date de
réveil */
}
```

```
}

void Tache2() {
/* Tâche apériodique */
/* Variable locale de la tâche: Tache2 */
TypeFlottant var_BAL_1;
/* end variable locale: Tache2 */

while (1) {
/* Corps de la tâche: Tache2 */
/* Attendre de déclenchement pour activer la tâche: Tache2 */
bal_recevoir(BAL_1,(char*)&var_BAL_1); /* Attente de message */

/* Partie d'utilisateurs de la tâche: Tache2 */
/* ----------------code----------------- */
/* end partie d'utilisateur: Tache2 */

/* L'instruction d'envoyer un message de la tâche: Tache2 */

}
}

void main() {
pthread_t thread_Tache1; /* Identificateur de tâche Tache1 */
pthread_t thread_Tache2; /* Identificateur de tâche Tache2 */
struct sched_param schedparam; /* Paramètre servant à affecter une
priorité aux tâche */
pthread_attr_t attributs; /* Attributs utilisés lors de la création des
tâches */

/* Initialisation du matériel (capteurs, actionneurs, etc) */

/* Initialisation des outils de communication et synchronisation */
BAL_1=bal_init(sizeof(t_BAL_1));

pthread_attr_init(&attributs); /* Initialisation des attributs aux
valeurs par défaut */
pthread_attr_setinheritsched(&attributs,PTHREAD_EXPLICIT_SCHED); /*
Force la prise en compte des paramètres d'ordonnancement de l'attribut:
sans cela, ils seraient hérités duprogramme principal */
/* Priorité de la prochaine tâche à créer*/
schedparam.sched_priority=[priority];
/* La priorité est placée dans les attributes */
pthread_attr_setschedparam(&attributs,&schedparam);
pthread-create(&thread_Tache1,&attributs,Tache1,NULL);
/* Priorité de la prochaine tâche à créer*/
schedparam.sched_priority=[priority];
/* La priorité est placée dans les attributes */
pthread_attr_setschedparam(&attributs,&schedparam);
pthread-create(&thread_Tache2,&attributs,Tache2,NULL);
pthread_join(thread_Tache1,NULL);
pthread_join(thread_Tache2,NULL);
}
```

Remarquons que le type de données des messages dans des outils de communication est déclaré en forme des types structures synonymes, avec des types composants exprimés par types de données LabVIEW en forme des commentaires. Il est requis que l'utilisateur remplace ces commentaires par le nom et le type C correspondant, par exemple dans le code généré dessus, il faut remplacer le code des déclarations de type de données des messages dans la BAL, nommée «BAL_1»

```
...
typedef struct {
/* type utilisé dans BAL_1 est:  DBL  */
```

```
} TypeFlottant;
...
```

par le code ci-après:

```
...
typedef struct {
double message;
} TypeFlottant;
...
```

III.5.4.c. Fichier XML-DTD pour la norme Ada

Suite à la présentation de la norme Ada aux chapitre I (réf. I.4.6.c.) et chapitre II (réf. II.3.3.), je propose une DTD pour le fichier XML spécifié à la norme Ada ci-après:

-- Fichier «Norme_Ada_DTD.dtd»

```
 1| <!-- DTD pour les normes temps-r&#xe9;el bas&#xe9;es sur le
        langage Ada -->
 2| <!ELEMENT NORME_Ada (En_Tete,SEMAPHORE,BAL,MDD,TACHE,Main)>
 3| <!ELEMENT En_Tete (#PCDATA)>
 4| <!ELEMENT SEMAPHORE (PackageSEM)>
 5| <!ELEMENT PackageSEM (#PCDATA)>
 6| <!ELEMENT BAL (Bal,Bal_Ecr)>
 7| <!ELEMENT Bal (Decla_Bal,PackageBal)>
 8| <!ELEMENT Decla_Bal (#PCDATA)>
 9| <!ELEMENT PackageBal (#PCDATA)>
10| <!ELEMENT Bal_Ecr (Decla_Bal_Ecr,PackageBal_Ecr)>
11| <!ELEMENT Decla_Bal_Ecr (#PCDATA)>
12| <!ELEMENT PackageBal_Ecr (#PCDATA)>
13| <!ELEMENT MDD (Decla_Mdd,PackageMdd)>
14| <!ELEMENT Decla_Mdd (#PCDATA)>
15| <!ELEMENT PackageMdd (#PCDATA)>
16| <!ELEMENT TACHE (SpecificationTache,BodyTacheP,BodyTacheAP)>
17| <!ELEMENT SpecificationTache (#PCDATA)>
18| <!ELEMENT BodyTacheP (#PCDATA)>
19| <!ELEMENT BodyTacheAP (#PCDATA)>
20| <!ELEMENT Main (#PCDATA)>
```

Selon cette DTD, l'élément `<NORME_Ada>` est défini en tant que l'élément racine du fichier XML spécifique à la norme Ada, qui possède six éléments enfants qui sont assez proches dans leurs fonctions des éléments obtenus pour C.

En se basant sur cette DTD, le fichier XML pour la génération de code spécifique à la norme Ada 95, appelé «Ada.xml», est donné ci-après:

-- Fichier «Ada.xml»

```
 1| <?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
 2| <!DOCTYPE NORME_Ada SYSTEM "Norme_Ada_DTD.dtd">
 3| <!-- La norme Ada -->
 4| <NORME_Ada>
 5| <En_Tete>pragma
Task_Dispatching_Policy(Fifo_Within_Priorities); --
 6| Ordonnancement FIFO par niveaux de priorit&#xe9;
 7| pragma Locking_Policy(Ceiling_Locking); --Protocole &#xe0;
 8| priorit&#xe9; plafond imm&#xe9;diat
 9| pragma Queuing_Policy(Priority_Queuing); --Gestion des
attentes d'objet
```

```
10| prot&#xe9;g&#xe9; par priorit&#xe9; des t&#xe2;ches
11| with Modele;
12| </En_Tete>
13| <!-- Outils de communication et synchronisation -->
14| <SEMAPHORE>
15| <!-- Fichier P_[nomSEM].ads -->
16| <PackageSEM>-- D&#xe9;claration de package du SEMAPHORE:
[nomSEM]
17| with SEMAPHORE;
18| package P_[nomSEM] is
19| [nomSEM]: SEMAPHORE.Semaphore;
20| end P_[nomSEM];
21| </PackageSEM>
22| </SEMAPHORE>
23| <BAL>
24| <Bal>
25| <!-- Fichier T_[nomBAL].ads -->
26| <Decla_Bal>-- D&#xe9;claration du type de la BAL: [nomBAL],
nomm&#xe9;
27| T_[nomBAL]
28| -- avec un message du type: t_[nomBAL] d&#xe9;finie dans le
fichier
29| LVTypes.ads
30| with LVTypes, BAL;
31| package T_[nomBAL] is new BAL(T => LVTypes.t_[nomBAL]);
32| </Decla_Bal>
33| <!-- Fichier P_[nomBAL].ads -->
34| <PackageBal>-- D&#xe9;claration de package de la BAL:
[nomBAL]
35| with T_[nomBAL];
36| use T_[nomBAL];
37| package P_[nomBAL] is
38| [nomBAL]: Bal;
39| end P_[nomBAL];
40| </PackageBal>
41| </Bal>
42| <Bal_Ecr>
43| <!-- Fichier T_[nomBAL_Ecr].ads -->
44| <Decla_Bal_Ecr>-- D&#xe9;claration du type de la BAL_Ecr:
[nomBAL],
45| nomm&#xe9; T_[nomBAL], avec un message du type: t_[nomBAL]
d&#xe9;finie
46| dans le fichier LVTypes.ads
47| with LVTypes, BAL_Ecr;
48| package T_[nomBAL] is new BAL_Ecr(T => LVTypes.t_[nomBAL]);
49| </Decla_Bal_Ecr>
50| <!-- Fichier P_[nomBAL_Ecr].ads -->
51| <PackageBal_Ecr>-- D&#xe9;claration de package de la BAL_Ecr:
[nomBAL]
52| with T_[nomBAL];
53| use T_[nomBAL];
54| package P_[nomBAL] is
55| [nomBAL]: Bal_Ecr;
```

```
56| end P_[nomBAL];
57| </PackageBal_Ecr>
58| </Bal_Ecr>
59| </BAL>
60| <MDD>
61| <!-- Fichier T_[nomMDD].ads -->
62| <Decla_Mdd>-- D&#xe9;claration du type de MDD: [nomMDD],
nomm&#xe9;
63| T_[nomMDD], avec un message du type: t_[nomMDD] d&#xe9;finie
dans le
64| fichier LVTypes.ads
65| with LVTypes, MDD;
66| package T_[nomMDD] is new MDD(T => LVTypes.t_[nomMDD]);
67| </Decla_Mdd>
68| <!-- Fichier P_[nomMDD].ads -->
69| <PackageMdd>-- D&#xe9;claration de package de la MDD:
[nomMDD]
70| with T_[nomMDD];
71| use T_[nomMDD];
72| package P_[nomMDD] is
73| [nomMDD]: Mdd;
74| end P_[nomMDD];
75| </PackageMdd>
76| </MDD>
77| <TACHE>
78| <SpecificationTache>-- D&#xe9;claration de la t&#xe2;che:
[nomTache]
79| task [nomTache] is
80| pragma Priority([priority]);
81| end [nomTache]; -- end d&#xe9;claration: [nomTache]
82| </SpecificationTache>
83| <BodyTacheP>-- Description de la t&#xe2;che: [nomTache], une
t&#xe2;che
84| p&#xe9;riodique
85| task body [nomTache] is
86| Periode: constant Time_Span := Milliseconds([periode]);
87| Prochaine: Time := Clock+Periode; -- Prochaine date de
r&#xe9;veil
88| -- Variables locales de la t&#xe2;che: [nomTache]
89| [variable]
90| -- end variables locales: [nomTache]
91| begin
92| -- Corps de la t&#xe2;che: [nomTache]
93| loop
94| -- L'instruction de recevoir un message de la t&#xe2;che:
[nomTache]
95| [instructions_1]
96| -- Partie d'utilisateur de la t&#xe2;che: [nomTache]
97| -- .................code..................
98| -- end partie d'utilisateur: [nomTache]
99| -- L'instruction d'envoyer un message de la t&#xe2;che:
[nomTache]
100| [instructions_2]
```

```
101| -- P&#xe9;riodici&#xe9;
102| delay until (Prochaine);
103| Prochaine:= Prochaine+Periode; --Calcul de la prochaine date
de
104| r&#xe9;veil
105| end loop;
106| end [nomTache]; -- end description: [nomTache]
107| </BodyTacheP>
108| <BodyTacheAP>-- Description de la t&#xe2;che: [nomTache],
une
109| t&#xe2;che ap&#xe9;riodique
110| task body [nomTache] is
111| -- Variables locales de la t&#xe2;che: [nomTache]
112| [variable]
113| -- end variables locales: [nomTache]
114| -- Corps de la t&#xe2;che: [nomTache]
115| begin
116| loop
117| -- Attendre un d&#xe9;clenchement pour activer la
t&#xe2;che:
118| [nomTache]
119| [instructions_1]
120| -- Partie d'utilisateurs de la t&#xe2;che: [nomTache]
121| -- .................code..................
122| -- end partie d'utilisateur: [nomTache]
123| -- L'instruction d'envoyer un message de la t&#xe2;che:
[nomTache]
124| [instructions_2]
125| end loop;
126| end [nomTache]; -- end description: [nomTache]
127| </BodyTacheAP>
128| </TACHE>
129| <Main>procedure Main is
130| begin
131| null;
132| end Main;
133| </Main>
134| </NORME_Ada>
```

Dans le contenu de la balise `<En_Tete>` (lignes 5-12), en plus des déclarations de l'ordonnancement des tâches, de protocole de la gestion des accès à un objet protégé, il existe une déclaration du paquetage, appelé «Modele» (c'est la ligne 11, «`with Modele;`») qui indique que toutes les tâches en Ada du modèle DARTSVIEW d'un système sont bien déclarées dans ce paquetage. ainsi, durant la génération de code, deux fichiers, «Modele.ads» et «Modele.adb», en langage Ada correspondants aux fichiers de la spécification et du corps de chaque tâche sont créés automatiquement. Et le nom «Modele» dans ce cas est fixé de façon délibérée pour le code Ada de chaque modèle DARTSVIEW afin de simplifier la phase de la génération de code Ada.

On trouve également les autres paquetages sur les lignes 17, 30, 47, 65 qui sont nommés respectivement «SEMAPHORE», «LVTypes», «BAL», «BAL_Ecr», et «MDD». Ce sont les paquetages prédéfinis pour la programmation multitâche en langage de façon assez semblable à [CG05]. Remarquons que le paquetage «LVTypes» sert à définir des types de données LabVIEW

en types de données Ada sous la forme des types synonymes, donc les types I32, I16, I8 dans LabVIEW peuvent être définis en tant que type Entier de Ada de manière suivante:

```
...
type I32 is new Integer;
type I16 is new Integer range -32768..32767;
type I8 is new Integer range -128..127;
...
```

Le type de données des messages dans des outils de communication est défini en Ada en forme d'un type structure synonyme de la façon suivante:

```
...
type [nomTypedeDonnee] is record
   valeur_1: [typededonnee_LabVIEW];
   valeur_2: [typededonnee_LabVIEW];
   ...
end record;
...
```

De même que les fichiers XML des normes POSIX et VxWorks, les parties entre crochets ([…]) dans les fichiers XML de la norme Ada sont remplacées par des informations correspondantes récupérées à partir du fichier XML de description du modèle DARTSVIEW lors de la génération de code.

III.5.4.d. Introduction au VI de génération de code pour la norme Ada

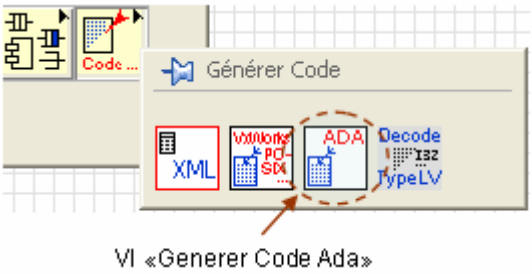


Figure III-39: VI «Generer Code Ada» dans la sous-palette «Générer Code»

Le VI «Generer Code Ada» dans la sous-palette «Générer Code» (voir Figure III-39 et Figure III-37) de la palette DARTSVIEW permet de générer le code d'un système conforme à une norme temps réel du langage Ada à partir d'un fichier XML de description de son modèle DARTSVIEW.

Le diagramme du VI «Generer Code Ada» est présenté sur la Figure III-40. Dans ce diagramme, l'utilisateur doit déclarer le chemin du fichier XML de description du modèle DARTSVIEW d'abord, et puis choisir le chemin pour les fichiers de code Ada générés. Dans ce cas, je suppose que le fichier XML spécifique à la norme Ada (par exemple, le fichier «Ada .xml») se trouve de façon délibérée dans le même répertoire que celui du VI «Generer Code Ada» de la bibliothèque DARTSVIEW.

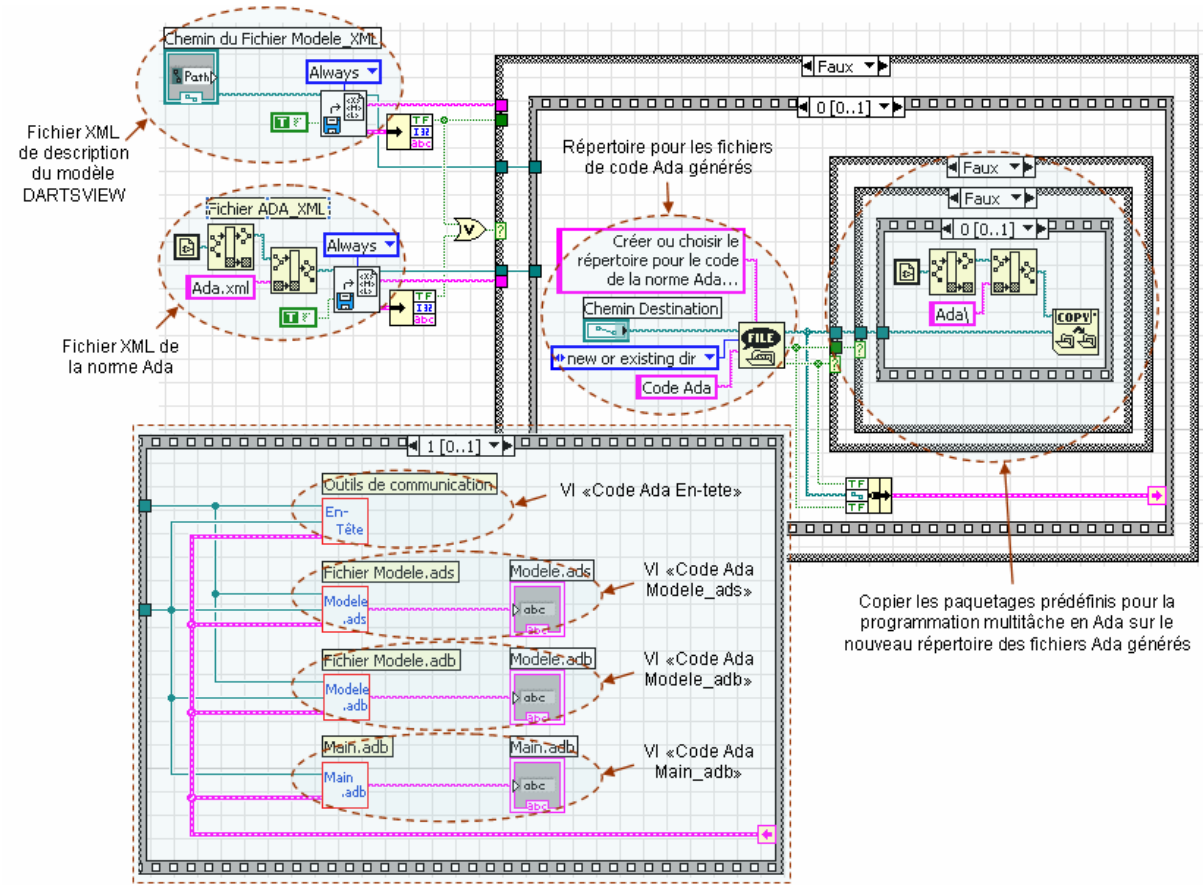


Figure III-40: Diagramme du VI «Generer Code Ada»

- *le VI «Code Ada En-Tete»*: génère le code pour la déclaration des paquetages des outils de communication/synchronisation utilisés (donc, chaque outil correspond à un paquetage, nommé «P_[nomOutil].ads»); le code pour la définition des types de données LabVIEW en Ada sous forme de types synonymes (le paquetage «LVTypes.ads»); et le code pour la déclaration des paquetages des types d'outil de communication/synchronisation avec un message du type de données bien défini dans le paquetage «LVType.ads» (donc, chaque outil correspond à un paquetage de type, nommé «T_[nomOutil].ads»). Ainsi, la syntaxe de déclaration du paquetage d'un outil de communication est suivante:

```
with T_[nomOutil];
use T_[nomOutil];
package P_[nomOutil] is
   [nomOutil]: Bal|Bal_Ecr|Mdd;
end P_[nomOutil];
```

- *le VI «Code Ada Modele_ads»*: génère le code de la spécification du paquetage «Modele», qui comporte les spécifications de toutes les tâches dans le modèle DARTSVIEW;
- *le VI «Code Ada Modele_adb»*: génère le code d'implémentation du paquetage «Modele», qui comporte les corps de toutes des tâches dans le modèle DARTSVIEW;
- *le VI «Code Ada Main_abs»*: génère le code du programme principal, nommé «Main».

Enfin, afin de montrer le code généré par ce VI, le code Ada du modèle DARTSVIEW de l'exemple présenté sur la Figure III-30 est donné ci-après (paquetages «Modele» et «LVTypes»).

```
-------------------------------------------------------------------------
-- Fichier «LVTypes.ads»
package LVTypes is
type I32 is new Integer;
type I16 is new Integer range -32768..32767;
```

```
type I8 is new Integer range -128..127;
type U32 is new Natural;
type U16 is new Natural range 0..65535;
type U8 is new Natural range 0..255;
type DBL is new Long_Float;
type SGL is new Float;
type Str is new String;
type t_BAL_1 is record
   -- valeur_1: DBL;
end record;
end LVTypes;
```

-- Fichier «Modele.ads»

```
package Modele is
-- Déclaration de la tâche: Tache1
task Tache1 is
pragma Priority([priority]);
end Tache1; -- end déclaration: Tache1

-- Déclaration de la tâche: Tache2
task Tache2 is
pragma Priority([priority]);
end Tache2; -- end déclaration: Tache2
end Modele;
```

-- Fichier «Modele.adb»

```
with Ada.Real_Time;use Ada.Real_Time;
with LVTypes;use LVTypes;

-- Définition des outils de communication
with P_BAL_1;use P_BAL_1;

-- Définit les fonctions d'accès aux capteurs/actionneurs
-- ...

package body Modele is
-- Description de la tâche: Tache1, une tâche périodique
task body Tache1 is
Periode: constant Time_Span := Milliseconds(1000);
Prochaine: Time := Clock+Periode; -- Prochaine date de réveil

-- Variables locales de la tâche: Tache1
var_BAL_1: t_BAL_1;
-- end variables locales: Tache1

begin
-- Corps de la tâche: Tache1
loop
-- L'instruction de recevoir un message de la tâche: Tache1


-- Partie d'utilisateur de la tâche: Tache1
-- .................code..................
-- end partie d'utilisateur: Tache1

-- L'instruction d'envoyer un message de la tâche: Tache1
BAL_1.Envoyer(var_BAL_1);

-- Périodicié
delay until (Prochaine);
Prochaine:= Prochaine+Periode; --Calcul de la prochaine date de réveil
end loop;
end Tache1; -- end description: Tache1

-- Description de la tâche: Tache2, une tâche apériodique
task body Tache2 is
```

```
-- Variables locales de la tâche: Tache2
var_BAL_1: t_BAL_1;
-- end variables locales: Tache2

-- Corps de la tâche: Tache2
begin
loop
-- Attendre un déclenchement pour activer la tâche: Tache2
BAL_1.Recevoir(var_BAL_1);

-- Partie d'utilisateur de la tâche: Tache2
-- .................code.................
-- end partie d'utilisateur: Tache2

-- L'instruction d'envoyer un message de la tâche: Tache2


end loop;
end Tache2; -- end description: Tache2
end Modele;

-----------------------------------------------------------------
```

III.5.5. Conclusion

En conclusion de cette section, DARTSVIEW est un outil d'aide au développement d'applications de contrôle/commande en LabVIEW, et bien un outil d'aide au développement d'applications temps réel embarquées pour un cycle en W. Le schéma de la génération de code et l'utilisation de DTD-XML que j'ai proposé dans ce chapitre, permettent aux concepteurs de faciliter la phase de la génération de code à partir d'un modèle DARTSVIEW conformément à différentes normes temps réel ou à différents outils de validation temporelles.

A cause des différences entre des normes temps réel basées sur le langage C et ceux basées sur le langage Ada, ou encore sur d'autre langage, il est difficile d'interpréter l'étape de la génération de code en un seul VI de LabVIEW. Le VI «Generer Code Norme Langage C» sert à générer le code pour toutes les normes du langage C; et le VI «Generer Code Ada» s'occupe du code Ada.

Les normes C présentées dans ce chapitre sont la norme POSIX et la norme VxWorks. Mais pour des autres normes C, le concepteur ou l'utilisateur peuvent créer aisément le fichier XML spécifique à cette norme avec les mêmes éléments bien déclarés dans la DTD, appelée «Norme_C_DTD.dtd» (ce qui a été fait pour OSEK/VDX dans le cadre d'un stage de Master 1). Le même principe doit être appliqué à MAST et PeNSMARTS pour les outils de validation.

IV. Cas d'étude

Le problème a été énoncé dans le chapitre I, il s'agit d'un système de chauffage d'une maison. Dans le diagramme d'architecture DARTS, ce système est décomposé en tâches concurrentes, et les relations entre tâches sont définies sous la forme des outils de communication et de synchronisation. En se basant sur un langage de programmation graphique flot de données tel LabVIEW, les éléments proposés dans la palette DARTSVIEW permettent non seulement d'implémenter les diagrammes d'architecture DARTS, pour tester l'aspect fonctionnel des systèmes considérés (i.e., tests unitaires, tests d'intégration…). Et puis, à partir des modèles DARTSVIEW, le code spécifique à des normes temps réel peut être généré.

Suite à la présentation de la palette DARTSVIEW dans le chapitre II, et à la génération de code à partir d'un modèle DARTSVIEW dans le chapitre III, dans ce chapitre le système de chauffage d'une maison est utilisé comme un cas d'étude pour montrer comment la palette DARTSVIEW s'applique bien au développement des systèmes temps réel.

Afin d'être mieux exposer comment DARTSVIEW implémente des éléments d'un diagramme DARTS, ou bien comment un simulateur avec une interface virtuelle peut être réalisé, nous considérons le diagramme de l'architecture logicielle DARTS suivant:

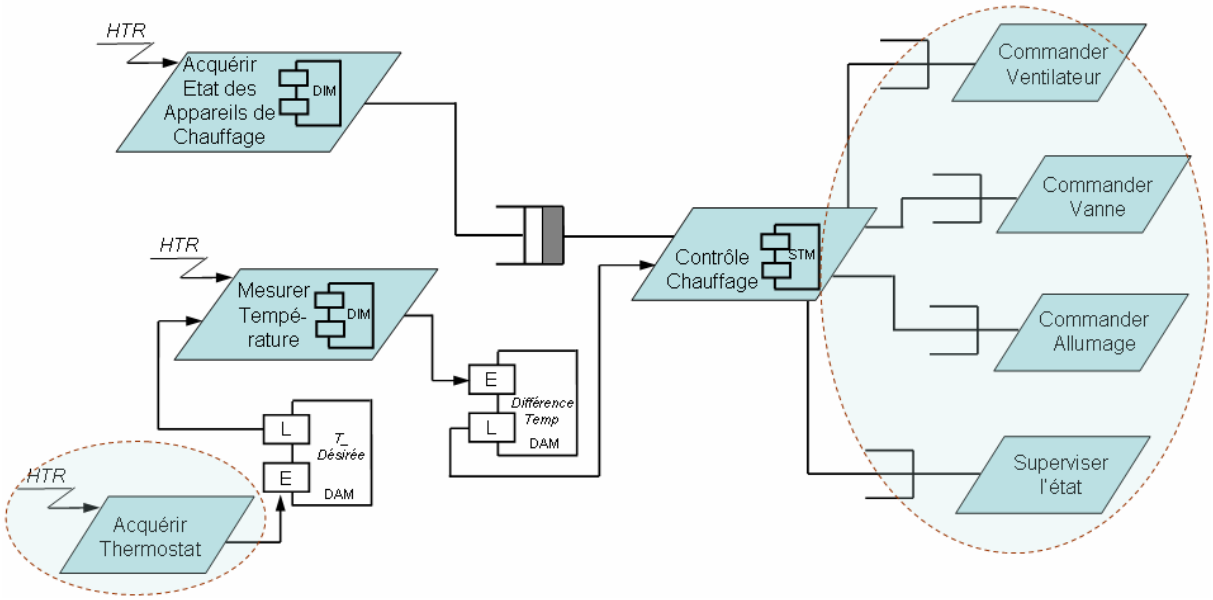


Figure IV-1: Diagramme d'architecture DARTS du système de chauffage

Par rapport au diagramme DARTS du système de chauffage présenté sur la Figure II-16, dans la Figure IV-1, les tâches de commande («Commander Ventilateur», «Commander Vanne», «Commander Allumage» et «Superviser l'état») ne sont pas intégrées dans la tâche de contrôle (le but étant dans ce chapitre de monter différents aspects, nous avons délibérément peu regroupé les tâches). De plus, une autre tâche a été ajoutée : appelée «Acquérir Thermostat», elle joue le rôle d'une tâche «mise à jour» de la valeur encapsulée du module de donnée «T_Désirée». C'est une tâche matérielle activée soit par une horloge temps réels (tâche périodique), soit par une interruption externe en provenant du procédé (tâche apériodique). Le modèle de cette tâche dans la Figure IV-1 est celui de tâche périodique (le modèle apériodique est donné aux paragraphes suivants).

IV.1.Diagramme DARTSVIEW du système de chauffage

La Figure IV-2 représente le diagramme de l'architecture DARTS du système de chauffage implémenté par des éléments dans la palette DARTSVIEW. Il est important de remarquer que,

conformément à ce que nous souhaitions, l'implémentation DARTSVIEW est très similaire au diagramme DARTS.

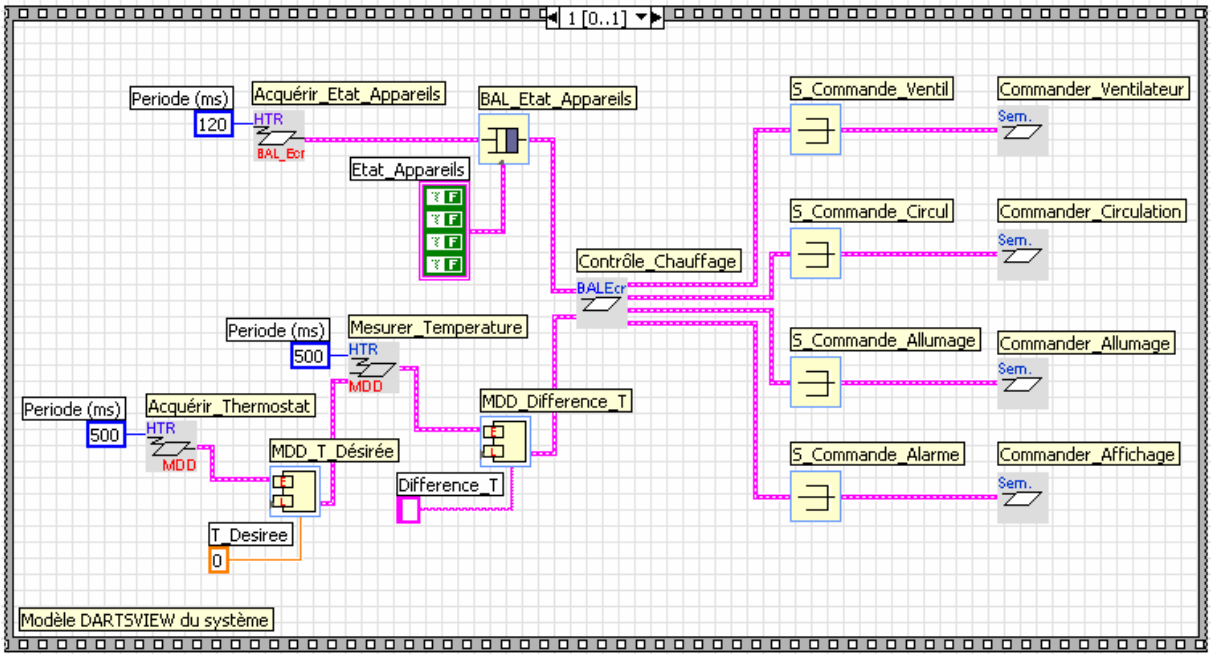


Figure IV-2: Diagramme DARTSVIEW du système de chauffage

IV.1.1. Simulateur du système de chauffage d'une maison

Dans le cadre du cycle de vie en W, nous développons initialement sur une station de travail qui en général n'a pas les dispositifs physiques nécessaires pour la connexion au procédé. De plus, lors du premier V, il est plus sûr de travailler sur un simulateur logiciel (pas de risque d'endommagement u procédé physique), et on peut changer d'échelle de temps (ici, il est évident que la vitesse du procédé thermique sera grandement accélérée), enfin on peut tester le comportement de l'application face à toutes les pannes matérielles plus simplement qu'avec le procédé physique.

Nous développons donc un simulateur du système de chauffage qui permettra de simuler le fonctionnement du procédé et de tester le fonctionnement de l'application. Il permet aussi de tester le système avec des simulations des cas de pannes (par exemple, capteurs, actionneurs, etc.) afin d'assurer que le système se comporte toujours correctement, ce qui serait difficile et coûteux à obtenir sur le procédé réel. Enfin, il permet d'accélérer le comportement du procédé pour rendre les tests faisables plus rapidement qu'avec le procédé réel.

IV.1.2. *Interface d'utilisateur virtuelle du système de chauffage*

L'interface graphique du simulateur du système de chauffage est présentée sur la figure ci-après:

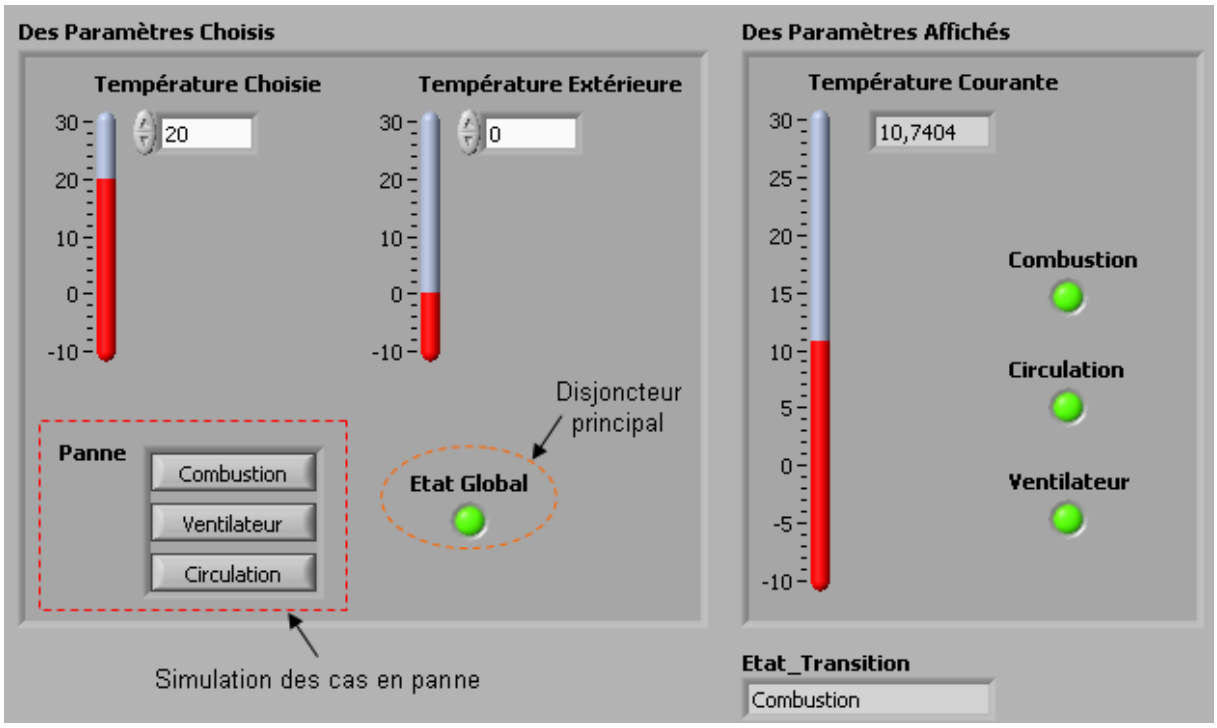


Figure IV-3: La «Face-avant» du simulateur du système de chauffage

Dans la Figure IV-3, à gauche, nous voyons les consignes, à droite les états qui seront lus par l'application via les capteurs, ainsi que l'état des actionneurs pilotés par l'application.

Comme j'en ai déjà parlé auparavant, les entrées (respectivement les sorties) d'un VI peuvent être connectées aux sorties (respectivement entrées) d'un autre VI de façon transparente grâce au mécanisme de VI non-réentrant contenant les registres à décalage. Cela facilite la communication entre le VI de simulateur du système et son modèle DARTSVIEW.

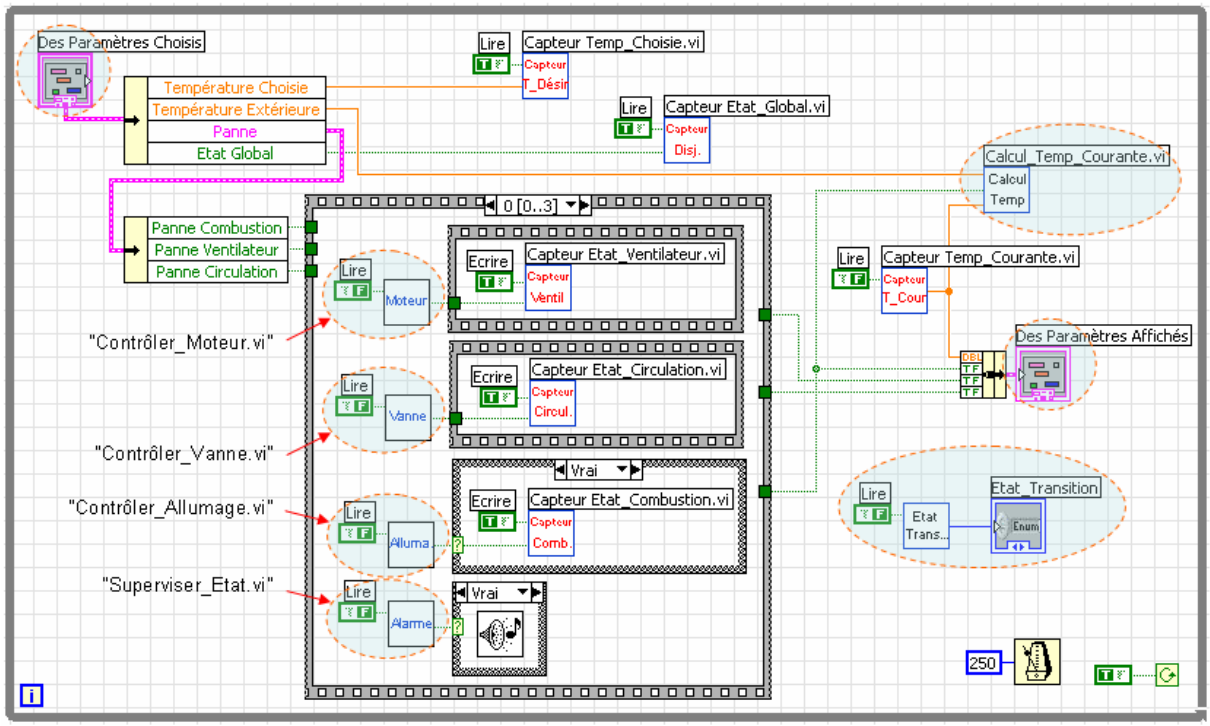


Figure IV-4: Diagramme d'un simulateur du système de chauffage

Par conséquent, dans le diagramme du simulateur du système de chauffage donné sur la Figure IV-4, les VIs non-réentrants représentent les interfaces entre ce simulateur et les tâches dans le diagramme du modèle DARTSVIEW du système.

En fonction des besoins, la simulation peut être plus ou moins exacte. Dans le cas présent nous avons choisi une représentation très grossière du procédé thermique.

IV.1.3. *Simulation des pannes du système de chauffage*

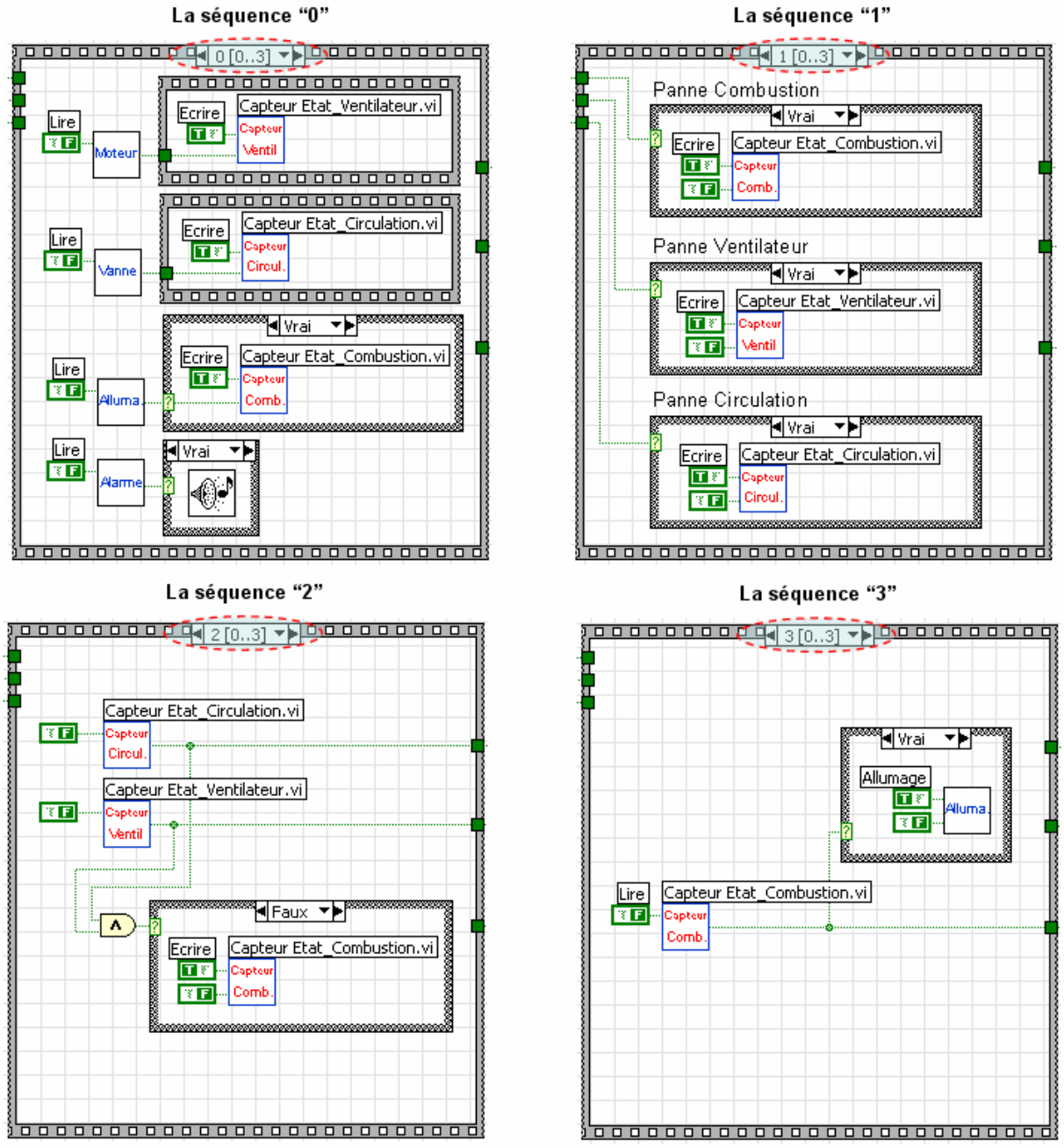


Figure IV-5: Simulation des pannes du système de chauffage dans LabVIEW

La simulation des pannes permet de tester le fonctionnement du système dans des états anormaux

Une fois les tests fonctionnels du système bien effectués sur le simulateur, il ne reste plus qu'à remplacer les accès du simulateur situés dans les tâches d'acquisition et de commande par des accès à une carte d'acquisition/commande.

IV.1.4. Modèle de tâche «Acquérir Etat des Appareils de Chauffage»

La tâche «Acquérir Etat des Appareils de Chauffage» est une tâche d'acquisition périodique, qui renvoie régulièrement des paramètres des états actuels du chauffage (il s'agit des états du disjoncteur principal, du ventilateur, de la vanne, et de la combustion) fournis par des capteurs correspondants à la tâche de contrôle «Contrôle Chauffage». Notons que dans la phase du premier V, ceux-ci sont obtenus à l'aide de VI non ré-entrants partagés avec le simulateur. Par conséquent, un module d'interface de périphérique (DIM) se trouve à l'intérieur de cette tâche de manière à ce qu'elle lors d'exécution déclenche l'acquisition des capteurs par l'intermédiaire des opérations dans l'interface de ce module. Ce DIM seul sera à changer lors du passage de la simulation au pilotage réel du procédé, son interface restant identique.

La Figure IV-6 représente le diagramme du modèle de la tâche «Acquérir Etat des Appareils de Chauffage», appelé «Acquérir_Etats_Appareils», implémenté par l'élément «HTR – (out) BAL_Ecr» dans la sous-palette «Tâche Matérielle» de la palette DARTSVIEW. A chaque itération, les états des appareils du chauffage générés par les VIs de simulateur des capteurs (les séquences «0» - «3») sont regroupés en un message pour l'envoyer dans la BAL_Ecr spécifique aux états des appareils (la séquence «4»). Un modèle de chien de garde est utilisé dans la dernière séquence (la séquence «5») afin d'assurer le respect de l'échéance de la tâche, une alarme survient si l'échéance est dépassée (je suppose que toutes les tâches périodiques dans le modèle DARTSVIEW du système de chauffage sont à échéance sur requête).

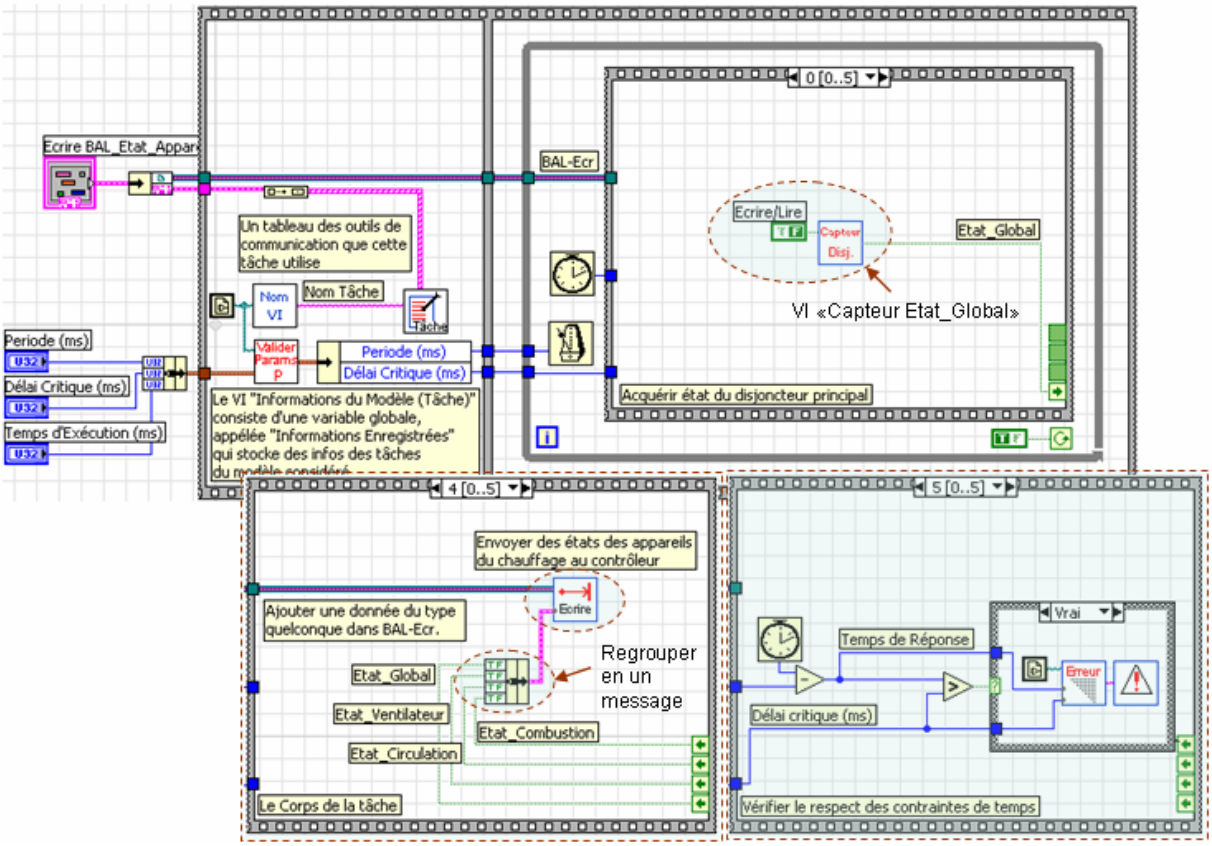


Figure IV-6: Diagramme du modèle de la tâche «Acquérir Etat des Appareils de Chauffage»

IV.1.5. Modèle de la tâche «Acquérir Thermostat»

La tâche «Acquérir Thermostat» dans ce cas est une tâche périodique activée par horloge temps réel. Lors de l'activation, s'il y a un changement de la température désirée, elle effectue une mise à jour du module de données pour cette valeur avec la valeur de la température désirée la plus récente, sinon elle ne fait rien.

Le diagramme du modèle de cette tâche est présenté sur la Figure IV-7. Il s'agit de l'élément «HTR – (out) MDD» de la sous-palette «Tâche Matérielle» de la palette DARTSVIEW. De même que le modèle de la tâche «Acquérir Etat des Appareils de Chauffage», un VI, appelé «Capteur Temp_Choisie» est utilisé pour simuler des changements de la température désirée (sauf que le type de données soit flottant au lieu de booléen). Grâce à des registres à décalage, la mise à jour de MDD de la température désirée n'est réalisée que lorsqu'un changement de la valeur de cette température est détecté (la séquence «0»).

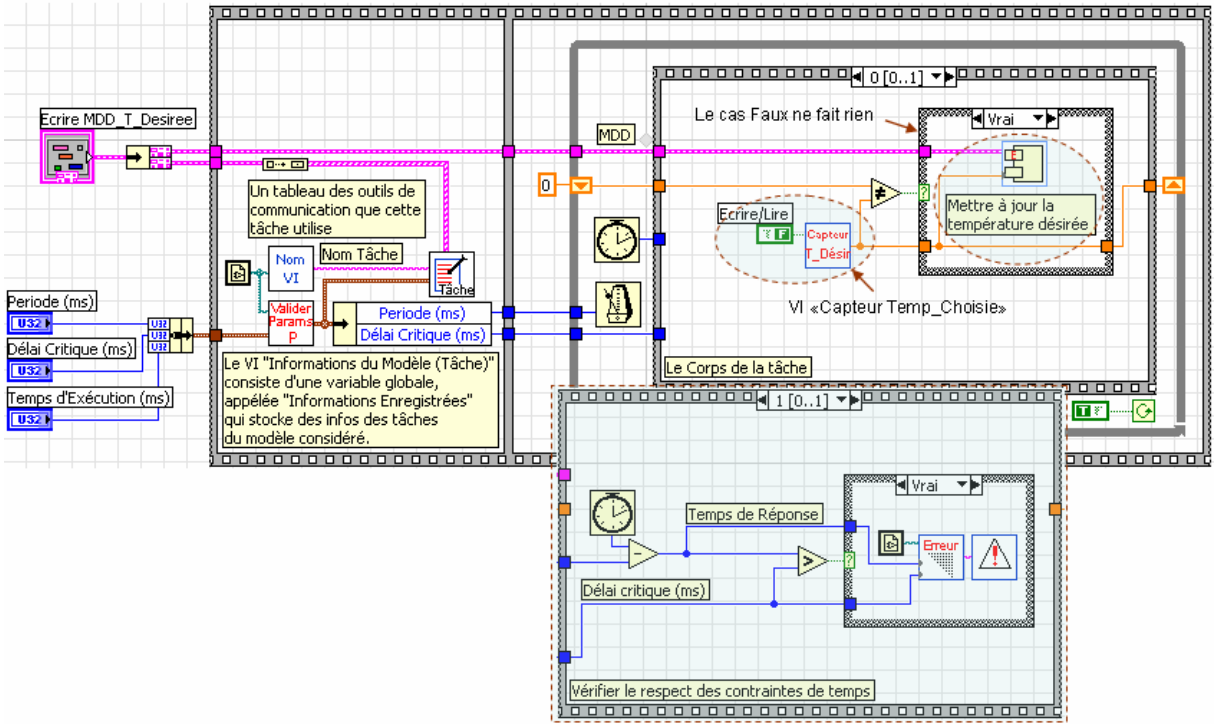


Figure IV-7: Diagramme du modèle de la tâche «Acquérir Thermostat»

En pratique, la tâche «Acquérir Thermostat» est souvent une tâche apériodique activée par une interruption en provenance du procédé (par exemple, le résident de la maison,…). Alors, un changement de la température désirée entraîne un déclenchement pour activer cette tâche. Et la mise à jour est effectuée à chaque exécution de la tâche «Acquérir Thermostat» sans avoir besoin de vérifier si une modification de la température désirée a lieu. Par conséquent, le modèle d'implémentation de cette tâche est généré par l'élément «Evénement – (out) MDD» avec un simulateur de déclenchement (j'en parlerai en détail aux paragraphes suivants de ce chapitre).

IV.1.6. Modèle de la tâche «Mesurer Température»

La tâche «Mesurer Température» est une tâche périodique déclenchée par horloge temps réel, qui calcule la différence entre la température courante (`T_Courante`) à l'intérieur de la maison et la température désirée par le résident (`T_Desiree`). Cette tâche lors d'exécution, acquiert d'abord le capteur pour la température courante (simulé par le VI «Capteur Temp_Courante»), et puis elle appelle l'opération «Lire» de MDD «T_Désirée» pour acquérir la valeur la plus récente stockée dans ce module. Ensuite, en fonction de la différence de ces deux valeurs, elle produit un message à la tâche de contrôle au moyen d'un autre module de données (MDD), nommé «Différence_Temp» (donc, «Trop_Haute», si `T_Courante - T_Desiree > 2`, et «Trop_Basse» si `T_Courante - T_Desiree < -2`).

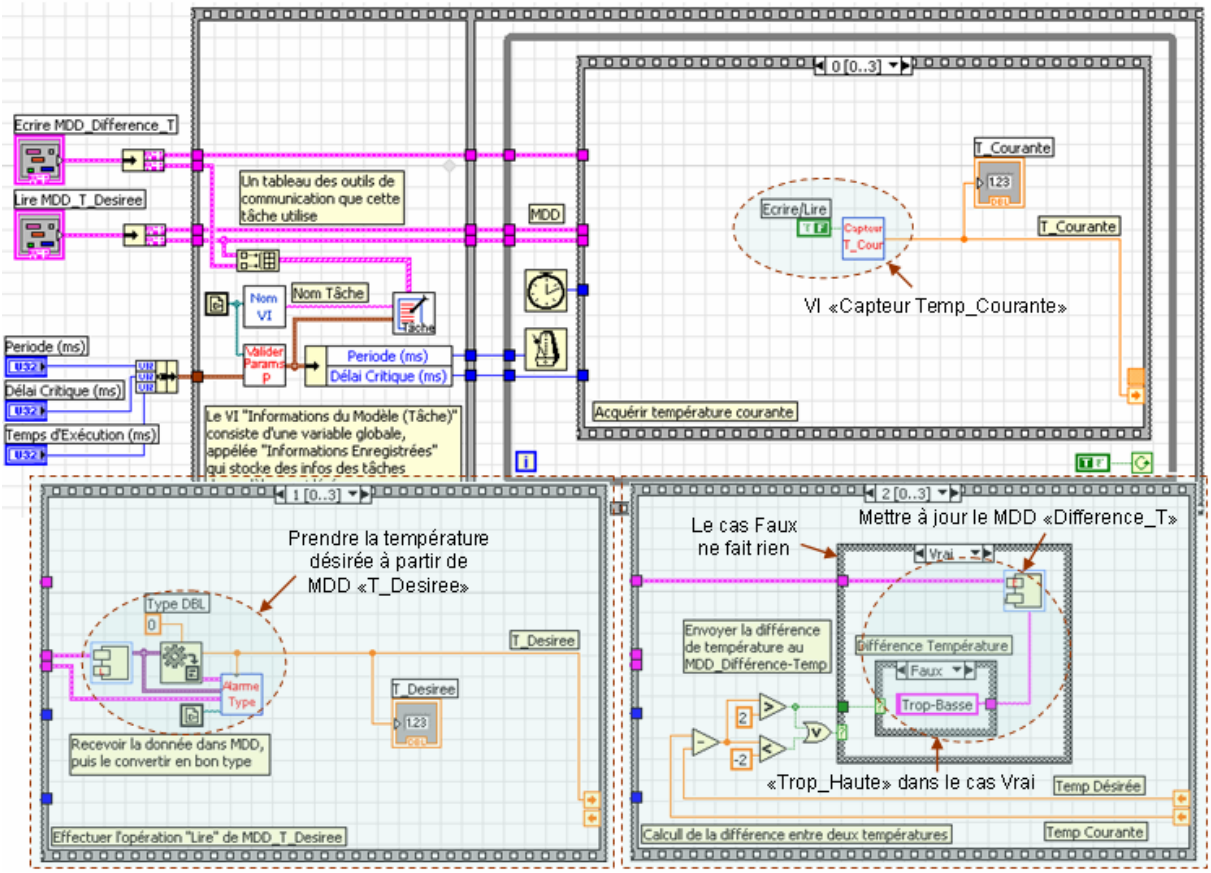


Figure IV-8: Diagramme du modèle de la tâche «Mesurer Température»

Le diagramme du modèle de la tâche «Mesurer Température» est donné sur la Figure IV-8. La séquence «0» implémente l'acquisition de la température courante simulée par le VI «Capteur Temp_Courante»; et puis dans la séquence «1» la température désirée est récupérée à partir de MDD «T_Desiree» en appelant l'opération «Lire»; la mise à jour de MDD «Difference_T» avec la nouvelle valeur de «Difference_T» (soir «Trop_Haute», soit «Trop_Basse») est effectuée si la condition de deux températures est satisfaite (le cas «Vrai» dans la séquence «2»); et comme pour les tâches présentées précédemment, un modèle de chien de garde est utilisé dans la dernière séquence afin d'assurer le respect d'échéance de la tâche. Puisque la tâche «Mesurer Température» interagit avec deux MDDs à la fois, le modèle DARTSVIEW de cette tâche est une extension de l'élément «HTR – (in) MDD» utilisant ajouter un autre MDD en mode «Ecrire» pour la différence de température, alors que le MDD déjà existant est en mode «Lire», et correspond à la lecture de la température désirée.

IV.1.7. Modèle de la tâche «Contrôle Chauffage»

IV.1.7.a. Diagramme du modèle de la tâche «Contrôle Chauffage»

La tâche «Contrôle Chauffage» joue le rôle de chef d'orchestre dans le système de chauffage. Elle est une tâche logicielle déclenchée par la tâche «Acquérir Etat des Appareils de Chauffage» sous la forme de la communication synchrone faiblement couplée avec écrasement. Le diagramme du modèle de la tâche «Contrôle Chauffage» est donné sur la Figure IV-9.

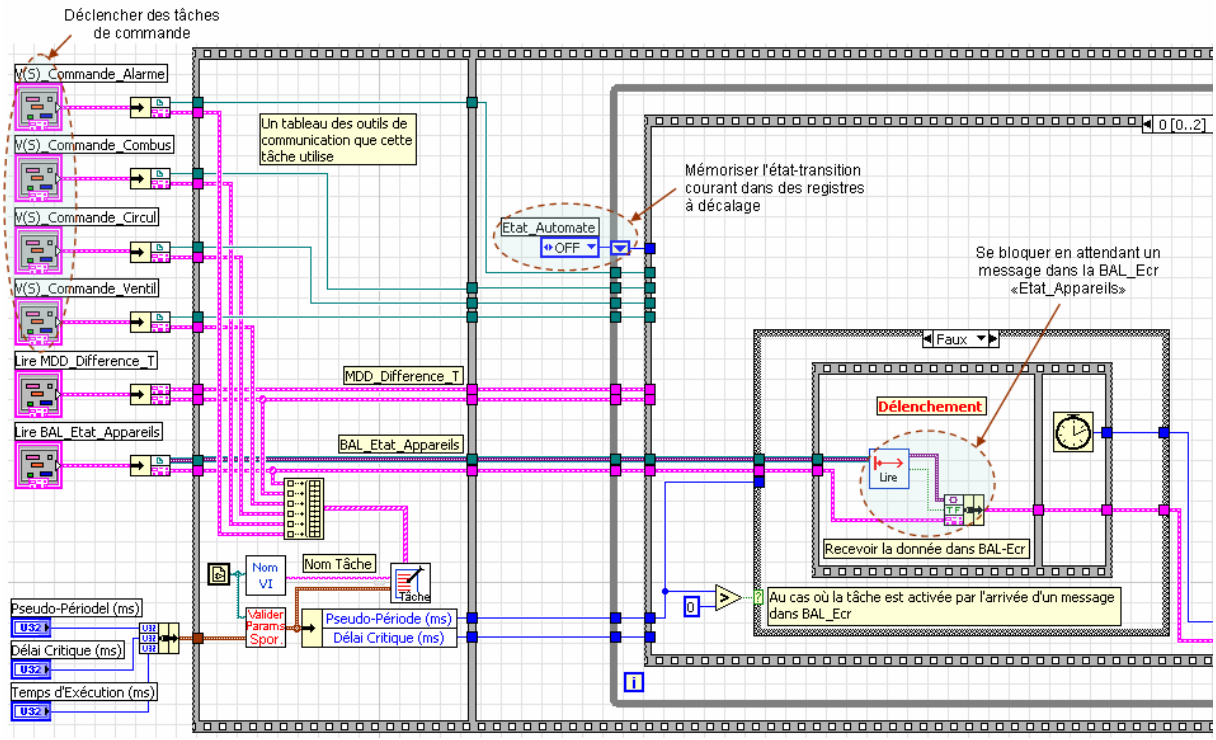


Figure IV-9: Diagramme du modèle de la tâche «Contrôle Chauffage»

Cette tâche est en attente de message provenant de «BAL_Etat_Appareils». Lors de l'exécution, en fonction des données reçues (les états des appareils du chauffage dans BAL_Ecr, la différence de température dans le MDD «Difference_T»), et de l'état transition actuel mémorisé dans les registres à décalage, elle exécute le module d'état-transition (STM) du système de chauffage. Ainsi, si un changement d'état a lieu, elle envoie des signaux afin de réveiller des tâches de commande appropriées. L'automate de «Contrôle Chauffage» sera présenté en détail dans la section suivante.

Comme il n'y a pas de passage d'information entre la tâche «Contrôle Chauffage» et les tâches de commande, la synchronisation est implémentée par sémaphores privés. Le modèle de la tâche «Contrôle Chauffage» est une extension de l'élément «Tache logicielle – (in) BAL_Ecr» de la sous-palette «Tâche Logicielle» de la palette DARTSVIEW en ajoutant un MDD en mode «Lire» pour la différence des températures, et quatre sémaphores en mode «Vendre (S)» pour la synchronisation avec les quatre tâches de commande correspondantes.

IV.1.7.b. L'automate de la tâche «Contrôle Chauffage»

Le contrôleur «Contrôle Chauffage» est le cœur du système de chauffage. Il joue le rôle indispensable dans le fonctionnement de ce système. Dans la Figure IV-1, le contrôleur se situe à la position centrale, et devient le «pont de communication» entre les tâches d'acquisition et les tâches de commande.

L'automate de la tâche «Contrôle Chauffage» est une machine à états finis comportant onze états, appelés «OFF», «Allumage_1», «Allumage_2», «Allumage_3», «Allumage_4», «Allumage_5», «Combustion», «Panne_Circulation», «Panne_Ventilateur», «Panne_ Combustion» et «Panne_Circul_Ventil» respectivement, comme celui présenté sur la Figure I-35. A un état-transition donné, selon des paramètres d'entrée, si une transition a lieu, un changement de l'état courant vers un nouvel état parmi ces onze états est effectué, et a pour effet éventuel de déclencher des tâches de commande.

Rappelons que dans LabVIEW l'automate est implémenté par une imbrication de structures conditionnelles. La Figure IV-10 représente le diagramme du VI d'implémentation de l'automate

de la tâche «Contrôle Chauffage», qui correspond au cas de la transition de l'état courant «Combustion» à l'état «Panne_Ventilateur». Alors, dans ce cas là, le contrôleur doit envoyer trois événements vers trois tâches de commande: «Commander Ventilateur», «Commander Circulation» et «Commander Affichage» afin d'arrêter le ventilateur, de fermer la vanne, et de déclencher une alarme respectivement. La mise à jour de l'état-transition courant avec l'état «Panne_Ventilateur» est bien-sûr réalisée (il s'agit de quatre séquence numérotées de «0» à «3» de la structure de séquence qui se trouve dans le cas «Faux» de la structure de condition la plus interne dans le modèle de l'automate de chauffage présenté sur la Figure IV-10).

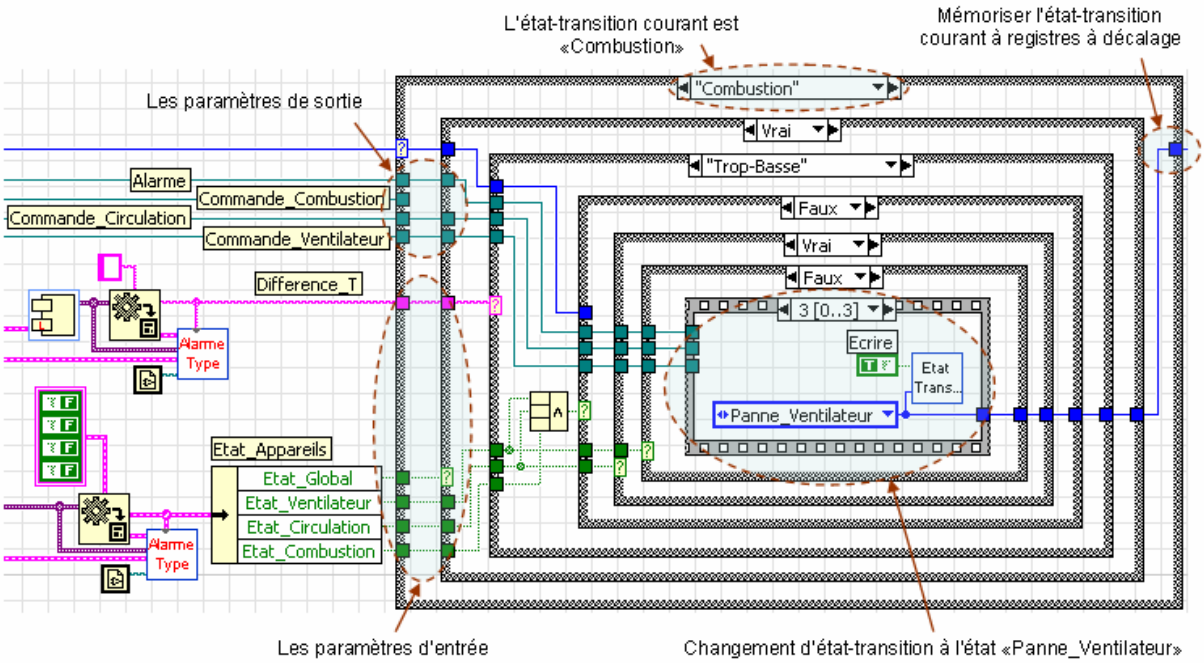


Figure IV-10: Implémentation d'automate de la tâche «Contrôle Chauffage»

Afin de terminer cette section, je parle de modèle des horloges internes qui sont utilisées dans certains états-transitions tels les états «Allumage_3», «Allumage_4» et «Allumage_5». Le rôle de ces horloges est de générer des signaux pour assurer le bon fonctionnement du chauffage si dans une certaine durée de temps, une action ou une condition n'est pas encore achevée ou satisfaite. Par exemple, dans l'état «Allumage_5», si après 5000 millisecondes l'état de la combustion n'est pas encore dans l'état «ON», l'horloge interne génère un signal pour déclencher une alarme et/ou remplacer l'exécution en cours par une ou des action(s) prédéfinie(s) correspondantes au cas en panne, «Panne_Combustion». Le diagramme du VI permettant d'implémenter des horloges internes est donné sur la figure ci-après:

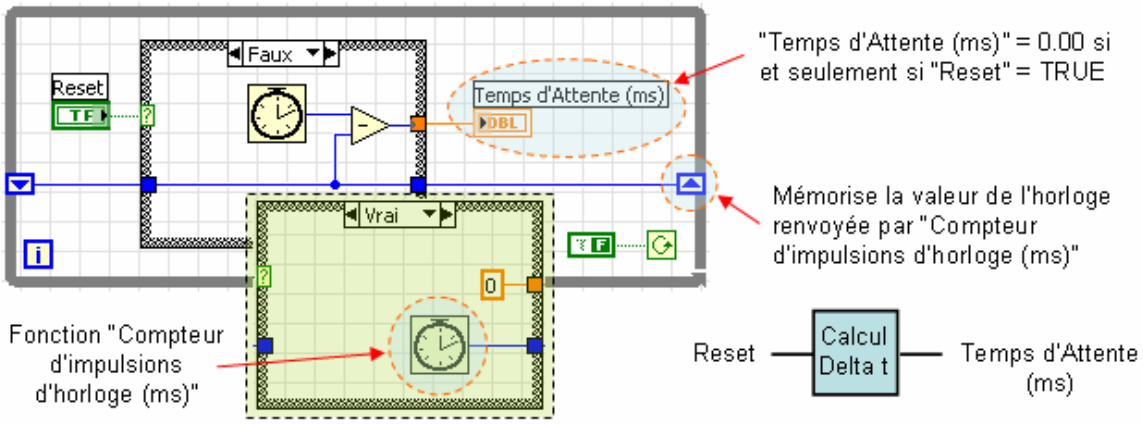


Figure IV-11: Diagramme du modèle des horloges internes dans LabVIEW

IV.1.8. Modèle des tâches de commande

Les tâches de commande («Commander Ventilateur», «Commander Circulation», «Commander Allumage», et «Commander Affichage») sont des tâches logicielles activées par la tâche «Contrôle Chauffage» à l'aide de sémaphores privés.

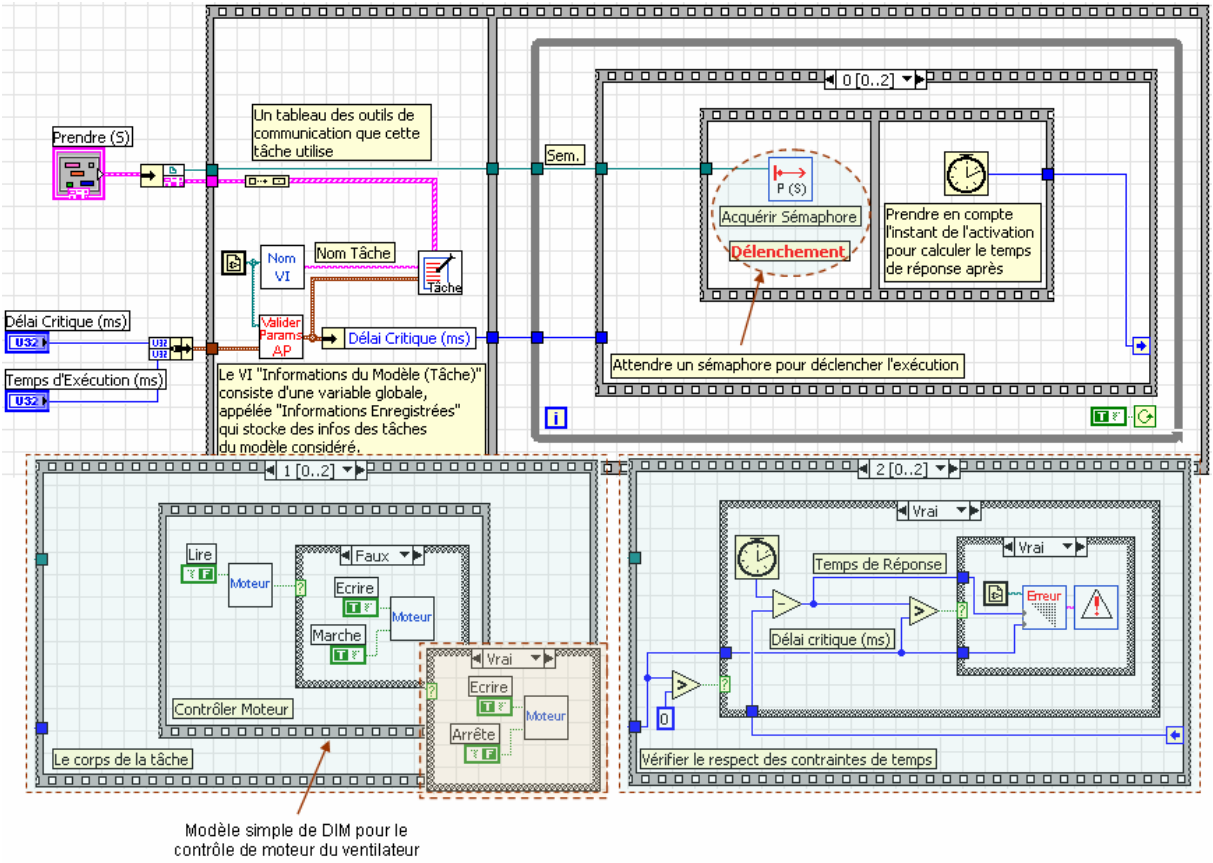


Figure IV-12: Diagramme du modèle de la tâche «Commander Ventilateur»

Dans le modèle DARTSVIEW du système de chauffage, les tâches de commande sont implémentées par l'élément «Tâche logicielle – (in) Semaphore» de la sous-palette «Tâche Logicielle» de la palette DARTSVIEW. Le diagramme du modèle de la tâche «Commander Ventilateur» présenté sur la Figure IV-12 en est un exemple.

Un modèle simple de DIM est inséré dans le corps de chaque tâche de commande pour simuler le contrôle de l'appareil de chauffage qu'elle prend en charge (voir Figure IV-12, la séquence «1»). Notons que dans le 1er V du cycle de vie, ces DIM correspondent à une communication via VI non-réentrant avec le simulateur, et seront remplacés par des accès physiques aux actionneurs dans le 2nd V.

IV.2.Génération de code du système de chauffage d'une maison

IV.2.1. Fichier XML de description du système de chauffage

Le fichier XML de description du modèle DARTSVIEW du système de chauffage est généré par le VI «Generer Fichier XML» dans la sous-palette «Générer Code» de la palette

Dans le fichier XML obtenu, on trouve les outils de communication et de synchronisation puis les tâches et les outils de communication et de synchronisation. Ce fichier ne contient que des informations du modèle DARTSVIEW du système de chauffage, qui sont indépendantes de l'implémentation. Il sert de base indispensable dans la phase de la génération de code spécifique à partir du modèle DARTSVIEW vers un langage de programmation, une norme temps réel, ou un outil de validation temporelle.

IV.2.2. Code des normes temps réel du langage C du système de chauffage

En associant le fichier XML de description du modèle avec le fichier XML spécifique à la norme POSIX, «POSIX.xml», présenté au chapitre III, on génère le code POSIX du système de chauffage.

Notons que pour le cas ou l'on souhaite générer le code pour VxWorks, sachant qu'il n'est pas possible d'implémenter des BAL à écrasement, le concepteur devra remplacer les BAL à écrasements par des BAL classiques.

IV.2.3. Code de la norme Ada du système de chauffage

En associant le fichier XML de description du modèle «Modèle DARTSVIEW du chauffage.xml» avec le fichier XML spécifique à la norme Ada, «Ada.xml», présenté au chapitre III, le code Ada du système de chauffage généré par le VI «Generer Code Ada» de la sous-palette «Générer Code» de la palette DARTSVIEW est donné dans 14 fichiers (1 procédure principale, et des paquetages, dont la plupart sont des instanciations de paquetages génériques sur des types).

IV.3.DARTSVIEW vs les modifications de modèle DARTS du système de chauffage

Le développement d'un système n'est pas réalisé de manière séquentielle. Ainsi, il est souvent nécessaire de modifier la conception après avoir mené des tests fonctionnels. Ainsi, supposons dans le diagramme d'architecture du système de chauffage (voir Figure IV-1), que la tâche «Acquérir Thermostat» soit une tâche matérielle activée par une interruption avec une nouvelle valeur de température désirée pour effectuer une mise à jour du module de donnée «T_Desiree». Cela permet de diminuer la charge du processeur requise pour cette tâche, et surtout lors d'exécution, la mise à jour du MDD «T_Desiree» est effectuée immédiatement. Ainsi, dans la spécification DARTS, on remplace le modèle de la tâche «Acquérir Thermostat» périodique activée par horloge temps réel par un modèle de la tâche «Acquérir Thermostat» activée par interruption externe provenant du procédé (voir Figure IV-13).

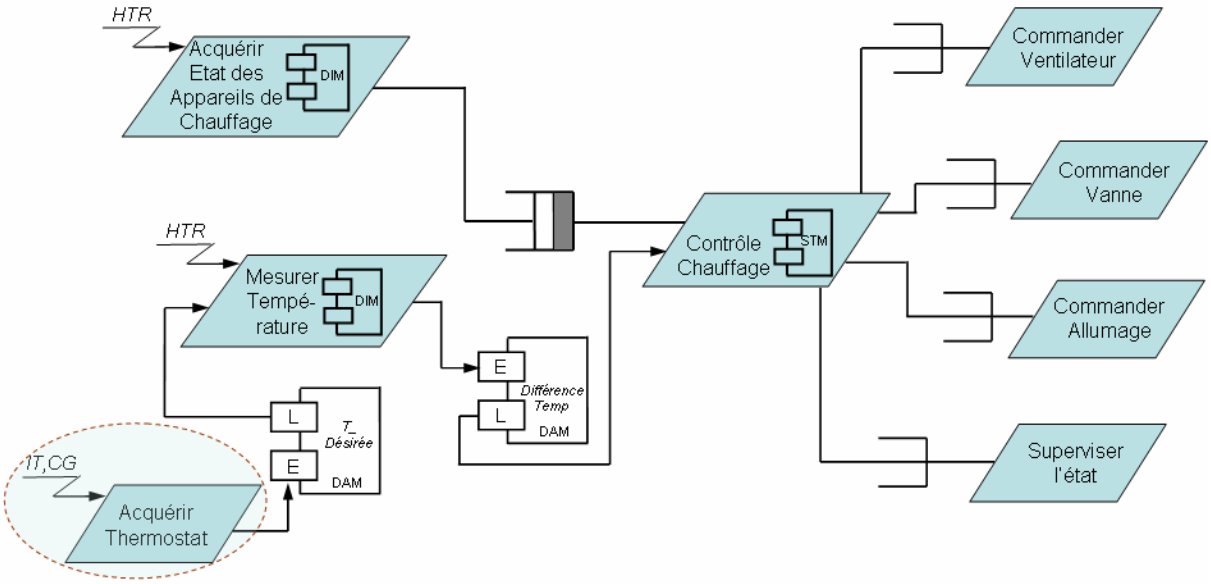


Figure IV-13: Modification du diagramme d'architecture DARTS du système de chauffage

Cette modification entraîne des changements nécessaires au VI du modèle DARTSVIEW du système, au VI du simulateur, et à l'interaction entre ces deux VIs. En mode simulation (premier V du cycle en W), ceci est réalisé en simulant l'interruption matérielle par une file d'attente entre simulateur et tâche d'acquisition du thermostat (voir Figure IV-14).

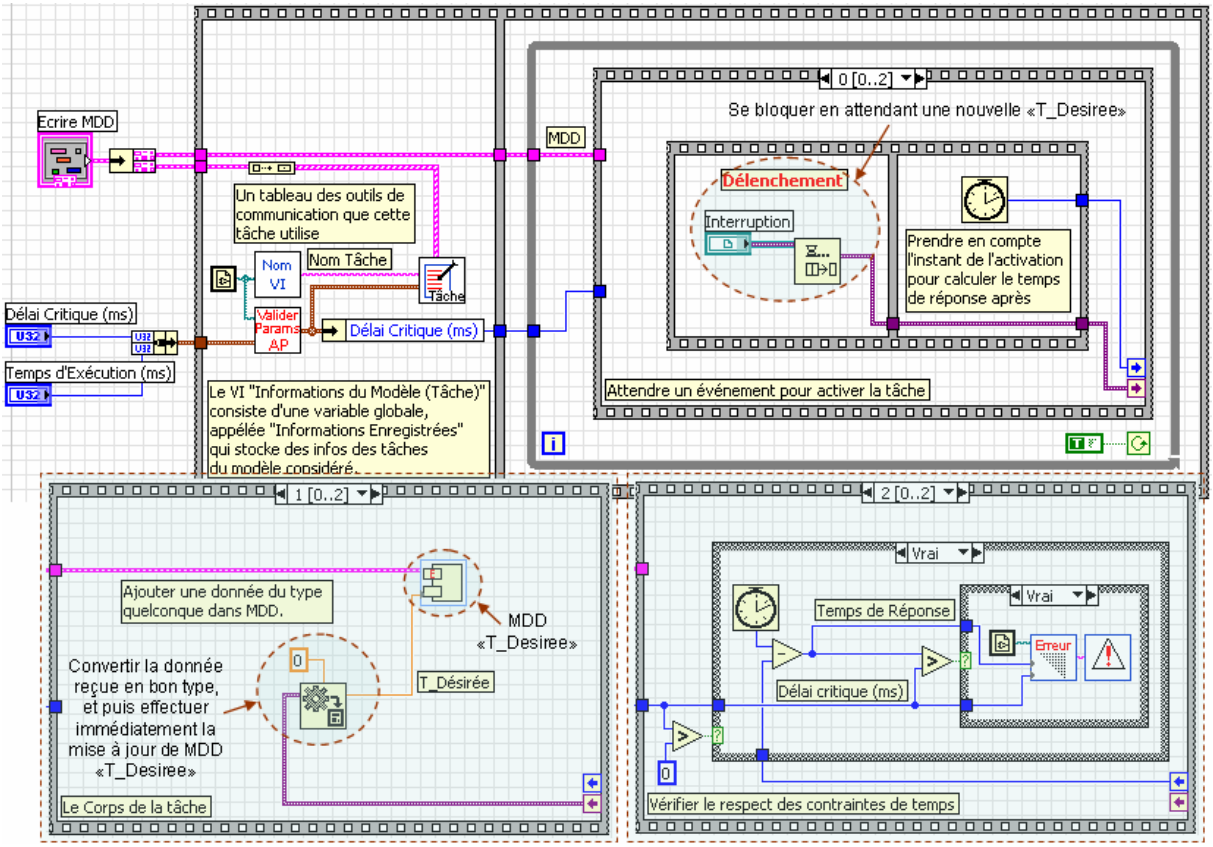


Figure IV-14: Diagramme du modèle de la tâche «Acquérir Thermostat» apériodique

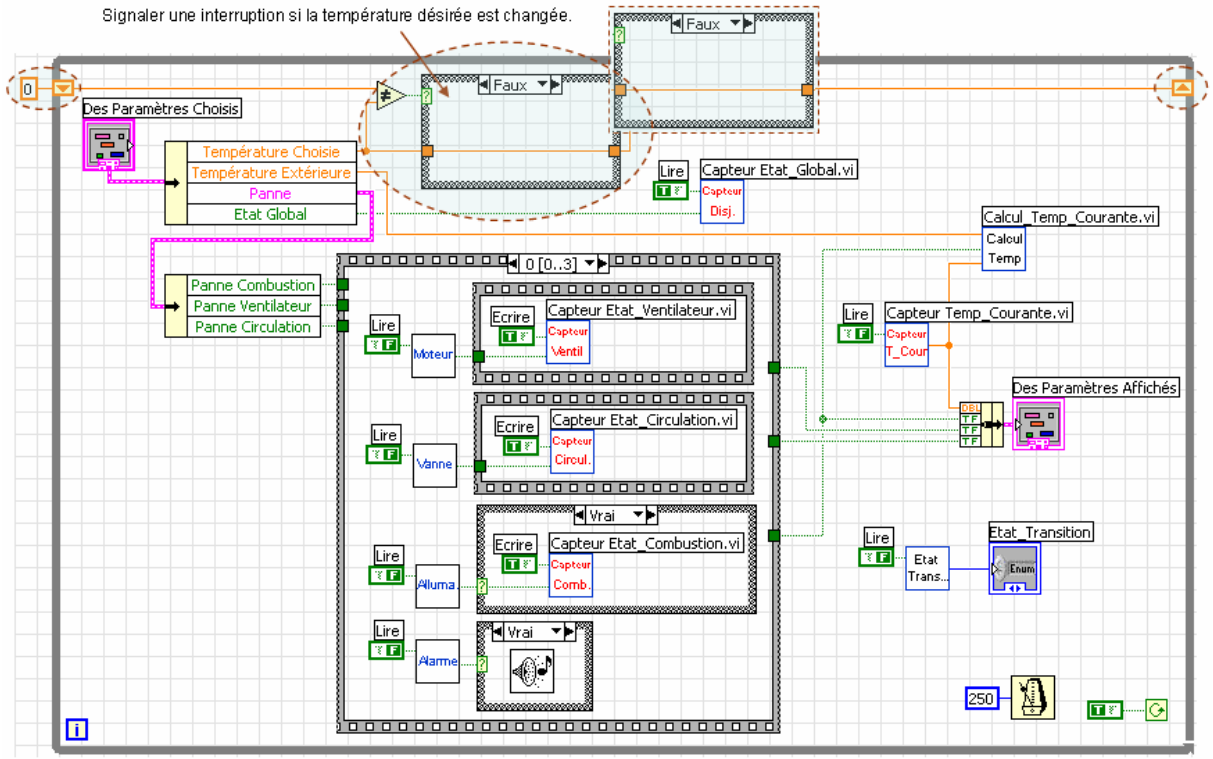


Figure IV-15: Diagramme du simulateur en cas de tâche «Acquérir Thermostat» apériodique

Nous voyons donc que dans le cas d'une tâche de scrutation, la communication entre simulateur et système est asynchrone :

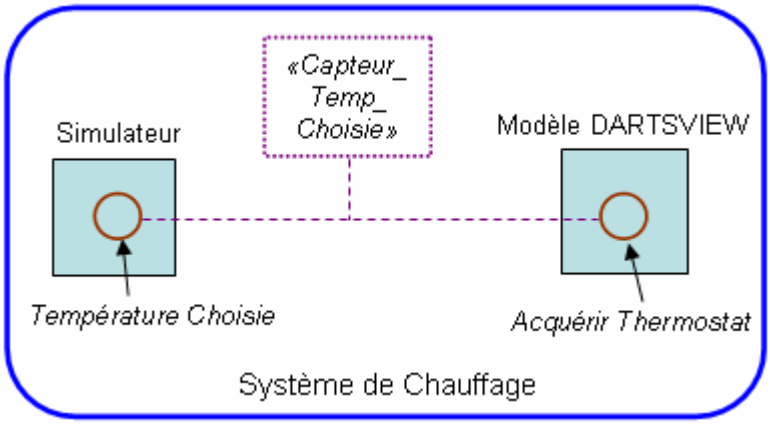


Figure IV-16: Interaction implicite entre le simulateur et le modèle DARTSVIEW en cas de tâche «Acquérir Thermostat» périodique

Alors qu'elle est synchrone dans le cas d'une tâche déclenchée par interruption matérielle :

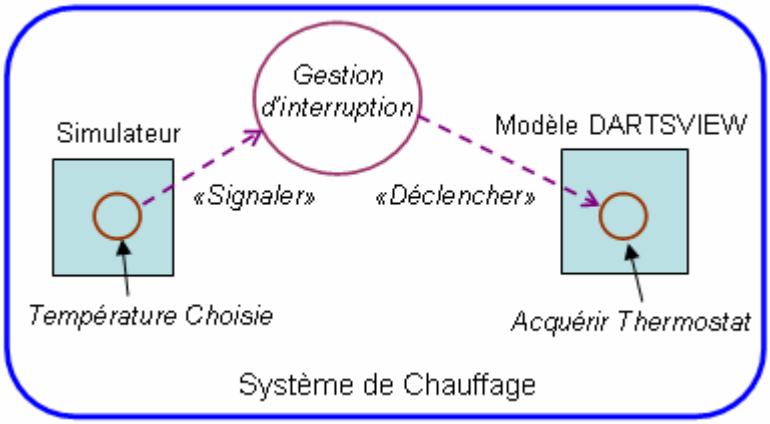


Figure IV-17: Interaction explicite entre le simulateur et le modèle DARTSVIEW en cas de tâche «Acquérir Thermostat» apériodique

Le diagramme DARTSVIEW du système de chauffage au cas où la tâche «Acquérir Thermostat» est activée par interruption en provenance du simulateur est donné ci-après:

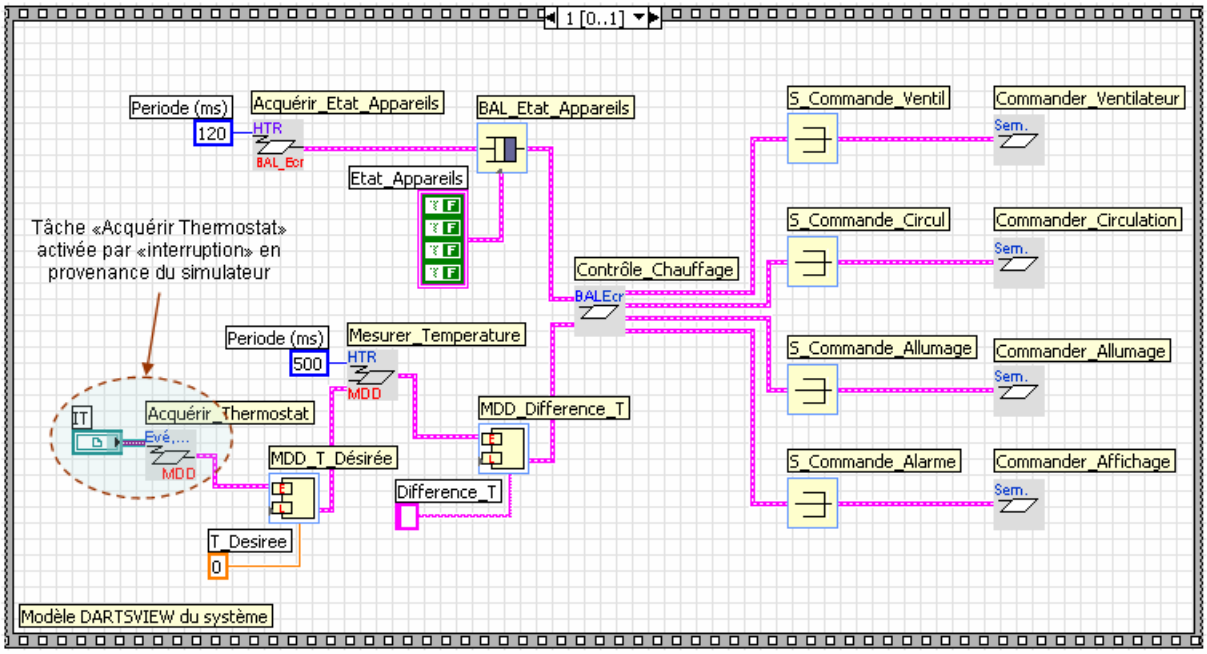


Figure IV-18: Diagramme DARTSVIEW du système de chauffage en cas de tâche «Acquérir Thermostat» apériodique

Pour la génération de code, la modification de la tâche «Acquérir Thermostat» n'entraîne que des changements correspondants dans la partie de code de cette tâche. Ainsi, le fichier XML de description du modèle DARTSVIEW n'est modifié que des informations concernant la tâche «Acquérir Thermostat»:

```
-------------------------------------------------------------------------
-- Fichier "Modèle DARTSVIEW du chauffage (modifié).xml"
...
<Tache nomTache="Acquérir_Thermostat">
<MDDU iden="Acquérir_Thermostat" nomMDD="MDD_T_Desiree"
typeDonnee="T_Desiree" modeMDD="Ecrire" />
</Tache>
...
-------------------------------------------------------------------------
```

De même pour le code des normes du langage C du système de chauffage généré par le VI «Generer Code Norme Langage C»:

```
-------------------------------------------------------------------------
-- Code Norme POSIX du système de chauffage (modifié)
...
void Acquérir_Thermostat() {
/* Tâche apériodique */
/* Variable locale de la tâche: Acquérir_Thermostat */
T_Desiree var_MDD_T_Desiree;
/* end variable locale: Acquérir_Thermostat */

while (1) { /* Corps de la tâche: Acquérir_Thermostat */
/* Attendre de déclenchement pour activer la tâche: Acquérir_Thermostat
*/


/* Partie d'utilisateurs de la tâche: Acquérir_Thermostat */
/* ----------------code----------------- */
/* end partie d'utilisateur: Acquérir_Thermostat */
```

```
/* L'instruction d'envoyer un message de la tâche: Acquérir_Thermostat
*/
pthread_mutex_lock(&s_MDD_T_Desiree);
MDD_T_Desiree=var_MDD_T_Desiree;
pthread_mutex_unlock(&s_MDD_T_Desiree);
}
}

...
-----------------------------------------------------------------------
```

Et code de la norme Ada généré par le VI «Generer Code Ada»:

```
-----------------------------------------------------------------------
-- Code Norme Ada du système de chauffage (modifié) - «Modele.adb»

...
-- Description de la tâche: Acquérir_Thermostat, une tâche
-- apériodique
task body Acquérir_Thermostat is
-- Variables locales de la tâche: Acquérir_Thermostat
var_MDD_T_Desiree: t_MDD_T_Desiree;
-- end variables locales: Acquérir_Thermostat

-- Corps de la tâche: Acquérir_Thermostat
begin
loop
-- Attendre un déclenchement pour activer la tâche:
-- Acquérir_Thermostat


-- Partie d'utilisateur de la tâche: Acquérir_Thermostat
-- .................code..................
-- end partie d'utilisateur: Acquérir_Thermostat

-- L'instruction d'envoyer un message de la tâche:
-- Acquérir_Thermostat
MDD_T_Desiree.Envoyer(var_MDD_T_Desiree);

end loop;
end Acquérir_Thermostat; -- end description: Acquérir_Thermostat

...
-----------------------------------------------------------------------
```

Il est important de remarquer que le code d'utilisateur et du matériel dans les fichiers de code du modèle DARTSVIEW précédent peut être récupéré et inséré dans ceux du modèle DARTSVIEW modifié lors de la génération de code à partir du fichier XML de description de ce modèle.

V. Conclusion

Les systèmes temps réel sont des systèmes dont le respect des contraintes temporelles est aussi important que l'exactitude du résultat. Le développement d'un système temps réel se concentre sur le développement de deux composants: matériel et logiciel. Pour les systèmes embarqués ou temps réel, par rapport à l'approche traditionnelle, l'approche en co-design permet de rendre efficace le «time-to-market» d'un développement, qui s'adapte bien à des changements rapides de la technologie, ou bien que des demandes d'utilisateurs de nos jours. Cependant, il nécessite a priori des expériences du concepteur afin de diminuer le surcoût. Au contraire, avec l'approche traditionnelle, le matériel et le logiciel peuvent être développés en parallèle. Cela permet de faciliter considérablement les modifications de chacun de ces composants de manière à ce qu'une modification de l'un n'influence pas le développement de l'autre, et vice versa jusqu'à l'intégration des deux. Dans ce cas, on requiert un cycle de développement de logiciel, qui permet d'une part d'assurer des spécifications sans prise en compte de matériel, d'autre part de rendre efficace la validation avec la prise en compte de matériel lors de l'intégration entre le logiciel et le matériel dans le développement d'un système considéré. Et le cycle de développement d'un logiciel en W présenté dans ce manuscrit répond à cet objectif.

Le choix des méthodes de développement est très important. Cela dépend non seulement des spécifications du système considéré, mais aussi des expériences des concepteurs car dans certains cas l'utilisation conjointe des différentes méthodes de développement est indispensable. La méthode DARTS de Gomaa est utilisée dans ce manuscrit en tant qu'une méthode de base n'est pas pour répondre la question "Quelle est la meilleure méthode de développement d'un système", mais est pour montrer la démarche classique de développement des systèmes temps réel avec le cycle de vie en W. Actuellement, par rapport à l'approche orientée-objet, ou bien l'approche incrémentielle et itérative, etc., l'approche fonctionnelle, représentée typiquement par la méthode DARTS, s'adapte bien au développement des systèmes de petite taille ou des systèmes centralisés. Avec les résultats obtenus dans ce manuscrit, je pense que LabVIEW ou la programmation graphique flots de données peut s'appliquer au développement des systèmes temps réel fondé sur une méthode de base orientée-objet telle CODARTS, HRT-HOOD, UML, etc.

La palette DARTSVIEW est une bibliothèque des éléments DARTS implémentés par LabVIEW. Grâce aux propriétés avancées de la programmation graphique de LabVIEW, DARTSVIEW permet de faciliter considérablement le premier V du cycle de vie en W avec des tests et des simulations de l'aspect fonctionnel d'un système. L'idée de génération de code à partir d'un modèle DARTSVIEW a pour le but de rendre efficace la validation dans le deuxième V du cycle W en offrant aux concepteurs la possibilité de génération de code conformément soit à un langage cible basé sur une norme temps réel, soit à un outil de validation tiers par l'intermédiaire de fichiers XML. Ainsi, la séparation de l'aspect fonctionnel et de l'aspect temporel dans l'approche traditionnelle de développement d'un logiciel est utile dans le développement des systèmes temps réel, où leurs spécifications sont fondées principalement sur des fonctionnements. Et DARTSVIEW devient une plate-forme de génération de code, où les concepteurs peuvent choisir leur outil de validation pour générer leur propre code.

Le code généré à partir de la palette DARTSVIEW présenté dans ce mémoire doit être conforme à la syntaxe soit d'une norme temps réel, soit d'un outil de validation. Quelques exemples sont donnés dans ce manuscrit, mais un concepteur devrait facilement pouvoir ajouter des formats XML spécifiques pour générer le code correspondant à ses besoins.

La palette DARTSVIEW est créée initialement en 2002 sous la version 6.i de LabVIEW grâce à ses VIs polymorphes et ses fonctionnalités avancées. Puis, l'année 2005 avec le type Variant et les VIs Express de LabVIEW 7.1, la palette DARTSVIEW est modifiée considérablement, et elle devient un Toolkit d'aide au développement des systèmes temps réel comme celle présentée dans ce manuscrit. Depuis la fin de l'année 2006 jusqu'à présent, les versions 8.x de LabVIEW sont

délivrées avec de nombreuses nouveautés vers la conception des systèmes distribués telles que la création des machines d'états graphiques pour les tâches événementielles avancées avec le module LabVIEW Statechart; le module LabVIEW pour le temps réel (NI LabVIEW 8.5 Real-Time) avec la mise au point améliorée grâce au Toolkit LabVIEW Real-Time Execution Trace, le support des trames cadencées de manière individuelle avec des échéances en microsecondes dans les structures cadencées LabVIEW, la réplique des systèmes temps réel pour l'évolutivité, la validation ou la conversion d'un PC de bureau en cible temps réel, etc. Toutes ces nouveautés permettent non seulement de perfectionner la palette DARTSVIEW, mais également d'élargir les champs d'application de LabVIEW dans le développement des systèmes temps réel au sens des approches orientées-objets (par exemple CODARTS, HRT-HOOD, UML,…), au lieu de l'approche fonctionnelle DARTS.

Bibliographie

[ACLS04] L. Apvrille, J.P. Courtiat, C. Lohr, P. de Saqui-Sannes, "TURTLE: A Real-Time UML Profile Supported by Formal Validation Toolkit", IEEE Trans. Software Engineering, 30(7):473-487, 2004.

[Ale04] A. Tesanovic, J. Hansson, “Structuring criteria for the design of component-based real-time systems”, Proceeding of the IADIS International Conference on Applied Computing 2004, Mar. 2004.

[And 98] Hugo. A. Andrade, Scott Kovner, “Software Synthesis form Dataflow Models for G and LabVIEW”, Thiry-second Annual Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, California, USA, 11-1998.

[AR03] E. Astesiano, G. Reggio, “Towards a well-founded UML-based development method”, Proceedings of the Software Engineering and Formal Methods, 2003.

[Artisan] Artisan Real Time Studio, 2001.

[Aud91] N.C. Audsley, “Optimal priority assignment and feasibility of static priority tasks with arbitrary start times”, University of York, YCS 164, 1991.

[Bak91] T.P. Baker, "Stack-based scheduling of real-time processes", The Journal of Real-Time Systems, vol. 3, pp. 67-99, 1991

[Bal94] Sandrine Balbo, “Evaluation ergonomique des interfaces utilisateur: Un pas vers l’automatisation”, rapport de Thèse, IMAG – Université Joseph FOURIER, Grenoble 1, 5 Septembre 1994.

[Bala99] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh et al., “Synthesis of Software Programs for Embedded Control Applications”, IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, No. 6, 1999.

[Bar 96] J. Barnes, "Programming in Ada 95", Addison-Wesley, Workingham England, 702p., 1996.

[BBB03] E. Bini, G.C. Buttazzo, G.M. Buttazzo, “Rate Monotonic analysis: the hyperbolic bound”, IEEE Transactions on Computers, Vol. 52(7), pp. 933-942, July 2003.

[BC98] J. P. Babau, F. Cottet, “Programmation et validation des applications temps reel à contraintes strictes sur l’analyse schedulability analysis”, Paris, 11-13 Janvier, 1995.

[BDV04] Alan Burns, Brian Dobbing and Tullio Vardanega “Guide for the use of the Ada Ravenscar Profile in high integrity systems”, ACM SIGAda Ada Letters, vol.XXIV(2), pp. 1-74, Juin 2004.

[Ben86] A. Benveniste, P. Le Guernic, P. Bournal, T. Gautier, “SIGNAL: a data-flow oriented language for signal processing”, IEEE Transactions on ASSP, ASSP-34, No 2, 362-374, 1986.

[Ber91] Gérard Berry, “A hardware implementation of pure Esterel”, Proc. Workshop on Formal Methods in VLSI Design, Miami, 1991.

[Berk03] Kirsten Berkenkötter, “Using UML 2.0 in Real-Time Development – A Critical Review”, In SVERTS, pp. 41-54, San Francisco, USA, October 2003.

[BHR93] S.K. Baruah, R.R. Howell, L.E. Rosier, “Feasibility problems for recurring tasks on one processor”, Theoretical Computer Science, Vol. 1(118), 1993.

[BNB06] E. Bini, M. Di Natale, G.C. Buttazzo, “Sensitivity Analysis for Fixed-Priority Real-Time Systems”, ECRTS, pp. 13-22, 2006.

[Bla76] J. Blazewicz, "Scheduling dependant tasks with different arrival times to meet deadlines", Modeling and performance evaluation of computer systems, ed. E. Gelenben et H. Beilner, Amsterdam, North-Holland, pp. 57-65, 1976.

[BLM02] V. D. Bianco, L. Lavazza, M. Mauri, “A formalization of UML statecharts for real-time software modeling”, The 6th Biennial World Conference on Engineering of Complex Computer Systems (ICECCS 2002), Greenbelt, Maryland, 2-4 December 2002.

[Boe76] Barry W. Boehm, “Software engineering”, IEEE Transactions on Software Engineering, vol. C-25, n12, 1976.

[Boe88] Barry W. Boehm, “A spiral model of software development and enhancement”, IEEE Transactions on Software Engineering, 5-1988.

[BRH90] S.K. Baruah, L.E. Rosier, R.R. Howell, “Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor”, Real-Time Systems, vol. 2, pp. 301-324, 1990.

[Bro87] F. P. Brooks, “No Silver Bullet. Essence and Accidents of Software Engineering”, IEEE Computer, 4-1987.

[Bur98] Alan Burns, Andy Wellings, “Concurrency in Ada”, Second edition, Cambridge University Press, 1998.

[Bur99] Alan Burns “The Ravenscar Profile”, ACM SIGAda Ada Letters, vol. XIX(4) pp. 49-52, Décembre 1999.

[But04] G. Buttazzo, “Hard real-time computing systems: predictable scheduling algorithms and applications”, Ed. Springer, 425p., 2004.

[BW95] A. Burns, A. J. Wellings, “HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems”, Elsevier, 1995.

[CDKM00] F. Cottet, J. Delacroix, C. Kaiser, Z. Mammeri, “Ordonnancement temps réel”, Hermes, pp. 207, 2000.

[CC89] H. Chetto, M. Silly-Chetto, “Some results of the earliest deadline scheduling algorithm”, IEEE Transactions on Software Engineering, vol. 15(10), pp. 1261-1269, 1989.

[Che93] Chettot H., Delacroix J., “Minimisation des temps de réponse des tâches sporadiques en présence des tâches périodiques”, RTS’93, p. 32-52, Paris 1993.

[Chi94] M. Chiodo, P.Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, “A formal methodology for hardware/software codesign of embedded systems”, IEEE Micro, Aug. 1994.

[CG05] F. Cottet, E. Grolleau, "Systèmes Temps Réel de Contrôle-Commande", Dunod, Paris, pp. 560, 2005.

[CL90] M. I. Chen, K. J. Lin, “Dynamic priority ceilings: A concurrency control protocol for real-time systems”, Real-Time Systems Journal, 2(4), p. 325-346, 1990.

[CNRS88] Groupe de Réflexion Temps Réel du CNRS, “Le Temps Réel”, Technique et Science Informatiques, 7(5), pp. 493-500, 1988.

[Cosyma] www.ida.ing.tu-bs.de/research/projects/cosyma/overview/cosyma.html

[Cot 01] Francis Cottet, “LabVIEW, Programmation et Applications”, Dunod, année 2001.

[CW96] P. Cornwell, A. Wellings, "Transaction Specification for Object-Oriented Real-Time Systems in HRT-HOOD”, Lecture Notes in Computer Science, vol. 1031. Proceedings of the Second International Eurospace-Ada-Europe Symposium on Ada in Europe, 1996.

[Dav02] L. David, “Contribution à la gestion de la régularité d'exécution des tâches d'une application temps réel à contraintes strictes, dans un contexte d'ordonnancement en ligne”, thèse, LISI/ENSMA, 2002.

[Dela01] Joëlle Delacroix, “Processus Ordonnancement”, Cours informatique, Méthode de Programmation Système B, 2000/2001.

[Der74] M.L. Dertouzos, “Control robotics: the procedural control of physical processes”, proc. IFIP Congress, pp. 807-813, 1974.

[DG00] R. Devillers, J. Goossens, “Liu and Layland’s Schedulability Test Revisited,” Information Processing Letters, vol. 73(5), pp. 157-161, Mar. 2000.

[DLS97] Z. Deng, J.W.S. Liu, J. Sun, “A scheme for scheduling hard real-time applications in open systems environment”, Proc. 9th Euromicro Workshop on Real-Time systems, pp. 191-199, June 1997.

[DM02] J. Delacroix, C. Ménival, "Système temps réel RTLinux", Techniques de l'Ingénieur, H 1 568, 11-2002.

[Edw97] S. Edwards, L. Lavagno, E. A. Lee, A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis", Proceeding of the IEEE, vol. 85, n3, pp. 366-390, 1997.

[FM02] S. Flake, W. Mueller, "A UML profile for real-time constraints with the OCL", 5th International Conference, Dresden, Germany, September 2002, Proceedings, vol. 2460 of LNCS, pages 179-195, Springer, 2002.

[Fra91] D. W. Franke, M. K. Purvis, "Hardware/Software CoDesign: A Perspective", Proceeding of the 13th international conference on Software Engineering, 1991.

[Gev98] Emmanuel Geveaux, "Conception d'un environnement de développement des applications de contrôle de procédé basé sur le modèle formel GRAFCET et fondé sur un langage graphique flot de données", rapport de Thèse, LISI – ENSMA, 29 Septembre 1998.

[GG04] A. Geniet, E. Grolleau, "Minimal schedulability interval for real-time systems of periodic tasks with offset", Theoretical Computer Science, vol. 310, pp. 117-134, 2004.

[GK06] A. Gherbi, F. Khendek, "UML Profiles for Real-Time Systems and their Applications", in Journal of Object Technology, vol. 5, no. 4, May-June 2006, pages 149-169.

[Gom84] Hassan Gomaa, "A Software Design Method for Real-Time Systems", Communication of the ACM, vol. 27, number 9, September 1984.

[Gom93] Hassan Gomaa, "Software Design Methods for Concurrent and Real-Time Systems", Addison-Wesley Publishing Company, 447 pages, 1993.

[Goo03] J. Goossens, "Scheduling of offset-free systems", Real-Time Systems, Vol. 24(2), pp. 239-258, 2003.

[GOO06] S. Graf, Ileana Ober, Iulian Ober, "A real-time profile for UML", International Journal on Software Tools for Technology Transfer (STTT), vol. 8, Issue 2, pages 113-127, April 2006.

[Gro99] Emmanuel Grolleau, "Ordonnancement temps réel hors-ligne optimal à l'aide de réseaux de Petri en environnement monoprocesseur et multiprocesseur", rapport de Thèse, LISI – ENSMA, 29 Novembre 1999.

[Har87] David Harel, "Statecharts: A visual formalism for complex systems", Science Computer Programming, vol.8, pp. 231-274, 1987.

[Har88] David Harel, "On visual formalisms", Communications of the ACM, vol. 31, no. 5, pp. 514-530, 1988.

[Har90] D. Harel et al., "STATEMATE: A working environment for the development of complex reactive systems", IEEE Transactions on Software Engineering, vol. 16, no. 4, pp. 403-414, April 1990.

[Har96] D. Harel, A. Naamad, "The STATEMATE Semantics of Statecharts", ACM Trans. Software Eng. Methodology, pp. 293-333, Oct. 1996.

[Har97] D. Harel, E. Gery, "Executable Object Modeling with Statecharts", IEEE Computer, 30(7):31-42, July 1997.

[Hat91] D. J. Hatley, I. A. Pirbhai, "Stratégie de specification des systèmes temps reel (SA-RT)", Masson, 346 pages, 1991.

[Hav68] J. W. Havender, "Avoiding Deadlocks in mutitasking systems", IBM Sytems Journal, 7, 2, p. 74-84, 1968.

[HG07] J.-F. Hermant, L. George "A C-space sensitivity analysis of Earliest Deadline First scheduling", proc. ISoLA, pp. 21-32, 2007.

[Horn74] W. Horn, "Some simple scheduling algorithms", Naval Research Logistic Quarterly, 21, 1974.

[Ilogix] Ilogix. Rhapsody development environment.

[Jac99] Ivar Jacobson, Grady Booch, James Rumbaugh, “Le processus unifié de développement logiciel”, Eyrolles, 488 pages, 1999.

[Jack55] J.R. Jackson, “Scheduling a production line to minimize maximum tardiness”, Management Science Research Project 43, University of California, Los Angeles, 1955

[Jack83] Michael Jackson, “System Development”, Prentice Hall, Englewood Cliffs, N.J., 1983.

[Jack98] Michael Jackson, “Formal Methods and Traditional Engineering”, Journal of Systems and Software, 40:191-194, 1998.

[JP86] M. Joseph and P. Pandya, “Finding response times in real-time system”, The Computer Journal, vol. 29(5), pp. 390-395, 1986.

[JS93] K. Jeffay and D.L. Stone, “Accounting for interrupt handling costs in dynamic priority task systems", proc. IEEE Real-Time Systems Symposium, Raleigh-Durham, NC, USA, 1993.

[Kai81] C. Kaiser, “De l’utilisation de la priorité en presence d’exclusion mutuelle", Rapport INRIA RR 84, 1981.

[Kai82] C. Kaiser, « Exclusion mutuelle et ordonnancement par priorité », Technique et Science Informatiques, Vol. 1, pp. 59-68, 1982.

[KM04] M. T. Kimour, D. Meslati, "An Approach to Building Object Models with UML in Embedded Systems", Journal of Computing and Information Technology – CIT 12, 2004, pages 223-235.

[Kop91] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner, W. Schütz, “The design of real-time systems: from specification to implementation and verification”, Software Engineering Journal, May 1991.

[KP92] Y. Kesten, A. Pnueli, “Timed and Hybrid Statecharts and their Textual Representation”, Weizmann Institute of Science, In Formal Techniques in Real-Time and Fault-Tolerant Systems 2nd International Symposium, 1992.

[KS01] J. M. Küster, J. Stroop, “Consistent design of embedded real-time systems with UML-RT”, Proceeding of the 4th International Symposium on Object-Oriented Real-Time Distributed Computing 2001, Los Alamitos, USA, IEEE Computer Society 2001, p. 31-40 Conference Paper.

[Lab74] Jacques Labetoulle, “Un algorithme optimal pour la gestion des processus en temps réel”, Revue Française d’Automatique, Informatique et Recherche Opérationnelle, pp. 11-17, Févr. 1974.

[LabV-1] “LabVIEW Express VI Development Toolkit User Guide – version 1.0”, Technical Report, August 2003.

[LabV-2] Norma Dorst, “Using LabVIEW to Create Multithreaded VIs for maximum Performance and Reliability”, Application Note 114, July 2000.

[Leh89] J. P. Lehoczky, L. Sha, Y. Ding, “The Rate Monotonic scheduling algorithm: Exact characterization and Average case behavior”, Proceedings of Real-Time Systems Symposium, p. 166-171, 1989.

[Leh90] J.P. Lehoczky, “Fixed priority scheduling of periodic task sets with arbitrary deadlines”, IEEE Real-Time Systems Symposium, Lake Buena Vista, Florida, USA, 1990.

[Leh92] Lehoczky J. P., Sacha L., Ding Y., “An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems”, Proceddings of the IEEE Real-Time Systems Symposium, p. 110-123, 1992.

[Liu00]J.W.S. Liu, “Real-time systems”, Ed. Prentice Hall, 610 p., 2000.

[LL73] C.L. Liu, J.W. Layland, “Scheduling algorithms for multiprogramming in a hard real-time environment”, Journal of the ACM, vol. 20, n°1, pp. 46-61, 1973.

[LM80] J. Leung and M. Merrill, “A note on preemptive scheduling of periodic real-time tasks”, Information Processing Letters, vol. 11(3), pp. 115-118, 1980.

[LP02] R. Laleau, F. Polack, “Coming and going from UML to B: a proposal to support traceability in rigorous IS Development”, ZB’2002 – Formal Specification and Development in Z and B.

[LQV01] L. Lavazza, G. Quaroni, M. Venturelli, “Combining UML and formal notations for modelling real-time systems”, Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), Wien, 10-14 September 2001.

[LR92] J.P. Lehoczky and S. Ramos-Thuel,"An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems", proc. Real-Time Systems Symposium, Phoenix, 1992, pp. 110-123.

[LSS87] J.P. Lehoczky, L. Sha and J. Strosnider,"Enhanced aperiodic responsiveness in hard real-time environment", proc. Real-Time Systems Symposium, San Jose, 1987, pp. 261-270

[LSST91] J. P. Lehoczky, L. Sha, J. K. Strosnider, H. Tokuda, “Fixed priority scheduling theory for hard real-time systems”, in Foundations of Real-Time Computing: Scheduling and ressource management, Kluwer Academic Publishers, pp. 1-30, 1991.

[LW82] J. W.-T. Leung, J. Whitehead, “On the complexity of fixed-priority scheduling of periodic real-time tasks”, Performance Evaluation, 2, pp. 237-250, 1982.

[Mar00] R. Marcano, E. Meyer, N. Levy, J. Souquières, “Utilisation de patterns dans la construction de spécifications en UML et B”, Journées Approches Formelles dans l’Assistance au Développement de Logiciels (AFADL’2000), p. 66-80, Grenoble – France, 26-28 Janvier 2000.

[May96] Elaine L. May, Barbara A. Zimmer, “The evolutionary development model for software”, Hewlett-Packard Journal, August 1996.

[Mic97] G. De Micheli, R. K. Gupta, “Hardware/Software Co-Design”, Proceeding of the IEEE, vol. 85, n3, pp. 349-365, 1997.

[MG06] K. L. Mills, H. Gomaa, “A Knowledge-base Approach for Automating a Design Method for Concurrent and Real-Time Systems”, Proceedings of the SEKE 96, ACM, June 10-12, 1996, pp. 529-536.

[Mil96] Kevin L. Mills, “Automated Generation of Concurrent Designs for Real-Time Software”, Ph.D Dissertation, George Masson University, FairFax, Virginia, March 1996, 588 pages.

[Mok83] A. K. Mok, “Fundamental design problems for the hard real-time environnements”, Ph. D. MIT, May 1983.

[NT04] J. A. Niaz, J. Tanako, “Mapping UML Statecharts to Java Code”. In M. H. Hamza, editor, International Conference on Software Engineering (SE’2004), Applied Informatics, pp. 111-116, Innsbruck (Austria), 2-2004.

[OMG1] OMG. UML 2.0 Infrastructure Specification, OMG Adopted Specification, September 2003.

[OMG2] OMG. UML 2.0 Superstructure Specification, OMG Adopted Specification, August 2003.

[OMG3] OMG. UML Profile for Schedulability, Performance and Time Specification, March 2002.

[OMG4] OMG. Unified Modeling Language Specification (Action Semantics), January 2002.

[OMG5] OMG. Unified Modeling Language Specification, version 1.4, Sept. 2001.

[Par72] D. L. Parnas, “On the Criteria To Be Used in Decomposing Systems into Modules”, Communications of ACM, vol. 15, number 12, December 1972.

[Ped93] B. M. Pedersen, “SDL-92 as an object oriented notation”, Telektronikk, 1993.

[Ped00] B. M. Pedersen, “SDL Combined with UML”, Telektronikk, 4-2000.

[POLIS] Balarin, F. Giusto, P. D. Jurecska…, “Hardware-Software Co-Design of Embedded Systems: The POLIS Approach”, Kluwer Academic Pulishers, pp. 324, 1997.

[RASSP] www.eda.org/rassp/index.html, Rapid Prototyping of Application Specific Signal Processors (RASSP), Module 14, “Hardware/Software Codesign Overview”.

[RCK01] P. Richard, F. Cottet, C. Kaiser, “Précédences généralisées et ordonnançabilité des tâches de suivi temps réel d’un laminoir", APII – JESA, 35(9), pp 1055-1071, 2001.

[Ree01] Rick Reed, “Notes on SDL-2000 for the new millennium”, Computer Networks, p. 709-720, 2001.

[RMA] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, M.G. Harbour, “A practitioner’s handbook for real-time analysis: guide to rate monotonic analysis for real-time systems”, 712 p., Kluwer Academic Publishers, 1994.

[Ros97] Jean-Pierre Rosen, “HOOD, An industrial approach for software design”, HOOD Technical Group, 1997.

[ROSE] Rational/IBM. Rose real-time development environment.

[Roy70] W.W. Royce, “Managing the development of large software systems”, Proc. WESTCON, Calif., USA, 1970.

[Roz94] Rozenblit, J. and K. Buchenrieder, “Codesign Computer-Aided Software/Hardware Engineering”, IEEE Press, Piscataway, New York, 1994.

[Sae94] Motoshi Saeki, “Software Specification & Design Methods and Method Engineering”, International Journal of Software Engineering and Knowledge Engineering, 1994.

[SB96] M. Spuri, G. Buttazzo, “Scheduling aperiodic tasks in dynamic priority systems”, Real-Time Systems, Vol. 10(2), pp.179-210, 1996.

[Scha02] S. R. Schach, “Object-Oriented and Classical Software Engineering” McGraw-Hill Science, 2002, p. 71.

[Ser72] O. Serlin, “Scheduling of time critical processes”, proc. Spring Joint Computers Conference, pp. 925-932, 1972.

[Sha95] Mary Shaw, "Comparing Architectural Design Styles", IEEE Software, 12(6):27-41, Nov. 1995.

[SLS95] J.K. Strosnider, J.P. Lehoczky and L. Sha,"The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environment", IEEE Transactions on Computers, vol. 44(1), pp. 73-91, 1995.

[Spu96] M. Spuri, “Analysis of deadline scheduled real-time systems”, Research Report INRIA, 2772, Jan, 1996.

[SRL90] L. Sha, R. Rajkumar, J. P. Lehoczky, “Priority inheritance protocols: An approach to real-time synchronisation”, IEEE Transaction on Computers, Vol. 39, n° 9, p. 1175-1185, 1990.

[Som98] Ian Sommerville, “Requirements Engineering: Processes and Techniques”, John Wiley&Sonc Inc., 282 pages, April 1998.

[Spr89] Sprunt B., Sha L., Lehoczky J. P., “Aperiodic task scheduling for hard-real-time systems”, The Journal of Real-Time Systems, p. 27-60, 1989.

[SRa04] H. Saiedian, S. Raguraman, “Using UML-Based Rate Monotonic Analysis to Predict Schedulability”, IEEE Computer, 37(10):56-63, 2004.

[SR98] B. Selic, J. Rumbaugh, “Using UML for modelling complex real-time systems”, Lecture Notes in Computer Science, 1474:250-262, 1998.

[SSL89] B. Sprunt, L. Sha and J. Lehoczky,"Aperiodic task scheduling for hard real-time systems", The Journal of Real-Time Systems, vol. 1, pp. 27-60, 1989

[Stau97] J. Staunstrup, W. Wolf, “Hardware/Software Co-Design: Principes and Practice”, Kluwer Academic Publishers, 1997.

[Ste99] W. P. Stevens, G. J. Myers, L. L. Constantine, “Structured design”, IBM Systems Journal, vol. 38, no. 2&3, 1999.

[TAU] Telelogic. TAU Generation 2 Reference Manual, 2002.

[Tra07] Karim Traoré, “Analyse et validation des applications temps réel en présence de transactions : application au pilotage d'un drone miniature”, Thèse, LISI-ENSMA, 2007

[VGC05] M. C. Valiente, G. Genova, J. Carretero, "UML 2.0 Notation for Modeling Real Time Task Scheduling", in Journal of Object Technology, vol. 5, no. 4, May-June 2005, pages 91-105.

[WEE07] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, “The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools”, ACM Transactions on Embedded Computing Systems (TECS), 2008.

[Wind 95] WindRiver, “VxWorks 5.3 – Programmer’s Guide”, 1995.

[YC79] E. Yourdon, M. Constantine, "Structured Design”, Prentice Hall, Englewood Cliffs, New Jersey, 1979.

[Z100] Z.100 (11/99) Specification and Description Language (SDL), ITU-T, 2000.

[Z109] Z.109 (11/99) SDL combined with UML, ITU-T, Geneva, 2000.

[Zur05] Richard Zurawski, “Embedded Systems Handbook”, Industrial Information Technology Series, Taylor&Francis, 2005.

Annexe A
Eléments d'XML

Entités

L'entité représente un phénomène (personne, concept, événement), qui peut être traité comme une unité indépendante, et dans lequel des données peuvent être stockées. Dans les documents XML-DTD, les entités peuvent être soit les entités représentant du texte (*Parsed General Entities*), soit des entités qui ne comportent pas du tout de caractères (*Unparsed Entities*), ou les entités paramètres (*Parameter Entities*).

- «Parsed General Entities»: sont utilisées souvent pour les textes dans les documents XML. Elles servent à référencer des textes et à simplifier les gestions des documents XML. Une texte-entité peut être déclarée dans une DTD de manière interne ou externe. En cas d'entités internes, la syntaxe de déclaration est la suivante:

```
<!ENTITY NomEntite "ContenuEntite">
```

- «ContenuEntite» ne peut comporter que des autres entités et des caractères (PCDATA). La syntaxe pour employer une texte-entité dans un fichier XML est vraiment semblable à celle des entités prédéfinies et des caractères référencés, c'est: «`&NomEntite;`».

- Les texte-entités externes sont utilisées de la même façon que celles internes, mais elles sont plus puissantes et plus contraintes, car leur contenu doit être stocké dans un autre fichier XML, et doit être bien-formé. Il est important de remarquer que dans certains cas un prologue peut être inclus, mais en forme réduite avec les paramètres indiquant la version et l'encodage (pas de paramètre «*standalone*»). Donc, la syntaxe acceptable de la déclaration d'une texte-entité externe est:

- <!ENTITY NomEntite SYSTEM "URI-au-Contenu">

- Les entités externes permettent de décomposer un document XML parties qui peuvent être standardisées et réutilisées, d'autre part des parties qui sont particulières à certain document et peuvent être mises à jour aisément.

- «Unparsed Entities»: sont les entités dont le contenu n'est pas du texte. Chacune comporte une référence pour le contenu de cette entité et une déclaration de type de contenu (fournie typiquement par une notation). La syntaxe de déclaration d'une «*Unparsed Entity*» ressemble à celle de texte-entité externe avec un mot-clé additionnel, appelé «NDATA», et l'identificateur de notation:

- <!ENTITY NomEntite SYSTEM "URI-au-Contenu" NDATA NomNotation>

- La notation doit être déclarée ailleurs dans la DTD. En outre, dans certains cas au lieu d'utiliser la syntaxe traditionnelle de référence des entités (c'est-à-dire, «`&NomEntite;`»), nous pouvons référencer une telle entité en tant qu'une valeur d'un attribut d'un élément dans les documents XML afin d'ajouter des informations supplémentaires, notamment en cas d'élément vide. (Les syntaxes de déclaration des éléments et des attributs seront présentées aux paragraphes suivants).

- «Parameter Entities»: sont les entités utilisées à l'intérieur des DTDs. De même que les texte-entités, les paramètres-entités peuvent être internes ou externes. Leur rôle est de représenter le contenu d'une DTD, ce qui permet de séparer chacune des DTDs en ses éléments constituants, et de faciliter la déclaration d'un ensemble des éléments partageant un ensemble des attributs communs. La syntaxe de déclaration des paramètres-entités internes est :

- <!ENTITY % NomEntite "ContenuEntite">

- Pour les paramètres-entités externes, la déclaration peut ressembler à la suivante :
- <!ENTITY % NomEntite SYSTEM "URI-au-Contenu">

  Pour référencer un paramètre-entité, nous utilisons la syntaxe suivante: «`%NomEntite;`»
- Remarquer que l'emploi des paramètres-entités externes est assez compliqué par rapport à celles internes, car un ensemble des déclarations internes dans une DTD peut supprimer celui externe en provenance de ces paramètres-entités.

IGNORE et INCLUDE

«IGNORE» et «INCLUDE» sont les deux mots-clés qui permettent de conditionner les traitements d'une ou plusieurs sections dans une DTD. Généralement, tous les deux sont utilisés uniquement dans les parties de déclarations d'une DTD externe, pas dans le contenu du document ou les parties de déclarations de DTD interne. Une fois qu'ils sont déclarés, ils servent à empêcher certaines sections d'une DTD d'être traitées. Par conséquent, cela permet de faciliter le changement des contextes des documents XML-DTDs avec un minimum de modifications. La syntaxe pour la déclaration d'une section marquée «INCLUDE» (c'est-à-dire d'autoriser le traitement), et celle marquée «IGNORE» (c'est-à-dire d'ignorer le traitement) sont les suivantes:

```
<![INCLUDE[ Declarations_completes ]]>
<![IGNORE[ Declarations_completes ]]>
```

Eléments

Les éléments de XML-DTD sont déclarés par la syntaxe suivante:

```
<!ELEMENT NomElement ModeleContenu>
```

Dans une DTD, les éléments peuvent être définis de manière soit interne, soit externe, mais il faut assurer qu'un élément est déclaré de façon unique. En fait, chaque DTD comporte une liste des déclarations d'éléments qui détermine la structure d'éléments dans les documents XML correspondants validés par cette DTD (chaque élément dans un document XML doit correspondre à une déclaration dans la DTD, et son contenu doit également être approprié à celui décrit par «`ModeleContenu`» de cette déclaration). Si ces deux conditions ne sont pas satisfaites, le document XML est dit «non-valide», et le «*parser*» rapportera une erreur.

Il existe quatre modèles de contenu d'un élément, ce sont «ANY», «EMPTY», «Eléments enfants», et «Contenu mixte». Il en résulte que le contenu d'un élément dans sa syntaxe des déclarations doit être conforme à un de ces quatre modèles.

- «ANY»: la syntaxe est la suivante,
- <!ELEMENT NomElement ANY>
- Le contenu des éléments de type «ANY» peut comporter du texte, des éléments enfants, ou aucun. Un tel type d'éléments est très flexible, mais dans certains cas cette flexibilité énorme peut entraîner des difficultés à créer une structure d'éléments monolithique. Donc, l'élément du type «ANY» est utilisé fréquemment en tant qu'une marque substitutive (*placeholder*), qui peut être remplacée par un modèle de contenu plus détaillé dès que le développement progresse.
- «EMPTY»: la syntaxe est suivante,
- <!ELEMENT NomElement EMPTY>

- Contrairement à l'élément du type «ANY», le contenu de l'élément du type «EMPTY» ne contient rien. Dans certains cas, ces éléments se servent des drapeaux (flags), mais typiquement ils sont utilisés pour porter les informations de l'élément dans leurs attributs.

- «Eléments d'enfant»: dans la syntaxe de déclaration de ces éléments, le «`ModeleContenu`» est spécifié par les éléments enfants sous forme d'un ensemble de possibilité, liées et regroupées par les ponctuations, qui identifient comment ce contenu peut apparaître. Donc, la séquence des éléments est exprimée par les virgules (,), tandis que leurs choix sont indiqués par les barres verticales (|), et les parenthèses sont utilisées pour grouper les éléments en sous-ensembles, qui peuvent être traités plus aisément.

  Par exemple, le modèle de contenu d'élément «NometPrenom» ci-après:

  ```
  <!ELEMENT NometPrenom (NomdeFamille,(Prenom|(Prenom,
  SecondPrenom)))>
  ```

  exige un élément «`NomdeFamille`», suivi par soit un élément «`Prenom`», soit une séquence des deux éléments «`Prenom`» et «`SecondPrenom`».

  Cependant il ne suffit pas toujours pour les éléments spécifiés en séquence, car parfois ils peuvent être apparus dans les documents XML de manière optionnelle ou itérative. Ainsi, les spécifications XML 1.0 fournissent les indicateurs d'occurrence afin de faciliter les descriptions du «`ModeleContenu`» (voir Tableau 1).

Indicateur	Définition
(aucun)	L'élément doit être apparu une seule fois
?	l'élément (ou un ensemble des éléments mis en parenthèses) peut être apparu une fois ou aucune. Cet élément est facultatif, mais n'apparaît pas plus d'une fois.
+	l'élément (ou un ensemble des éléments mis en parenthèses) doit être apparu au moins une fois dans un document XML.
*	l'élément (ou un ensemble des éléments mis en parenthèses) peut être apparu jamais ou plusieurs fois dans un document XML. Cette option est l'option la plus flexible, qui permet de déclarer un élément plusieurs fois consécutives s'il est nécessaire, ou pas du tout.

Tableau 1: Indicateurs d'occurrence dans XML-DTD

  Il en résulte que l'élément «`NometPrenom`» dans l'exemple dessus peut être déclaré avec les indicateurs d'occurrence de façon suivante:

  ```
  <!ELEMENT NometPrenom (Nom,Prenom,SecondPrenom?)>
  ```

  L'élément «`SecondPrenom`» est optionnel (indiqué par le point d'interrogation), donc dans l'élément «`NometPrenom`.

- «Contenu mixte»: XML est un langage hiérarchique. Les éléments XML constituent un squelette dans lequel les informations sont stockées. Il existe un seul élément racine, qui est l'ancêtre de tous les autres éléments. Au niveau de l'élément terminal (*leaf element*), il n'y a pas de descendant, le modèle de contenu est soit du texte (PCDATA), soit nul (élément vide). Donc, le modèle de contenu mixte dans ce cas là est une extension de celui d'élément terminal, car le contenu mixte peut comporter les textes et les éléments enfants à la fois, ou aucun. Il est important de remarquer qu'il n'est pas possible de spécifier l'ordre des composants définis dans le contenu de l'élément mixte. Ce modèle doit être utilisé avec prudence. La syntaxe de déclaration mixte ressemble à celle-ci-dessous:

```
<!ELEMENT NomElement (#PCDATA|Element|Element...)*>
```

Attributs des éléments

L'attribut est un paramètre qui détermine une caractéristique d'un élément placé dans un modèle. Les attributs XML fournissent une couche de puissance supplémentaire aux éléments permettant de définir les relations entre eux.

Chaque élément XML peut comporter une liste d'attributs déclarations qui contient au moins une déclaration d'attribut de cet élément. La syntaxe de déclaration d'une liste d'attributs dans XML-DTD est suivante:

```
<!ATTLIST NomElement
   NomAttribut  TypeAttribut     DefautDeclaration
   [... autres attributs ...]
>
```

«`TypeAttribut`» identifie la définition de contenu d'un attribut que sa valeur dans les documents doit satisfaire. Ceci est assuré par le «*parser*» durant la validation. Le Tableau 2 ci-dessous représente quelques types d'attributs que j'utilise dans ce manuscrit.

Indicateur	Définition
CDATA	la valeur d'attributs peut être n'importe quelle chaîne de caractères XML légaux.
ID	la valeur doit être unique par rapport à tous les autres attributs du type «ID» dans le document XML. Pour chaque élément, il existe un seul attribut du type «ID» parmi ses attributs déclarés. Et les attributs «ID» n'ont pas jamais de valeur par défaut (#FIXED).
IDREF	la valeur d'attributs doit être identique à celle d'un attribut du type «ID» d'un élément dans le même document XML.
ENTITY	la valeur d'attributs doit être identique au nom d'une entité du type «Unparsed Entity» déclarée dans la DTD.
énuméré	ceci fournit une liste des valeurs acceptables pour un attribut. Le mot «énuméré» n'apparaît pas dans la déclaration de la liste d'attributs, il est remplacé par une liste des valeurs possibles selon la syntaxe suivante : *(valeur | valeur ...)*

Tableau 2: Types d'attributs disponibles dans les spécifications XML 1.0

Une valeur par défaut est une valeur attribuée automatiquement par le programme en l'absence d'une situation explicite de la part de l'utilisateur et qui représente habituellement le choix ou le réglage le plus probable, compte tenu du contexte. Les défauts d'attribut XML («`DefautDeclaration`») représentent les nouvelles fonctions, regroupées en quatre situations possibles par défaut, décrites en détail dans le tableau ci-après:

Défaut	Définition
#REQUIRED	l'élément instancié doit fournir explicitement une valeur pour cet attribut une fois qu'il est utilisé.
#IMPLIED	l'élément instancié peut fournir une valeur pour cet attribut si l'utilisateur le désire.
"ValeurEntreGuillemets"	une valeur pour cet attribut est optionnelle, mais une fois qu'elle est déclarée, elle se sert d'une valeur d'attribut par

	défaut attribuée automatiquement en l'absence d'une indication explicite dans l'élément instancié.
#FIXED "ValeurEntreGuillemets"	la valeur de cet attribut est fixée en permanence à "ValeurEntreGuillemets". Un changement de cette valeur dans un élément instancié va engendrer une erreur.

Tableau 3: Défauts d'attributs disponibles dans les spécifications XML 1.0

En associant un nom, un type, un défaut et un nom d'élément, nous pouvons déclarer aisément des attributs d'un élément déterminé. Grâce à des fonctions supplémentaires, l'attribut XML devient un outil très utile, qui permet de définir non seulement plus de contraintes pour un élément, mais également des interrelations de deux ou plusieurs éléments dans un document XML. Afin d'éclaircir ces annotations, les informations de l'élément «`NometPrenom`» auparavant peuvent être stockées en tant que les valeurs d'attributs de cet élément de manière suivante:

```
<!ELEMENT NometPrenom EMPTY>
<!ATTLIST NometPrenom
   Nom     CDATA      #REQUIRED
   Prenom      CDATA      #REQUIRED
   SecondPrenom      CDATA      #IMPLIED>
```

Remarquer que l'élément «`NometPrenom`» dans ce cas là est un élément vide, mais il faut que chaque élément instancié fournisse explicitement une valeur pour chacun des attributs, appelés «`Nom`» et «`Prenom`». Alors qu'une valeur pour l'attribut «`SecondPrenom`» est optionnelle. Par conséquent, l'absence de la valeur de cet attribut dans les éléments instanciés de «`NometPrenom`» n'entraîne aucune erreur durant la validation d'un document XML.

DTD (Déclaration de Type de Document)

Une DTD (Déclaration de Type de Document) est un ensemble de règles qui définissent la sémantique de la structure interne des éléments d'un fichier XML. Ainsi lorsqu'un fichier XML est traité, il faut vérifier la conformité de sa structure d'éléments interne à celle définie dans sa DTD. Cela permet d'assurer d'une part l'aspect sémantique (c'est-à-dire que ce document est bien-formé), d'autre part l'aspect syntaxique (c'est-à-dire la partie de la grammaire définissant les règles qui régissent la formation correcte des énoncés du fichier). En fait, la DTD et le Schéma, une autre forme de déclaration des syntaxes des documents XML, sont les deux outils de validation communs dans les spécifications de XML 1.0. Grâce à eux, XML devient de nos jours un langage des marquages de plus en plus répandu pour l'échange d'informations. Les développeurs peuvent trouver dans les documents XML non seulement la lisibilité (avec un arbre d'informations), mais également l'intégrabilité et l'extensibilité.

La validation du fichier «contacts.xml» auparavant peut être réalisée par le «*parser*» avec une simple DTD telle celle-ci-après:

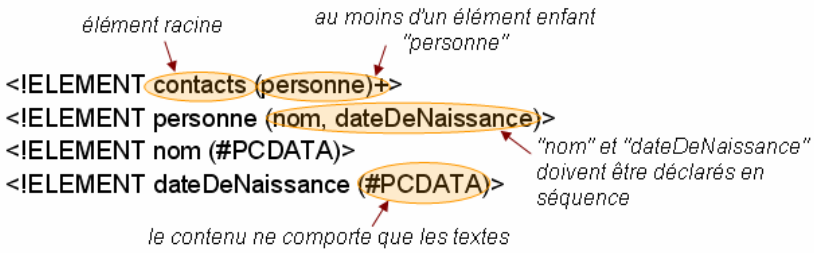


Figure 0-1: «contacts.dtd», validation de fichier «contacts.xml»

Il est important de noter que si le fichier «contacts.xml» contient et se conforme à cette DTD, il est dit valide. Sinon, il est non-valide, mais il peut être bien-formé.

XML-DTD: limitations

XML est un langage indépendant de langages de programmation. Les spécifications XML fournissent les syntaxes et les structures pour les traitements des informations de n'importe quel domaine. Plusieurs modèles XML peuvent être créés pour la description d'un document ou des types de données. Cela rend les documents XML la convivialité et la portabilité.

Pour valider un document XML, une DTD (Déclaration de Type de Document) est utilisée. Cependant, XML-DTD n'offre qu'un nombre limité des syntaxes pour les déclarations de contenu des éléments XML et leurs attributs. Effectivement, les DTDs ne supposent aucune contraintes pour les textes (#PCDATA), le contenu d'éléments XML. Pour les attributs, il est possible d'imposer quelques contraintes aux éléments instanciés, par exemple les attributs de type «ID», de type «IDREF», ou les attributs requis/optionnels (#REQUIRE, #IMPLIED). Mais nous ne trouvons d'aucun moyen permettant d'assurer que la valeur d'un attribut ou le contenu d'un élément est conforme à une sémantique déterminée (par exemple, un nombre entier, une valeur flottante, etc.). Considérer le fichier XML-DTD ci-dessous:

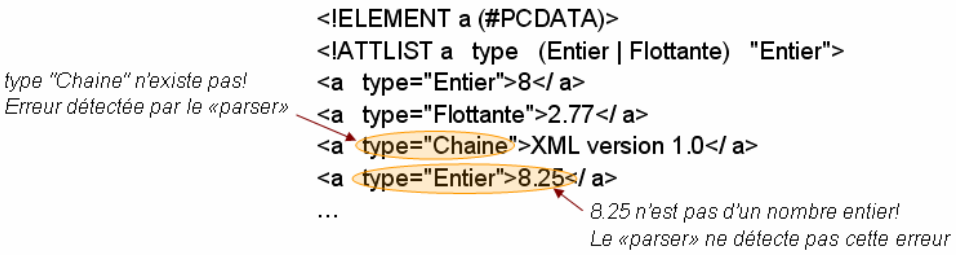


Figure 0-2: XML-DTD n'impose aucune contrainte aux textes

«a» est un élément qui ne contient que des textes. «a» possède un attribut, nommé «type», qui permet d'ajouter une information concernant son type de données. Donc, dans ce cas-là «a» peut être soit du type entier, soit du type flottant, et le type entier est la valeur par défaut de cet attribut. Nous voyons que le type du troisième élément de «a» est «Chaine» (Chaîne de Caractères), si ce type n'est pas défini dans la liste des valeurs acceptables de «a», une erreur survient. Elle est détectée par le «*parser*» durant la validation. Par contre, pour le quatrième «a» bien que la valeur de son attribut soit correcte (type Entier), 8.25 n'est pas un nombre entier. Cette incohérence de type de données ne peut pas être détectée par le «*parser*». Alors, en supprimant le troisième élément, ce fichier XML est dit «valide»!

En utilisant le XML-Schéma au lieu de XML-DTD, la conformité de contenu d'éléments XML à son type de données peut être assurée. XML-Schéma (www.w3.org/TR/xmlschema-0) permet de décrire une structure de document XML de façon très précise (ex., spécification des types de données autorisés, des intervalles de valeurs, de patterns, etc.). Donc pour valider le fichier XML au-dessus, un schéma tel que celui-ci peut être utilisé,

```
<xsd:element   name="a">
    <xsd:complexType>
      <xsd:choice>
        <xsd:simpleContent>
          <xsd:extension   base="xsd:integer">
            <xsd:attribute name="type" type="xsd:string" used="required" fixed="Entier">
          </ xsd:extension>
        </ xsd:simpleContent>
        <xsd:simpleContent>
          <xsd:extension   base="xsd:float">
            <xsd:attribute name="type" type="xsd:string" used="required" fixed="Flottante">
          </ xsd:extension>
        </ xsd:simpleContent>
```

```
        </ xsd:choice>
    </ xsd:complexType>
</ xsd:element>
```

Par rapport à XML-DTD, les schémas sont eux-mêmes les fichiers XML (contrairement aux DTD utilisant une syntaxe spécifique), qui décrivent précisément des structures de documents XML. Cela permet d'une part d'assurer l'échange de données, d'autre part de vérifier la validité de ces documents. Cependant, les formes de XML-Schéma sont très lourdes et verbeuses, surtout pour des utilisateurs qui ne connaissent pas bien XML. De plus, la validation des documents XML par des schémas nécessite soit une interface de programmation programmatique (*programmatic API*) telles Xerces de Apache (www.apache.org/xerces-j/index.html), MSXML4.0 (www.microsoft.com), soit une interface utilisateur graphique orientée (*GUI oriented*) telles XML Spy (www.xmlspy.com), Turbo XML (www.extensibility.com), ou un outil tiers

Avec ses simplicités, XML-DTD est encore utilisé dans plusieurs domaines où l'échange de données est requis. Par ailleurs, le problème de l'incohérence de type de données (ou la possibilité de typer les contenus) dans la validation par DTD au dessus peut être résolu aisément au moment des traitements de ces documents au fur et à mesure de l'interpréteur. En conséquence, étant donné que l'utilisateur de DARTSVIEW peut être amené à définir ses propres fichiers XML, dans ce manuscrit, XML-DTD sert d'outil d'échange d'informations entre le modèle DARTSVIEW et le générateur de code malgré ces limitations. Je garde XML-Schéma comme une perspective pour les lecteurs qui s'intéressent à la description en détail et précise des documents XML.

Aide au développement de systèmes temps réel à l'aide d'un langage graphique flots de données

Présentée et soutenue publiquement par

NGO Khanh Hieu

Directeurs de thèse

Francis COTTET et Emmanuel GROLLEAU

Résumé : Les systèmes temps réel sont des systèmes pour lesquels le respect des contraintes temporelles est aussi important que l'exactitude du résultat. Le développement d'un système temps réel embarqué se concentre sur le développement de deux composants: matériel et logiciel. Par rapport à l'approche en co-design, l'approche traditionnelle permet de développer en parallèle le matériel et le logiciel. Cela facilite considérablement des modifications de chacun des composants de manière à ce qu'une modification de l'un n'influence pas le développement de l'autre, et vice versa jusqu'à l'intégration des deux. Cependant afin de réduire le «time-to-market» d'un développement avec cette approche, il est nécessaire d'utiliser un cycle de développement de logiciel, qui permet d'une part d'assurer des spécifications indépendantes du matériel, d'autre part d'être capable d'effectuer le codage croisé prenant en compte de matériel lors de l'intégration entre le logiciel et le matériel. C'est le cycle de vie d'un logiciel en W. La méthode DARTS, centre d'intérêt de la thèse, est une méthode de conception de systèmes multitâches qui se focalise sur l'aspect communications/rythmes des tâches composant le système.

La palette DARTSVIEW, développée au cours de cette thèse, est une bibliothèque des éléments DARTS implémentés en langage G (LabVIEW™). Grâce aux propriétés avancées de la programmation graphique de G, DARTSVIEW permet de faciliter considérablement le premier V du cycle de vie en W permettant tests et des simulations de l'aspect fonctionnel. L'idée de génération de code à partir d'un modèle DARTSVIEW a pour but de rendre efficace la validation dans le deuxième V du cycle en W en offrant aux concepteurs la possibilité de génération de code conformément à un langage sur cible basé sur une norme temps réel, soit à un outil de validation tiers par l'intermédiaire des fichiers XML. Ainsi, la séparation de l'aspect fonctionnel et de l'aspect temporel dans l'approche traditionnelle de développement d'un logiciel est encore utile pour des systèmes temps réel, où les spécifications sont principalement fonctionnelles. DARTSVIEW est une plate-forme de génération de code, permettant aux concepteurs de choisir leur outil de validation et de générer le code dans le langage choisi.

Avec les résultats obtenus, cette thèse montre que la démarche DARTSVIEW et la programmation graphique flots de données peut s'appliquer au développement des systèmes temps réel fondé sur une méthode de base orientée objet telles CODARTS, HRT-HOOD, UML, etc.

Mots-clés: cycle de développement, temps réel, systèmes embarqués, contrôle/commande, méthode DARTS, programmation multitâche par LabVIEW, codage croisé, DTDs-XML, DARTSVIEW