



HAL
open science

Conception d'un générateur d'intergiciels temps réel embarqués dans l'automobile.

Karim Dahmen

► **To cite this version:**

| Karim Dahmen. Conception d'un générateur d'intergiciels temps réel embarqués dans l'automobile..
| [Stage] 2007. inria-00586813

HAL Id: inria-00586813

<https://inria.hal.science/inria-00586813>

Submitted on 18 Apr 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

RÉPUBLIQUE TUNISIENNE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR,
DE LA RECHERCHE SCIENTIFIQUE ET DE LA TECHNOLOGIE
UNIVERSITÉ DE TUNIS EL MANAR



INSTITUT SUPÉRIEUR D'INFORMATIQUE

MÉMOIRE DE PROJET DE FIN D'ÉTUDES

Présenté en vue de l'obtention du
Diplôme National d'Ingénieur en Informatique
Spécialité : Génie du Logiciel et des Systèmes d'Information

Par

Karim DAHMEN

**CONCEPTION D'UN GÉNÉRATEUR D'INTERGICIELS
TEMPS RÉEL EMBARQUÉS DANS L'AUTOMOBILE**

Organisme d'accueil

LORIA - INRIA Lorraine

Laboratoire Lorrain de recherche en Informatique et ses Applications - UMR 7503.

Encadrants à l'établissement : **Françoise SIMONOT-LION** - Professeur

Xavier REBEUF - Maître de conférence

Encadrant à l'ISI : **Adel KHALFALLAH** - Maître de conférence

Il faut apprendre pour connaître, connaître pour comprendre, comprendre pour juger.

[Narada]

... juger par nos sens pour progresser au mieux de nos intérêts.

Généralités

Institution

ISI – Institut Supérieur d’Informatique – Université de Tunis El Manar
2, Abou Raihane Bayroumi – L’Ariana 2080 – Tunisie – (www.isim.rnu.tn).

Organisme d’accueil

LORIA – INRIA Lorraine – Laboratoire lorrain de Recherche en Informatique et ses Applications.
UMR 7503 – Équipe TRIO – Bâtiment C – Campus scientifique – B.P. 239 – 54506
Vandoeuvre lès Nancy Cedex – France – (www.loria.fr/equipes/TRIO/).

Encadrants de l’organisme d’accueil

Xavier REBEUF – Maître de conférence à l’ École Nationale Supérieure d’Électricité et de Mécanique – INPL – (Xavier.Rebeuf@loria.fr).

Françoise SIMONOT-LION – Responsable de l’équipe – Professeur à l’École Nationale Supérieure des Mines de Nancy – INPL– (simonot@loria.fr).

Répondant de l’institution

Adel KHALFALLAH - Maître de conférence à l’Institut Supérieur d’Informatique – Université de Tunis El Manar – (Adel.Khalfallah@fst.mu.tn).

Durée

4.½ mois, du 02 février au 10 juin 2007.

Table des matières

Remerciements	3
Table des illustrations	3
Liste des tableaux	4
Glossaire	4
I Introduction générale.....	7
I - 1 Objectifs du document	10
I - 2 Conditions de réalisation du stage.....	11
II Cadre de référence et problématique	14
II - 1 Un sujet, un stage, un mémoire	14
II - 2 Problématique et objectifs du projet.....	15
II - 3 Notre vision du système	18
II - 4 Plan du mémoire.....	19
III Contexte des systèmes temps réel embarqués dans l'automobile	20
III - 1 Introduction : Application, système informatique, environnement.....	20
III - 2 Qualité de service et contextes temporels des systèmes temps réel	34
III - 3 Du temps réel embarqué au temps réel embarqué appliqué à l'automobile.....	35
III - 4 Méthodologie de développement logiciel dans l'automobile	41
III - 5 La distribution dans l'automobile	45
III - 6 Une architecture cible abstraite	51
III - 7 De l'intergiciel à l'intergiciel temps réel appliqué à l'automobile.....	53
III - 8 Dimensionnement temps réel de l'architecture cible abstraite.....	59
III - 9 Conclusion sur le contexte	62
IV Réalisation.....	64
IV - 1 Méthode et contraintes du processus de développement.....	64
IV - 2 Spécification générale	67
IV - 3 Modélisation UML	72
IV - 4 Le générateur de code	81
IV - 5 L'intergiciel.....	84
IV - 6 Développement et déploiement.....	99
IV - 7 Conclusion sur la réalisation	108
V Conclusion et perspectives.....	110
V - 1 Les résultats du projet	111
V - 2 Conclusion.....	113
V - 3 Perspectives	115
VI Annexes	119
VII Bibliographie.....	132

Remerciements

Je remercie tout d'abord les membres du jury qui me font honneur en jugeant ce travail.

Ce mémoire de projet de fin d'études d'ingénieur en informatique est le fruit d'un travail mené au sein de l'équipe *Temps Réel et InterOpérabilité* (TRIO) du laboratoire lorrain de recherche en Informatique et ses applications (LORIA) dirigée par le Professeur **Françoise Simonot-Lion**, à qui je voudrais exprimer toute ma gratitude pour sa confiance et les moyens techniques et scientifiques qu'elle a mise à ma disposition et sans qui ce travail n'aurait jamais été envisageable...« شكرا ».

Je tiens à exprimer mes plus sincères remerciements à mon tuteur Monsieur **Xavier Rebeuf**, maître de conférence à l'École Nationale Supérieure d'Électricité et de Mécanique de Nancy, pour avoir encadré ce travail de fin d'études. Sa disponibilité, ses conseils éclairés et son soutien m'ont été d'une aide inestimable et ont largement contribué à ma formation. Je n'oublierais certainement jamais cette extraordinaire collaboration...« большое спасибо ».

Je souhaite également remercier Monsieur **Adel Khalfallah**, maître de conférence à l'institut supérieur d'Informatique et mon maître de stage, et qu'il trouve ici l'expression de mon indéfectible reconnaissance pour cette merveilleuse opportunité qu'il m'a offerte, en me permettant d'intégrer cette équipe. Sans oublier toutes ces années où il a été mon professeur à l'ISI, et pendant lesquelles il a si bien su nous transmettre sa rigueur scientifique et certainement le goût de la perfection... « un grand merci ».

Enfin, je remercie sincèrement tous les membres de l'équipe, pour leur chaleureux accueil et pour les bons moments passés ensemble et la bonne ambiance qu'ils entretiennent au sein de TRIO ...« grazie a tutti ».

Je dédie ce travail à ma mère Natalia et à ma sœur Anissa pour leur indéfectible soutien, les marques d'amour et les sacrifices dont elles ne se sont jamais départies ...« я вас люблю »... Un grand merci à mon ami Achraf ... Clin d'œil particulier à ma chère Alevtina.

*Regretter ce que l'on aime est un bien, ... [Jean de La Bruyère]
Et Bien ... À tous ceux que j'ai perdu et qu'aujourd'hui je regrette.*

Table des illustrations

Figure 1 : Méthodologie de construction de l'intergiciel.	18
Figure 2 : Logo du projet <i>GenIETeR</i>	18
Figure 3 : Application temps réel.	20
Figure 4 : Un système Temps Réel - Interaction procédé contrôleur.	23
Figure 5 : Architecture générale d'une application temps réel monoprocesseur.	27
Figure 6 : Les systèmes d'exploitation temps réel.	28
Figure 7 : Modélisation d'une tâche périodique.	32
Figure 8 : Enfouissement d'un contrôleur d'injection diesel de Scania. [MRTC05].	35
Figure 9 : De l'électricité à l'électronique dans l'automobile. [PSA07].	36
Figure 10 : Architecture conventionnelle. [ETR05].	38
Figure 11 : Architecture centralisée. [ETR05].	39
Figure 12 : Architecture faiblement distribuée.	39
Figure 13 : Architecture multi-calculateurs.	40
Figure 14 : Architecture fortement distribuée.	40
Figure 15 : Exemple d'un système embarqué de contrôle de freinage [MRTC05].	41
Figure 16 : Augmentation de la taille du code. [PSA07].	42
Figure 17 : Croissance de l'importance du logiciel dans l'automobile. [HAR04].	43
Figure 18 : Flot de conception logicielle global. [ALB02].	44
Figure 19 : Intervenants et rôles dans le processus de réalisation logicielle.	45
Figure 20 : Modèle d'interaction distante ente 2 tâches.	50
Figure 21 : Une abstraction de l'architecture cible.	52
Figure 22 : (a, b) Couche intergiciel dans l'organisation d'un système.	53
Figure 23 : Couche intergiciel dans l'organisation d'un système distribué.	54
Figure 24 : Contrainte de fraîcheur d'un signal (exécutif préemptif).	60
Figure 25 : Modèle logiciel d'Interception/Interposition.	61
Figure 26 : Temps de bout en bout.	62
Figure 27 : Situation de la conception générique dans 2TUP. [ROQ03].	65
Figure 28 : Synoptique des différents résultats d'un test d'ordonnançabilité.	69
Figure 29 : Diagramme des cas d'utilisation.	73
Figure 30 : Diagramme de séquences (utilisation de l'intergiciel).	76
Figure 31 : Organisation en couches de l'intergiciel.	78
Figure 32 : Organisation du modèle logique du système.	78
Figure 33 : Modèle structurel.	80
Figure 34 : Diagramme de classes du générateur.	82
Figure 35 : Illustration d'une tâche configurée en GMF. [RSM06].	84
Figure 36 : Taxonomie des structures de données du système.	85
Figure 37 : Modèle d'interaction tâches applicatives/intergiciel.	86
Figure 38 : Modèle générique de l'intergiciel (1/2).	87
Figure 39 : Modèle générique de l'intergiciel (2/2).	88
Figure 40 : Diagramme de séquence du mécanisme de prélude (1/3).	90
Figure 41 : Diagramme de séquence du mécanisme de prélude (2/3).	91
Figure 42 : Diagramme de séquence du mécanisme de prélude (3/3).	92
Figure 43 : Diagramme de séquence du mécanisme de postlude (1/3).	93
Figure 44 : Diagramme de séquence du mécanisme de postlude (2/3).	93
Figure 45 : Diagramme de séquence du mécanisme de postlude (3/3).	94
Figure 46 : Ebauche du modèle fonctionnel pour la communication locale.	95
Figure 47 : Modèle de tâches de l'intergiciel.	96
Figure 48 : Modèle de gestion d'erreur pour l'intergiciel.	97
Figure 49 : Organisation des <i>design patterns</i> dans l'intergiciel.	98
Figure 50 : Programmation et déboguage in situ pour PIC. [MOREN].	103
Figure 51 : Solution d'architecture pour le développement sur CAN. [WCANS].	104
Figure 52 : La méthodologie de développement globale.	106
Figure 53 : Courbe d'efforts impartis au projet.	114
Figure 54 : Passerelle distribuée. [WVECT].	118

Liste des tableaux

Tableau 1 : Classification des réseaux multipléxé dans l'automobile. [ETR05]	47
Tableau 2 : Preuve de la limite du temps de bout en bout.....	62
Tableau 3 : Structure et nomenclature de la spécification du trafic.	70

Glossaire

[1] *Système temps réel embarqué* : C'est tout d'abord un système dont le comportement dépend, non seulement de l'exactitude des traitements effectués, mais également du temps où les résultats de ces traitements sont produits. Il est dit « temps réel » parce que ses actions logicielles dirigent les activités d'un processus en cours d'exécution. Si de plus, il est physiquement intégré à l'environnement qu'il contrôle et qu'il est soumis aux mêmes contraintes physiques (température, pression, etc.) que son environnement, il est alors dit « embarqué ».

[2] *Système réparti* : On peut employer le terme de système réparti ou distribué lorsque le système dispose d'un ensemble d'entités communicantes, installées sur une architecture de processeurs reliés par un réseau de communication, dans le but de résoudre en coopération une fonctionnalité applicative commune.

[3] *Intergiciel* : Un intergiciel (*Middleware*) désigne un logiciel présent sur un noeud d'un système réparti et servant d'intermédiaire de communication entre des applications complexes, distribuées sur un réseau d'échange d'informations. Il offre pour cela les abstractions d'un modèle de distribution des services de haut niveau liés aux besoins de communication de ces composants applicatifs présents sur ce nœud.

[4] *Composant logiciel ou applicatif* : La définition qu'on pourrait retenir pour un composant logiciel se base principalement sur la deuxième phrase de celle donnée par [SZYP98] : «*Un composant logiciel est une unité de composition spécifiant, par contrats, ses interfaces*

(fournies et requises) et ses dépendances explicites aux contextes. Un composant logiciel peut être déployé indépendamment et peut être sujet de composition par un tiers pour la conception d'applications logicielles». Nous utilisons le terme composant logiciel par analogie avec les composants utilisés dans l'industrie de l'électronique. Contrairement au contexte des CBSE [BEL01] (*Component-Based Software Engineering*) nous n'en retiendrons que les notions liées à sa faculté d'adaptation. Nous ferons alors le rapprochement avec le terme «composant logiciel réutilisable» pour dire que c'est une entité logicielle capable de résoudre un même problème dans des contextes différents. Paradoxalement, c'est un composant avec peu de dépendance.

[5] *CAN* : *Control Area Network* [CAN] est un bus multiplexé/réseau de terrain à faibles coûts. C'est d'une part, un réseau de terrain à moyens débits respectant le modèle d'interconnexion des systèmes ouverts OSI [WOSI] de l'ISO (ISO 7498) et qui est particulièrement adapté à un fonctionnement dans un environnement critique et géographiquement limité. D'autre part, c'est un protocole de communication série qui supporte des systèmes temps réel avec un haut niveau de fiabilité.

[6] *OSEK/VDX OS* : OSEK/VDX (cf. Annexe 1) est un système d'exploitation standard [WOSEK] dans l'industrie automobile européenne. Il est spécialement adapté aux applications temps réel embarquées qui exigent une utilisation minimale de la mémoire et du temps d'une unité centrale.

[7] *Design patterns* : Un patron de conception est un concept destiné à résoudre les problèmes récurrents du génie logiciel suivant le paradigme objet. Les patrons de conception décrivent des solutions standards pour répondre à des problèmes d'architecture et de conception des logiciels par une formalisation de bonnes pratiques, ce qui signifie que l'on privilégie les solutions éprouvées (un patron de conception n'est considéré comme « prouvé » qu'une fois qu'il a été utilisé avec succès au moins dans trois cas).

[8] *Frame packing* : Techniques et algorithmes [TIN95, TIN93] de validation d'une collection d'objets (dans notre cas, un ensemble de tâches et/ou de trames) reposant sur des critères analytiques ou des heuristiques (contraintes temporelles) pour conclure à la construction d'une séquence d'ordonnancement (ou configuration ordonnancée) de cet ensemble et à sa faisabilité. L'utilisation de ces méthodes se justifie dès lors que l'on souhaite ordonnancer des applications fortement couplées (forte utilisation de ressources et de synchronisations). En effet, la construction d'une séquence d'ordonnancement permet de s'assurer du respect des

contraintes temporelles pendant sa construction et donc de valider simultanément la séquence produite puisque celle-ci sera utilisée telle quelle dans un séquenceur (dispositif d'analyse). Ces algorithmes [SAND00] sont une adaptation des algorithmes de *bin packing* visant à trouver l'arrangement optimal (ou un arrangement) le plus économique possible (économie des ressources matérielles) pour cette séquence.

[9] *Modèle de répartition* : Un modèle de répartition est un ensemble de définitions d'interactions entre des entités abstraites, associées à une projection de ces entités sur les noeuds d'un système réparti.

[10] *Générateur de code* : Un générateur de code est un programme qui partant d'une configuration donnée produit du code *ad hoc* (spécifique) optimisé pour cette configuration. En d'autres termes, la génération permet de généraliser les bonnes pratiques capitalisées et met à disposition des développeurs les éléments nécessaires aux développements additionnels. Les développements complémentaires sont réalisés indépendamment du code généré et non pas sur les éléments générés garantissant ainsi la possibilité de re-générer certains éléments lors de l'apparition de nouvelles bonnes pratiques.

[11] *Intergiciel générique* : Un intergiciel générique est un intergiciel dans lequel certains modules sont rendus indépendants du choix de la structuration d'une répartition particulière, et réalisés sous une forme réutilisable. L'adaptation ou la personnalisation est alors obtenue en concrétisant ces modules génériques avec des composants de personnalisation.

[12] *Intergiciel configurable ou adaptable* : Un intergiciel est dit configurable lorsque les composants qu'il intègre peuvent être choisis et adaptés leur comportement en fonction des besoins de l'application et des fonctionnalités offertes par l'environnement.

I Introduction générale

Les récentes controverses sur certains des incidents survenus sur des véhicules équipés de régulateurs de vitesse (le *pédalgate* Renault entre autres) dont la vocation est d'éviter aux conducteurs de confondre frein et embrayage (seule hypothèse avancée à ce jour par Renault pour justifier les nombreux accidents) ont beaucoup défrayé la chronique. Le manque de moyen d'urgence donné au conducteur indépendamment de tout système électronique pour lui permettre de reprendre le contrôle de son véhicule en cas de dysfonctionnement, est un fait. À qui attribuer la responsabilité : au constructeur ou au conducteur ?

En d'autres termes la loi de Murphy s'applique aussi à l'électronique, « *tout ce qui peut potentiellement dysfonctionner dysfonctionnera* » [WREN], et cela s'applique nécessairement à l'industrie automobile comme aux autres industries.

Conscients que les dysfonctionnements électroniques existent, et que la vie de leurs clients dépend de la qualité de leurs produits, les constructeurs concentrent tous leurs efforts sur l'amélioration de la sécurité des véhicules. Et par conséquent, améliorent leurs procédés de conception et de fabrication, tout en réduisant les coûts et les délais.

Parce que ce genre de système est appelé « système réactif », il est soumis à des contraintes temporelles fortes (temps de réponse borné, fréquence d'échantillonnage d'un signal, etc.), spécifiées lors de la conception. La vérification de celles-ci avant utilisation est nécessaire pour la validation du système. La criticité de ces systèmes se traduit alors, non seulement par leur fiabilité, mais également par leur rigoureuse réactivité.

Un autre objectif poursuivi par les constructeurs est celui d'offrir aux utilisateurs un plus grand confort d'utilisation, ainsi que des services nouveaux. Tout ceci nécessite la mise en place de plateformes de développement performantes, qui traitent d'une part des aspects fonctionnels, et d'autre part des aspects opérationnels de construction. C'est ce second point qui nous intéresse dans le présent mémoire. La validation opérationnelle d'une application sous-entend la nécessité de procéder à des vérifications temporelles quantitatives. La sécurité nécessite que l'on puisse garantir qu'entre toute évolution du procédé et la commande produite en réaction par le système, s'écoulera une durée limitée.

En d'autres termes, il s'agit de garantir que le temps de réponse de l'Informatique soit, au moins, aussi rapide que le procédé qu'elle contrôle. Cette propriété doit être garantie avant et après la mise en exploitation de l'application. L'application concernée consiste en un ensemble de tâches, chacune d'elles ayant vocation à traiter une information reçue du procédé

(la réception de l'information pouvant être assimilée à l'évènement) ou à réagir à un évènement spécifique. L'application est donc exposée comme un ensemble d'activités, décrites par des tâches, qui doivent s'exécuter de manière répétitive tout au long de la vie de l'application. Par ailleurs, celle-ci s'exécute sur une architecture souvent dédiée, qui peut être monoprocesseur, multiprocesseur ou répartie. Dans ce mémoire, nous considérerons des systèmes distribués monoprocesseur. L'exécution des tâches doit donc être répartie entre les différents processeurs (ou sites dans la terminologie des systèmes distribués) du système pour réaliser les activités de l'application concernée.

Ainsi, le développement d'applications distribuées temps réel nécessite aujourd'hui l'introduction de paradigmes existants d'origines diverses (notamment issus du génie du logiciel). De manière très classique, la répartition de l'architecture est à la charge d'une couche logicielle spécifique appelée intergiciel [3] (*Middleware*). Pour ne pas alourdir notre présentation, nous parlerons dans cette introduction de plateforme ou d'intergiciel, en gardant en mémoire le caractère plus ou moins étendu que peuvent revêtir ces notions. La notion la plus ancienne, celle de plateforme, désigne un ensemble de logiciels dont l'objectif est de servir de fondation au développement et au fonctionnement de nouvelles applications. La notion d'intergiciel est, elle, beaucoup plus récente et désigne plus particulièrement un ensemble de services réalisés entre le système d'exploitation et l'application. Parmi les intergiciels les plus populaires ces dernières années, l'on peut citer les intergiciels orientés communication, notamment ceux basés sur la norme CORBA [OMG02]. Toutefois, peu d'entre eux permettent de garantir une qualité de service temporelle.

Les intergiciels rendent les applications réparties indépendantes d'un environnement particulier, mais au prix d'une surcharge de l'architecture logicielle de ces applications. Ils masquent ainsi une partie de la complexité de la programmation répartie en permettant à un développeur de manipuler de la même manière des entités locales et distantes. De plus, la spécificité vient ici du fait que l'objectif premier (voire même unique) de l'intergiciel est de garantir le respect des contraintes temporelles induites par le procédé.

La détermination des fonctions fondamentales de l'intergiciel et leur isolement dans des modules indépendants permet également de rendre celui-ci adaptable à des besoins et à des environnements variés par assemblage de « briques logicielles » [4] encapsulant des mécanismes prédéfinis. Cet assemblage est effectué au moyen de générateurs plus ou moins automatisés, qui réalisent les abstractions d'un modèle de répartition et permettent aux tâches applicatives d'interagir. Selon la nature et la richesse des « briques » fournies, on pourra parler de « kit de développement » ou de « canevas » (*framework*).

Des architectures d'intergiciels génériques [11] ont été par ailleurs proposées, introduisant la notion de paramétrisation. À partir de ces architectures d'intergiciels génériques, nous pouvons retenir un certain nombre de fonctions communes de communication, mais néanmoins dépendant d'un choix de modèle de communication : une communication asynchrone orientée messages entre tâches, inspiré de celui des MOMs (*Message-Oriented Middlewares*). Il est alors possible de mettre en oeuvre ces fonctions sous forme d'une couche neutre. Cette couche neutre est ensuite dotée de services, qui sont une façade offrant une interface standard à toutes les couches applicatives. La séparation entre couche neutre et noyau de l'intergiciel permet de découpler, tout en garantissant le respect des contraintes temporelles fournies par les exigences opérationnelles, les deux aspects architecturaux : une facette applicative présentée aux applicatifs de l'utilisateur, et une facette protocolaire présentée aux réseaux.

Par garantir, il faut concevoir ; pour concevoir, il faut formaliser. Il est donc nécessaire de modéliser l'application, et dans le cas qui nous occupe, c'est une modélisation temporelle de l'application qui s'impose. Une telle modélisation est utilisée depuis très longtemps dans la littérature, et les travaux (qui nous intéressent) qui s'appuient sur une modélisation souvent appelée « modélisation de Liu Layland » [LAY73]. Il s'agit de proposer une abstraction temporelle de fonctionnement de l'application. Une application consiste en un ensemble de tâches, chacune de ces tâches est décrite à l'aide de paramètres temporels qui peuvent indiquer (le cas échéant) l'instant de création de la tâche, sa période (s'il s'agit d'une tâche de contrôle, par essence même périodique), son délai critique (qui quantifie la « durée adaptée » entre la création d'une instance et sa terminaison), et sa durée. Parmi tous ces paramètres, les trois premiers sont issus du cahier des charges, et dépendent du concepteur de l'application. Mais le dernier dépend directement du code de la tâche. Comme il est difficile (voire même dénué de sens) de déterminer une durée d'exécution unique et exacte, nous travaillons en fait avec un majorant de la durée (WCET, *Worst Case Execution Time*). Mais cette approche « pire cas » masque la réalité des choses. Quand les variations de durées sont dues à l'indéterminisme de la gestion de la mémoire par exemple, il est difficile de faire autrement. Mais cette vue de durée unique masque également les variations liées au code lui-même. Les études réalisées en masquant les communications distantes sont alors d'une part sur-contraintes (particulièrement dans le cas où le protocole de communication est pris en compte) et d'autre part incapables de décrire le fonctionnement réel de l'application. Alors, comment modéliser et construire un générateur d'intergiciel temps réel embarqués dans l'automobile ?

... Et c'est ainsi que commence ce mémoire...

I - 1 Objectifs du document

La réalisation de ce générateur d'intergiciels temps réel embarqués dans l'automobile intervient dans le cadre du stage du projet de fin d'études pour l'obtention du diplôme d'ingénieur en génie du logiciel et des systèmes d'information. Ce sujet m'a valu un déplacement à Nancy (France) et une entrée au Laboratoire Lorrain de Recherche en Informatique et ses Applications pour un séjour de plus de quatre mois. Le document que vous avez entre les mains se veut être un mémoire et le rapport témoin de ce stage. Ce mémoire est destiné à l'honorable jury chargé de l'évaluation du stage, ainsi qu'aux encadrants de l'organisme qui y retrouveront les acquis et le résumé de mes activités pour le sujet qui me fût confié.

Il s'adresse par ailleurs aux concepteurs et développeurs qui me reliaieront, et qui peut-être y trouveront un intérêt, ou à tout autre personne qui s'intéresserait à ce sujet.

A ces personnes, je souhaite une bonne lecture.

Le stage touchant à sa fin, j'ai voulu et j'ai le devoir de réaliser le présent document à l'échelle d'un rapport de mes réalisations durant toute la période du stage. Et d'autre part le considérer comme un témoignage des compétences et connaissances acquises lors de cette nouvelle expérience professionnelle. Une expérience au sein d'un laboratoire de recherche en Informatique qui, s'est révélé extrêmement instructive.

Voici, les principaux objectifs de ce document :

- Comprendre les enjeux des missions qui m'ont été confiées,
- Synthétiser les concepts théoriques que j'ai pu appréhender durant ce stage au travers de l'étude bibliographique,
- Montrer comment j'ai résolu les situations délicates en y confrontant mes acquis théoriques,
- Présenter les réalisations effectuées et la méthodologie de mise en oeuvre de ce système applicatif,
- Procéder à une analyse des fonctionnalités requises pour l'utilisation de notre système applicatif et en présenter les conclusions,
- Décrire les principales fonctions et les scénarii de transactions du système applicatif,
- Déterminer les exigences du système applicatif, les objectifs généraux et les critères de réussite qui permettront d'assurer son bon fonctionnement et de juger de l'avancement du projet,
- Déterminer les conditions et les contraintes opérationnelles reliées au système défini afin d'évaluer les délais de déploiement,
- Présenter les réalisations que j'ai pus effectuer durant ce stage,


- Mettre en exergue mes appréciations pour ce stage et ceux de mes répondants, ainsi que les conclusions de mon travail,
- Faire le point de mes nouvelles connaissances sur le métier qui régit le secteur d'activité de l'organisme qui a bien voulu m'ouvrir ses portes,
- Prouver en quoi le stage a réussi à développer mes compétences (rigueur, esprit d'équipe, créativité, gestion de projet),
- Prouver l'utilité de mon choix pour ce stage,
- Faire le tour de cette expérience qui m'a fait découvrir les prémisses de la recherche scientifique et qui n'a fait que renforcer mes motivations pour un avenir de chercheur.

I - 2 Conditions de réalisation du stage

C'est pour toute la période de ces quatre mois et demi, au sein du LORIA, que j'ai pu confronter les compétences techniques du génie logiciel acquises durant ma formation académique à l'Institut Supérieur d'Informatique, à la réalité d'un environnement où la rigueur, l'esprit d'équipe et l'innovation doivent rallier l'esprit de recherche, afin de faire avancer la science.

I - 2 - 1 Présentation de l'organisme d'accueil : le LORIA



Le , Laboratoire Lorrain de Recherche en Informatique et ses Applications, est une Unité Mixte de Recherche - UMR 7503 - commune à plusieurs établissements partenaires :

- **CNRS** - Centre National de Recherche Scientifique,
- **INRIA** - Institut National de Recherche en Informatique et en Automatique,
- **INPL** - Institut National Polytechnique de Lorraine,
- **UHP** - Université Henri Poincaré - Nancy 1,
- **NANCY 2** - Université Nancy 2.

I - 2 - 2 Identité du LORIA

La création de cette unité a été officialisée le 19 décembre 1997 par la signature du contrat quadriennal avec le ministère de l'éducation nationale, de la recherche et de la technologie et par une convention entre les cinq partenaires. Cette unité, renouvelée en 2001, succède ainsi au CRIN (Centre de Recherche en Informatique de Nancy), et associe les équipes communes entre celui-ci et l'Unité de Recherche INRIA Lorraine.

Le LORIA est un Laboratoire de plus de 450 personnes parmi lesquelles :

- cent cinquante chercheurs et enseignants-chercheurs,
- un tiers de doctorants et post doctorants,
- des ingénieurs, techniciens et personnels administratifs,

organisés en équipes de recherche et services de soutien à la recherche.

C'est aussi chaque année :

- Une trentaine de chercheurs étrangers invités,
- Des coopérations internationales avec des pays des cinq continents,
- Une quarantaine de contrats industriels.

Les missions du LORIA sont essentiellement :

- La **recherche** fondamentale et appliquée au niveau international dans le domaine des sciences et technologies de l'information et de la communication,
- La **formation par la recherche** en partenariat avec les universités lorraines,
- Le **transfert technologique** par le biais de partenariats industriels et au travers de la création d'entreprises.

I - 2 - 3 **Identité de l'équipe d'accueil : TRIO, *the Dream Team***

TRIO (Temps Réel et InterOpérabilité) est l'une des équipes de recherche du LORIA, son objectif premier est le développement par l'innovation des techniques et des méthodes pour la conception, la validation et l'optimisation des applications distribuées temps réel. Pour se faire, ses travaux sont structurés selon deux axes complémentaires :

- *La spécification de mécanismes exécutifs* (protocoles, ordonnanceurs, intergiciels, ...) offrant des services et une qualité de service garantissant le respect des contraintes de sûreté de fonctionnement, en particulier des contraintes de temps ; ceci intègre la détection de fautes, la signalisation de fautes et la tolérance aux fautes.
- *La modélisation, l'analyse et l'évaluation* d'applications temps réel distribuées en vue d'exploiter les modèles pour des activités de validation et /ou de configuration automatique de tout ou partie des applications.

Les problématiques à résoudre proviennent principalement de trois caractéristiques des applications ciblées :

- Il s'agit de systèmes à événements discrets intégrant des caractéristiques temporelles (performances du matériel support, propriétés temporelles); ceci accroît la complexité de leur modélisation et de leur analyse. Aussi, une partie des objectifs de recherche consiste à maîtriser cette complexité en faisant un compromis entre précision d'un modèle et son exploitabilité.

- L'environnement de ces systèmes peut être cause de perturbations. Les travaux doivent en particulier prendre en compte l'impact d'un environnement incertain (par exemple, l'influence des perturbations électromagnétiques sur le support matériel des applications) sur le respect des propriétés requises. Des approches stochastiques doivent, alors, être mises en oeuvre.
- Enfin, la caractéristique principale des travaux repose sur le fait qu'ils traitent de temps physique continu.

Les trois axes mentionnés ci-dessus contribuent à couvrir l'ensemble des activités allant de la résolution de problèmes théoriques (modélisation et analyse de systèmes temporisés à événements discrets) à leur mise en oeuvre en applications industrielles à échelle réelle, et en particulier dans des systèmes électroniques embarqués dans le domaine de l'automobile. Certains résultats ont conduit à des prototypes et à des collaborations fructueuses avec l'industrie automobile.

L'équipe-projet TRIO maintient des relations à la fois avec des institutions académiques et des partenaires industriels, en France aussi bien qu'à l'étranger :

- National Laboratory of Industrial Control Technology of Zhejiang University.
- MOSIC team, Université de Tunis El Manar - Tunisie.
- Anatoly Manita.
- Joël Goossens, Université Libre de Bruxelles - Belgique.
- Kottering University.

Elle collabore activement dans des projets et des programmes aussi bien européens que nationaux au travers de contrats conclus avec ses partenaires :

- Programme européen : NNE 2001-00825, REMPLI.
- Projets nationaux : ANR - PREDIT Project SCARLET, ARA SSIA SAFE_NECS.
- Action régionale : QSL Operation TT_SAFETY - Evaluation of the Safety of Systems Distributed onto a TDMA-Based Network and Subject to "Agressive Environment".

L'équipe TRIO a monté, en 2001, une équipe de recherche technologique en coopération avec le groupe PSA Peugeot-Citroën, CARMELS (CAractérisation des Réseaux embarqués dans l'Automobile et Mécanismes En Ligne pour leur Sûreté) supportée par le ministère de la recherche. La thématique traitée concerne l'évaluation des caractéristiques de sûreté de fonctionnement des architectures de communications supportant les applications embarquées X-by-Wire dans l'automobile.

II Cadre de référence et problématique

II - 1 Un sujet, un stage, un mémoire

Le contexte de notre travail émane directement des propositions de la méthodologie présentée dans la thèse [RSM06], et qui concerne le développement d'un intergiciel à embarquer [1] dans l'automobile. En raison de l'impact que peut avoir un intergiciel [3] sur le bon fonctionnement des fonctions automobiles réparties, et en particulier sur le respect des contraintes temporelles imposées aux échanges entre ces fonctions, l'auteur propose la construction d'un intergiciel devant fournir des services de communication asynchrone au niveau applicatif. Cette construction repose sur une génération d'intergiciels « légers » (rapide et peu encombrant) et adaptés [12] à un type d'application, et ce par assemblage automatisé de composants prédéfinis, optimisant ainsi la consommation des ressources.

Les objectifs traités dans ces travaux sont :

- Offrir un ensemble de services de communication distante permettant aux entités applicatives d'effectuer des échanges entre elles tout en garantissant le respect des propriétés liées à la criticité du système globale ;
- Cacher la distribution, en ce que les services de communication fournis doivent être indépendants de la localisation des intervenants dans les échanges ;
- Garantir une qualité de service temporelle, où le respect de toutes les contraintes temporelles soit assuré.

Ces objectifs fournissent deux axes importants qui ont été explorés :

- D'une part, l'intergiciel sera un logiciel de plus dans chaque ECU provoquant un encombrement logiciel dont il est impératif d'étudier l'impact ;
- D'autre part, la garantie du respect des contraintes impose à l'intergiciel un comportement totalement déterministe.

La méthodologie proposée traite alors de l'implémentation et la configuration d'un intergiciel automobile optimal. En ce qui concerne l'implémentation de cet intergiciel, la définition d'un ensemble optimal d'entités, dans le sens où il est adapté aux propriétés du support d'exécution, a été faite pour qu'elles puissent être capables de représenter l'intergiciel sur chaque ECU cible avec une moindre utilisation de ressources (par *frame packing* [8]). Le volet suivant, la configuration de l'intergiciel, essaye d'attribuer des paramètres au système de manière à ce que les contraintes temporelles et fonctionnelles imposées soient respectées.

Par conséquent, la thèse ouvrait le champ à une perspective immédiate d'une implémentation sur une plateforme réelle. Ce travail, qui nous a été proposé et que nous nous

sommes approprié dans le cadre du stage de notre projet de fin d'études, a pour objectif de vérifier l'implémentabilité de l'approche proposée, de quantifier l'influence de l'architecture logicielle basée sur des *design patterns* sur la performance de l'intergiciel et d'évaluer la possibilité d'automatiser l'implémentation de l'empreinte d'un intergiciel sur chaque ECU.

II - 2 Problématique et objectifs du projet

Un véhicule est traversé de part et d'autre par des faisceaux de fils électrique afin de transmettre énergie et informations à l'ensemble des différents organes électriques (des calculateurs ou des unités de contrôle électroniques, ou leur acronyme anglais ECU : *Electronic Control Unit*, ou des capteurs, etc.) composant le système. L'augmentation croissante des fonctions dans l'automobile fût à l'origine de l'accroissement considérable du nombre des liaisons filaires, et par conséquent, a augmenté le poids des véhicules ainsi que leurs coûts. Une mutation de ces liaisons vers des réseaux multiplexés permet de limiter ces contraintes tout en introduisant d'autres difficultés à surmonter, comme le dimensionnement temps réel et la réactivité.

Certes l'introduction des réseaux permet de réduire la quantité de liaisons filaires reliant les nombreux organes d'un véhicule, mais rajoute une autre caractéristique à ces systèmes : la distribution [2]. Par ailleurs, cette situation provoque des modèles de répartition [9] spécifiques à chaque système.

L'interaction des différents programmes participant à un système distribué pose également des problèmes algorithmiques particuliers. L'hétérogénéité des architectures matérielles et logicielles au sein d'un système réparti rend nécessaire la mise en oeuvre de composants d'adaptation : la répartition est une source de complexité qui s'ajoute aux problèmes propres à une application temps réel.

Historiquement, il incombait aux développeurs d'applications de résoudre eux-mêmes les problèmes liés à la répartition : fiabilisation des communications, algorithmes répartis, hétérogénéité des environnements, etc. En particulier, ils devaient programmer directement le matériel de communication de chaque machine, s'assurer de la coordination entre composants, et choisir une représentation de l'information commune acceptée par tous les participants.

Pour faciliter la conception et le développement d'applications réparties, des modèles de répartition ont été proposés, c'est-à-dire des ensembles d'abstractions permettant de spécifier les interactions entre composants suivant leur organisation logique et leur déploiement, plutôt qu'en termes de moyens techniques utilisés pour transporter l'information de l'un à l'autre.

L'utilisation de modèles de répartition, mis en oeuvre par des intergiciels (*Middlewares*), permet donc aux développeurs de se concentrer sur les aspects fonctionnels de l'application. Elle affranchit le programmeur de certaines contraintes, telles que la gestion de la communication entre noeuds, ou l'hétérogénéité des architectures matérielles et logicielles.

Entre les organes de l'automobile, les échanges se font par des signaux au niveau applicatif et par des trames au niveau du protocole réseau. Par exemple, certaines applications utilisent des intergiciels basés sur le passage de messages (MOM, *Message-Oriented Middlewares*), d'autres les objets distants (ORB, *Object Request Broker*) ou la mémoire partagée répartie.

Les intergiciels suppriment la dépendance d'une application répartie vis-à-vis d'un environnement matériel et d'un système d'exploitation, mais introduisent en contrepartie un encombrement supplémentaire. Cette situation constitue le paradoxe de l'intergiciel

Cependant, les exigences en temps réel, la variété des modèles de répartition et des intergiciels introduit une nouvelle problématique.

De plus, le fort besoin en prévisibilité (et par conséquent le déterminisme) des systèmes temps réel embarqués dans l'automobile, du fait de leur criticité, implique impérativement que les comportements des applications développées soient connus au départ et ne changent pas au cours de la vie du système.

Il paraît donc inenvisageable d'utiliser des intergiciels « génériques » [11] (spécifiés une seule fois et instanciés à la demande) car de surcroît, à défaut de disposer des ressources matérielles suffisantes, l'optimisation est primordial.

Les deux caractéristiques des systèmes temps réel embarqués, de prévisibilité et de limitation de ressource, rendent possible l'utilisation des techniques de validation hors-ligne dites *tests d'ordonnancements* sur le comportement de ces systèmes distribués pour en valider les contraintes temporelles tels qu'elles sont imposées par le cahier des charges.

Ces tests permettront de statuer, non seulement sur la faisabilité des applications, mais aussi de déterminer une configuration des échanges entre les entités du réseau. Ils concluent sur l'aptitude, d'une part, de l'architecture matérielle à supporter des exigences d'utilisations optimales des ressources en terme de consommation de bande passante, et d'autre part celle d'un intergiciel (exigences de qualité de service) et à s'assurer de la faisabilité de la communication distante, et à conclure quant à la sûreté de l'application temps réel.

Comme nous l'avons précédemment fait remarquer, [RSM06] a défrâché le terrain en répondant à :

- Comment spécifier une architecture d'implémentation ? (quels sont les composants logiciels et comment interagissent-ils entre eux) ;

- Comment construire une configuration qui respecte les contraintes ? (configuration de messagerie, déploiement des composants logiciels au sein des tâches, configuration des tâches).

La méthodologie qu'il propose est basée sur l'utilisation de schémas d'assemblage [7] de composants conçus dans un même langage, mais l'ingénierie de conception des applications logicielles à base de composants logiciels [4] est ainsi née du besoin de faciliter, au maximum, la réutilisation du code, ce qui permet de concevoir des applications robustes, réduites, légers (peu encombrant) et très rapidement.

L'idée de développer des applications par utilisation, assemblage ou réutilisation de composant est ancienne, mais les débuts de sa mise en pratique à grande échelle sont récents. Le formalisme de programmation par objets est déjà au coeur des possibilités actuelles dans ce domaine (architectures logicielles génériques ou extensibles : *frameworks*, architectures pour l'interopérabilité : CORBA [OMG02], COM/OLE, composants répartis : *JAVA Beans*, contrôles Active/X), mais la prise en compte du temps réel parmi les propriétés nécessaires de tels composants n'est pas encore usuelle.

Une barrière technologique s'oppose cependant à l'émergence de tels intergiciels : l'automatisation de leur construction par assemblage adapté à la configuration du support d'exécution et du modèle de répartition en minimisant la surcharge introduite par l'exécution des composants constituant l'intergiciel. Et le tout pour le temps réel.

L'objectif du stage est de mettre en oeuvre de telles techniques pour la génération de code [10] d'intergiciels de communication embarqués pour l'automobile. Plus précisément, cette approche doit permettre de concevoir un générateur d'un ensemble de tâches implémentant cette couche de communication pour un type d'application donné. Pour cela, elle prend en compte l'ensemble des événements requis et offerts par l'application tout en assurant la qualité de service requise par cette dernière (propriétés temps réel, garanties de convergence des exécutions) et minimisant la surcharge due à l'exécution de l'intergiciel.

Le processus de réalisation, illustré par la Figure 1, que nous suivrons est : partant d'une spécification temporelle donnée des interactions entre entités applicatives (trafic des signaux et messagerie des trames, *Temporal traffic specification*), il devra générer un intergiciel (*Concrete Middleware model*) spécifiquement configuré à l'environnement (*Environmental properties specification*) sur lequel il sera déployé.

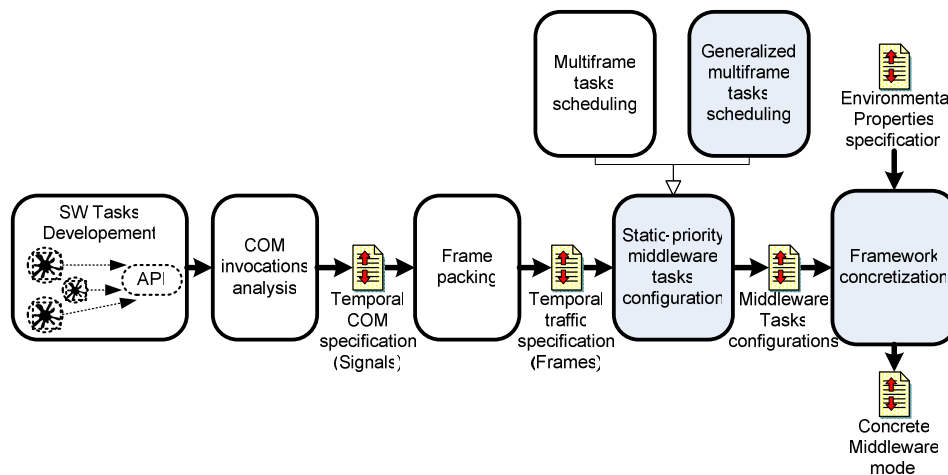


Figure 1 : Méthodologie de construction de l'intergiciel.

De plus, le résultat généré (une empreinte logiciel personnalisé) devra être exploitable sur une plateforme temps réel de démonstration.

Technologiquement parlant, notre réalisation devrait rejoindre les hypothèses des modèles théoriques de [RSM06], à savoir :

- Les ECUs sont munis d'OSEK/VDX OS [6] ;
- Ces ECUs sont connectés à même réseau (local) de communication CAN [5] ;

II - 3 Notre vision du système

Le nom que nous avons retenu pour identifier le système logiciel est « *GenIETeR* » en référence à l'acronyme de « GÉNérateur d'Intergiciels Embarqués Temps Réel » d'une part. Et d'autre part, faisant référence à la consonance que pourrait donner ce même acronyme avec le substantif « géniteur » et qui exprimerait le fait que le générateur serait le géniteur des empreintes logicielles cibles qu'il produirait. Ce plus indéniable a eu l'agrément du répondant du laboratoire, et représentait notre système comme le géniteur d'une longue lignée d'applications.



Figure 2 : Logo du projet *GenIETeR*.

Le logo ayant pour intérêt d'exprimer une empreinte logicielle générée pour un environnement cible particulier.

II - 4 Plan du mémoire

La suite de ce mémoire est organisée en deux parties :

Contexte de la distribution des systèmes temps réel embarqués dans l'automobile

La première partie est consacrée à l'étude théorique du contexte des systèmes temps réel, en passant en revue les modèles et les architectures qui les caractérisent. Nous introduisons, par la suite, le dimensionnement temps réel des applications embarqués dans l'automobile en mettant en exergue les contraintes temporelles imposées à leurs constructions. En second lieu, nous nous intéresserons à la distribution dans l'électronique des véhicules ainsi qu'aux exigences conséquentes de maîtrise des méthodologies de développement logiciel.

Ensuite, nous dégagerons la notion de modèle de répartition dans les architectures distribuées dans le domaine automobile, et nous motiverons la nécessité de recourir aux intergiciels. La section suivante situera alors notre problématique par rapport aux intergiciels existants. Elle dégagera les axes pertinents de ces projets par rapport aux besoins que nous avons identifiés. En conclusion, nous terminerons par la proposition d'une architecture d'intergiciel innovante, qui permet à un générateur de produire des instances d'intergiciels configurés et adaptés conformément au modèle de répartition de la solution à la quelle ils sont destinés.

Réalisation

La seconde partie de ce mémoire présente une synthèse des contributions de l'étude dont émane notre travail. Nous y retracerons les éléments ayant motivés nos choix pour la démarche de réalisation que nous avons suivi dans la construction d'une conception générique, et nous dégageons les aspects innovants de notre projet.

À la suite, nous ferons le tour du périmètre fonctionnel de notre solution générique, indépendamment du choix de tout modèle de répartition, pour ensuite présenter notre conception des mécanismes de base et des patrons de conception que nous avons retenue pour faciliter la réalisation de notre intergiciel par assemblage et configuration de composants au travers d'un générateur.

En dernier lieu, nous décrirons notre développement logiciel et les choix technologiques adoptés pour construire les prototypes de notre solution.

Le mémoire s'achèvera alors en rappelant les réalisations essentielles de notre travail et les perspectives ouvertes par l'architecture que nous proposons.

III Contexte des systèmes temps réel embarqués dans l'automobile

III - 1 Introduction : Application, système informatique, environnement

Une *application* est composée d'un *système informatique* (Lorsqu'il n'y aura pas de confusion possible nous l'appellerons simplement système) et d'un *environnement* physique avec lequel il interagit. Le système est composé d'un *calculateur* et d'un ensemble de programmes qu'il doit exécuter appelé *Applicatif* (cf. Figure 3).

Le système informatique, considéré par l'automaticien comme un simple maillon du système automatisé constitue l'unique centre d'intérêt de l'informaticien. L'environnement est défini comme l'ensemble de tous les éléments physiques qui sont extérieurs au système informatique et qui sont en interaction avec lui. La frontière entre l'environnement et le système informatique est parfois difficile à établir, c'est pourquoi nous avons choisi ici de considérer que l'ensemble des composants physiques qui peuvent être programmés (microprocesseurs, bus de communication, mémoire, interface E/S, etc.) fait partie du système informatique, alors que tous les composants physiques non-programmables constituent l'environnement. Il représente, du point de vue de l'automaticien, l'ensemble des éléments qui constituent le processus discrétisé à l'exception des convertisseurs analogique-numérique et numérique-analogique.

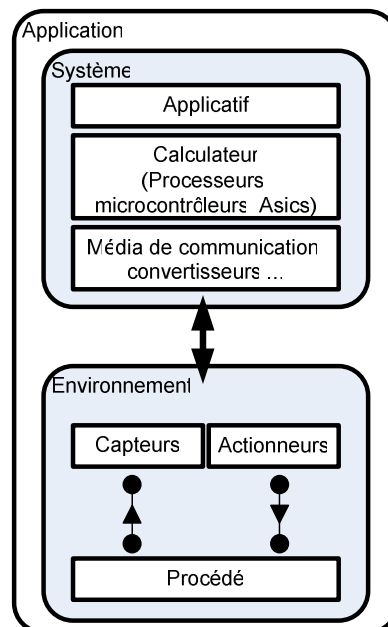


Figure 3 : Application temps réel.

III - 1 - 1 Définitions et taxinomie

Chaque système informatique(isé) possède ses propres spécificités, étroitement dépendantes de son domaine d'utilisation. Toutefois, nous pouvons regrouper les applications informatiques parmi les trois catégories suivantes [ELL97], que la bibliographie qualifie de :

- *Transformationnelles* : toutes les applications permettant d'élaborer un résultat à partir de données connues et disponibles à l'initialisation de l'application. Leur traitement effectif est de plus non contraint dans la durée. Les applications de calcul scientifique ou de gestion de bases de données sont des exemples représentatifs de cette catégorie d'application ;
- *Interactives* : toutes les applications permettant de fournir des résultats en fonction de données produites par l'environnement du système (essentiellement par l'interaction avec les utilisateurs), dans un délai de production satisfaisant au vue de contraintes aussi bien fonctionnelles, que non fonctionnelles. Parmi ces applications, les progiciels de bureautique sont certainement les plus connus ou les logiciels qu'on qualifie d'aide à un cœur de métier particulier ;
- *Réactives* : toutes les applications dont les résultats sont entièrement liés à l'environnement dans lequel elles évoluent. De plus, la dynamique de cet environnement conditionne les instants de production de ces résultats.

Cette dernière catégorie est souvent assimilée aux systèmes temps réel puisque d'une part, elle suggère par le terme réactif, les paradigmes essentiels aux systèmes temps réel. Parmi les caractéristiques des systèmes réactifs [HAL93], il convient de souligner les suivantes :

- *La concurrence* : L'évolution du système se fait de façon concurrente avec son environnement. Il est naturel de voir un tel système comme un ensemble de composants parallèles qui coopèrent pour réaliser la tâche assignée au système entier. Ainsi, le système comprend au moins deux types de processus coopérants qui assurent d'une part l'échange des données et d'autre part le traitement de celles-ci ;
- *Les contraintes temporelles* : Les systèmes réactifs sont soumis à des contraintes temporelles fortes (temps de réponse borné, fréquence d'échantillonnage d'un signal, etc.), spécifiées lors de la conception. La vérification de celles-ci avant utilisation est nécessaire pour la validation du système. On parle dans ce cas de systèmes temps réel;
- *La sûreté* : Du fait de leurs divers rôles critiques (par exemple, le contrôle d'une centrale nucléaire ou bien celui du train d'atterrissage d'un avion) impliquant des enjeux importants aussi bien en vies humaines qu'en termes économiques, il est primordial d'éviter tout risque de dysfonctionnement. La sûreté de fonctionnement devient de fait une question essentielle pour ces systèmes. Cela entraîne un besoin de

méthodes de développement qui intègrent des outils et techniques adéquates pour la vérification et l'analyse ;

- *Le déterminisme.* C'est une propriété fortement souhaitée dans les systèmes temps réel. Elle facilite la prédiction sur les comportements d'un système. Elle permet aussi de reproduire des comportements d'un système dans l'optique, par exemple, de mettre en évidence des erreurs susceptibles de survenir lors d'une exécution de celui-ci ;
- *La maintenance.* Les systèmes réactifs sont parfois embarqués (dits aussi enfouis de façon équivalente dans le reste de ce mémoire), posant ainsi quelques difficultés pour les modifier après la réalisation. Dans ce cas, une stratégie prédictive de maintenance, basée sur des modèles a priori, peut jouer un rôle important dans le choix d'une structure de système facile à modifier. Les systèmes embarqués sont surtout caractérisés par des contraintes matérielles (ressources limitées en espace mémoire et puissance de calcul) ;
- *La répartition.* Ce sont souvent des systèmes géographiquement distribués pour diverses raisons : la délocalisation des éléments d'un système (capteurs chargés de récupérer les entrées, actionneurs responsables de la production des sorties, processeurs servant de support d'exécution), le gain de performance grâce à l'utilisation de plusieurs calculateurs pour améliorer les temps de réponse des systèmes, la tolérance aux fautes à travers la duplication de certaines parties d'un système. Nous reviendrons plus loin en détail sur ce type système et qui bien sûr nous intéresse ici.

L'interactivité et le temps et d'autre part l'émergence de l'informatisation du contrôle de procédés de plus en plus complexes contribue à une utilisation en forte croissance de ce type de systèmes. Les domaines concernés vont des chaînes de production aux transports aériens en passant par des robots de plus en plus perfectionnés, tels que les modules d'exploration planétaire, l'aide à la conduite automobile ou aux applications multimédia contraintes par le temps. Si le caractère réactif prédomine dans ces exemples, la criticité du temps est autant importante pour ces systèmes. En effet, alors que le retard de quelques millisecondes ou la perte d'une trame vidéo sur un réseau ne nuira qu'au confort d'un utilisateur, le retard de la décision de redresser les roues d'une voiture lors d'un tête-à-queue peut provoquer une catastrophe sur le plan humain ainsi que des pertes financières considérables et surtout l'image de marque du constructeur.

Par référence à ces caractéristiques, la conception des systèmes réactifs temps réel distribués requiert des méthodologies suffisamment élaborées pour répondre aux exigences d'une mise en œuvre fiable. Pour toutes ces raisons, il est nécessaire de s'assurer du bien saisi

des notions qui régissent ce domaine du temps réel. Ce que nous tenterons de présenter dans la suite de ce mémoire.

III - 1 - 2 Les systèmes réactifs temps réel

Il existe de nombreuses définitions des systèmes temps réel [STA88, CNR88] qui évoquent les caractéristiques précitées au paragraphe précédent. Nous citerons la définition donnée par [AD92] qui introduit les notions de critères temporels :

« *Système de traitement de l'information ayant pour but de commander un environnement imposé en respectant les contraintes de temps et de débit (temps de réponse à un stimulus, taux de perte d'information toléré par entrée) qui sont imposées à ses interfaces avec cet environnement* ».

Cette définition met en évidence deux éléments distincts : une ou plusieurs entités physiques constituant le procédé, dont le rôle est d'agir et de détecter, et un contrôle informatique, nommé contrôleur ou application temps réel qui est le décideur des actions (ou réactions) du procédé. Le contrôleur reçoit des informations sur l'environnement du procédé à l'aide de capteurs et commande les changements d'état du procédé via des actionneurs. La Figure 4 donne un aperçu des interactions qui existent entre procédé et contrôleur d'un système temps réel.

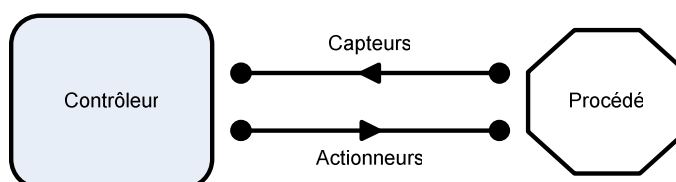


Figure 4 : Un système Temps Réel - Interaction procédé contrôleur.

III - 1 - 3 Application temps réel et applications concurrentes

Pour gérer la dynamique des périphériques (ou interfaces) du procédé et de son environnement, il est donc nécessaire de développer des techniques logicielles permettant de traiter les informations issues des capteurs sur l'unité de calcul pour produire les actions adéquates sur les actionneurs. Pour cela, citons le principe des applications concurrentes dont nous pouvons trouver une définition dans [BW90] :

« *La programmation concurrente est le nom donné aux techniques et notations de programmation pour exprimer le potentiel parallèle et résoudre les problèmes inhérents de synchronisation et de communication. L'implémentation du parallélisme est un problème des systèmes informatiques (logiciel ou matériel) qui est avant tout indépendant de la programmation concurrente. La programmation concurrente est importante puisqu'elle permet de faire abstraction de l'étude du parallélisme sans être confronté aux détails de l'implémentation* ».

Cette approche est particulièrement prometteuse en ce qu'elle permet d'augmenter la puissance d'expressivité et de réduire le coût de développement par rapport à une application fonctionnant sans parallélisme. Toutefois, l'utilisation des applications concurrentes n'est pas sans inconvénients. En effet, il devient nécessaire de fournir un moteur d'exécution (*Runtime* est un programme ou une bibliothèque qui permet l'exécution d'un autre programme) ou noyau temps réel pour superviser l'exécution sur l'unité de calcul des différentes tâches du système.

III - 1 - 4 Notion de tâche

La multiplicité des capteurs et des actionneurs introduit implicitement, du point de vue logiciel, l'utilisation de différentes tâches permettant de les piloter. Ce sont des programmes (flot de contrôle propre, distinct) séquentiels dédiés au traitement d'un des composants du système temps réel (aussi bien d'un point de vue fonctionnel que structurel). Par exemple un programme temps réel peut être constitué d'une collection de tâches telles que :

- Des exécutions périodiques de mesures de différentes grandeurs physiques (pression, température, accélération, etc.). Ces valeurs peuvent être comparées à des valeurs de consignes liées au cahier des charges du procédé ;
- Des traitements à intervalles réguliers ou programmés ;
- Des traitements en réaction à des événements internes ou externes ; dans ce cas les tâches doivent être capables d'accepter et de traiter en accord avec la dynamique du système les requêtes associées à ces événements.

Ainsi nous pouvons caractériser une application temps réel d'application multitâches.

III - 1 - 5 Interactions entre les tâches

Les tâches, dont les comportements sont séquentiels, peuvent interagir entre elles pour assurer le bon fonctionnement global de l'application que le système pilote. Il est donc nécessaire de fournir parallèlement à ces tâches des moyens de communication et de synchronisation permettant de gérer tous les problèmes liés aux accès à des ressources communes, comme par exemple les périphériques (terminaux, imprimantes, etc.), ou l'exécution des tâches ordonnées par des critères de précedence. Ces critères de précedence des tâches sont pour la plupart issus, soit d'un désir d'échange de données entre deux tâches, soit de la volonté de synchroniser deux tâches pour que la suite de leur exécution s'effectue en parallèle par un mécanisme de rendez-vous. Dans le premier cas, on identifie une tâche

émettrice et une tâche réceptrice. Nous parlons de tâches indépendantes lorsque l'application n'utilise ni ressources critiques, ni synchronisation.

III - 1 - 6 Architecture matérielle d'une application temps réel

Nous appelons *architecture matérielle*, l'ensemble des composants physiques qu'il est nécessaire d'ajouter à un processus pour réaliser l'application temps réel. L'architecture matérielle est donc composée d'un calculateur, de capteurs et d'actionneurs.

Le calculateur : Microprocesseurs, microcontrôleurs

Un *microprocesseur* est composé d'un CPU (*Central Processing Unit*) et d'unités de communication pour communiquer avec des périphériques externes ou d'autres microprocesseurs. Le CPU est une machine séquentielle constituée généralement d'un *séquenceur d'instruction* (SI), d'une *unité de traitement* (UT) et d'une *mémoire*. Les dernières générations de microprocesseurs peuvent aussi intégrer une *unité de calcul flottant* (FPU) dont le but étant d'accélérer considérablement certains calculs mathématiques (multiplication, division, sinus, arrondi, etc.).

Un *microcontrôleur* est un microprocesseur intégrant un certain nombre d'interfaces supplémentaires (mémoires, timers, PIO : *Parallel Input Output*, décodeurs d'adresse, etc.). Ces nombreuses entrées-sorties garantissent un interfaçage aisé avec un environnement extérieur tout en nécessitant un minimum de circuits périphériques ce qui les rend particulièrement bien adaptés aux applications temps réel embarquées. Du fait de leur forte intégration en périphérique (certains microcontrôleurs vont jusqu'à intégrer des fonctions spécialisées dans la commande des moteurs), les microcontrôleurs sont souvent moins puissants que les microprocesseurs; le CPU qu'ils intègrent est généralement en retard d'une ou même de deux générations.

Dans la suite du document, nous utiliserons le terme de ECU (*Electronic Control Unit*) pour désigner indifféremment un microprocesseur ou un microcontrôleur pourvu qu'il soit doté d'une architecture logicielle.

Le capteur

Le capteur est un dispositif conçu pour mesurer une grandeur physique (température, pression, accélération, etc.) en la convertissant en une tension ou un courant électrique. Le signal électrique issu d'un capteur est un signal continu qui doit être discrétisé pour pouvoir être traité par le calculateur. Cette discrétisation ou numérisation est réalisée par un circuit appelé, Convertisseur Analogique-Numérique (C.A.N).

Actionneur

Un actionneur est un dispositif qui convertit un signal électrique en un phénomène physique (moteur, vérin électrique, voyant, haut-parleur, etc.) censé modifier l'état courant du processus. Le signal de commande fourni par le calculateur est un signal numérique qu'il faut convertir en signal électrique analogique à l'aide d'un Convertisseur Numérique Analogique (C.N.A).

Environnement d'exécution

Nous avons vu précédemment qu'un classement des systèmes temps réel pourrait être fait selon leurs caractéristiques. D'un autre point de vue, matériel cette fois, les systèmes temps réel peuvent être classés selon leur couplage avec des éléments matériels avec lesquels ils interagissent. Ainsi, l'application concurrente et le système d'exploitation qui lui est associé peuvent se trouver :

- Soit directement dans le procédé contrôlé : c'est ce que l'on a appelé des systèmes embarqués (*embedded systems*). Le procédé est souvent très spécialisé et fortement dépendant du calculateur. Les exemples de systèmes embarqués sont nombreux : contrôle d'injection automobile, stabilisation d'avion, électroménager, etc. C'est le domaine des systèmes spécifiques intégrant des logiciels sécurisés optimisés en encombrement et en temps de réponse ;
- Soit le calculateur est détaché du procédé : c'est souvent le cas lorsque le procédé ne peut être physiquement couplé avec le système ou dans le cas général des contrôle/commandes de processus industriels. Dans ce cas, les applications utilisent généralement des calculateurs industriels munis de systèmes d'exploitation standard ou des automates programmables industriels, comme par exemple dans les chaînes de montage industrielles.

Ayant introduit la notion de calculateur ou de processeur, nous distinguons trois grandes catégories d'architecture matérielle pour les Systèmes Temps Réel en fonction de leur richesse en terme de nombre de cartes d'entrée/sortie, de mémoires, de processeurs et de la présence de réseaux.

- L'architecture monoprocesseur : un unique processeur exécute toutes les tâches de l'application concurrente. Dans ce cas, la notion de parallélisme n'a plus vraiment de sens puisque le temps processeur est partagé entre toutes les tâches. Nous parlons plutôt de pseudo-parallélisme ou d'entrelacement des exécutions. En effet, le parallélisme des tâches semble réel à l'échelle de l'utilisateur mais le traitement sur l'unique processeur s'opère de façon séquentielle.

- L'architecture multiprocesseurs : l'exécution de toutes les tâches est ici répartie sur plus de deux processeurs partageant une unique mémoire centrale. La coopération entre tâches se fait par partage des informations placées en mémoire. Donc ici, le traitement est réellement parallélisé.
- L'architecture distribuée : c'est le cas des architectures multiprocesseurs ne partageant pas de mémoire centrale. Ces processeurs sont reliés entre eux par l'intermédiaire de réseaux permettant d'assurer les communications entre les différentes tâches. La coopération se fait ici par communication par réseau ou par bus.

Pour le contexte de ce mémoire, nous retiendrons de cette taxonomie des différentes formes que peuvent prendre les systèmes temps réel, le cas des architectures distribuées monoprocasseur. La figure ci-dessous montre plus particulièrement l'architecture monoprocasseur d'un tel système en affinant l'interaction entre le procédé et le contrôleur.

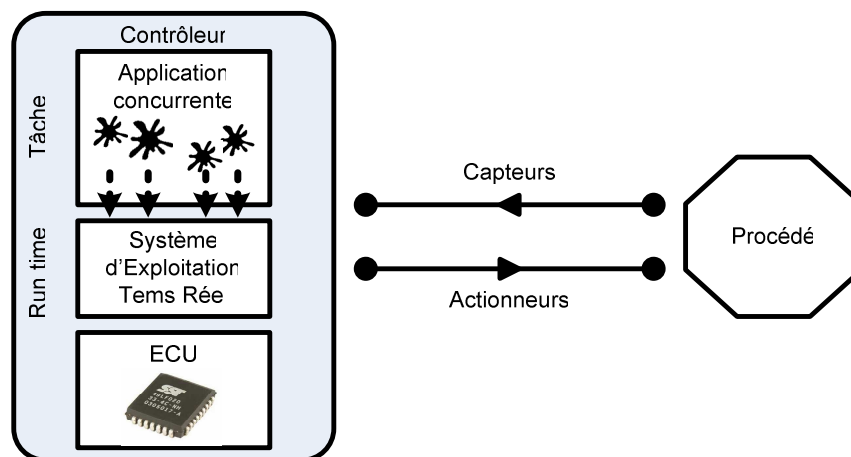


Figure 5 : Architecture générale d'une application temps réel monoprocasseur.

III - 1 - 7 Architecture logicielle d'une application temps réel

Avant de passer à l'architecture logicielle, il est intéressant de dire que si cette architecture matérielle permet de mettre en avant les qualités réactive de ces systèmes, il n'en va pas de même pour la seconde notion primordiale : le temps. Il est vital pour le procédé de pouvoir garantir que le traitement des différents évènements interviendra dans des délais adaptés à l'évolution du procédé. Pour cela, il est nécessaire d'associer à un système réactif des contrôles temporels. Ceux-ci doivent évaluer la vitesse de réaction du système de contrôle. Deux approches peuvent être envisagées : l'approche synchrone et l'approche asynchrone. Dans le premier cas, le respect des délais est implicite. Dans le deuxième cas, il devra être contrôlé. De plus, il sera nécessaire de se placer dans un environnement d'exécution adapté, qui sera fourni par un noyau temps réel ou exécutif temps réel, proposant des routines temporelles spécifiques.

Nous pouvons maintenant décomposer l'architecture logicielle d'un système temps réel en deux couches. La première, consiste en une application concurrente composée d'un ensemble de tâches comme nous l'avons cité plus haut. Nous utilisons également le terme d'application multi-tâches. La deuxième, de plus bas niveau, joue le rôle d'un système d'exploitation minimal chargé de faire le lien entre le procédé physique et l'application multitâches. Ce système d'exploitation, appelé exécutif temps réel, est de par la considération de l'asynchronisme (quantification du retard entre l'occurrence d'un évènement et la terminaison de l'exécution de la tâche qui lui est associée), dirigé par les évènements, ceux-ci pouvant provenir de différentes sources :

- Du procédé physique par l'intermédiaire d'interruptions matérielles associées à chaque évènement ;
- Du temps : chaque système est muni d'une horloge temps réel pouvant générer des interruptions ;
- De l'application multi-tâches lorsque par exemple l'exécution d'une tâche est conditionnée par l'exécution d'autres tâches. Dans ce cas il faut que l'exécutif retarde l'exécution de cette tâche pour permettre au préalable au processeur d'exécuter les autres.

III - 1 - 8 Le rôle de l'exécutif temps réel

L'exécutif temps réel a pour rôle de décrire les conditions événementielles d'exécution des tâches de même que les conditions de préemption (réquisition du processeur à une tâche et sa mise en attente forcée) et de reprise d'exécution sur le processeur. On appelle donc exécutif temps réel, la partie du système d'exploitation chargée de gérer l'exécution des tâches conformément aux contraintes exprimées. Cet exécutif est constitué d'une base communément appelée ordonnanceur, encapsulé par des niveaux qui offrent aux tâches les services requis pour leurs synchronisations, communications, temporisations, etc.

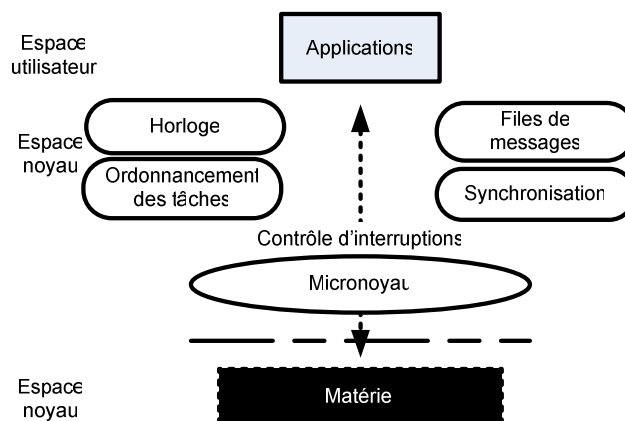


Figure 6 : Les systèmes d'exploitation temps réel.

L'exécutif temps réel propose différents services et garanties facilitant l'exécution et la communication des tâches. Ces services appelés primitives temps réel peuvent être directement utilisés dans les tâches et sont de différentes natures :

- Gestion des tâches ;
- Gestion des ressources partagées ;
- Gestion des communications entre tâches ;
- Gestion du temps ;
- Gestion des interruptions et de la mémoire, etc.

Gestion des communications entre tâches

Il existe principalement deux types de communications entre tâches. La première dite asynchrone utilise le concept de boîte aux lettres. Ce dernier est fondé sur l'utilisation d'un tampon d'échange de données : une tâche émettrice dépose chaque donnée dans ce tampon qui les gère en mode FIFO; la tâche réceptrice, lorsqu'elle a besoin de la donnée, doit soit se mettre en attente si le tampon est vide, soit recueillir la donnée la plus ancienne. La deuxième forme de communication est dite synchrone et est la plupart du temps utilisée par les langages synchrones. Elle utilise la méthode du rendez-vous qui permet à deux tâches de se synchroniser à un instant précis de leur exécution pour s'exécuter par la suite de façon conjointe.

Gestion des interruptions et de la mémoire

La gestion des interruptions doit permettre de prendre en compte toutes les sollicitations matérielles et logicielles. On utilise pour cela un service de routines d'interruption (ou ISR pour *Interrupt Service Routine*) permettant d'associer un traitement à chaque exécution. La durée de chaque routine doit être la plus courte possible puisque les routines s'exécutent de manière atomique (les interruptions sont masquées durant leurs exécutions). La gestion de la mémoire peut être faite suivant deux modèles : soit l'exécutif et les tâches ont chacun une zone de mémoire réservée, soit chaque tâche ainsi que l'exécutif possèdent une zone mémoire séparée et protégée.

Toutes ces fonctions de l'exécutif temps réel existent sous forme de primitives ou routines élémentaires dont la plupart possèdent des bribes atomiques, c'est-à-dire ne pouvant pas être interrompues par la gestion des interruptions matérielles. Ces portions ininterrompibles engendrent des retards dans la gestion des événements qu'ils soient logiciels ou matériels. Pour assurer un service optimal aux traitements des tâches, il faut réduire ces portions au minimum. C'est justement l'un des critères d'évaluation des exécutifs temps réel du marché

(ou RTOS pour *Real Time Operating System*), ce qui les différencie des systèmes d'exploitation classiques. Les RTOS ou les systèmes à micronoyau temps réel assurent ainsi une borne temporelle pour chacune des primitives temps réel qu'elles proposent. Parmi ces RTOS, nous pouvons citer par exemple OSEK/VDX, Vxworks, RTEMS, Lynx OS, Linux RT.

Nous reviendrons en détails sur notre choix pour l'exécutif OSEK/VDX dans la section consacrée à la réalisation.

III - 1 - 9 Les systèmes à micronoyau

L'objectif principal des systèmes à micronoyau est de maîtriser la complexité croissante du système d'exploitation en les rendant synthétiques et homogènes. La première caractéristique des systèmes à micronoyau est la taille réduite de leur code (de quelques kilos octets à quelques centaines de kilos octets). Cette réduction de taille du noyau a pour conséquence directe la limitation des services proposés qui sont complétés par des couches transversales de service (au sens programme fournissant des services) sous forme de modules ou de bibliothèques. Actuellement, presque seuls les systèmes d'exploitation temps réel sont à micronoyau et pour la plupart d'entre eux, les programmes applicatifs temps réel sont chargés dans l'espace noyau. Il n'y a donc aucun transfert de données entre espace utilisateur et espace noyau pour accéder aux fonctionnalités du système d'exploitation et donc une minimisation du temps d'exécution due aux nombreuses transmissions de données vers le micronoyau.

Le fait que les applicatifs temps réel résident dans l'espace noyau (cf. Figure 6) permet de les « rapprocher du matériel », améliorant ainsi grandement les performances en termes de temps de réponse. Toutefois, l'inconvénient majeur de cette approche réside dans l'impossibilité de tirer profit des services uniquement accessibles dans l'espace utilisateur tels que la protection de la mémoire (les erreurs de segmentation qui protègent le système d'exploitation et les autres applications des problèmes d'adressage), d'utiliser de primitives d'allocation dynamique de mémoire et l'utilisation de *drivers* matériels spécifiquement dédiés à l'espace utilisateur.

III - 1 - 10 Quantification du temps

L'architecture logicielle des applications temps réel permet d'identifier le traitement d'un événement à une tâche. Nous avons vu que ce traitement doit intervenir dans des délais appropriés. Il faut donc être à même de vérifier que les contraintes temporelles soient bien

respectées. Pour cela, nous devons introduire des indications temporelles quantitatives permettant par exemple, d'exprimer les délais à respecter. Ceci est mis en oeuvre par la modélisation temporelle des tâches. De plus, il est nécessaire de préciser la façon dont ces délais doivent être pris en compte. Nous devons, en d'autres termes, préciser la qualité de service attendue pour l'évaluation de l'application temps réel.

III - 1 - 11 Les tâches en temps réel

Il existe trois types de tâches en temps réel qui diffèrent par leurs caractéristiques temporelles. Les tâches dites périodiques sont la plupart du temps stimulées par l'horloge temps réel (HTR) de l'exécutif temps réel de façon à assurer une activité régulière, par exemple lors de l'acquisition de données (comme dans le cas d'une lecture échantillonnée du signal continu de la vitesse des tours moteurs) ou la génération périodique d'évènements.

Les tâches aperiodiques sont quant à elles activées de façon aléatoire en fonction par exemple, d'évènement aléatoire. Nous pouvons noter qu'il existe une sous famille de ce type de tâches qui est la famille des tâches sporadiques pour lesquelles une durée minimale sépare deux occurrences successives de l'évènement déclencheur.

Enfin les tâches cycliques sont très proches des tâches périodiques à la différence près que leur activation n'est pas liée à l'horloge temps réel mais à des déclencheurs (ou alarmes), ce qui induit une périodicité approximative. La durée séparant deux activations successives d'une tâche périodique est constante, alors qu'elle appartient à un intervalle pour les tâches cycliques. Nous ne nous intéresserons par la suite qu'aux tâches périodiques, car comme nous allons le démontrer, les modèles que nous utilisons peuvent être aisément étendus au reste des tâches. Nous utiliserons aussi le terme de tâche pour désigner le programme informatique compilé qui sera exécuté sur le processeur du système. Lorsque nous parlons d'activation d'une tâche, qu'elle soit périodique ou non, nous utiliserons le terme d'instance. Une instance de tâche désignera donc une exécution spécifique du code de la tâche concernée et il peut y avoir dans le système plusieurs instances de la même tâche.

III - 1 - 12 Les tâches périodiques dans le temps réel

Le modèle de tâche périodique que nous utiliserons dans ce mémoire représente les tâches activées à intervalles réguliers (constants). Ce modèle [LAY73, MOK83] s'appuie sur le modèle temporel utilisé dans la thèse [RSM06], en effet notre travail se doit d'être, je cite « *la vérification de l'implémentabilité de l'approche proposée* » et une extension conceptuelles des modèles architecturaux proposés. Il est donc nécessaire d'introduire ces concepts

temporels puisqu'ils sont inhérents à la compréhension de la méthodologie de construction de notre système, et plus précisément à la phase de configuration des tâches de l'intergiciel.

Modèle temporel d'une tâche périodique

Soit une tâche périodique Φ_i parmi un ensemble de n tâches affectée à un ECU, et incluant un système d'exploitation temps réel préemptif, alors Φ_i est modélisée par les quatre paramètres temporels : r_i, C_i, D_i, T_i avec :

- r_i la date à laquelle la première instance de Φ_i est activée ;
- C_i la pire durée d'exécution (ou charge maximale) dans le pire cas de Φ_i ;
- D_i , le délai critique (ou échéance relative) associé à Φ_i ;
- T_i la période de la tâche Φ_i .

La Figure 7 illustre le rôle de chacun de ces paramètres.

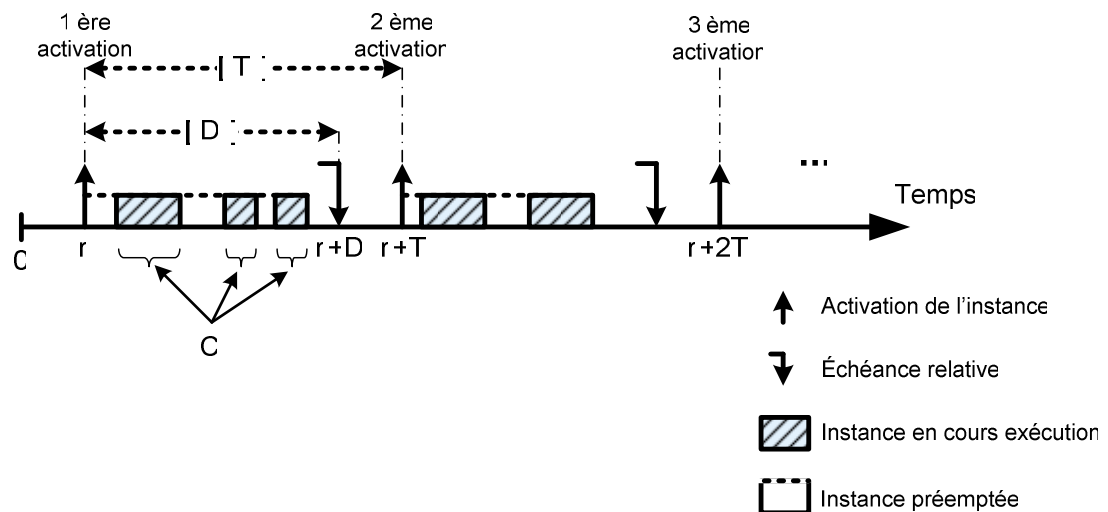


Figure 7 : Modélisation d'une tâche périodique.

La date r_i , de première instance de la tâche Φ_i , est appelée date de réveil (ou *offset*). Lorsque toutes les tâches ont la même date de réveil que nous pouvons alors supposer nulle ($r_1 = r_2 = \dots = r_n = 0$), alors à l'instant $t = 0$, il existe une instance de chaque tâche prête à être exécutée. Nous parlons de tâches à départs simultanés. Dans le cas contraire nous parlons de tâches à départs différés. Lorsque la date de réveil n'est pas connue à l'avance, nous parlons de tâche non concrète (ou *Offset free*).

La durée d'exécution C_i est en réalité une borne supérieure du temps d'exécution de la tâche Φ_i . Cette borne du pire cas d'exécution est très complexe à obtenir. En effet, que ce soit par analyse dynamique (mesure directe d'une exécution) ou analyse statique (exploration du code de la tâche), son évaluation est rendue difficile par la présence d'instructions conditionnelles, de boucles indéterministes, ou encore des améliorations des processeurs

jusqu'à se révéler impossible (due à l'indéterminisme des mémoires cache, au changement de contexte provoqué par les interruptions). L'étude de ce paramètre appelé également WCET (*Worst Case Execution Time*) fait l'objet de nombreux travaux. Il va de soi, que cela passe par la définition de la situation la plus pénalisante. La difficulté consiste donc à déterminer cette dernière qui correspond à un intervalle de temps sur lequel le processeur se voit attribué la plus grande charge de travail. Pour de plus amples explications, les lecteurs peuvent se référer à [PB00, PUA05]. Ce paramètre est par conséquent l'un des plus critiques pour la modélisation du système de tâches, puisqu'il est le seul a priori à être une valeur approchée.

C'est pourquoi, il existe de nombreuses autres approches de modèles de tâches permettant de tenir compte des évaluations plus ou moins fines de cette durée d'exécution d'une instance de tâche à une autre. Cette grandeur pourra, entre autres, être utilisée pour mettre en œuvre, en hors-ligne, un test dit *test d'ordonnabilité* (incluant une définition d'heuristiques temporelles). Ce test permettra de statuer, comme dans notre cas [RSM03] de configuration des signaux dans les trames réseau, sur l'aptitude de l'architecture matérielle (exigence d'utilisation optimale des ressources en terme de consommation de bande passante) d'une part, et celle de l'intergiciel (exigences de qualité de service) d'une autre part, à assurer de la faisabilité de la communication distante, et à conclure quant à la sûreté de l'application temps réel.

Le délai critique D_i ou échéance relative, détermine le temps alloué à chaque instance de la tâche pour son exécution c'est-à-dire le délai maximal autorisé entre l'activation et la terminaison de l'instance.

Enfin, le dernier paramètre, T_i , quantifie l'intervalle de temps séparant deux activations successives d'une même tâche. Ainsi, il est possible de déterminer les dates d'activation pour chaque instance de la tâche Φ_i par des dates de réveil successifs. Il en vient que la date de réveil de la k -ième instance de Φ_i est $r_i^k = r_i + (k - 1) \times T_i$ (pour un $k > 1$). De la même façon, nous déterminons les dates successives des échéances de chaque instance d'une tâche Φ_i avec $k > 1$ par $d_{ki} = r_{ki} + D_i$.

Dans le cadre du projet *GenIETeR*, l'affectation d'une valeur à ces paramètres (au travers de la spécification des contraintes temporelles de la communication distante) est réalisée par le programmeur d'application. On attend donc, que celui-ci fournisse des valeurs convenables (pas trop surestimées) mais majorantes de la durée d'exécution effective et du délai critique en toutes circonstances. S'il ne lui est pas possible de fournir de telles valeurs, il ne pourra en aucun cas mettre en œuvre un mécanisme de communication distante. Y compris lorsque la durée d'exécution effective d'une trame (cf. configuration des tâches de l'intergiciel) dépasse le WCET fourni initialement, on ne tolérera point, à ce stade du projet, un mode de

fonctionnement dégradé de l'intergiciel. Ce mode de fonctionnement devra toujours être en conformité avec les exigences temporelles de l'application.

Il existe d'autres modèles de tâches : apériodique, enrichis incluant le modèle à multi représentations, le modèle *multiframe* (que nous détaillerons plus loin). Il faut retenir, pour la suite de ce mémoire, que lorsque nous parlons de caractéristiques temporelles d'une tâche, nous nous en tiendrons au modèle de tâches périodiques à départs simultanés. Théoriquement, ce même modèle peut être aisément étendu au modèle de tâches apériodiques en introduisant un paramètre supplémentaire [MOK83] permettant de définir un intervalle minimal entre deux activations successives, et ainsi étendre notre implémentation à ce type de tâches.

III - 2 Qualité de service et contextes temporels des systèmes temps réel

Dans ce qui précède, nous avons défini les paramètres temporels caractérisant une tâche. On peut distinguer ainsi trois familles de systèmes temps réel suivant la rigidité des contraintes temporelles qui leurs sont imposées. En effet, les systèmes temps réel n'ont pas tous le même degré d'exigence vis-à-vis de ces critères.

- Les systèmes temps réel à *contraintes strictes* : Ce type de système impose que toutes les contraintes temporelles soient impérativement respectées et plus particulièrement le délai critique D_i .
- Les systèmes temps réel à *contraintes souples* : À l'opposé de la classe précédente, un non respect d'une échéance n'entraîne pas la défaillance du système. Ces dépassements sont donc tolérés, mais entraînent des perturbations qu'il faudra alors minimiser.
- Les systèmes temps réel à *contraintes mixtes* : Ces derniers sont soumis à la fois aux exigences des systèmes à contraintes strictes pour certaines tâches et à celles des systèmes à contraintes souples pour d'autres.

Les systèmes temps réel à contraintes strictes ont un comportement déterministe. Ils sont employés dans des systèmes embarqués. Les systèmes temps réel à contraintes souples, dont la mesure d'efficacité s'opère généralement par une analyse statistique des temps de réponse moyens, peuvent se rencontrer dans les systèmes du traitement multimédia, comme par exemple le *streaming*. Enfin, les systèmes temps réel à contraintes mixtes regroupent des applications temps réel composées de tâches dont un sous ensemble doit impérativement respecter des contraintes temporelles, à l'inverse des autres tâches dont le critère d'évaluation cherchera à minimiser les fautes temporelles.

Pour les besoins du projet *GenIETeR*, nous nous intéresserons qu'aux systèmes temps réel à contraintes strictes pour qualifier notre système de tel. Mais disons-nous bien qu'il s'agisse

seulement de catégoriser notre intergiciel, puisque les tâches applicatives développées par le programmeur d'application peuvent arbitrairement respecter un degré de contraintes temporelles plus au moins strictes. On ne s'intéresse ici qu'aux contraintes temporelles devant être respectées par la communication distante entre tâches, et la considération ne sera pas faite pour l'entropie du système.

III - 3 Du temps réel embarqué au temps réel embarqué appliqué à l'automobile

III - 3 - 1 Les systèmes temps réel embarqués

Lorsque le système temps réel est physiquement intégré à l'environnement qu'il contrôle et qu'il est soumis aux mêmes contraintes physiques (température, pression, etc.) que son environnement, il est dit *embarqué* (cf. Figure 8).

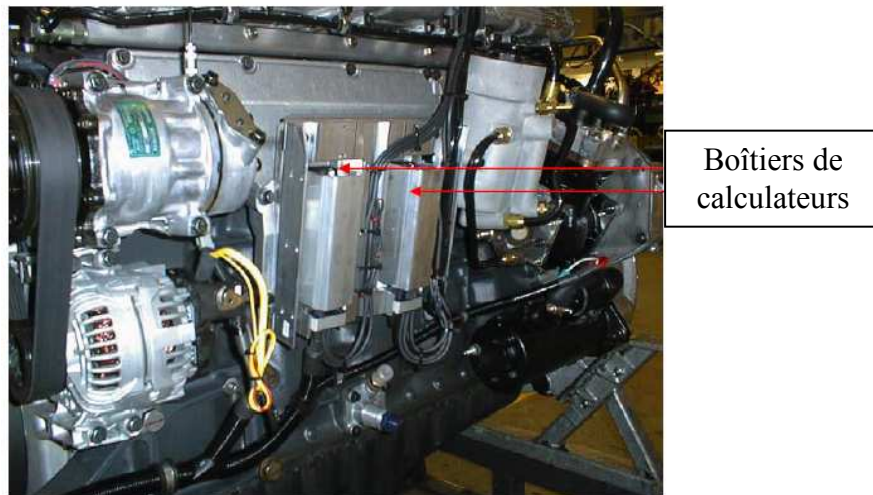


Figure 8 : Enfouissement d'un contrôleur d'injection diesel de Scania. [MRTC05]

(Source de l'image : Scania)

Les systèmes embarqués sont généralement soumis à des contraintes spécifiques de *coûts* pris au sens large du terme. Ces contraintes sont dues, d'une part, au type d'applications couvertes par cette catégorie de système, d'autre part, à l'intégration du système à son environnement. Ces contraintes particulières de coût sont de plusieurs types : encombrement, consommation d'énergie, prix, etc. Les applications automobiles, par exemple, nécessitent l'utilisation de systèmes embarqués dont les contraintes sont principalement des contraintes de coût financier (forte compétitivité du secteur commercial), d'encombrement (habitabilité du véhicule) et de consommation d'énergie. Concevoir un système temps réel embarqué nécessite une bonne maîtrise des coûts matériels mais aussi une bonne maîtrise des coûts de développement car ces derniers représentent une grande part du coût financier d'une application.

III - 3 - 2 Spécificité des systèmes temps réel embarqués dans l'automobile

« Les besoins en électronique des véhicules automobiles, ont évolué de façon considérable au cours de ces dernières années. Si naguère les seuls composants électroniques que l'on rencontrait à l'intérieur d'un véhicule étaient destinés uniquement à la radio, aujourd'hui une automobile peut compter près d'une vingtaine de calculateurs, et ce nombre devrait doubler vers l'an 2000. Les raisons de cet accroissement sont principalement dues aux normes antipollution de plus en plus sévères, ainsi qu'à la sécurité et au confort accru du conducteur et de ses passagers.

En 1950, la Peugeot 203 comportait un faisceau électrique de 50 fils. En 1997, une Renault Safrane en comporte quelque 800, voire 1000 pour des voitures de très haut de gamme, et les longueurs cumulées atteignent des sommets impressionnants : plusieurs kilomètres. En même temps que leur nombre augmentait, les équipements électroniques n'ont cessé de se sophistication. La quantité et la complexité des informations que doivent échanger les divers organes sont telles que le nombre de connexions est rapidement devenu prohibitif. Un calculateur n'est plus un élément isolé, mais utilise les informations des autres calculateurs et des divers capteurs, pour augmenter son efficacité. Un véhicule n'est plus un ensemble d'équipements isolés fonctionnant indépendamment les uns des autres, mais un seul et unique système. » (Source PSA et Auto-Volt 1997).

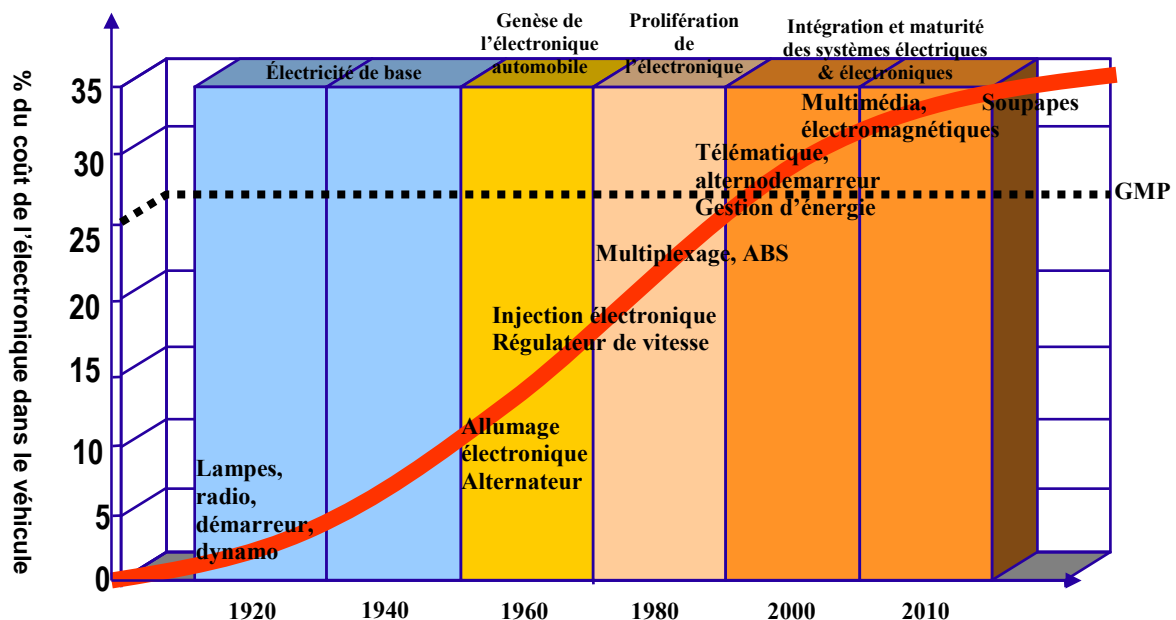


Figure 9 : De l'électricité à l'électronique dans l'automobile. [PSA07]

Aujourd'hui, il est évident que l'électronique tient une grande importance dans l'automobile, à en croire les chiffres [PSA07] (cf. la Figure 9), elle représenterait environ 30% du coût de revient globale du véhicule. Beaucoup de fonctions ont été et sont encore en cours de réalisation (particulièrement ceux liés à la sécurité, l'aide à la conduite, l'approvisionnement en énergie hybride et renouvelable), provoquant ainsi une prolifération des équipements électroniques à l'intérieur des véhicules qui confrontent les constructeurs aux problématiques de conception, fabrication, coût, encombrement, de fiabilité et de recherche de pannes. D'autre par cette prolifération impose une évolution constante des architectures et une croissance de la puissance de calcul embarquée. A savoir, qu'une Peugeot 607 atteint

aujourd'hui 2 Mega-octets en taille du code (cf. Figure 16), soit approximativement 1800 fois plus qu'une Citroën CX, c'est aussi cent fois plus qu'un A 300, le premier Airbus lancé en 1975.

Il est nécessaire de souligner que de tels systèmes sont parmi les meilleurs exemples des systèmes temps réel à contraintes temporelles strictes. Prenons l'exemple [WWIKI] de :

« L'airbag (coussin gonflable), vu généralement comme un sac qui étouffe, est en réalité une véritable prouesse technologique. En effet, il est constitué d'un sac en nylon ultra résistant et d'un système pyrotechnique qui, grâce à diverses formules chimiques et à un calculateur électronique (détectant le choc en 5 ms), va produire de l'azote gazeux qui va, en moins de 40 ms, gonfler le sac en direction des passagers. Grâce à des événements situés sur le côté du sac, l'excédent de gaz est évacué afin d'assurer au mieux l'absorption d'énergie, assurée à 75 % par le bloc avant. Il se dégonfle ensuite en 0,5 s environ. »

En résumé, un cycle d'airbag dure moins longtemps qu'un clignement d'œil. Il assure la sécurité passive des passagers, avec ce qu'il y a de plus concret en terme de sûreté, de déterminisme, de maintenance et de répartition.

III - 3 - 3 Maîtrise des coûts matériels

En effet, de fortes contraintes temps réel imposeront l'utilisation de calculateurs puissants basés sur des processeurs cadencés à des fréquences élevées. Cependant, en micro-électronique, l'utilisation de fréquences élevées est synonyme de consommation électrique plus élevée ce qui va à l'encontre des contraintes de consommation d'énergie. De la même manière, réduire la consommation électrique pourra nécessiter l'utilisation des composants spécifiques à faible consommation qui sont souvent plus coûteux. C'est ainsi, que les constructeurs automobile s'attèlent à réaliser une solution non optimale, mais globalement satisfaisante pour toutes les contraintes. Un bon moyen de satisfaire globalement toutes les contraintes est de minimiser le nombre de composants nécessaires à l'application. Voici une vue globale des moyens employés :

- **Limiter le nombre de capteurs** en les partageant entre plusieurs fonctionnalités que doit assurer le système.
- **Réduire les câblages** en rapprochant le calculateur des actionneurs grâce à l'utilisation d'un calculateur multiprocesseur, dont chacun des microprocesseurs est positionné le plus près possible d'un ensemble de capteurs et actionneurs.
- **Limiter le nombre de composants du calculateur** en cherchant à utiliser des microcontrôleurs, plutôt que des microprocesseurs.

III - 3 - 4 Evolution des architectures

Ainsi, l'architecture matérielle du système temps réel a évolué en prenant en compte les différentes solutions visant à minimiser le nombre de composants que nous venons d'énumérer. C'est à travers l'exemple significatif de l'automobile, que nous passons en revue les différents types d'architecture témoins de cette évolution pour pouvoir légitimer le positionnement de notre projet.

Les architectures conventionnelles

Dans les architectures conventionnelles comme le montre la Figure 10, il est fait en sorte que par agrégation, chaque boîtier électronique rassemble un certain nombre de traitements fonctionnellement identiques.

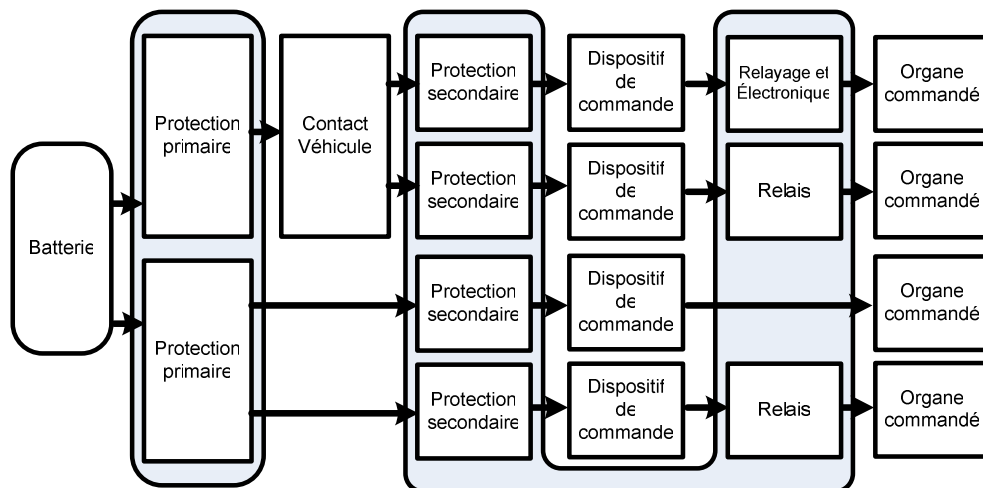


Figure 10 : Architecture conventionnelle. [ETR05]

Cette approche nécessite une duplication par boîtier des composants de protection et d'alimentation de l'électronique, et bien sûr le raccordement des broches pour connecter les alimentations, les entrées et les sorties nécessaires à la réalisation de la fonction.

Ainsi dans cette architecture, les contraintes de temps réel son propre à chaque calculateur : l'exécutif embarqué dans les calculateurs est dédié à la fonction réalisée. Mais de telles architectures tendent à disparaître des choix des acteurs du marché de l'automobile car elles supportent mal l'augmentation du nombre de fonctions dans le véhicule, et l'explosion combinatoire des branchements provoque des coûts énormes.

Les architectures centralisées

La Figure 11 présente ce type d'architecture. Elle est composée d'un calculateur qui peut être monoprocesseur ou multiprocesseur à mémoire partagée et d'un ensemble de capteurs et

actionneurs, tous reliés au calculateur. Ce type d'architecture conduit à un câblage de type « étoile » souvent important et coûteux.

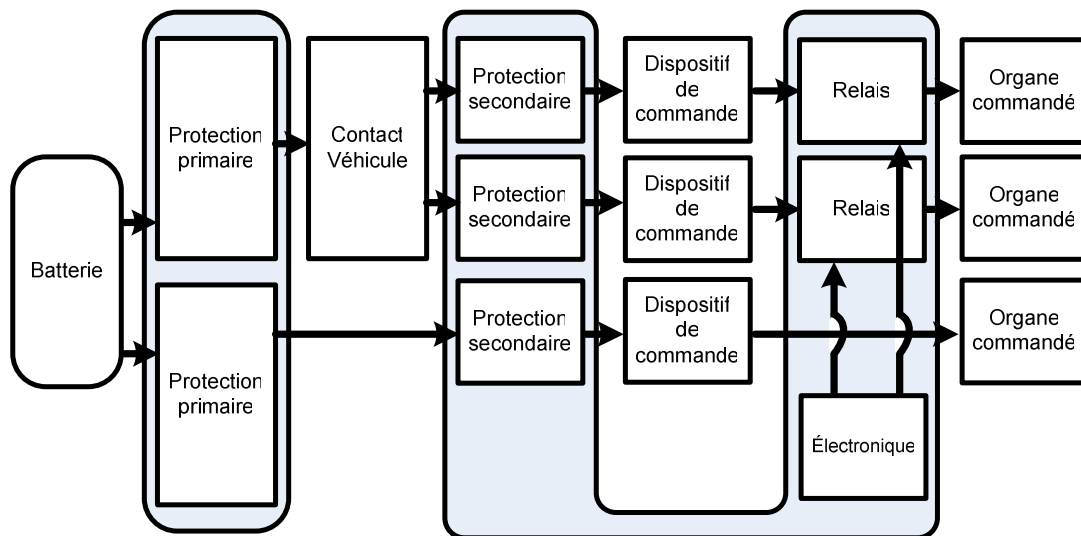


Figure 11 : Architecture centralisée. [ETR05]

Architecture faiblement distribuée

Dans les années 80, des bus de terrain ont été développés par les constructeurs automobiles et équipementiers : bus CAN (*Controller Area Network*, développé par Bosch), Van (*Vehicle Area Network*, présenté par PSA et Renault), J1850 (mis au point par la Société Américaine des Ingénieurs de l'Automobile), A-Bus (par Volkswagen) et K-Bus (par BMW), SCP, ECP, I-BUS, P-BUS et bien d'autres encore. Ces bus bifilaires dédiés aux environnements perturbés tels que les automobiles permettent de relier entre eux les calculateurs. Le bus le plus utilisé actuellement est le bus CAN, il est devenu un standard dans les applications automobiles.

« Aujourd'hui seul le protocole CAN, même si il n'est pas parfait, répond bien aux besoins et a été adopté par de nombreux industriels pour tout type d'application, et est en passe de devenir le standard pour le multiplexage automobile. » [WNET1]

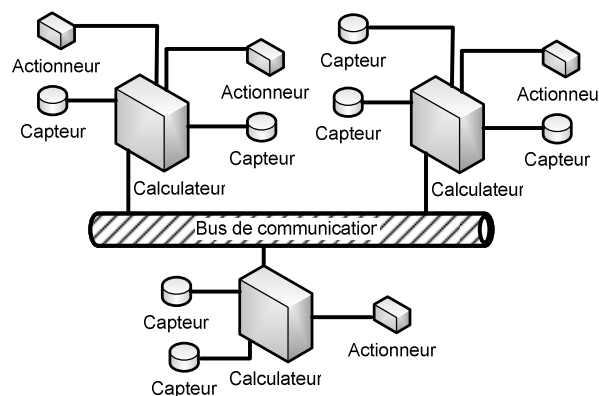


Figure 12 : Architecture faiblement distribuée.

Grâce à ces nouveaux bus, sont apparues les applications temps réel embarquées avec des architectures que l'on qualifie de « faiblement distribuées » (cf. Figure 12). Elles sont constituées d'un ensemble de calculateurs reliés entre eux par un bus. Par rapport à l'approche multi-calculateurs le gain est indéniable : il est possible de contrôler le comportement global de l'ensemble de tous les calculateurs et il est possible de partager des capteurs entre les calculateurs.

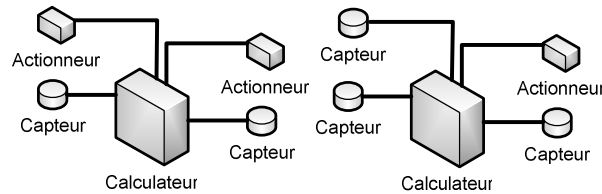


Figure 13 : Architecture multi-calculateurs.

Dans ces architectures distribuées, une fonction peut être répartie entre plusieurs calculateurs électroniques reliés par des réseaux multiplexés.

Architecture fortement distribuée

Cette architecture est la plus aboutie. Les capteurs et actionneurs deviennent intelligents, ils peuvent directement être connectés sur le bus (cf. Figure 14). Cette approche permet de réduire considérablement tous les câblages, car tous les organes électriques du véhicule peuvent être reliés au bus. C'est aussi l'approche la plus complexe à mettre en oeuvre au niveau logiciel. Toute la difficulté consiste à gérer efficacement le multiplexage des données issues des capteurs et des calculateurs sur le bus de telle sorte que les contraintes temporelles de chacun des signaux soient satisfaites.

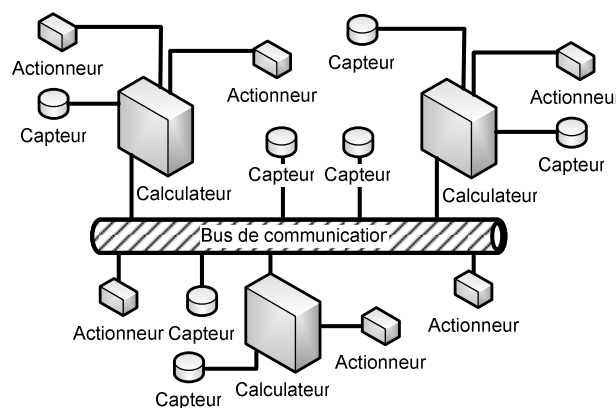


Figure 14 : Architecture fortement distribuée.

L'introduction des réseaux permet de réduire la quantité de liaisons filaires reliant les nombreux organes d'un véhicule. Cependant, ceci impose la définition d'une messagerie pour ces réseaux et de revoir la conception des circuits de protection et des alimentations. Dans ce type d'architecture, les contraintes de temps réel influent le dimensionnement du système en

entier. Elles entrent dans la définition de la communication des réseaux, leurs modes de fonctionnement et le choix de la distribution d'une même fonction entre plusieurs organes du logiciel embarqué.

Comme notre intérêt porte sur les applications temps réel à contraintes temporelles strictes pour les communications distantes entre tâches, nous retiendrons le cas des architectures distribuées au sens large pour le contexte de notre projet *GenIETeR*.

Voici une illustration du contexte d'un système embarqué de contrôle de freinage :

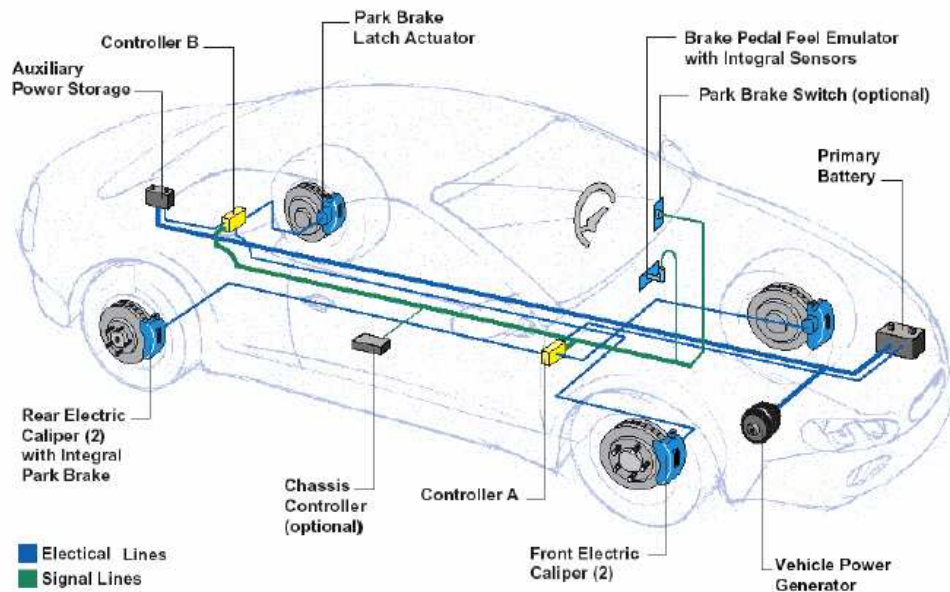


Figure 15 : Exemple d'un système embarqué de contrôle de freinage [MRTC05].

Ce système est construit autour d'une architecture matérielle comprenant :

- Une source d'énergie incluant une batterie et un générateur de courant. Il est évident qu'il est possible d'avoir d'autres sources sinon de répliquer certaines sources particulièrement critiques ;
- Un câblage de distribution d'énergie (cf. Figure 15, *Electrical Lines*) qui permet d'alimenter aussi bien les actionneurs que les autres composants électroniques ;
- Des actionneurs transformant l'énergie électrique en mouvements mécaniques ;
- Des capteurs qui notifient le système de contrôle et/ou l'utilisateur de l'état instantané du véhicule ;
- Une distribution des fonctionnalités matérialisées par des réseaux embarqués, de type CAN [5] par exemple (cf. Figure 15, *Signal Lines*).

III - 4 Méthodologie de développement logiciel dans l'automobile

Généralement les voitures sont fabriquées à très grande échelle, de l'ordre de quelques millions d'exemplaires. Pour atteindre ces volumes et pouvoir toujours offrir de larges

gammes de produits à leurs clients, les constructeurs font en sorte de construire des plateformes flexibles contenant les technologies communes et pouvant être adaptées en différentes déclinaisons. Ce concept s'inspire du modèle de processus PLP (*Product Line Practice*) développé par le *Software Engineering Institute of Carnegie Mellon University in Pittsburg* [HAR04]. Par exemple, la Volvo XC90 [MRTC05], mise sur le marché en 2002, est issue de la même plateforme qu'une précédente Volvo lancée depuis 1998. Les technologies contenues dans ces plateformes sont très diverses, rendant ainsi leur maîtrise très difficile par un seul et même constructeur automobile (ou OEM, *Original Equipment Manufacturers*). Ce processus de fabrication par composition nécessite donc un grand nombre d'externalisation de composants à différents équipementiers (ou fournisseurs, sous-traitants, systémiers) qui fournissent des pièces détachées. Le rôle du constructeur est alors de fournir aux équipementiers les spécifications nécessaires de sorte à adapter les composants fournis et de les intégrer dans ses produits par assemblage. Traditionnellement, les fournisseurs n'offrent que les pièces physiques (le matériel), mais les besoins évoluent, pour fournir le logiciel aussi.

Du fait de l'émergence du *Full By-Wire* (FBW), et pour le qualifier en d'autre terme c'est le « tout numérique », ce qui veut dire que toute l'information échangée dans le réseaux local des ECUs devient numérique. Même les commandes du contrôle mécanique sont électriques, et ceci au travers de l'utilisation de moteurs électriques et des actionneurs électromécaniques.

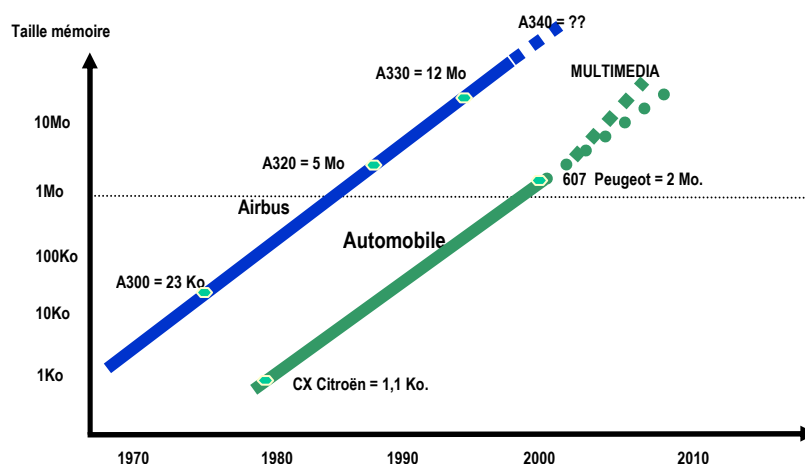


Figure 16 : Augmentation de la taille du code. [PSA07]

Cette course à l'intégration et au multiplexage accompagnée d'une augmentation de la puissance des ECU a comme conséquence direct un accroissement considérable de la complexité d'interfacer plusieurs applicatifs en provenance de différents équipementiers dont le résultat est une augmentation phénoménale du code (cf. Figure 16). D'après [HAR04], si dans l'industrie automobile, l'électronique constitue 90% de l'innovation, il n'en demeure pas moins que 80% de cette innovation est faite dans le secteur du logiciel. D'autant plus qu'une

estimation du *Mercer Management Consulting and Hypovereinsbank* [MER01] voudrait qu'en 2010 la proportion du coût logiciel passe à 13% du coût global d'un véhicule (cf. Figure 17).

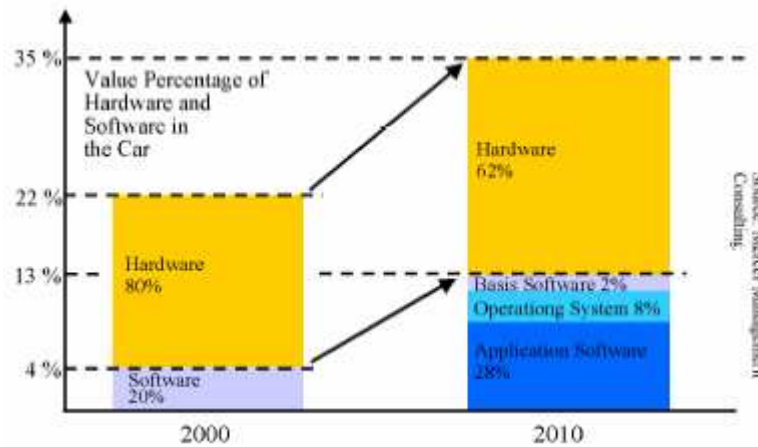


Figure 17 : Croissance de l'importance du logiciel dans l'automobile. [HAR04]

Cela pour dire qu'il y a un réel enjeu économique à changer les procédés de réalisation et à innover dans les méthodes de développement. Dans ce contexte, pour n'en citer que quelques-uns, bien des travaux ont été réalisés dans le domaine de l'ingénierie des besoins, la qualité du logiciel ou le développement orienté par les modèles. L'objectif premier est de réduire le temps d'élaboration, et d'améliorer la qualité du logiciel. Un défi existe, qui est aussi le nôtre, est celui de la réutilisation du logiciel (horizontale : au travers d'un large éventail d'applications, et verticale : dans un domaine d'application donnée). Le besoin majeur de la réutilisation est la séparation du logiciel embarqué contenu dans un ECU de l'architecture matérielle qui le compose. Pour cela il faut que :

- Les composants logiciels [4] réutilisables soient indépendants de l'environnement matériel ;
- Les interfaces de ces composants logiciels doivent fournir des mécanismes de communication inter-composants aussi bien localement à un ECU et/ou à travers le bus de données ;
- Le développement de logiciel réutilisable doit être capable d'intégrer de nouveaux besoins du fait de l'évolution des fonctionnalités fournies par ces composants ;
- La taille du code et le temps d'exécution des composants logiciels doivent être optimisés durant le développement pour consommer le minimum de ressources ;
- Augmentation de l'utilisation des COTS (*Components Of the Shelf*).

On pourrait citer dans ce contexte la loi de Lanergan émettant : « *Plus importante et décentralisée est l'organisation, meilleur est le potentiel de réutilisation* ».

La modularité, la scalabilité et la transférabilité sont aussi des points importants dans le processus de développement de logiciels réutilisables nécessitant ainsi de procéder à des standardisations et des regroupements en consortiums pour normaliser les architectures. Le consortium AUTOSAR [WASAR] et le projet OSEK/VDX [WOSEK] (cf. Annexe 1) en sont l'illustration. Notons que les techniques et les outils de vérification, que nous n'aborderons pas dans ce mémoire, ont une grande part dans l'industrie automobile à en voir le processus global de réalisation (cf. Figure 18).

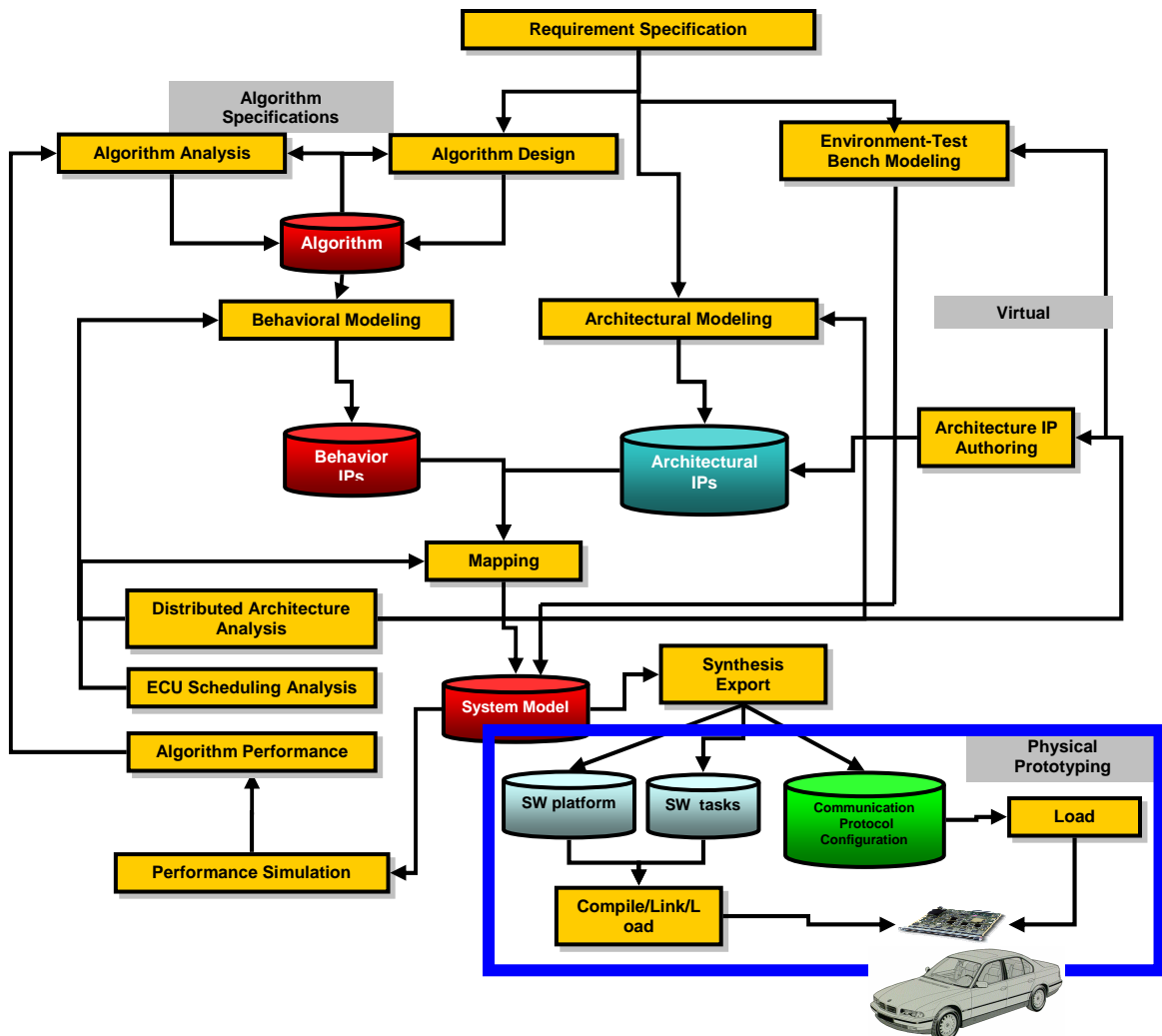


Figure 18 : Flot de conception logicielle global. [ALB02]

Pour définir le périmètre de notre projet, nous nous placerons dans cette illustration sommaire du flot de réalisation logicielle dans la phase dédiée au prototypage physique (cf. Figure 18, *Physical Prototyping*). Cette phase de prototypage fait intervenir différents rôles du génie logiciel que nous pouvons illustrer par la Figure 19. Il est intéressant de remarquer que comme nous l'avons décrit précédemment, l'architecture logicielle d'un système temps réel se décomposait en deux sous parties ; un RTOS ou exécutif temps réel et une application concurrente. Cette même distinction se reflète dans le processus de développement logiciel

(cf. Figure 18) par une séparation concrète entre le flot de réalisation de la plateforme et celui de l'application concurrente (ici dénommée système par abus).

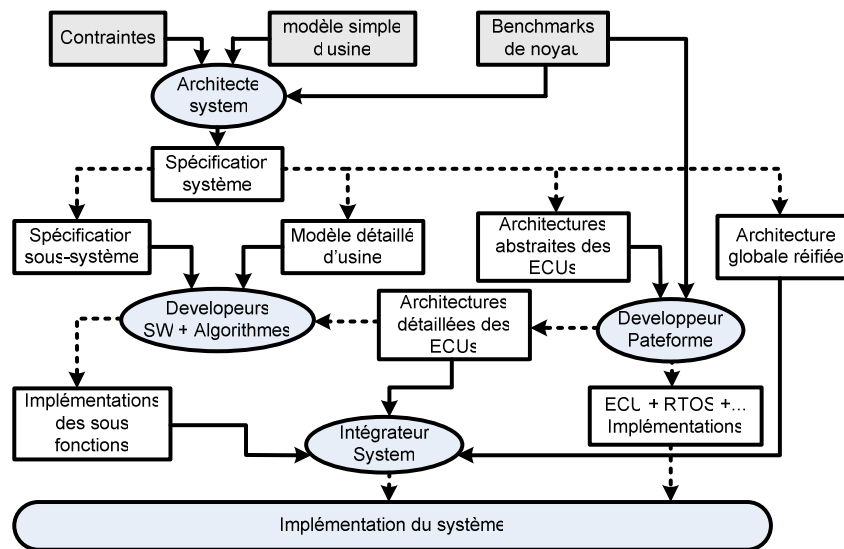


Figure 19 : Intervenants et rôles dans le processus de réalisation logicielle.

(Inspirée de [ALB02])

Le rôle qui nous intéresse dans le cadre de notre projet est l'intégrateur système, puisqu'il est le premier concerné par l'ouverture, l'interopérabilité des interfaces et surtout l'interopérabilité temporelle (respect des contraintes de temps). D'ailleurs, il sera l'acteur principal de notre système.

Dans ce contexte de croissance constante de la taille du code embarqué, et où l'hétérogénéité des plateformes, des protocoles, des systèmes et l'importance de l'interopérabilité rend les techniques traditionnelles de développement inefficaces. Face à ce fort besoin d'intégration à moindre coût et en un temps optimale, il devient donc impératif et crucial d'innover par de nouvelles méthodologies et d'outils, qui en plus de l'automatisation de la production doivent garantir, par construction, le respect de toutes les propriétés. Ceci forme la problématique de fond de notre projet qui a été soulevé par [RSM06-2]. Ces travaux proposent la spécification/développement d'un intergiciel [3] et définissent une méthode de configuration/déploiement de composants logiciels [4] de celui-ci, tout en préservant les propriétés temporelles requises au niveau applicatif.

III - 5 La distribution dans l'automobile

Précédemment, nous avons pu présenter l'architecture distribuée comme une classe de la taxonomie des systèmes temps réel. Par l'exemple du système de contrôle de freinage (cf. Figure 15), nous avons pu illustrer l'effectivité de la répartition des fonctionnalités et organes dans l'automobile. Il est à présent nécessaire de passer aux concepts liés à la communication dans les systèmes temps réel embarqués dans l'automobile.

Un véhicule est traversé de part et d'autre par des faisceaux de fils électrique afin de transmettre énergie et informations à l'ensemble des différents organes électriques composant le système. L'augmentation croissante des fonctions dans l'automobile a accru considérablement le nombre de ces fils et par conséquent, augmenté le poids des véhicules ainsi que leur coût. Une mutation de ces liaisons vers des réseaux multiplexés (se référer à [WNET1] pour de plus de détails) permet de limiter ces contraintes tout en introduisant d'autres difficultés à surmonter, comme le dimensionnement temps réel et la réactivité.

« C'est surtout la Mercedes classe S qui a donné un début d'existence au concept de réseau local dans l'automobile. Les 6 calculateurs relatifs à l'allumage, l'injection, le pilotage du remplissage, l'ASR » ...Système de contrôle de la motricité ou système d'anti-patinage... «la commande de boîte et le système de diagnostic étaient branchés en réseau. En configuration maximale, les fonctions électroniques et électriques intégrées comptaient encore 48 microcontrôleurs pilotant 62 moteurs et 3 000 points de contacts. La longueur des câbles parvenait tout juste à rester inférieure à 3 kilomètres. » [WNET1]

Devant les problèmes de coût, de conception, de fiabilité, de montage et de contrôle, les constructeurs et les équipementiers ont décidé de relier les différents organes au moyen d'un réseau local. Ainsi, plutôt que d'utiliser plusieurs fils pour interconnecter point-à-point les composants électriques, transportant chacun une seule information, les différents organes sont reliés par un ou deux uniques fils partagés à tour de rôle. C'est pour cette raison que cette technique d'interconnexion est plus communément appelée « bus » multiplexé ou plus simplement multiplexage.

III - 5 - 1 Les systèmes de communication

Un réseau (local) élémentaire se constitue d'au moins deux nœuds (dans notre cas nous considérons des ECUs) : un émetteur et un récepteur, et d'un médium de transmission des données. Les données à transmettre sont encapsulées dans des trames répondant à un protocole de communication. Un mécanisme d'arbitrage définit comment gérer les conflits entre plusieurs nœuds voulant émettre simultanément. Une caractéristique d'un réseau est sa vitesse de transmission qui correspond au nombre de bits par secondes (ou Bauds) que le réseau est capable de transmettre avec un médium physique de transmission de données. Les besoins des systèmes temps réel embarqués varient actuellement de 10 Kb/s à 25Mb/s [ETR05]. La vitesse de transmission est l'un des premiers critères pour le choix d'un réseau. Une classification selon ce critère peut être faite pour les réseaux multiplexés dans les architectures réparties de l'automobile.

Classe	Vitesse	Support	Exemple d'application
A	< 10kb/s	un fil	Ouverture coffre
B	10-125kb/s	Paire torsadée	Tableau de bord
C	125 kb/s-1Mb/s	câble coaxial	ABS, Motorisation
D	> 1Mb/s	fibre optique	Multimédia, X-by-Wire

Tableau 1 : Classification des réseaux multiplexé dans l'automobile. [ETR05]

Dans le cadre de notre travail, les considérations sont restreintes à une liaison point-à-point monodirectionnelle. Impliquant ainsi une vision selon le modèle producteur-consommateur pour définir les tâches en tâche productrice, et tâche consommatrice.

Les techniques d'accès au réseau de communication les plus répandus sont CSMA/CD (*Carrier Sense Multiple Access/Collision Detection*), CSMA/CA (*Carrier Sense Multiple Access/ Collision Avoidance*), TDMA (*Time Division Multiple Access*), *Tokens*, *Central Master*, et *Mini Slotting*. Ces protocoles avec des caractéristiques propres sont aussi bien utilisés pour le temps réel que pour des applications moins contraignantes. Le choix d'un protocole dépend de son aptitude à détecter et/ou à résoudre les collisions entre les trames. Le principal défaut du CSMA/CD est que la collision est détectable, provoquant une retransmission non déterministe des trames en conflit. Il est utilisé par exemple dans *Ethernet*. D'autre part le CSMA/CA est plus approprié aux applications temps réel par le fait qu'il propose une prévention des collisions et offre ainsi des comportements déterministes en comparaison du CSMA/CD. Ce protocole est implémenté dans CAN et ARNIC 629 [AUD97]. TDMA utilise un multiplexage temporel, dont le principe est de découper le temps disponible entre les différentes connexions. Son comportement est très déterministe et il est propice pour les analyses du temps réel. TTP [WTTP] en est une illustration. Une première alternative pour éliminer tout risque de collision est d'utiliser le paradigme de jetons (*Tokens*), comme par exemple dans *Profibus*. Le jeton matérialise le droit de transmettre. Chaque station le passe (le répète) sur l'anneau à une autre station, qui est prédéfinie comme la station suivante. En contrepartie, on se crée des contraintes topologiques. La seconde alternative est de définir un nœud du réseau qui soit un nœud maître pour contrôler le trafic en décidant, quand et quelles trames doivent être transmises. Cette approche est utilisée dans LIN et TTP/A. Enfin le *Mini Slotting* [RV93] peut être employé pour éliminer des collisions comme dans *FlexRay*. En associant à chaque nœud un quantum de temps prédéfini pour toutes ses utilisations du canal de transmission, on garantit l'unicité de l'accès au réseaux, ce qui se prête mal à la scalabilité puisque les nœuds les plus actifs seront pénalisés par les nœuds utilisant moins le bus.

Controller Area Network (CAN)

CAN [CAN] est un bus système série développé par Bosch pour l'automobile. Il fut présenté avec Intel en 1985. Le CAN a été standardisé par l'ISO dans les normes 11898 pour les applications à hauts débits, et ISO 11519 pour les applications à bas débits :

- CAN standard ou CAN 2.0 A : (*Standard Frames*) avec un identifiant d'objet codé sur 11 bits, qui permet d'accepter théoriquement jusqu'à 2 048 types de messages (limité à 2 031 pour des raisons historiques) ;
- CAN étendu ou CAN 2.0 B : (*Extended Frames*) avec un identifiant d'objet codé sur 29 bits, qui permet d'accepter théoriquement jusqu'à 536 870 912 types de messages. À la demande du SAE qui est à l'origine du standard J1939.

Le CAN [WCAN] a été créé pour opérer dans des environnements embarqués et c'est pourquoi il comprend de nombreux mécanismes de détection d'erreur. Les erreurs survenant sur tous les noeuds sont détectables à 100 %. Il utilise entre autre un mécanisme d'arbitrage à priorité fixe (FPS : *Fixed-Priority Scheduling*) [TBW95] pour une modèle de communication multipoints. Chaque ECU connecté au bus est capable de recevoir et de transmettre des messages. Les messages émis sont accessibles à tous les participants au bus. Un ECU n'évalue néanmoins que les informations qui lui sont utiles et ignore les autres. Dès que le bus n'est plus occupé, chaque unité de commande peut procéder à la transmission. Un seul message peut être transmis par le bus. Si deux unités de commande procèdent en même temps à la transmission, le message le plus important est transmis en priorité. Cette priorité est fixée par un champ d'arbitrage. La structure du protocole du bus CAN possède implicitement les principales propriétés suivantes :

- Hiérarchisation des messages ;
- Garantie des temps de latence ;
- Souplesse de configuration ;
- Réception de multiples sources avec synchronisation temporelle ;
- Fonctionnement multi-maître ;
- Détections et signalisations d'erreurs ;
- Retransmission automatique des messages altérés dès que le bus est de nouveau au repos ;
- Déconnexion automatique des nœuds défectueux.

Time-Triggered Protocol Class C (TTP/C)

Le protocole TTP/C [TTT99] supporte différentes structures physiques d'interconnexions (bus, étoile..), appliquées au fibre optique et aux autres supports physiques. Si le système est configuré en bus, chaque noeud a son propre gardien de bus qui permet l'accès correct au bus aux instants prévus [STE2001]. L'architecture développée autour du protocole TTP/C est appelée TTA : *Time Triggered Architecture* [KOP94], récemment adopté par Audi et Volkswagen dans les applications automobiles.

Local Interconnect Network (LIN)

LIN est un système de communication série développé en complément des réseaux déjà existants. LIN présente une alternative moins chère au CAN dans les connexions entre capteurs/actionneurs, où la vitesse et la fiabilité du CAN ne sont pas requises. La communication s'effectue sur la base du format SCI (UART) et sur un paradigme maître unique/esclaves multiples.

Bien d'autres protocoles multiplexés appelés « *Fieldbus* » existent en réponse au besoin de réduction de câblage dans l'industrie automobile. [CHAR] en fait d'ailleurs une étude comparative. On peut citer par exemple: *Time-Triggered CAN* (TT-CAN), *Flexible Time-Triggered CAN* (FTT-CAN), *FlexRay*, *ARINC 629* [AUD97], *Profibus*, *Train Communication Network* (TCN), *WorldFIP*, *Vehicle Area Network* (VAN), etc.

III - 5 - 2 Communication distante entre tâches

Les tâches (cf. III - 1 - 4) sont des entités fonctionnelles qui concourent à la réalisation d'un travail commun. Pour ce faire, elles interagissent par l'émission et la réception de signaux (pression dans le circuit de freinage dans un ABS, par exemple) qui peuvent s'accompagner d'informations. Suivant le type d'architecture utilisée, la communication inter tâches revêtira une forme différente tels que l'utilisation d'une boîte aux lettres en mémoire commune, l'utilisation d'un système centralisé et le transfert par messages dans un système distribué. Sa mise en œuvre, dans ce dernier cas, est rendue complexe à la fois par les délais de transmission non négligeables inhérents à tout support de communication, l'absence de cohérence d'horloges locales mais aussi la difficulté de construire un schéma de contrôle global pour le respect des contraintes de fraîcheur temporelle (cf. Figure 20). Car de part la criticité du système, l'âge d'un signal (le temps entre l'activation de la tâche émettrice et la réception effective du signal par la tâche consommatrice distante) doit être inférieure à sa durée de vie (contrainte fonctionnelle imposée par le cahier des charges) pour garantir la vivacité du système. Dans les travaux de [RSM06], cette complexité est réduite par le fait de

considérer une communication asynchrone distante orienté message (cf. ci-dessous) (un signal ou un message est caractérisé par une durée d'exécution et une échéance).

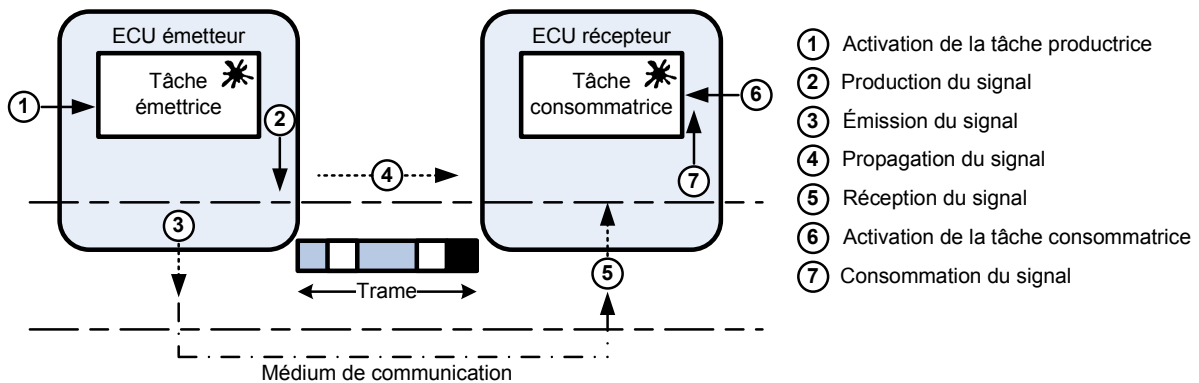


Figure 20 : Modèle d'interaction distante ente 2 tâches.

Lorsque les fonctionnalités d'une application temps réel sont réparties sur des calculateurs interconnectés en réseau, la structure de l'exécutif en est rendue plus complexe par la présence d'un service de communication approprié. Bien que chaque calculateur soit pourvu d'un exécutif dit local, la spécification de l'exécutif réparti doit garantir les mêmes propriétés (aussi bien temporelles que fonctionnelles) que chacun des exécutifs locaux. Cela implique que les primitives de communication inter-tâches distantes doivent pouvoir délivrer un signal (en respectant les mêmes contraintes temporelles définies sur ce signal qu'une communication locale) à n'importe quelle tâche distante, définie au niveau d'un exécutif local (appartenant au réseau), sans qu'il soit nécessaire de connaître sa localisation dans le réseau. Cette approche permet alors de considérer l'ensemble des calculateurs hétérogènes d'un réseau comme un unique calculateur virtuel, d'où une liberté de choix laissée à l'utilisateur pour la répartition des tâches.

Quel que soit le type d'utilisation auquel l'exécutif temps réel est destiné, la conception et la mise au point de ce dernier devront toujours être réalisées dans un souci de *criticité*, *portabilité*, *extensibilité* et *modularité*. Généralement, ces exécutifs sont conçus d'une manière monolithique du fait de leur complexité, ce qui les rend difficilement configurables. Du point de vue du développeur ainsi que celui de l'intégrateur système, l'exécutif temps réel est considéré comme une boîte noire qu'il n'est pas possible d'adapter. Pour palier à ce besoin de personnalisation, maîtriser la complexité des hétérogénéités aussi bien des exécutifs que des mediums de communication et la distribution des application, il est classiquement admis de développer une couche logicielle qui encapsule les fonctionnalités responsable de l'échange entre le système d'exploitation et les tâches. Sauf, que dans le cas des systèmes temps réel embarqués, il faut envisager les différentes contraintes inhérentes à ce type de système en terme de qualité de service temporelle, et de défaut de ressources matérielles

suffisantes. C'est donc dans cet esprit, que les bibliothèques de modules du système ont été développées dans le cadre du projet *GenIETeR*.

III - 6 Une architecture cible abstraite

Avant de commencer toute réalisation, il est intéressant et nécessaire de définir une vision générale de l'architecture qui concerne notre travail. Cette vision globale passe par une caractérisation d'un modèle abstrait qui doit être indépendant de toute architecture cible pouvant existés dans les systèmes embarqués dans l'automobile.

Tout d'abord nous émettons l'hypothèse sur la localisation des entités (tâches ou ECU) de notre système. Si on considère les scénarii conversationnels, nous pouvons distinguer deux types de localisation : absolue et relative. Dans le cas absolu, un intervenant est positionné par rapport à son « univers », indépendamment de tout autre objet, alors que dans le cas relatif, cette position dépend d'un autre objet. Les deux visions sont utiles et complémentaires mais nous retiendrons les localisations comme entités absolues. De ce fait, la position exacte d'une tâche n'est pas vraiment importante, nous exploitons surtout le fait qu'elle soit différente de toute autre localisation, ce qui nous permet d'identifier cette tâche de façon unique. Et analogiquement, nous exploitons ce fait pour localiser les ECUs dans notre architecture.

Pour parler de distribution dans une architecture, il nécessaire d'y associer un canal supportant l'échange d'informations. Un tel canal permet de modéliser la distribution du système, en associant un émetteur d'information à au moins une destination ou récepteur de l'information. Ainsi, lors d'une interaction, il est possible d'identifier la source de façon unique. Et pour éviter toute confusion, nous partons du principe qu'une seule information est émise à un instant donné. Cette information, pour être unique, doit être associée à une identification elle-même unique : la localisation de l'entité émettrice. Un signal sera donc identifié de façon unique dans tout le système.

Comme le montre la Figure 21, on considère une abstraction de l'architecture cible qui consiste en un réseau d'ECUs connectés sur un même bus. Chaque ECU est caractérisé par un nœud contenant un système d'exploitation, un contrôleur du medium de communication et une ou plusieurs tâches applicatives, et une couche de communication dédiée.

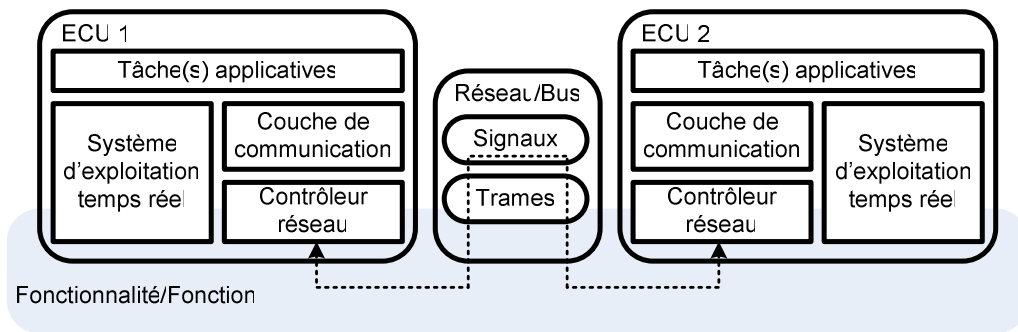


Figure 21 : Une abstraction de l'architecture cible.

Le cadre horizontal en bas de la Figure 21 indique que la fonctionnalité contenue dans ce modèle de haut niveau du système peut être arbitrairement distribuée entre les nœuds du réseau. Selon cette considération du système, la distribution d'une fonction réalisée par le système est ainsi transparente. Dans ce contexte, nous faisons l'amalgame entre les deux termes « fonction » et « fonctionnalité » pour exprimer une certaine habilité ou une propriété du système. La couche dédiée à la communication fait office d'une interface commune pour la communication entre tâches, qu'elles soient locales ou distribuées entre les ECU. Sa vocation est d'encapsuler les traitements liés aux signaux en provenance des couches applicatives et qui doivent être transmis entre différentes tâches. Supposons, qu'à ce stade, et pour l'assimilation des concepts de dimensionnement temps réel, que cette couche en charge de la communication soit matérialisée par une tâche concurrente aux tâches applicatives (nous reviendrons sur l'argumentation de ce choix). La communication, à proprement parler, est réalisée par le biais de trames au travers du module dédié au contrôleur réseau, et qui dépend du protocole et de la nature du bus.

Le comportement (explication établissant un rapport de cause à effet entre un stimulus et le comportement étudié) des systèmes embarqués est totalement connu à la spécification et n'évolue pas au cours de la vie du système. C'est pour cette raison qu'ils sont dits « prédictibles ». Nous utilisons, comme présenté dans [RSM06], une communication asynchrone entre les tâches applicatives et la couche dédiée à la communication, je cite :

«Un choix important au niveau de l'implémentation de l'intergiciel a été fait : nous considérons un intergiciel asynchrone par rapport aux tâches applicatives. En conséquence, l'intergiciel sera implémenté sous la forme de tâches. La raison principale de ce choix réside dans le fait que nous voulons appliquer des stratégies de minimisation de la consommation de bande passante, ce qui implique que l'intergiciel doit contrôler lui-même la construction des trames. Ceci serait aussi possible avec un intergiciel synchrone, mais notre choix permet une séparation plus efficace entre la production de signaux et leur mise en trame. De plus, ce choix au niveau de l'implémentation favorise le non blocage des tâches applicatives, empêchant que celles-ci attendent que les signaux produits soient mis en trames et envoyés. »

Cette communication est validée par des techniques d'analyse qui évitent toute collision et minimise la consommation des ressources matérielles.

Nous pouvons donc retenir que notre architecture, support de l'interaction unique (signaux et trames), relie une source unique (tâche ou ECU), appelée propriétaire du canal, et un ou plusieurs récepteurs aussi unique dits connectés au canal.

III - 7 De l'intergiciel à l'intergiciel temps réel appliqué à l'automobile

L'intergiciel (plus connu sous l'appellation anglo-saxonne middleware), comme le montre la Figure 22.a, a pour rôle d'assurer l'interfaçage entre la partie applicative d'un système structuré en couches, et le système d'exploitation. Cela permet de résoudre les problèmes d'interopérabilité et d'intégration des applications.

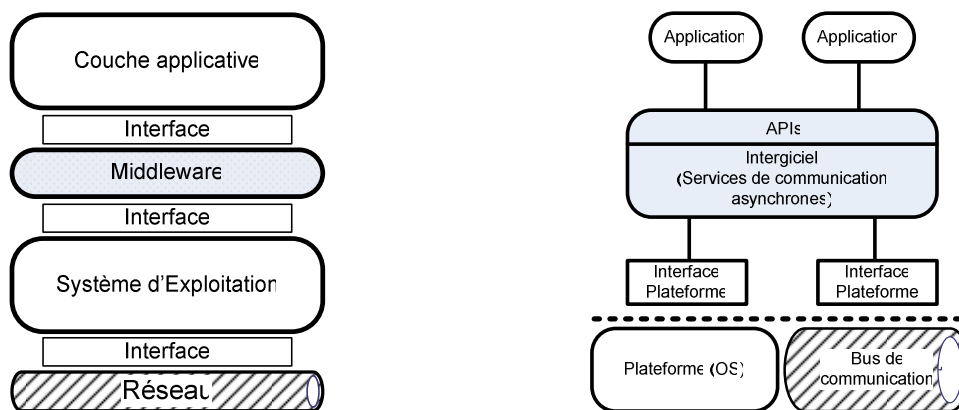


Figure 22 : (a, b) Couche intergiciel dans l'organisation d'un système.

Une infrastructure répartie s'organise traditionnellement en deux couches distinctes. Cette séparation en couches ne correspond pas forcément à une architecture du système, mais plutôt à une classification et à un positionnement grossiers de fonctions logicielles. Ainsi, la frontière entre système d'exploitation et intergiciel reflète plus l'état des technologies et des offres actuelles qu'un invariant fondamental. Le système d'exploitation fournit aux applications un accès partagé aux ressources matérielles (processeurs, mémoires, réseaux) via un ensemble d'abstractions de plus ou de moins haut niveau (par exemple tâches, mémoire virtuelle, système de fichiers, etc.). L'intergiciel, couche située entre système d'exploitation et applications (cf. Figure 23), fournit un ensemble de fonctions dédiées à la mise en œuvre d'applications réparties (par exemple fonctions de localisation, de communication, de gestion de transactions, de persistance, etc.).

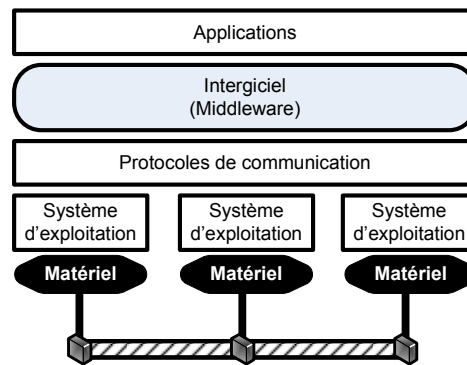


Figure 23 : Couche intergiciel dans l'organisation d'un système distribué.

Une telle organisation est notamment intéressante pour des systèmes où résident des applications réparties concurrentes. Ces dernières peuvent être à priori de natures différentes, mais grâce à la transparence offerte par l'intergiciel, elles accèdent de façon uniforme, par le biais d'interfaces standards (API : *Application Programming Interface*), aux primitives de communication distribuée, qui ne sont pas forcément compatibles au niveau du format (cf. Figure 22.b). Cette abstraction permet de dissimuler les embarras de la programmation liés à la localisation, la concurrence, la réplication, la fiabilité et la mobilité.

III - 7 - 1 Rôle d'un intergiciel

Pour un développeur d'application utilisant un modèle de répartition de haut niveau (envoi et réception de signaux, par exemple), la répartition est introduite de façon « transparente », au sens où elle lui est dissimulée. Les opérations de communication sous-jacentes sont réalisées de façon implicite ; le développeur utilise les mêmes procédés de développement d'une application que dans le cas de « non répartition ». Il définit les limites des fonctionnalités, les interfaces et les relations des fonctions de la même façon. La répartition consiste alors à projeter les entités ainsi délimitées, sur les différents ECU participant à l'application répartie. Les échanges de signaux sous-tendus par le modèle de répartition sont effectués par l'intergiciel. Celui-ci réalise ainsi les abstractions du modèle de répartition. De plus, il réduit la dépendance des composants vis-à-vis de l'environnement d'exécution, en masquant les aspects d'interfaçage de bas niveau avec les primitives de communication. La Figure 23 illustre l'organisation générale d'une application répartie utilisant un intergiciel : les composants de l'application utilisent les fonctions de l'intergiciel pour communiquer entre eux de façon transparente ; C'est l'intergiciel qui utilise les services de base du système d'exploitation et l'environnement matériel sous-jacents pour fournir ces fonctions de communication de haut niveau. Chaque nœud possède donc une instance de l'intergiciel. Ces instances communiquent entre elles au moyen de primitives de bas niveau, et jouent le rôle

d'intermédiaires entre les composants de l'application qui résident sur le même noeud et ceux qui résident sur d'autres noeuds. Les interactions avec ces derniers sont réalisées par échange de signaux entre les instances d'intergiciel locale et distante.

Adressage

L'intergiciel doit avant tout définir un moyen de nommer les diverses entités qui participent à une application répartie : il leur attribue des identifiants susceptibles d'être échangés et compris par chacun des noeuds de l'application. Ces identifiants permettent à une entité d'en désigner une autre, et ainsi d'interagir avec elle. L'intergiciel remplit une fonction d'adressage des signaux. Cette fonction est rendue accessible aux composants applicatifs à travers des constructions de programmation qui permettent d'associer la localisation concrète du programme utilisateur à des abstractions d'un modèle de répartition.

Transport

À partir des identifiants des signaux et des ECUs, une instance d'intergiciel achemine ces signaux pour les interactions entre tâches applicatifs. À cet égard, l'intergiciel assure le transport de signaux entre les noeuds, en utilisant le réseau de communication sous-jacent.

Protocole et représentation

L'intergiciel doit mettre en oeuvre un protocole. Il se charge également de mettre les données à échanger sous une forme transmissible, et compréhensible par les noeuds distants. La syntaxe et la sémantique des signaux reflètent les abstractions d'un modèle de répartition. L'utilisation d'un intergiciel permet donc de faire interagir, au sein d'une même application répartie, des tâches s'exécutant sur des architectures matérielles et dans des environnements logiciels différents. En contrepartie, le choix d'un protocole et d'une représentation limite les interactions possibles aux composants, dont l'intergiciel supporte le protocole. Il devient donc primordial, dans la quête à la réutilisation, de rechercher des moyens permettant de rendre les représentations indépendantes de leur environnement.

C'est ainsi que la recherche de la flexibilité s'est exercée tant au niveau des intergiciels, qu'à celui des noyaux de systèmes. Sans viser l'exhaustivité, nous présentons quelques voies d'approche suivies pour mettre en exergue les techniques d'adaptation et de configuration dynamique.

III - 7 - 2 Les intergiciels adaptables

Composer une application comme assemblage de parties réutilisables est un objectif ancien, pour lequel les progrès ont été lents. La notion comme celle de modules a fait son apparition, et puis successivement, celle d'objet. La notion de composant logiciel [4] vise à aller au delà de la programmation par objets en introduisant notamment l'explicitation des dépendances (entre composants, vis-à-vis de l'environnement), la notion d'architecture logicielle traduite dans un langage approprié, la possibilité de composition hiérarchique et le partage contrôlé des fonctions liées au déploiement. Construire une application à base de composants implique de disposer d'un modèle, d'outils de composition et d'une infrastructure support. Cette dernière doit mettre en œuvre le modèle et fournir des services génériques (persistance, transactions, sécurité, etc.). L'importance grandissante des méthodes orientées composants amène les intégrateurs à assembler des entités hétérogènes, impliquant de disposer d'un modèle, d'outils de composition et d'une infrastructure support. Parmi les technologies prévues à cet effet, on peut trouver CORBA (*Common Object Request Broker Architecture*) [OMG02] défini par l'OMG, ou Jini [WJINI] défini par la « communauté Java », PolyORB [WPOLY] (prototype d'un intergiciel schizophrène permettent le partage de code par plusieurs personnalités réalisant des modèles de répartition différents). Bien que des infrastructures industrielles à composants soient maintenant présents sur le marché, seule CORBA offre une extension temps réel : RTCORBA (*Real-Time CORBA*). Nous pouvons citer ACE/TAO (*Adaptive Communication Environment/ The Ace Orb*) [SCHM98] comme *framework* à destination du temps réel.

Les architectures proposées pour les intergiciels adaptables contiennent généralement trois parties. Une première partie, dite modifiable constituée par des éléments modifiables et de différentes interconnexions entre ceux-ci, permet de caractériser les services offerts par l'intergiciel, appelé dans ce cas «réflexif ». Une seconde partie de monitoring représente des moniteurs qui observent les ressources et les profils utilisateurs. Ces éléments fournissent les données nécessaires à une description complète du service, appelée méta-description de l'ensemble du service-contexte. Et enfin, une troisième partie dite de contrôle, cette partie est composée par : un adaptateur, qui à partir de la description de l'ensemble du service-contexte, décide de la modification nécessaire pour adapter le service à la nature du profil de l'appel utilisateur; ainsi que d'un assembleur qui exécute ce que l'adaptateur lui prescrit. L'adaptateur utilise des composants existants qu'il puise dans une base de composants comme des solutions d'adaptation.

III - 7 - 3 Les intergiciels dans l'automobile

Ces architectures éprouvées offrent une bonne adaptation dynamique, en gérant à la volée les ordres de reconfiguration. Sauf que dans le contexte des systèmes temps réel embarqués, il est inenvisageable de recourir à une telle dynamique de l'architecture logicielle, même si elle est déterministe. Car si ces systèmes sont prédictibles, il est inutile de surcharger l'intergiciel embarqué de mécanismes d'adaptation qui pénaliseraient inutilement le temps d'exécution. D'autre part, les ressources matérielles étant limitées, il serait incongru de les réquisitionner pour le seul bénéfice de la base des solutions d'adaptation, alors qu'ils sont d'une grande importance pour la couche applicative. C'est alors que les méthodes de configuration statique et d'analyse hors-ligne pour le support des propriétés non fonctionnelles (ordonnancement, fiabilité, sécurité) sont considérées comme un moyen efficace de construire des applications distribuées.

Dans sa thèse, au deuxième chapitre, [RSM06] procède à une étude comparative entre les différentes propositions d'intergiciels embarqués dans l'automobile, telles que Volcano [VOL98], le projet Eureka ITEA EST-EEA [WEURK] et les travaux du consortium AUTOSAR. Il en dégage les constatations suivantes :

- La conception de ces intergiciels est fortement conditionnée par l'utilisation de design patterns [7] qui en augmente la qualité, la réutilisation et la portabilité ;
- Le développement s'oriente vers les modèles orientés composant pour une plus grande flexibilité du processus, pouvant être répartie entre différents acteurs tout en assurant leur interopérabilité lors de l'intégration ;
- Les standards proposés fournissent des méthodologies normatives d'un haut niveau d'abstraction, sans pour autant traiter, ni des problèmes d'ordonnancement (étude de la faisabilité du système), ni de la configuration des propriétés temporelles des objets actifs accomplissant l'intergiciel ;

La conclusion sera de définir une méthodologie [RSM05] de développement d'un intergiciel inspirée de celle du consortium AUTOSAR, et qui aborde les spécifications de l'architecture fonctionnelle de l'intergiciel d'une part. Et d'autre part, présente la configuration temporelle aussi bien du code déployable de l'intergiciel, que celle de ses interactions avec son environnement (contraintes temporelles imposées sur les tâches et les signaux).

III - 7 - 4 Et les intergiciels orientés messages alors ?

L'utilisation d'intergiciels asynchrones orientés messages, fondés sur l'envoi de messages (abstraction faite de ce que l'on a surnommé jusqu'ici de signal), est reconnue comme un moyen efficace de construire des applications distribuées constituées d'entités faiblement couplées, et communiquant par le biais de réseaux à grande échelle (Internet). En effet, on s'accorde à penser que les modèles de communication asynchrones sont mieux adaptés que les modèles synchrones (de type client-serveur) pour gérer les interactions entre entités fonctionnelles faiblement couplés. Le couplage faible résulte de plusieurs facteurs de nature spatiale ou temporelle : l'éloignement géographique des entités communicantes, la possibilité de déconnexion temporaire d'un partenaire en raison d'une panne ou d'une interruption de la communication (pour les usages critiques par exemple). Les modèles de communication asynchrones prennent en compte l'indépendance entre entités communicantes, et sont donc mieux armés pour traiter ces problèmes. Aujourd'hui, les intergiciels asynchrones (comme des systèmes de communication), appelés MOM (*Message Oriented Middleware*), sont très largement répandus dans la mesure où ils constituent la base technologique pour la réalisation des classes d'applications suivantes :

- Intégration de données et intégration d'applications (EAI, B2B, etc.)
- Systèmes ubiquitaires et les usages de la mobilité.
- Surveillance et contrôle d'équipements en réseau.

D'un point de vue du modèle de communication, les intergiciels à messages partagent les concepts suivants :

- Les entités communicantes sont découplées. L'émission (dite aussi la production) d'un message est une opération non bloquante. Par ailleurs, émetteur (producteur) et récepteur (consommateur) ne communiquent pas directement entre eux mais utilisent un objet de communication intermédiaire (boîte aux lettres).
- Deux modèles de communication sont fournis : un modèle point-à-point dans lequel un message produit est consommé par un destinataire unique, et un autre modèle multipoints dans lequel un message peut être adressé à une communauté de destinataires (communication de groupe). Le mode multipoints est généralement doublé d'un système de désignation associative dans lequel les destinataires du message sont identifiés par une propriété du message.

Pour notre système, nous utilisons ces deux modèles de communication respectivement à des niveaux particuliers du système. Par analogie à l'architecture cible abstraite que nous avons définie au § III - 5 - 2, la communication inter-tâches applicative est définie par un modèle point-à-point matérialisé par un échange de signaux. Alors que, à un niveau plus bas,

la communication multiplexée entre contrôleurs réseaux (plus généralement entre ECUs) se fait par un modèle multipoints d'échange de trames.

Plusieurs MOM ont été développés [WMSMQ, WJORAM, WSMQ]. Les travaux de recherche se sont concentrés sur le support de propriétés non fonctionnelles variées (ordonnancement des messages, fiabilité, sécurité, etc). En revanche, la configurabilité des MOM a fait l'objet de moins de travaux. D'un point de vue fonctionnel, les MOM existants implantent un modèle de communication figé : publication/abonnement, événement/réaction, files de messages, etc. D'un point de vue non fonctionnel, les MOM existants fournissent souvent les mêmes propriétés non fonctionnelles pour tous les échanges de messages. Cela réduit leurs performances et rend difficile leur utilisation pour des systèmes et des équipements aux ressources restreintes. En effet, ces propriétés non fonctionnelles n'ont souvent pas été développées de manière modulaire, et les enlever nécessite une ré-ingénierie du MOM. La limitation principale de ces systèmes réside dans le fait qu'ils utilisent des modèles de composants simplistes, n'offrant pas de support pour l'analyse temporelle, c'est-à-dire, leur capacité à respecter les échéances sur une plateforme donnée. De ce fait, les architectures construites sont temporellement difficiles à analyser et ne sont pas prise en compte dans la phase de conception.

Une solution à ces limitations serait de construire des architectures modulaires à base de composants. Ces composants sont assemblés de façon statique (à la construction, par opposition à la dynamique lors de l'exécution en environnement) pour répondre à la fois aux besoins fonctionnels et aux contraintes de l'architecture matérielle embarquée. En pratique, la construction d'intergiciels asynchrones adaptés, passe par l'intégration du modèle des tâches de l'intergiciel lui-même (selon le système d'exploitation utilisé) dans le modèle fonctionnel global des tâches du système pour en déterminer la faisabilité. La conclusion à la faisabilité du système temps réel et déduite aussi bien de l'étude de l'ordonnancement des tâches que de l'analyse de la configuration de la communication entre celles-ci (*frame packing* [8]).

III - 8 Dimensionnement temps réel de l'architecture cible abstraite

Dans les architectures conventionnelles, le dimensionnement temps réel de la réponse maximum d'une fonction dans un calculateur existe d'ores et déjà. Mais ce temps n'était pas influencé par la réparation. Il dépendait essentiellement de l'architecture du logiciel temps réel lui-même embarqué dans le calculateur. Dans l'architecture distribuée que nous considérons, le dimensionnement du pire temps de réponse (WCRT) d'une fonction distribuée doit tenir compte des effets que produit l'architecture physique du système et les autres fonctions concurrentes. Ces effets sont, d'une part, des délais des traitements (production et

consommation), et d'autre part, des délais de communication entre calculateurs (émission et réception). Nous parlons alors de délai de « bout en bout » ou de traversée d'un signal. Ce délai est alors la période de temps calculé entre l'activation de la tâche productrice et la consommation effective du signal (cf. Figure 24). Par analogie à la Figure 20, nous illustrons dans la figure ci-dessous les différentes phases par lesquelles passe un signal lors d'une communication distante entre deux tâches.

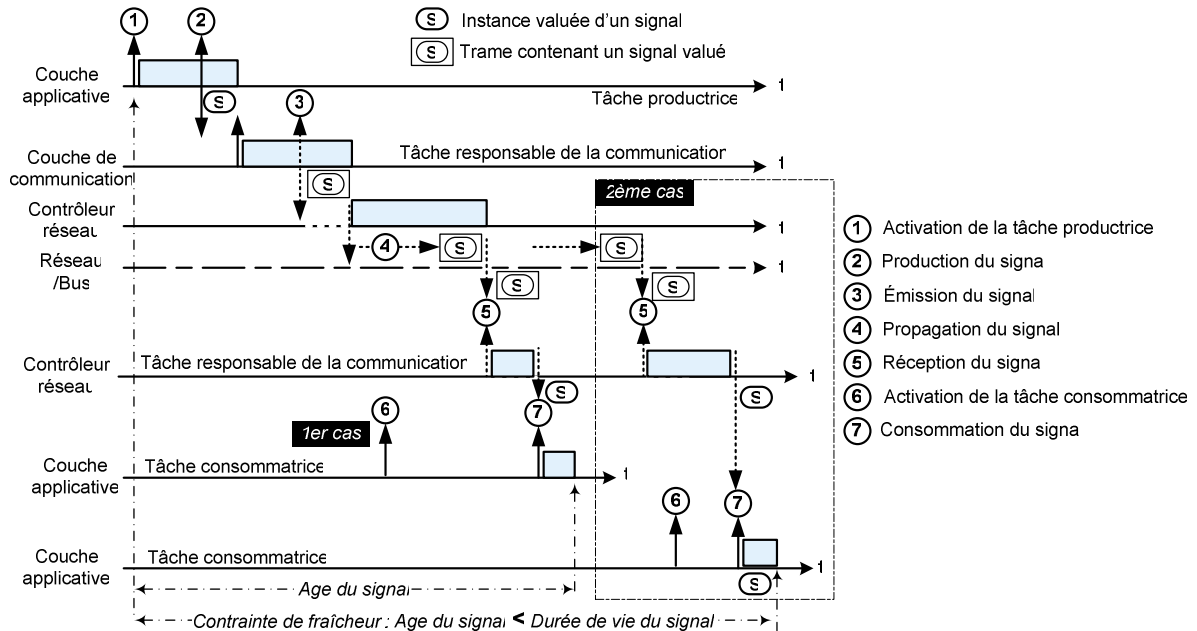


Figure 24 : Contrainte de fraîcheur d'un signal (exécutif préemptif).

Nous pouvons voir (cf. Figure 24) que l'âge d'un signal n'est pas uniquement influencé par le temps qu'il lui est nécessaire pour traverser le réseau d'un site à un autre, mais également par les instants d'activation des tâches consommatrices. Un premier cas pourra être le fait que l'activation de la tâche consommatrice précède la réception d'une trame au niveau la couche de communication. Dans ce cas de figure, l'âge du signal contenu dans cette trame, sera nécessairement plus court, que si une instance de la tâche consommatrice s'active après la réception de la trame.

Comme le suppose [RSM06], les échanges distants entre tâches se font selon le mode à écrasement, ce qui veut dire que la considération n'est pas faite pour l'évaluation des temps de mise en file d'attente (aussi bien en émission qu'en réception) d'une part. Et d'autre part, il est supposé, que comme les signaux sont ordonnancés et empaquetés dans des trames (par un ordonnancement stricte hors-ligne réduisant ainsi la consommation en bande passante) dont ils ne dépassent pas la taille. C'est-à-dire, au pire des cas une trame contiendra au minimum une instance de signal et qu'une instance de signal sera au maximum contenu dans une seule et unique trame. Concrètement il n'y a pas de mécanisme de filtrage des trames ni de mise en tampons.

À défaut de disposer d'un modèle temporel global, il nous est difficile d'estimer à priori le temps d'exécution des différentes routines de service qu'offrira notre système. Pour cela nous avons décidé d'évaluer la qualité de service temporelle une fois que le développement sera achevé, et ce par le biais d'une analyse dynamique (mesure directe de l'exécution) et de réintégrer les résultats temporelles obtenues dans des modèles temporels du système.

Considérant d'une part, (cf. Figure 25) que la couche applicative *A* caractérisée par l'ensemble des tâches soumettant leurs opérations d'envoi et de réception à une couche *B* de communication matérialisée par un bus. Et d'autre part, l'intergiciel comme défini précédemment est une troisième couche intercalée à la réunion des deux autres. Puisque le système qu'on s'est proposé de réaliser est un intergiciel asynchrone orienté messages (*MOM : Message Oriented Middleware*), celui-ci devra offrir des mécanismes d'interposition et d'interception en précédant les opérations de couche *A* par ce que l'on caractérisera par prélude et qui signifiera toutes les interactions de service en émission entre cette couche *A* et notre système. La désignation de postlude servira à décrire les interactions de service en réception entre notre système et la couche applicative *A*.

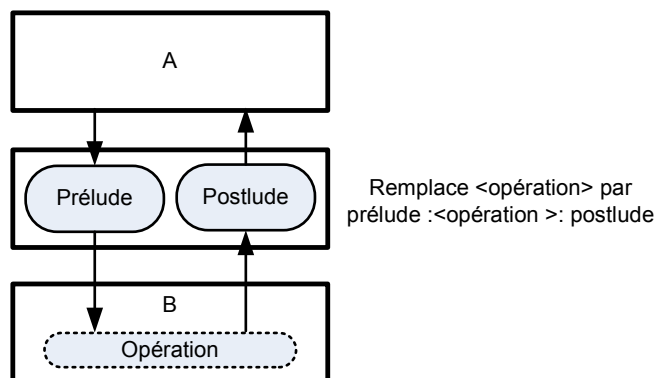


Figure 25 : Modèle logiciel d'Interception/Interposition.

Une seule considération pourra être faite à ce niveau de la réalisation, c'est que conscient de l'asynchronisme de la communication on peut émettre l'hypothèse que le temps d'exécution au pire des cas des mécanismes d'interception/interposition ($T_{interception/interposition}$) est borné, en supposant qu'il soit identique en prélude et en postlude. Ce qui implique qu'il est possible, dû au fait que les hypothèses sur le canal font que le temps de transmission ($T_{transmission}$) soit borné (car nous considérant un canal de transmission idéale), de définir une borne temporelle supérieure pour la traversée ($T_{bout\ en\ bout}$) des signaux au travers de notre système. Ce temps pourra servir pour l'estimation du pire temps d'exécution des signaux en l'additionnant au pire temps d'exécution intrinsèque de chaque signal. Comme le montre la Figure 26 :

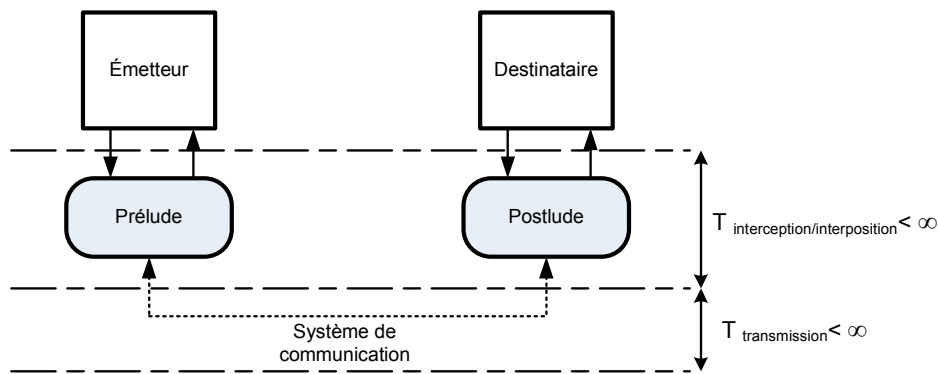


Figure 26 : Temps de bout en bout.

$$\text{Puisque } T_{\text{bout en bout}} = (2 * T_{\text{interception/interposition}}) + T_{\text{transmission}} < \infty$$

$$\text{Et } T_{\text{interception/interposition}} < \infty,$$

$$\text{Ainsi que } T_{\text{transmission}} < \infty,$$

$$\text{Il vient que } T_{\text{bout en bout}} < \infty.$$

Tableau 2 : Preuve de la limite du temps de bout en bout.

Un système temps réel, comme le notre, doit s'assurer à tout instant du respect des contraintes de fraîcheur (cf. Figure 24) de tout signal qu'il gouverne. Cette contrainte impose que l'âge d'un signal (ou délai de bout en bout) soit toujours inférieur à sa durée de vie (fraîcheur limite). Cela dit, l'âge maximal concret d'un signal dépend des différents traitements qu'il subit (production, transmission, réception, consommation) et du déroulement de l'acheminement sur le réseau de la trame qui le renferme. Il est donc nécessaire d'introduire une contrainte temporelle sur la trame aussi exprimée par une contrainte de fraîcheur et une échéance relative. Dans le modèle point-à-point, l'échéance relative d'une trame contenant un seul signal est l'intervalle de temps calculé par soustraction des différents temps d'interception/interposition à partir de la fraîcheur de ce signal. Par contre, lorsqu'une trame contient plus d'un signal, cette échéance de la trame est calculée comme le plus petit intervalle de temps relativement à chaque signal. Enfin, dans le modèle multipoints, il suffit de considérer la fraîcheur la plus stricte imposée l'un des consommateurs des signaux contenus dans une trame.

III - 9 Conclusion sur le contexte

Les systèmes réactifs embarqués appartiennent à une classe de systèmes informatiques caractérisés par la présence de contraintes non fonctionnelles dans leurs exigences. Parmi ces contraintes, nous avons identifié les contraintes de temps et les exigences liées à la sûreté de fonctionnement. Nous avons également rappelé que la nature critique de ces systèmes

(imposée dans leur environnement) nécessite qu'ils soient prédictibles et que les techniques de développement imposent de conserver une certaine simplicité pour les aspects logiciels et matériels tout en augmentant leur réutilisation.

L'exemple du § III - 3 - 2 nous a permis de souligner le lien entre les systèmes temps réel embarqués et le domaine automobile, lien qui conduit naturellement à une architecture logicielle composée d'entités concurrentes. Nous avons illustré cette décomposition en décrivant une mise en oeuvre asynchrone, c'est à dire basée sur l'utilisation d'un exécutif temps réel qui effectue (entre autres) l'interface entre les événements du procédé et les traitements de la partie applicative.

Nous avons ensuite souligné qu'il est souvent nécessaire d'envisager l'utilisation d'une architecture matérielle distribuée autour d'un bus multiplexé afin de répondre aux différentes exigences non fonctionnelles : coût, encombrement, etc. Cela ajoute des problèmes liés à la distribution. De plus, le besoin de prédictibilité du comportement du système global se répercute nécessairement sur l'architecture matérielle et en particulier sur les protocoles de communication qui doivent être déterministes et offrir des délais de transmission bornés.

Après avoir conclu par les travaux de la thèse [RSM06], faisant foie de référence de ce mémoire, nous avons défini une architecture logicielle abstraite dans laquelle le dimensionnement temps réel impose l'adjonction d'un intergiciel qui soit le garant d'une distribution respectant toutes les contraintes fonctionnelles et temporelles du système. La construction à base de composants de cet intergiciel devra donc répondre à toutes les contraintes d'un système temps réel embarqué.

IV Réalisation

Après avoir présenté le contexte de notre mémoire, nous nous intéressons dans le reste de ce document à la conception et à la mise en œuvre du projet *GenIETeR*.

IV - 1 Méthode et contraintes du processus de développement

L'application des meilleures pratiques méthodologiques au sein d'une démarche réfléchie, permet de construire un processus de développement adapté à la typologie d'un projet. Dans notre cas, il est intéressant et nécessaire de positionner notre démarche dans un processus de développement dont la finalité est de produire des applications de qualité qui répondent aux besoins de leurs utilisateurs dans des temps et coûts prévisibles. En conséquence, le processus devra alors permettre de contrôler le développement technique, qui se concentre principalement sur la qualité de la production.

IV - 1 - 1 Modèles, patrons et canevas d'architecture répartie

La course à la réutilisation, l'intégration et la fiabilité nécessitent la recherche d'une conception générique, et paradoxalement communes aux systèmes répartis, qui revêtent trois aspects : définition de modèles d'architectures (formel ou semi-formel), élaboration de patrons génériques d'architecture, construction de canevas logiciels associés.

La modélisation des systèmes répartis introduit par rapport aux architectures centralisées un degré important de complexité dû en particulier à l'asynchronisme induit par le système de communication, à la prise en compte de l'hétérogénéité des architectures matérielles ; Qui plus est si on modélise des systèmes temps réel (qu'ils soient embarqués) induisant des considérations de caractéristiques temporelles (temps d'exécution, période d'activation, échéance relative et priorités). Il est donc essentiel de concevoir une solution qui répond à ces spécifications techniques, cette conception est qualifiée de générique car elle est entièrement indépendante des aspects fonctionnels spécifiés dans une application concurrente (cf. § III - 1 - 2). Pour illustrer ces propos, on citera par exemple la branche d'activité droite : branche technique du processus 2TUP ([ROQ03], chapitre 9) qui recense toutes les contraintes et les choix dimensionnant la conception d'un système (cf. Figure 27).

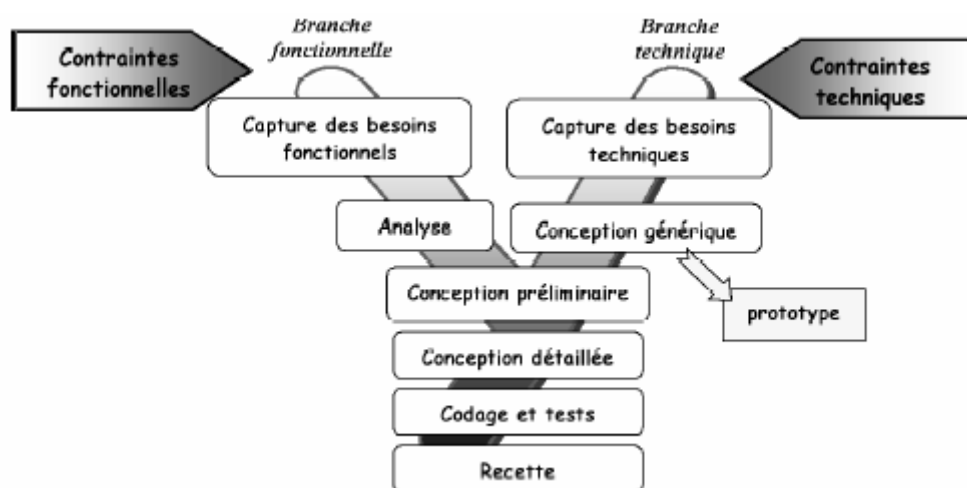


Figure 27 : Situation de la conception générique dans 2TUP. [ROQ03]

Cette branche comporte une conception générique, qui définit les composants (sous forme de classes techniques réutilisables) nécessaires à la construction de l'architecture technique. Cette conception est la moins dépendante possible des aspects fonctionnels. Elle a pour objectif d'uniformiser et de réutiliser les mêmes mécanismes pour tout un système. L'architecture technique construit le squelette du système informatique et écarte la plupart des risques de niveau technique. L'importance de sa réussite est telle qu'il est conseillé de réaliser un prototype pour assurer sa validité. Cette phase de prototypage est fortement conseillée, car la qualité d'une conception générique conditionne généralement celle du développement pour le reste du projet.

C'est pour son universalité que nous avons décidé d'utiliser UML [WUML] comme notre langage de conception du projet *GenIETeR*. Ceci pour spécifier d'abord sous forme de canevas logiciel (cf. ci-dessous) les besoins techniques auxquels notre système devra répondre, avant d'en implémenter la solution. Il devient aujourd'hui, sans aucun doute, un langage semi-formel de communication efficace pour à la fois spécifier, esquisser et construire une conception.

Dans une conception générique, le concept d'intégrité se manifeste par un ensemble de classes dites techniques que l'on réutilise pour développer les différentes composantes fonctionnelles du système. Cependant, une classe technique n'ayant rarement d'effet toute seule, c'est pour cela que l'on y associe le concept-clé de la conception générique : le *framework* technique.

IV - 1 - 2 Canevas logiciel technique

Un canevas logiciel (*software framework* en anglais) [BOO99] est un squelette de programme réutilisable pour une famille d'applications. Les canevas sont souvent mis en

œuvre par des objets et prennent alors la forme d'un ensemble de classes (souvent abstraites) à être adaptées (par extension ou surcharge) à des cas d'usage spécifique (par exemple le package Swing de Java ou les classes MFC de Microsoft). Celles-ci collaborent à l'unisson pour une responsabilité qui dépasse celle de chacune prise individuellement. Un *framework* technique, au sens du processus 2TUP, ne concerne que les responsabilités fonctionnelles. Un canevas logiciel peut être abstrait ou concret. Pour le cas abstrait, il est composé de classes ou d'interfaces pouvant être directement réutilisées dans un projet ce qui permet juste de structurer le modèle de configuration logicielle. Le cas du canevas logiciel concret, détermine à la fois le modèle de configuration logicielle et le modèle d'exploitation par une combinaison d'interfaces à implémenter et de classes à réutiliser. Dans le modèle d'exploitation, un *framework* peut revêtir la forme de composants distribués ou de bibliothèques partagées.

Les classes, les interfaces et les diagrammes de classes que nous présenterons dans cette partie constituent les briques de construction d'un modèle logique de conception générique. Ils seront étoffés de diagrammes dynamiques pour en illustrer le fonctionnement de la structure. On utilise fréquemment des valeurs étiquetées ou stéréotype : une extension des propriétés d'un concept UML en apportant de nouvelles informations de spécification, pour alléger les diagrammes de leurs implémentations. Chaque valeur étiquetée partage et factorise une même représentation (comportement et structure) de conception.

Comme nous l'avons énoncé dans la section précédente notre conception de l'intergiciel est fortement conditionnée par l'utilisation de *design patterns* [7].

IV - 1 - 3 Design Patterns

Un patron d'architecture (*design pattern*) [GOF] vise à capturer des règles de conception permettant de répondre à une classe de besoins spécifiée. Les patrons doivent identifier les abstractions sous-jacentes et intégrer l'expérience acquise lors de la résolution de problèmes apparentés. Leurs usages apportent évolutivité, lisibilité et efficacité aux développements. La recherche sur les patrons de conception pour l'intergiciel est apparue dans les années 1990, et a été particulièrement active depuis 1995. Elle a permis d'identifier les prescriptions d'architecture de base [POSA1] dans la mesure où ils représentent pour les néophytes un catalogue des meilleures pratiques à adopter. Les prototypes de recherche dans le domaine de l'intergiciel sont maintenant principalement développés sous forme de canevas pour pouvoir être réutilisés. Ce que nous avons essayé d'appliquer dans notre projet en puisant dans ces catalogues [GOF, POSA1] et ceux proposés par Douglas C Schmidt dans [WSCHM] à savoir les *Patterns for Concurrent, Parallel, and Distributed Systems (Concurrency Patterns and*

Idioms, Event Patterns) pour le canevas de notre intergiciel adaptable (cf. § IV - 3 - 2). Nous reviendrons sur les patrons utilisés lorsque nous présenterons le modèle d'exploitation de notre intergiciel.

IV - 2 Spécification générale

Comme en génie du logiciel la spécification est une étape du développement d'un programme ou d'un système informatique qui consiste à décrire ce que le système en construction doit faire, et puis comment il doit le faire. Ceci passe par une définition de sa structure et de son comportement pour délivrer les services qu'il offre.

IV - 2 - 1 Synthèse de l'existant

Initialement l'accord avec les tuteurs du LORIA était que notre travail consiste en une implémentation sur une plateforme réelle des travaux de [RSM06] avec une conception du générateur de l'intergiciel. Après exploration de ces travaux nous nous sommes aperçu que les modèles conceptuels proposés répondaient bien aux concepts énoncés dans la thèse, mais manquaient clairement la prise en considérations des besoins fonctionnels. Quelques modèles d'implémentation sous forme de diagrammes de classes UML existent sans pour autant rendre compte des détails liés au dimensionnement concurrent des tâches caractérisant l'intergiciel. Des classes de ces diagrammes sont annotées par un pseudo code pour aider à leur assimilation mais ne donnent aucune explicitation sur leur nature, c'est-à-dire, quelles sont celles actives et quelles sont d'entre elles qui représentent des bibliothèques du système. La considération est faite pour que la communication entre les tâches applicatives et l'intergiciel soit asynchrone, sauf qu'il n'y a aucune mention dans les représentations de ces mécanismes.

En théorie, la documentation d'un projet informatique professionnel devrait toujours être assez complète et explicite. Cependant la pratique montre que c'est rarement le cas, et malheureusement, les projets de recherche n'échappe pas à cette constatation. Car il est incontestable qu'un travail de thèse aboutisse généralement à un prototype. Ce manque de documentation nous a obligé à faire une étude approfondie du code source. Cependant, il est indéniable qu'un effort a été fourni pour que la thèse puisse être en même temps la documentation du projet et la capitalisation des contributions réalisées.

C'est à la suite de ce constat que l'on s'est fixé pour objectif de refondre la conception aussi bien de l'intergiciel que de concevoir le générateur qui nous incombait depuis le début. La conception devrait se rapprocher de la meilleure façon aux hypothèses de la méthodologie

énoncée dans [RSM05]. Et nous voulons que ce mémoire soit à l'image de la documentation de ce projet.

Pour simuler sa technique [RSM03] de configuration des trames (*frame packing* [8]), R. Santos Marques s'est appliqué à réaliser un codage basé sur les heuristiques de *Bandwidth-Best-Fit decrecreasing* [COFF96] (BBFd) et de *Semi-Exhaustive* [RSM06, chapitre 7, page 82] (SE) avec une génération aléatoire de l'ensemble des trames selon une méthode détaillée dans [WORLO02], sous les hypothèses suivantes :

- Les ECUs composant le système sont connectés à un seul réseau local de type CAN (cf. § III - 5);
- Les tâches sont périodiques à départ simultané (cf. § III - 1 - 11) ;
- La consommation de plusieurs signaux peut être faite à toute instance d'une tâche (cf. § III - 1 - 10);
- La production d'un signal par une tâche est unique mais peut être produit à toute instance de la tâche ;
- Le signal produit pourra toujours être contenu dans une seule trame sans dépasser la taille des données utilisées imposées par le protocole du réseau.

Le premier algorithme BBFd et une approche hors-ligne (effectuée avant l'exécution du système temps réel, et ce en fonction des caractéristiques du jeu de test) construisant une séquence (une solution) de trames dont les caractéristiques temporelles assurent l'exigence de fraîcheur des signaux qu'elles transportent tout en minimisant la consommation de bande passante (sans tenir compte du nombre de trames). A contrario, l'algorithme *Semi-Exhaustive* parcourt exhaustivement l'espace des solutions (limité à 12 signaux par ECU), trié par ordre incrémental, à la recherche de la séquence de trames qui minimise la consommation de la bande passante. La faisabilité (ou l'ordonnabilité) des trames est ensuite vérifiée par un algorithme d'Audsley [AUD91]. Cet algorithme optimal, basé sur une affectation incrémentale des priorités de la moins prioritaire à la plus prioritaire, s'appuie sur une condition suffisante de propriétés des temps de réponse. En effet, Audsley a constaté qu'une diminution du niveau de priorité (appliquée dans son étude à une tâche, par analogie à une trame dans notre cas) n'engendre pas d'augmentation du temps de réponse des autres tâches. De plus, une tâche est insensible à l'ordre des affectations des tâches plus prioritaires qu'elle. L'algorithme détermine donc en premier lieu la trame de plus faible priorité en effectuant un test d'ordonnabilité basé sur le temps de réponse (en utilisant l'instant critique sachant que les autres trames sont plus prioritaires) permettant de vérifier que cette affectation conduira à un ordonnancement valide et faisable.

Comme indiqué sur le schéma synoptique suivant (cf. Figure 28) si un test d'ordonnançabilité s'appuyant sur une condition suffisante d'ordonnançabilité produit un résultat positif, donc l'ensemble de trames est définitivement ordonnançable. Par contre, si le résultat est négatif, l'ensemble peut être ordonnançable ou pas. De la même manière, si un test d'ordonnançabilité s'appuyant sur une condition nécessaire d'ordonnançabilité produit un résultat positif, l'ensemble des tâches peut ne pas être ordonnançable alors que s'il est négatif, l'ensemble des trames est définitivement non ordonnançable.

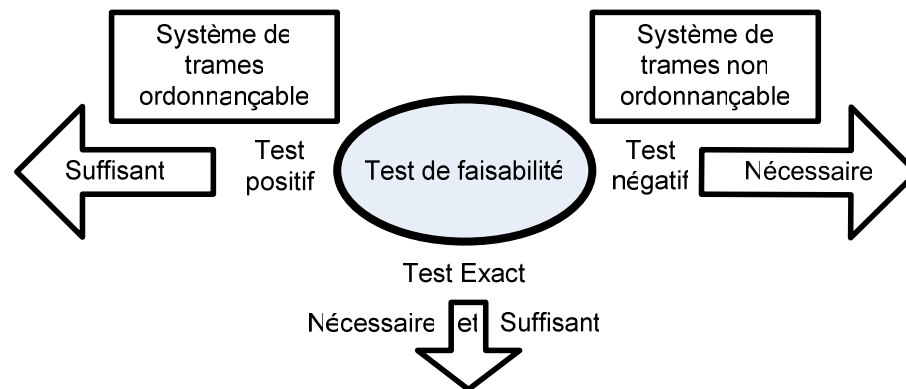


Figure 28 : Synoptique des différents résultats d'un test d'ordonnançabilité.

Ainsi pour le *frame packing*, si le test d'Audslay réussit, alors une optimisation locale sur les signaux composant la trame est réalisée pour produire une configuration temporelle de la messagerie. Cette structuration des contraintes temporelles en un trafic réseau des trames et en genèse des signaux, permet de savoir :

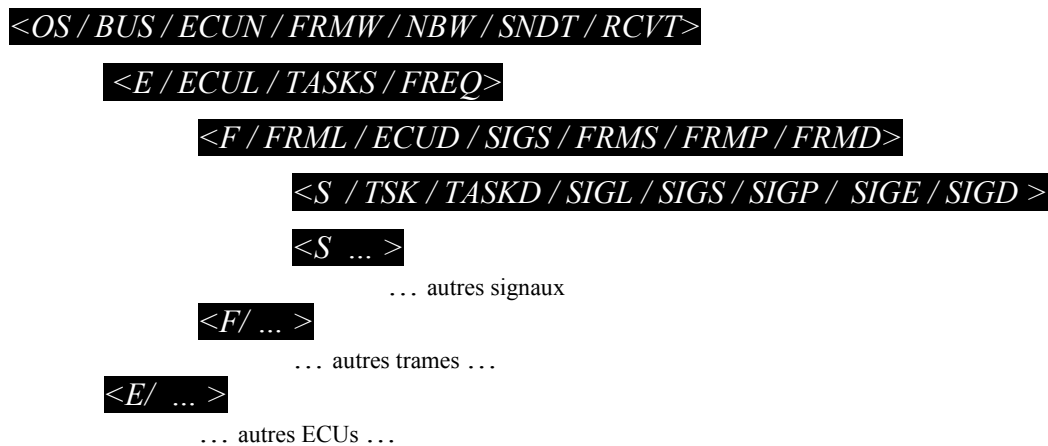
- Quelles sont les trames à envoyer par chaque ECU ;
- Quels sont les signaux qui composent chacune des trames ;
- Quelles sont les caractéristiques des trames ;
- Quelle est la priorité de chaque trame pour le protocole d'accès au medium de communication.

Ainsi notre intergiciel, qui n'est pas réflexif, exécutera en-ligne (par un dispositif appelé ordonnanceur, *Scheduler* en anglais) la configuration des trames engendrées par ce *frame packing* (réalisé en hors-ligne) qui devra :

- Minimiser la consommation en bande passante ;
- Garantir la faisabilité de la messagerie (ensemble des trames sur le réseau) ;
- Assurer le respect des contraintes de fraîcheur associées aux signaux.

Pour interfacer la sortie cette implémentation à notre système, nous nous sommes permis d'adapter le code source réalisé à la génération d'un fichier à plat (cf. Figure 1, illustré par *Temporal traffic specification*). Ce fichier que l'on considère comme un point d'entrée à notre système permet de spécifier d'une part la distribution des tâches sur chaque ECU (par l'unicité

de leur localisation, § III - 5 - 2) et d'autre part la genèse des signaux des trames devant être gérés par notre système. La structure de ce fichier (cf. Tableau 3) est à l'image de la distribution physique (ou modèle de répartition) d'une application temps réel embarquée, c'est-à-dire que sa lecture permet de connaître l'architecture globale de l'application et son dimensionnement temps réel.



Abréviation	Synonyme	Unité
OS	<i>Operating system</i> , type d'OS utilisé	
BUS	<i>Bus</i> , type de bus utilisé	
ECUN	<i>ECU Number</i> , nombre d'ECU de l'application	
FRMW	<i>Frame Width</i> , taille maximale d'une trame	bits
NBW	<i>Network Bandwidth</i> , bande passante du bus	kb/s
SNTD	<i>Send Time</i> , temps d'exécution du préluide	ms.
RCVT	<i>Receive Time</i> , temps d'exécution du postluide	ms.
E	<i>ECU</i> , début de balise d'un ECU	
ECUL	<i>ECU Label</i> , désignation unique de cet ECU	
TASKS	<i>Tasks</i> , tâches continues dans cet ECU	
FREQ	<i>Frequency</i> , fréquence de cet ECU	MHz
F	<i>Frame</i> , début de balise d'une trame	
FRML	<i>Frame Label</i> , désignation unique de cette trame	
ECUD	<i>Destination ECU</i> , ECU destinataire de cette trame	
SIGS	<i>Signals</i> , signaux contenus dans cette trame	
FRMS	<i>Frame Size</i> , taille de cette trame	bits
FRMP	<i>Frame Period</i> , période de cette trame	ms.
FRMD	<i>Frame Deadline</i> , échéance de cette trame	ms.
S	<i>Signal</i> , début de balise d'un signal	
TSK	<i>Task</i> , tâche productrice de ce signal	
TASKD	<i>Destination Task</i> , tâche(s) destinataire(s) de ce signal	
SIGL	<i>Signal Label</i> , désignation unique de ce signal	
SIGS	<i>Signal Size</i> , taille de ce signal	bits
SIGP	<i>Signal Period</i> , période de ce signal	ms.
SIGE	<i>Signal Execution Time</i> , temps d'exécution de ce signal	ms.
SIGD	<i>Signal Deadline</i> , échéance de signal	ms.

Tableau 3 : Structure et nomenclature de la spécification du trafic.

Suite à ce processus de configuration de la messagerie, et dans le contexte de sa méthodologie qui vise à définir un ensemble de tâches capables de représenter l'intergiciel sur

chaque ECU, [RSM06] propose une deuxième étape consistant en une caractérisation de ces tâches (sous un exécuteurs OSEK/VDX OS) réalisant l'intergiciel par une configuration de leurs flots de contrôle. Cette étape est complétée par une détermination des priorités de l'ensemble des tâches applicatives et celles de l'intergiciel formant le système (applicatif et celui intergiciel) sur chaque ECU. Il propose ainsi de résoudre :

- Combien de tâches quantifient l'intergiciel sur chaque ECU ;
- Quel est le rôle optimal de chaque tâche pour qu'elle puisse réaliser les services de communication de l'intergiciel ;
- Par quel modèle temporelle faut-il caractériser ces tâches.

Les mécanismes proposés doivent répondre à ces interrogations tout en permettant que l'intergiciel puisse réaliser les quatre services de communication suivant :

- Interception des signaux : réception et mémorisation des signaux valués en provenance des tâches applicatives ;
- Construction et transmission des trames : recherche des valeurs des signaux mémorisés, empaquetage des trames avec ses valeurs, et leur mise sur le réseau pour transmission ;
- Interposition et traitement des trames : interception des trames, dépaquetage des trames, et mémorisation des valeurs des signaux reçus ;
- Délivrance et consommation des signaux : recherche de la valeur mémorisée d'un signal et sa délivrance à une tâche applicative consommatrice.

Il conclut pour cela sur le fait de réquisitionner deux tâches sur chaque ECU pour implémenter ces mécanismes. La première se chargera des deux premiers mécanismes (interception des signaux, construction et transmission de trames) définie selon deux modèles de configuration temporelle : *multiframe* [MOK96] ou *generalized multiframe* [TAKA97, BAR99] (cf. § IV - 4, pour plus de détails). La seconde prendra en charge les deux autres fonctions : interposition et délivrance des signaux. Elle est pour cela caractérisée par un modèle de tâche sporadique (cf. § III - 1 - 10) qui peut être implémenté comme un service de routines d'interruption (ou ISR pour *Interruption Service Routine*) permettant d'associer un traitement à chaque réception de trame. Nous reviendrons sur cette configuration des tâches de l'intergiciel dans le paragraphe consacré à la réalisation du générateur. Par la suite, nous qualifierons la première tâche de tâche en charge de l'envoi et la seconde de tâche en charge de la réception.

Pour ce qui est de l'assignation des priorités aux tâches, [RSM06] prévoit d'attribuer la plus haute (au sens de OSEK/VDX OS, cf. Annexe 1 - 2) à la tâche en charge de l'envoi. Et le

fait de définir la tâche en charge de la réception comme ISR cela lui confère, au sens d'OSEK/VDX OS [WOSEKISR], une priorité supérieure aux tâches.

IV - 3 Modélisation UML

La notion d'architecture logicielle est centrée sur les entités qui vont constituer le système (quelles sont-elles, comment sont-elles organisées, comment communiquent-elles, etc. ?) par opposition à celle de la conception détaillée qui s'attache à l'élaboration des algorithmes et des structures de données des entités et qui se concentre donc successivement sur des petites parties du système, l'étude et la conception de l'architecture nécessitent d'adopter une vision globale du système. L'utilisation de méthode de conception permet de structurer ce passage en proposant des modèles d'implémentation pour certaines classes de besoins.

Mais l'utilisation d'une méthode restreint cependant l'espace des implémentations à un sous ensemble particulier (par exemple l'utilisation d'une méthode orientée objet conduit à une implémentation suivant une architecture orientée objet). Cette restriction est levée par l'utilisation du niveau intermédiaire constituée par l'architecture logicielle (différentes structurations peuvent être envisagées et leur capacité à résoudre les problèmes posés peut être évaluée). Une fois le type d'architecture déterminé, le passage à l'implémentation (dans un langage particulier) peut être entrepris.

Comme il s'agit de décrire un système applicatif temps réel, il est nécessaire de mettre l'accent sur le fait que nous n'utilisons pas de profil UML particulier pour notre modélisation. Premièrement, ceci est dû au fait que notre système n'intègre pas particulièrement de contraintes temporelles strictes en soit (mais celles imposées par la configuration temporelle du trafic de la communication distante) car nous considérons l'hypothèse que la définition de celles-ci est à la charge du développeur d'application utilisant notre intergiciel.

IV - 3 - 1 Contexte d'utilisation du système

Ainsi comme présenté au § III - 3 - 4, nous avons démontré le positionnement de notre méthodologie de développement dans le contexte d'un processus global de réalisation logicielle dans le domaine automobile. Pour le contexte d'utilisation de notre système, et dans ce qui suit, nous ne ferons pas de distinction entre acteurs principaux et acteurs secondaires mais on devra faire la différence entre les acteurs humains et ceux non humains. Ceci dit, comme dans toute méthodologie de réalisation de système embarqué temps réel, il est nécessaire de prévoir le plus tôt possible, dès les premières phases du cycle de vie, aussi bien l'environnement matériel que logiciel.

Donc voici le contexte d'utilisation de notre système :

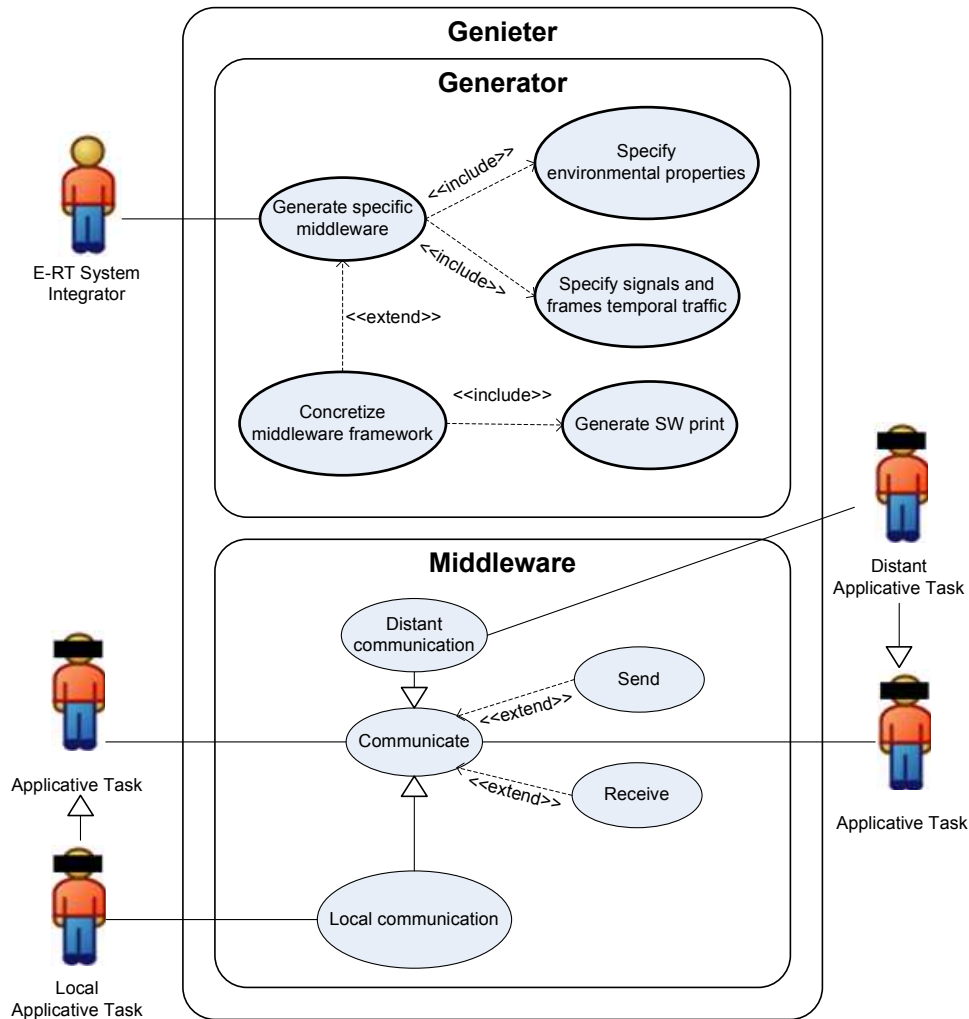


Figure 29 : Diagramme des cas d'utilisation.

Notre système applicatif, *GenIETeR*, se décompose en deux sous-systèmes (regroupement d'éléments de modélisation dont le but est de fournir une même unité de comportement au sein d'un même système mais dont le procédé de fabrication est indépendant). Le premier sous-système est le générateur. Il est censé produire l'empreinte d'un intergiciel à la demande d'un acteur humain. Cette génération de code source se produit par la concrétisation d'un invariant du *framework* de l'intergiciel grâce à une injection de code source (données et flots de contrôle des tâches de l'intergiciel qui font appel à des bibliothèques de composants élémentaires statiques). Le code source de l'intergiciel ainsi produit doit être déployable sur une plateforme cible. Il peut être soit importé dans le projet du développeur logiciel du système distribué en tant que source non compilé, soit être utilisé en tant que bibliothèque à la génération de liens dans le processus de compilation. Ainsi, les tâches applicatives développées pourront invoquer les primitives de cet intergiciel adapté par des appels de service minimisant la consommation des ressources.

Le second sous-système représente un invariant des fonctionnalités de l'intergiciel générique qui permettent une communication asynchrone orientée messages entre les différentes tâches de la couche applicative. Même si [RSM06] ne s'intéresse qu'à des mécanismes de communication distante entre tâches applicatives car il considère un *frame packing* excluant les signaux locaux, nous avons décidé pourtant de prévoir que notre intergiciel offrirait des services de communication locale. Cette communication locale pourrait aussi bien faire l'objet d'une analyse d'ordonnabilité à priori qui n'est pas cruciale à ce stade du projet, mais qui pourrait être envisagée ultérieurement pour donner lieu à un intergiciel fonctionnellement complet pour tout type d'application temps réel embarqués dans un véhicule.

Dans ce qui suit nous adopterons cette dichotomie du système en générateur et intergiciel générique adaptable pour organiser la rédaction de ce mémoire.

IV - 3 - 2 Description des acteurs

Les acteurs du système sont donc :

- *Embedded Real Time System Integrator* : L'Intégrateur de système temps réel embarqué est un acteur humain d'interface utilisant le système *GenIETeR* pour générer un intergiciel concret en lui fournissant une spécification (sous forme de fichier à plat, cf. § IV - 2) aussi bien du trafic réseau (signaux et trames) que celle de l'architecture globale du système (propriétés liées à l'environnement matériel et la distribution).
- *Applicative Task* : Une tâche applicative, comme définie au § III - 1 - 3, est un flot de contrôle séquentiels dédiés au traitement d'une des composantes de l'application temps réel embarquée, et qui interagit d'une manière asynchrone avec le système au travers d'une API des primitives orientées messages pour communiquer avec d'autres tâches de l'application.
- *Local Applicative Task* : Une tâche applicative locale (en opposition à une tâche applicative distante) est un stéréotype d'acteur non humain définissant une tâche périodique (cf. § III - 1 - 11) se trouvant physiquement sur un même ECU (partageant les mêmes ressources) qu'une autre tâche concurrente avec laquelle elle devrait communiquer pour réaliser les fonctionnalités de l'application temps réel.
- *Distant Applicative Task* : Une tâche applicative distante est un stéréotype de tâche périodique, qui physiquement placée sur un ECU distant (mais appartenant au même réseau local) de celui d'une tâche locale doit interagir avec cette dernière par échange de signaux. Ces échanges, en plus de respecter des contraintes temporelles strictes

(instants adéquats d'envoi et limites de fraîcheur en réception), doivent être indépendants de la localisation des intervenants et de l'hétérogénéité des plateformes traversées par ces communications.

IV - 3 - 3 Les cas d'utilisation

Génération d'un intergiciel spécifique

- Titre : *Generate specific middleware.*
- But : Permettre à l'intégrateur d'un système temps réel embarqué de générer l'empreinte logicielle d'un intergiciel adapté à la spécification de son application.
- Résumé : L'intégrateur de l'application temps réel embarquée soumet au système la spécification de la genèse des interactions entre les tâches ainsi que les spécificités de la distribution du système pour produire le code source d'un intergiciel répondant à ces spécifications.
- Acteur : Intégrateur du système temps réel embarqué.
- Enchaînements : La concrétisation du *framework* de l'intergiciel débute lorsque l'intégrateur soumet au système le fichier à plat de la spécification de son application. Cette spécification ayant préalablement fait l'objet d'un *frame packing* (cf. § IV - 2) donne une vue sur la configuration des signaux dans les trames, ainsi que l'envoi de celles-ci aux instants adéquats (c'est-à-dire aux instants tels que les contraintes de fraîcheur sur les signaux soient satisfaites). Cette concrétisation passe par la configuration de l'intergiciel en attribuant des paramètres aux tâches en charge des services de l'intergiciel de manière à ce que les contraintes temporelles imposées sur les signaux et les trames soient respectées. Cette configuration construit une architecture logicielle adéquate en puisant dans un *framework* technique constitué d'un catalogue de composants pré-implémentés. Par assemblage de ces composants et en injectant le flot de contrôle et le flot de données nécessaires à l'entrelacement des tâches représentant l'intergiciel (vis-à-vis des tâches applicatives), un code source déployable sur une plateforme contenant un exécutif temps réel est généré pour chaque ECU du système.

Interaction avec l'intergiciel

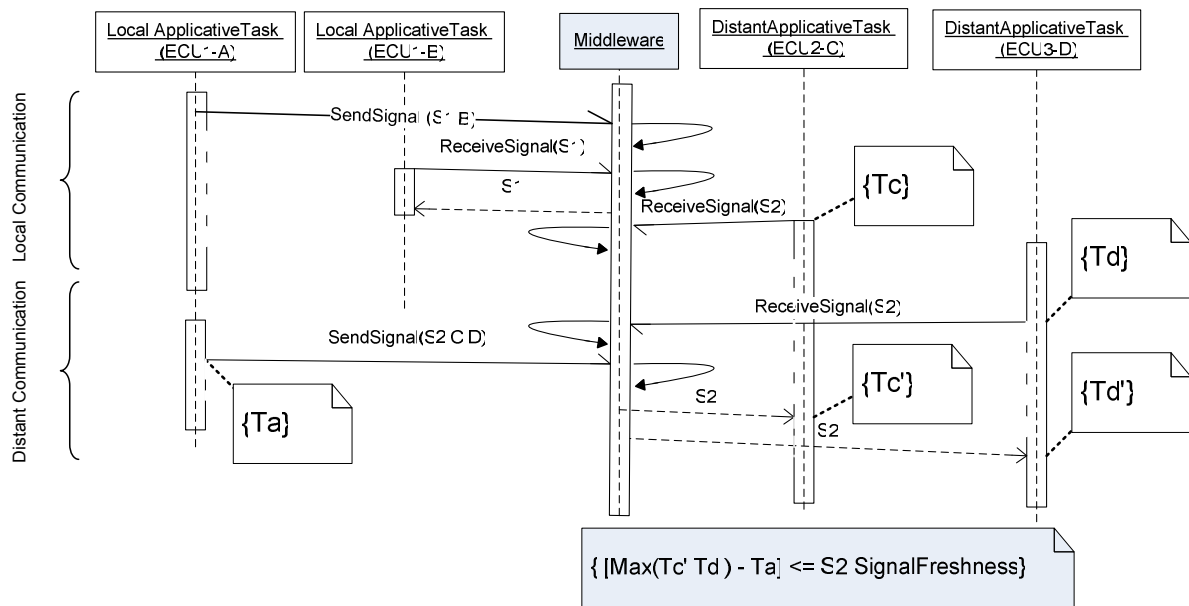


Figure 30 : Diagramme de séquences (utilisation de l'intergiciel).

- Titre : *Communicate*.
- But : Permettre aux tâches applicatives d'accéder par une interface standard à des services de communication asynchrones indépendamment de la localisation des intervenants et de l'hétérogénéité des supports de communication.
- Résumé : Les tâches applicatives d'un ECU soumettent leurs ordres asynchrones d'envoi et de réception des signaux au système (à l'intergiciel généré) qui se charge de les empaqueter/dépaqueter dans les trames adéquates en respectant les contraintes temporelles (cf. Figure 1) exprimées par la spécification et de les transmettre/réceptionner aux tâches appropriées.
- Acteurs : Une tâche applicative locale et une tâche applicative distante.

Nous nous permettons de labelliser les entités du système grâce aux hypothèses émises au § III - 5 - 2 et ce pour mettre accent sur des interactions indépendantes de la localisation des tâches :

- Enchaînements : Dans le cas de deux tâches applicatives (*productrice* : *A* et *consommatrice* : *B*) appartenant à un même ECU (*ECU 1*) voulant échanger un signal (*S1*) (cf. Figure 30, *local communication*). La tâche productrice (*A*) appelle la primitive d'envoi de l'intergiciel (`SendSignal(S1,B)`) pour notifier le système d'un ordre asynchrone d'envoi d'un signal valué. La tâche consommatrice (*B*) fait appel à l'interface de l'intergiciel (`ReceiveSignal(S1)`) pour demander une réception d'un signal (*S1*). L'intergiciel se charge alors de fournir le signal à la tâche consommatrice sans en acquitter la tâche productrice. Pour autant les tâches ne sont pas bloquées par

les appels mais peuvent être préemptés par l'exécutif temps réel. Pour le cas d'une communication distante multipoint asynchrone (cf. Figure 30, *distant communication*) entre plusieurs tâches (A : productrice et C, D : consommatrices) se trouvant sur des ECU distants (A sur un ECU 1, C sur un ECU 2 et D sur un ECU 3). Par activation, l'instance de la tâche productrice (A) appelle l'intergiciel ($SendSignal(S2,C;D)$ à l'instant T) pour un envoi de signal ($S2$) à une ou plusieurs autres tâches (C et D) qui en font le vœux de réception dans leurs instances respectives par appels interposés ($ReceiveSignal(S2)$ aux Tc et Td tels que $Tc, Td \leq Ta$ et/ou $Tc, Td \geq Ta$). L'intergiciel se charge alors de localiser les tâches consommatrices, d'empaqueter le signal dans une trame préalablement définie, de l'acheminer jusqu'aux ECU requis et de le délivrer (aux instants Tc', Td'). Pendant les exécutions des services de l'intergiciel les instances des différentes tâches ne sont pas bloquées mais peuvent être préemptées. Le temps d'exécution de ce service ($Max[Tc', Td'] - Ta$) doit cependant garantir le respect de la contrainte de fraîcheur imposée sur le signal ($S2.SignalFreshness$) (cf. III - 7 - 4).

- Besoins d'interface : l'API proposée par l'intergiciel doit être standardisée et compatible à la classe de conformité CCCB d'OSEK/VDX COM (cf. Annexe 1 - 3) pour une plus grande portabilité des applications d'une part, et d'autre part offrir la possibilité de réutiliser les développements antérieurs autour de cette spécification.

IV - 3 - 4 Modèle logique de conception technique

Un modèle de conception technique est généralement organisé en *packages* (regroupement logique et non fonctionnel) de classes qui représentent le *framework* développé pour résoudre les problèmes purement techniques. Rappelons que c'est pour cette raison que nous parlons de conception générique. Le modèle logique que nous présentons par un diagramme de packages UML est organisé suivant les dépendances qui s'établissent entre sous-canevas techniques.

La Figure 32 indique l'organisation retenue pour le projet *GenIETeR*. Notez la façon dont cette organisation du *framework* technique est influencée par les couches logicielles (cf. Figure 31) dans lesquelles nous avons exprimé les besoins aussi bien fonctionnels que non fonctionnels.

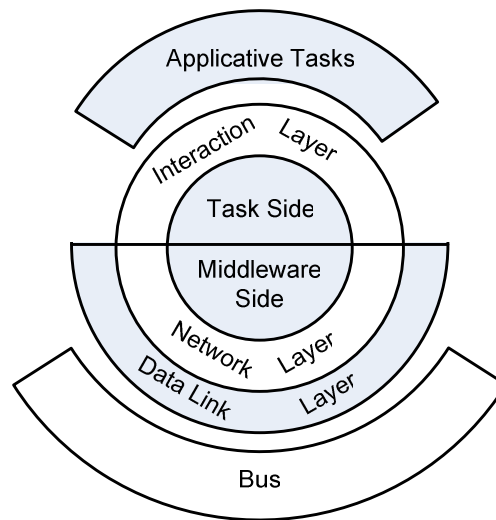


Figure 31 : Organisation en couches de l'intergiciel.

L'obstination du découpage en couches est une conséquence du périmètre des responsabilités techniques affectées à chaque *package*. En ce sens une responsabilité technique doit concerner une seule couche logicielle pour veiller à sa cohérence et son homogénéité (donc réduire le couplage et augmenter la cohésion). Le modèle de conception technique ajoute cependant des services transverses aux couches. Le sous-canevas *DataTypes* est typiquement l'un des services utilisables depuis toutes les couches logicielles.

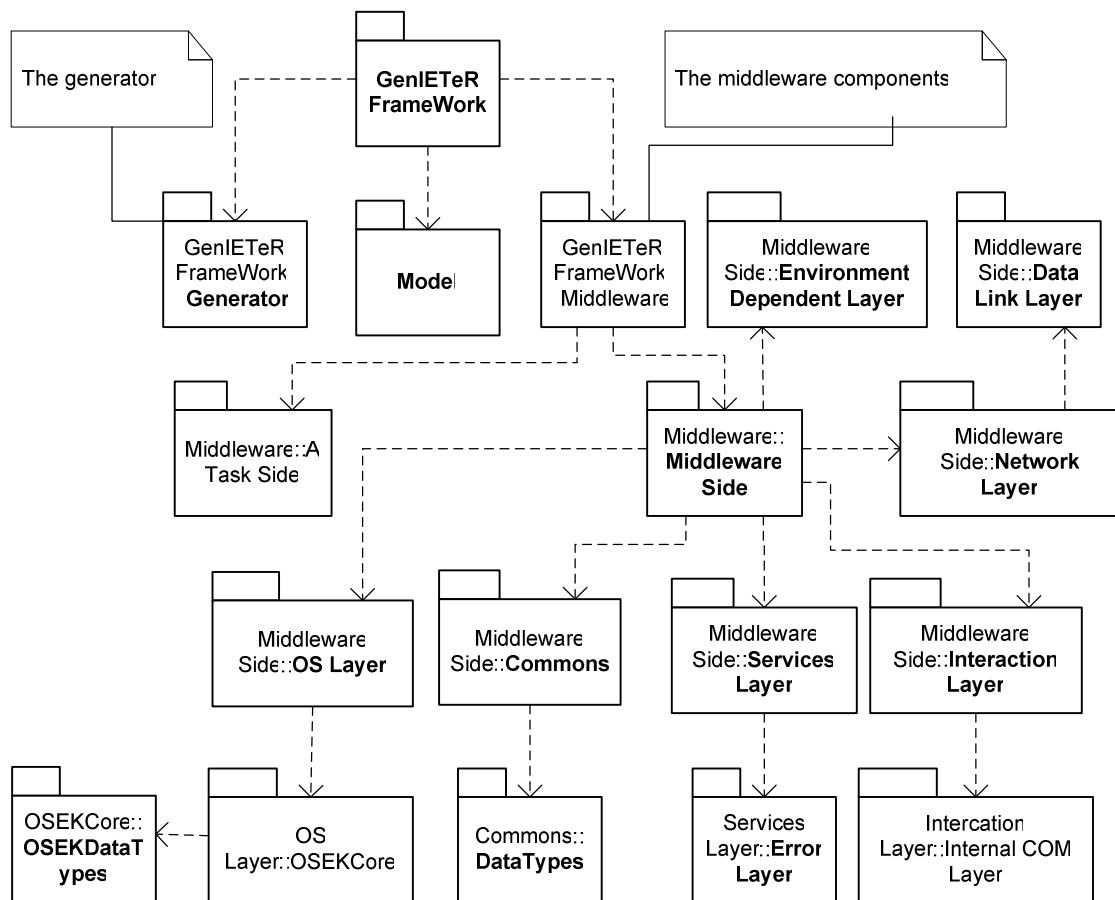


Figure 32 : Organisation du modèle logique du système.

Organisation du framework

L'organisation du modèle logique reprend les couches logicielles. À chaque couche, ou presque, correspond un sous-canevas technique, en partie abstrait, qui définit des interfaces génériques de réalisation des responsabilités logicielles :

- Le *Generator* définit les mécanismes de lecture des spécifications, de la configuration de l'intergiciel et de sa génération ;
- Le *Middleware* définit l'architecture adaptable de l'intergiciel et les patrons de classes imminents à son fonctionnement ;
- Le *Model* définit le noyau du système et de sa logique fonctionnelle ;
- Le *Task Side* regroupe les interfaces d'interaction entre les tâches concurrentes de la couche applicative et reste de notre système ;
- Le *Middleware Side* définit l'organisation et les services de communication offerts par l'intergiciel ;
- Le *EnvironmentDependent Layer* définit les classes d'implémentation de l'intergiciel qui sont générés au niveau du générateur ;
- L'*Interaction Layer* définit les mécanismes de communication asynchrones internes ;
- Le *Network Layer* établit les mécanismes de *marshalling* (action de mettre une structure de données sous une forme universelle indépendante des plateformes) de type et de données pour les échanges de trames et de signaux ;
- Le *Data Link Layer* définit les composants nécessaires à l'accès aux médias de communication ;
- L'*OS Layer* définit les dépendances et les interfaces avec l'exécutif temps réel ;
- L'*OSEKCore* définit l'API de la spécification OSEK/VDX (cf. Annexe 1).

Les autres sous-canevas répondent à des services transverses et complètent la conception des couches logicielles. Les services correspondent aux problématiques de portabilité et de la gestion d'erreurs.

Ainsi, même si la réutilisation des parties du système en cours de développement ne serait pas requise, l'envisager apporte des qualités de découplage, d'évolutivité, de compréhension et de robustesse du code. Autant de propriétés qui vont permettre la conception d'une solution générique. La solution est en effet indépendante de toute fonctionnalité du niveau applicatif. Il est donc bien dans notre intention de pouvoir réutiliser les mêmes concepts techniques pour toutes les applications ayant des besoins rejoignant la solution que nous proposons. Remarquez enfin comment la cartographie des dépendances entre les sous-canevas est importante pour la réutilisation. Celle-ci permet de maîtriser le couplage entre couches modulaires et d'en réaliser une maintenance ciblée et des tests unitaires.

IV - 3 - 5 Modèle structurel

En conception générique, la création de stéréotypes permet une personnalisation plus approfondie des moyens de représentation prévus par le langage UML. Dans notre cas, les stéréotypes étant utilisés pour définir la nature des éléments, l'utilisation de stéréotypes personnalisés permettra de différencier certains éléments de modélisation de ceux dédiés au traitement des données

Destiné à englober dans une même description structurelle et qualitative les constituants communs à toute architecture distribuée qui s'apparenterait à notre abstraction de l'architecture cible, le modèle structurel (cf. Figure 33) montre l'organisation des composants appartenant aux différents sous-canevas que l'on a défini. Pour établir ce modèle, nous avons défini les stéréotypes de classes suivants :

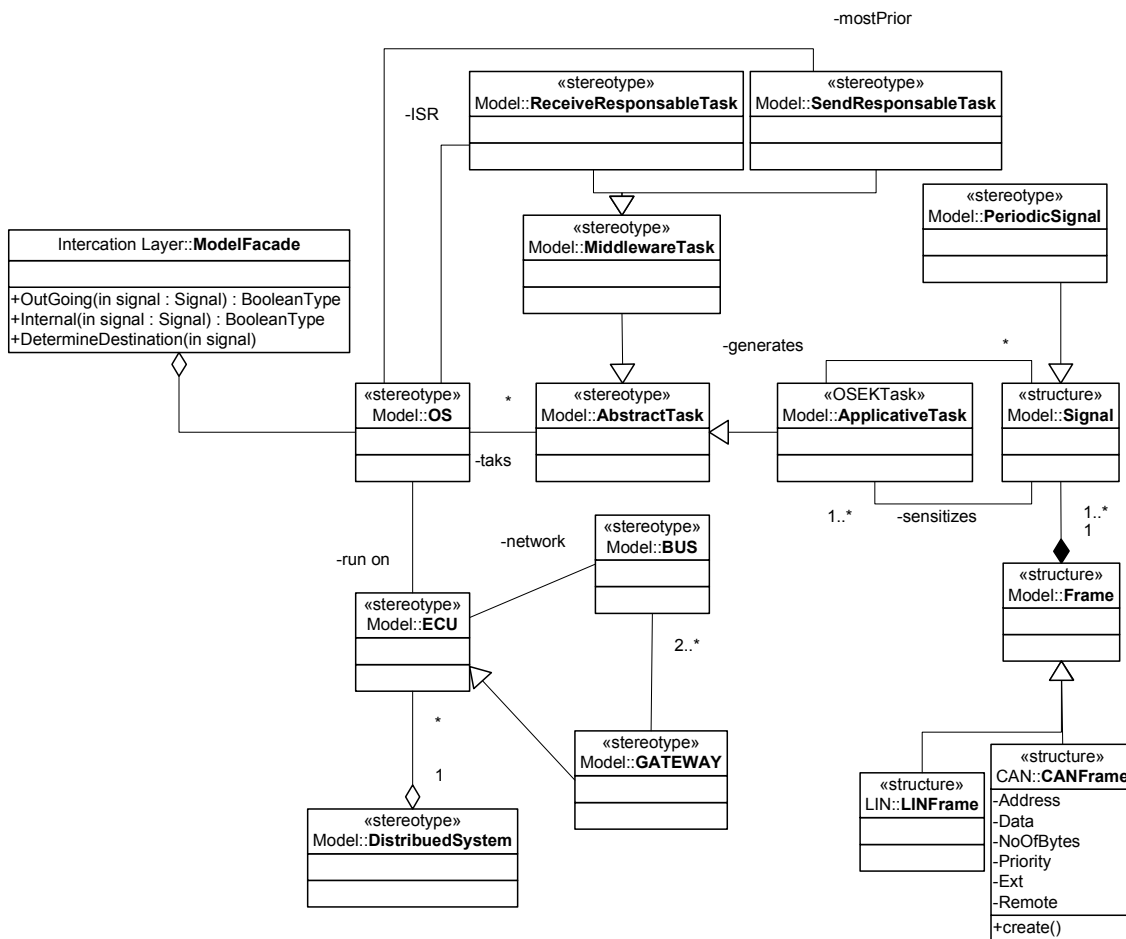


Figure 33 : Modèle structurel.

- « AbstractTask » représente une classe active décrivant une classe possédant son propre flot de contrôle. Nous l'utilisons ici pour illustrer notre modèle de tâches (cf. § III - 1 - 11) ;
- « MiddlewareTask » définit une tâche abstraite de l'intergiciel ;
- « ApplicativeTask » représente une tâche du niveau applicatif ;

- « *SendResponsibleTask* » représente la tâche de l'intergiciel exécutant la fonction de construction et de transmission de trames ;
- « *ReceiveResponsibleTask* » correspond à la tâche de l'intergiciel exécutant la fonction de réception et du traitement des trames ;
- « *COMDriver* » correspond à un *driver* de communication destiné à la prise en charge complète d'un périphérique d'accès à un bus.
- « *OSEKTask* », « *OSEKMessage* », « *OSEKAlarm* », « *OSEKEvent* », « *OSEKIsr* » sont des stéréotypes pour caractériser l'API d'OSEK/VDX [WOSEK] (cf. Annexe 1).

Les autres stéréotypes que nous avons retenus s'apparentent à la configuration logicielle [WUML] :

- « *structure* » représente une structure de données.
- « *file* » représente un fichier de code ;
- « *utility* » correspond à une classe utilitaire définie par une collection d'opérations.

Nous avons aussi opté pour des stéréotypes de méthodes de classes pour faire la distinction entre des méthodes prédéfinies et celles qui seront générés par le générateur de code :

- « *generated* » correspond à une méthode générée.

IV - 4 Le générateur de code

Pour l'architecture de notre générateur de code, nous avons retenu une génération de corps des méthodes fonctionnelles. Le générateur produit du code source en complétant des méthodes préalablement définies et qui sont appelées par le flot de contrôle des tâches de l'intergiciel générique. Mais avant pouvoir générer ce code, il doit procéder à une analyse de la spécification temporelle du trafic qui lui est soumise par l'utilisateur d'interface sous forme de fichier à plat (cf. § IV - 2). C'est ensuite qu'il pourra configurer le modèle temporel de la tâche responsable de l'intergiciel exécutant la fonction de construction et de transmission de trames en produisant un vecteur des instants d'exécution (*TimePattern*), des périodes d'activation et des échéances relatives.

IV - 4 - 1 Modèle de configuration de l'intergiciel et modèle fonctionnel du générateur

Nous présentons ici l'architecture de notre générateur (cf. Figure 34). Cette architecture est composée de trois modèles implicites chacun représentant une facette de ce générateur. La première facette, composée des classes *Genieter*, *SpecificationReader*, *SpecificationFile*, *CodeWriter*, *SourceCode*, représente les deux fonctionnalités principales du générateur, à

savoir : lire une spécification et en générer le code source de tâches de l'intergiciel. La seconde facette (représentée par la partie supérieure du diagramme, Figure 34) correspond au modèle de spécification de l'application distribuée pour laquelle un intergiciel va être généré.

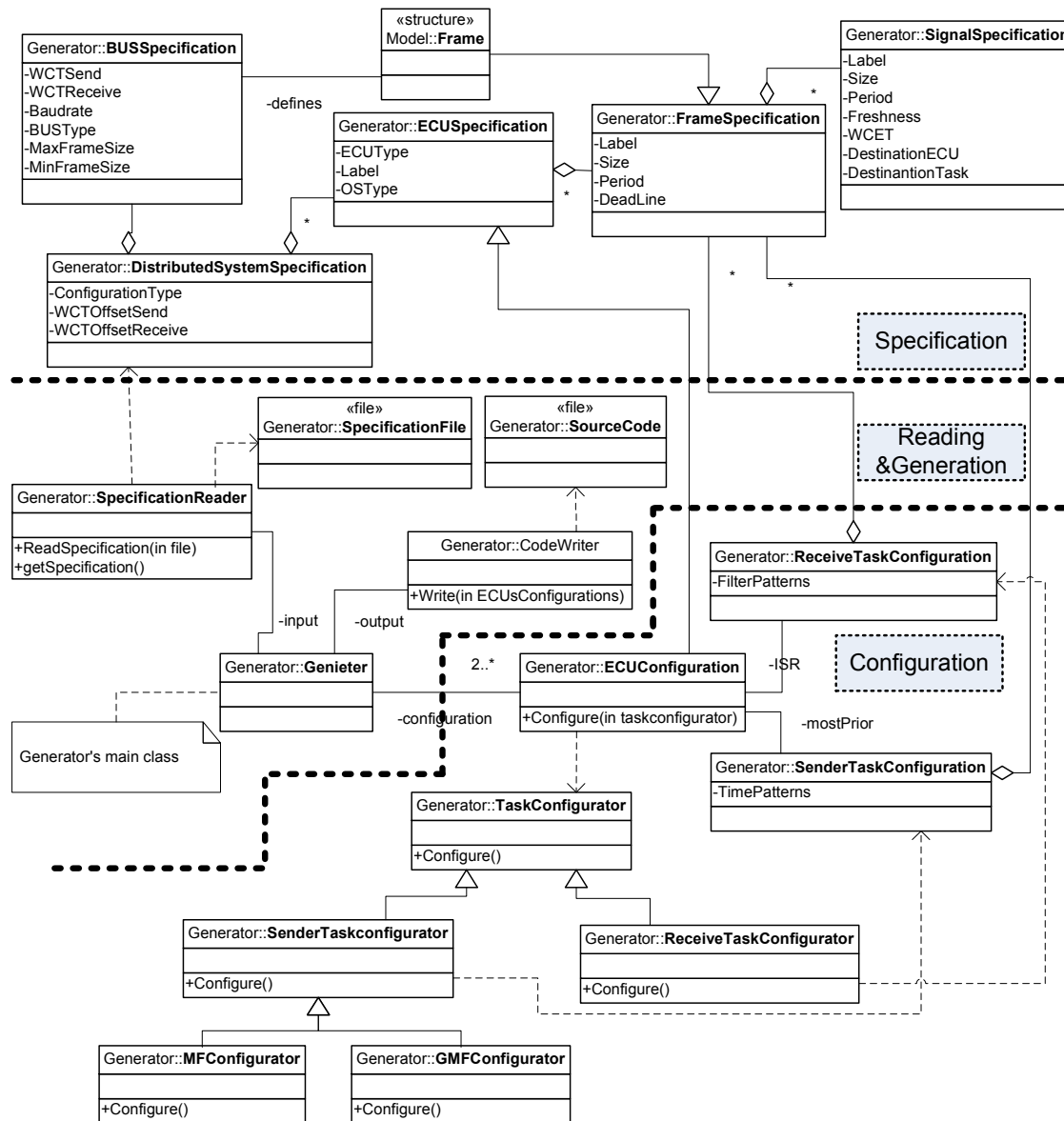


Figure 34 : Diagramme de classes du générateur.

La troisième facette du générateur correspond à la configuration du modèle temporel des tâches de l'intergiciel (*SenderTaskConfiguration* et *ReceiveTaskConfiguration*). La classe *TaskConfigurator* adapte la configuration de ces tâches en un modèle *multiframe* [MOK96] (*MFConfigurator*) ou *generalized multiframe* [BAR99] (*GMFConfigurator*) pour construire la configuration de l'intergiciel sur chaque ECU (*ECUConfiguration*).

Le modèle *multiframe* permet d'avoir des temps d'exécution qui suivent un motif donné. La supposition que nous connaissons à l'avance les durées d'exécutions des différentes instances d'une même tâche est bien fondée, puisque nous connaissons le flot de contrôle de toutes ces instances. Cela est dû au fait que la genèse de la messagerie et des interactions entre

les tâches applicatives et celle de l'intergiciel est donnée par la spécification préétablie. Mais ce premier modèle ne tient pas compte des échéances. Explicitement, il faut ainsi se référer à la généralisation des *multiframe* (*Generalized Multiframe Model*) pour prendre en compte les échéances relatives.

Pour notre prototype du générateur nous nous sommes limité à une implémentation de la configuration temporelle de la tâche responsable de l'envoi en *generalized multiframe* (GMF) qui est considérée comme une extension à l'approche *multiframe*. Nous présentons ici la formulation de ce problème [RSM06, chapitre 8, page 92].

Considérons une tâche périodique Φ_i destinée à un ECU i (cf. § III - 1 - 11) définie selon le modèle d'activation GMF. Celle-ci est caractérisée alors par le tuple $(C_{\Phi_i}, T_{\Phi_i}, D_{\Phi_i})$ tels que :

- $C_{\Phi_i} = \{c_{\Phi_i}^0, c_{\Phi_i}^1, \dots, c_{\Phi_i}^{N-1}\}$ soit un ensemble de N pires durées d'exécution de Φ_i ;
- $T_{\Phi_i} = \{t_{\Phi_i}^0, t_{\Phi_i}^1, \dots, t_{\Phi_i}^{N-1}\}$ soit un ensemble de N périodes d'activation de Φ_i ; et
- $D_{\Phi_i} = \{d_{\Phi_i}^0, d_{\Phi_i}^1, \dots, d_{\Phi_i}^{N-1}\}$ soit un ensemble de N délais critiques associés à Φ_i .

En concédant que les envois de cette tâche soient à départs simultanés, soit le vecteur $Q_i = \{(Q_{f_{i,1}}, T_{f_{i,1}}), \dots, (Q_{f_{i,k}}, T_{f_{i,k}})\}$ où les $Q_{f_{i,k}}$ représente les durées de construction et d'émission d'une trame $f_{i,k}$ par la tâche Φ_i , et où $T_{f_{i,k}}$ correspond à sa période. La construction temporelle du vecteur des périodes de cette tâche GMF Φ_i sera comme suit :

$$t_{\Phi_i}^0 = \min (T_{f_{i,1}}, \dots, T_{f_{i,k}}) ;$$

$$t_{\Phi_i}^{j+1} = \min \forall k ([a \cdot T_{f_{i,k}}] - \sum_{n=0}^j t_{\Phi_i}^n) / j > 0 \quad \text{et où} \quad a = \min \{i \in \mathbb{N}^+ / i \cdot T_{f_{i,k}} > \sum_{n=0}^j t_{\Phi_i}^n \}$$

Ce vecteur des périodes est construit sous la condition que :

$$\min \forall k (a \cdot T_{f_{i,k}}) \leq \text{lcm} (T_{f_{i,1}}, \dots, T_{f_{i,k}})$$

et où *lcm* représente la fonction du plus petit facteur commun (*Least Common Multiple*).

L'expression suivante doit être vérifiée à la fin de la construction :

$$\sum_{j=0} t_{\Phi_i}^j = \text{lcm} (T_{f_{i,1}}, \dots, T_{f_{i,k}}) .$$

La procédure utilisée pour construire le vecteur des durées d'exécution est :

$$c_{\Phi_i}^0 = \sum_k Q_{f_{i,k}} ;$$

$$c_{\Phi_i}^{j+1} = \sum_k Q_{f_{i,k}} \quad \text{pour} \quad \{k / \sum_{n=0}^j t_{\Phi_i}^n \bmod T_{f_{i,k}} = 0\}$$

Les délais critiques sont calculés par sommation des échéances relatives des trames envoyées dans chaque instance de la tâche.

Prenons un exemple d'illustration tel que $Q_i = \{(3, 15), (6, 60), (4, 20)\}$ qui représente une collection de trois trames $\{f_{i,1}, f_{i,2}, f_{i,3}\}$ chacune identifiée par son temps d'exécution (sa

construction et sa transmission) et sa période de transmission (ici on supposera que l'échéance relative d'une trame est égale à sa période de transmission). La construction du vecteur des périodes d'activation de la tâche Φ_i est effectuée, comme suit, itérativement jusqu'à ce que la condition $\sum_{j=0} t_{\phi_i}^j = \text{lcm}(15, 60, 20)$ soit satisfaite. Il en résulte cette configuration :

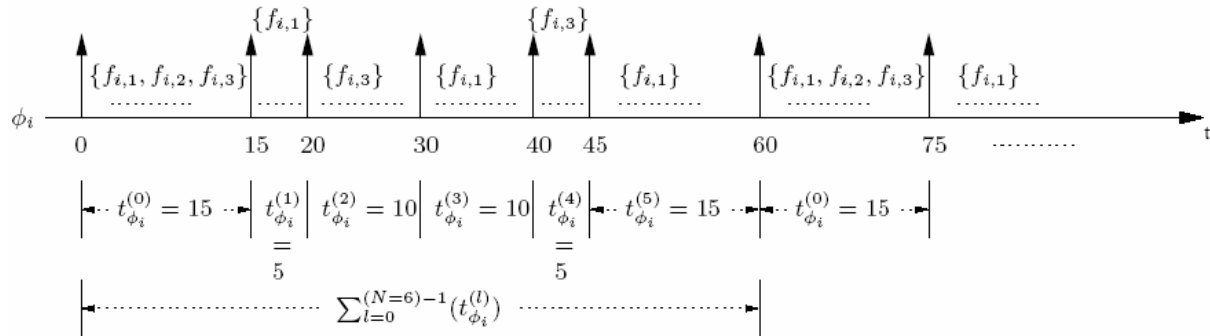


Figure 35 : Illustration d'une tâche configurée en GMF. [RSM06]

Nous introduisons le terme « fenêtre » pour caractériser l'exécution d'une instance de tâche entre deux périodes d'activation successives pendant laquelle cette tâche peut envoyer les trames nécessaires.

La configuration de l'ISR en charge de la réception des trames sur chaque ECU se caractérise par :

- un unique temps d'exécution correspondant au temps d'exécution le plus grand de toutes les trames définies dans la spécification de messagerie et produite par la *frame packing* ;
- une unique période d'activation définie par le plus petit temps d'exécution des trames destinées à l'ECU contenant la tâche ;
- le délai critique est égal à la période d'activation pour veiller à la réception en totalité des trames.

IV - 5 L'intergiciel

Comme nous l'avons énoncé plus haut, notre intergiciel offre au niveau applicatif une API conforme à la classe de conformité CCB d'OSEK/VDX COM (cf. § Annexe 1 - 3). Il nous est nécessaire d'introduire le concept de « message » défini dans ce standard. À la différence d'un message comme nous avons pu le voir dans un intergiciel orienté messages (MOM, cf. § III - 7 - 3), cette spécification des services de communication dans une architecture OSEK/VDX préconise que les tâches applicatives ou les ISRs et cette couche de communication interagissent par un échange de message. Ce message étend le concept de signal que nous avons utilisé jusqu'à présent pour caractériser l'échange entre les tâches. En

effet, mise à part qu'il contient les données spécifiques aux signaux échangés entre les tâches, cette idée de message possède des attributs propres à la configuration aussi bien fonctionnels (taille, valeur initiale, direction : en émission ou en réception, quelles actions entreprendre lorsque se message est reçu, etc.) que non fonctionnels (interne à un ECU ou externe par exemple, contraintes temporelles, etc.). Ceci nous vaut de rajouter un niveau d'abstraction « message » à notre architecture du système (cf. Figure 21). Ceci permet d'avoir la taxonomie des structures de données qui suit :

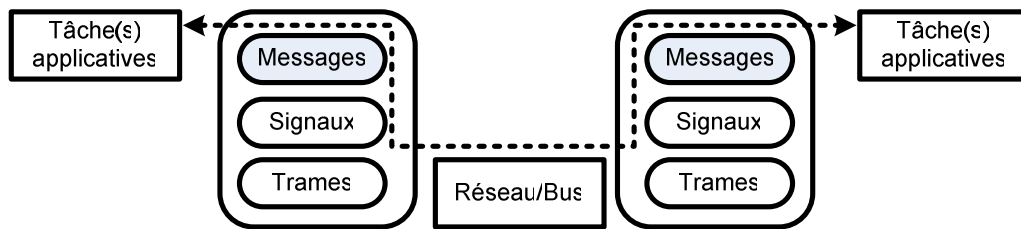


Figure 36 : Taxonomie des structures de données du système.

OSEK/VDX COM étant destiné aux architectures logicielles du domaine automobile qui, comme nous l'avons présenté au § III - 3 - 1, voient leurs processus de développement scindés entre constructeurs (OEM, *Original Equipment Manufacturers*) et équipementiers (*Suppliers*). Dans cette même logique et pour une plus grande souplesse d'intégration des applications constructeurs/équipementiers, OSEK/VDX OIL [WOIL] (spécification d'un langage d'exécution de définition statique de services OSEK/VDX) introduit une séparation entre types de messages. Cette distinction définit un objet *MESSAGE* qui présente des attributs spécifiques aux fournisseurs (*Supplier-specific*) et un objet *NETWORKMESSAGE* contenant les attributs appartenant aux constructeurs (*OEM-specific*).

Pour notre interface de l'intergiciel, il était donc nécessaire de prendre en compte cette API d'OSEK/VDX COM (cf. Annexe 1 - 6, Figure 56) sans pour autant en être dépendant. De plus, l'exécutif sur lequel nous sommes arrêté (PICos18, cf. Annexe 3) pour déployer notre solution n'implémentait pas ce standard, ce qui nous a motivé à prévoir dans notre conception les spécificités d'OSEK/VDX COM.

Cette interface est définie par la classe utilitaire *OSEKCOM* dans le modèle d'interaction entre tâches applicatives et l'intergiciel suivant :

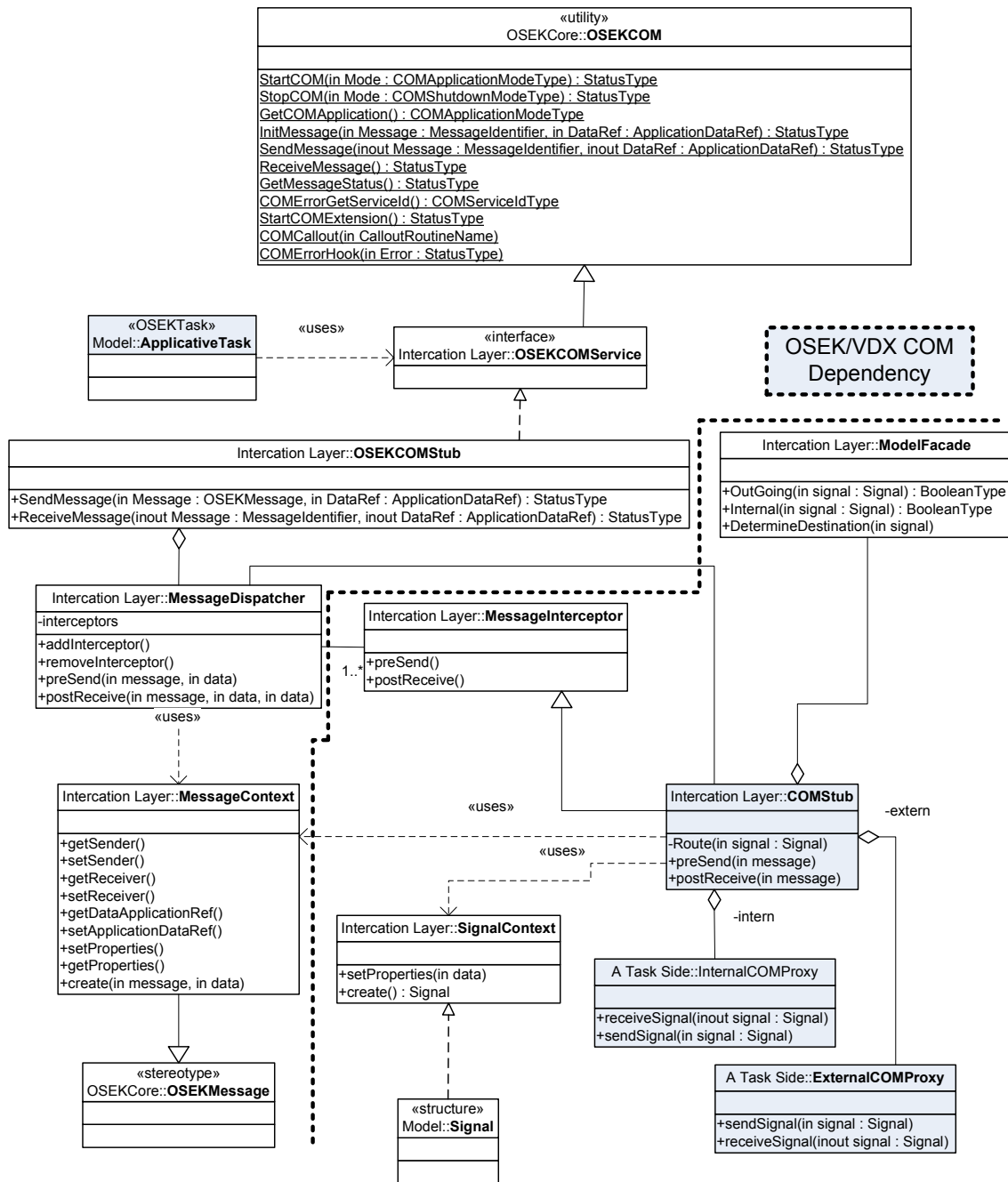


Figure 37 : Modèle d'interaction tâches applicatives/intergiciel.

Ce modèle d'interaction met en évidence le service en charge de la communication inter-tâches (locales et distantes). Le point d'entrée du service de communication de l'intergiciel accessible aux tâches applicatives (*ApplicativeTask*) est l'interface *OSEKCOMService* offrant les mêmes méthodes conformément à OSEK/VDX COM. Le talon de communication *OSEKCOMStub* alloué à chaque tâche applicative, par un mécanisme d'interception et d'interposition (les composant transformateurs : *MessageDispatcher*, *MessageInterceptor*), traduit un message en un format de signal dans le cas du prélude (c'est-à-dire de l'envoi) ou inversement dans le cas du postlude. Ce signal est transmis à la classe *COMStub* qui le route

selon sa destination (locale ou distante) à l'une des passerelles de communication (*InternalCOMProxy* ou *ExternalCOMProxy*) qui se chargera de la communication effective.

IV - 5 - 1 Modèles fonctionnels de l'intergiciel

L'architecture que nous avons retenue pour notre intergiciel se subdivise en deux parties. L'une destinée au service de communication asynchrone distante entre les ECUs et l'autre correspond à la communication interne (intra-ECUs). Commençons donc par le premier service :

Communication distante

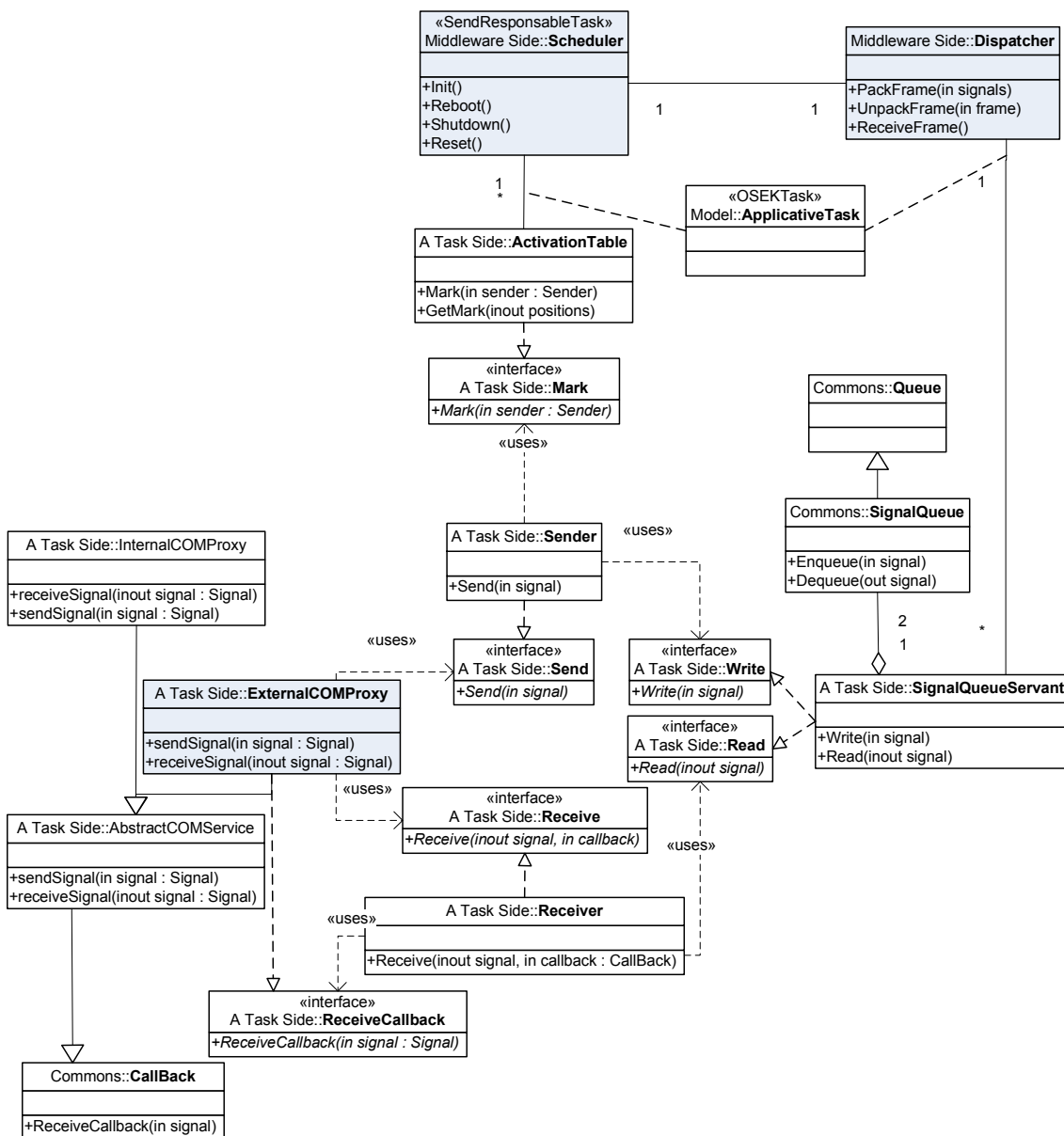


Figure 38 : Modèle générique de l'intergiciel (1/2).

Le service de communication *ExternalCOMProxy*, selon les besoins des tâches applicatives, peut-être amené à réaliser soit des mécanismes en charge de l'envoi de signaux (*Sender*) que nous appellerons prélude, soit des mécanismes de réception que nous désignerons par postlude (*Receiver*).

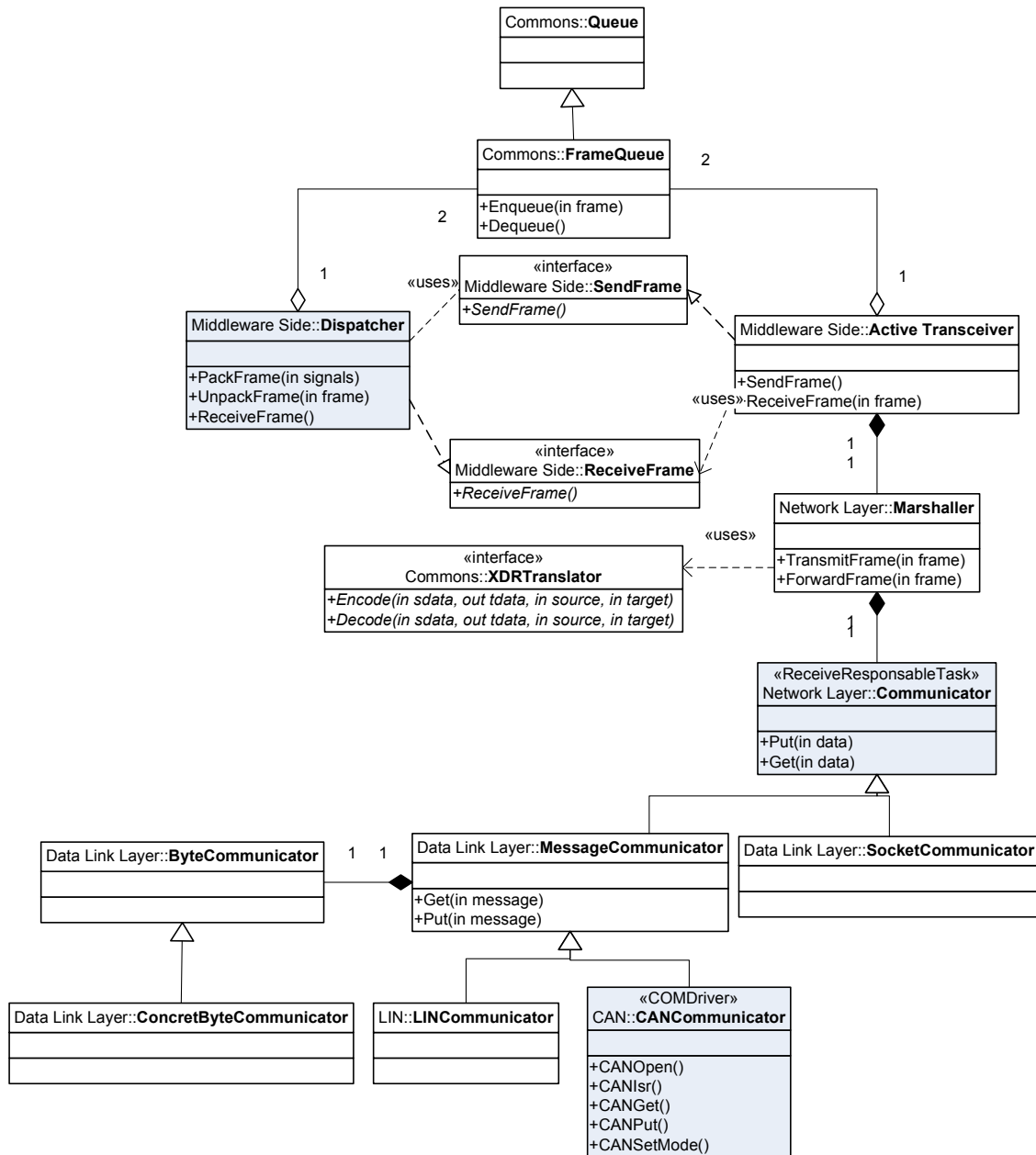


Figure 39 : Modèle générique de l'intergiciel (2/2).

Dans le modèle générique ci-dessus (cf. Figure 38 et Figure 39), nous présentons les composants de base de la bibliothèque de l'intergiciel. Ces composants encapsulent les fonctions que l'on trouve de façon commune dans les différents intergiciels générés et adaptés aux spécifications en entrées du générateur. La librairie possède également des composants développés pour des paramétrisations spécifiques.

Les files de signaux et de trames (*SignalQueue*, *FrameQueue*) servent à stocker aussi bien les signaux et trames en attente d'envoi, que ceux reçus par l'intergiciel et en attente d'être

délivrés. Elles ont une entrée qui est utilisée par les autres composants pour stocker ces objets, et une sortie que la file utilise pour délivrer les objets stockés. La capacité d'une file est représentée par le nombre maximum de messages qu'elle est capable de stocker. Même si [RSM06] préconise que le mode de stockage des signaux soit par écrasement (car fonctionnellement, pour une tâche la valeur la plus récente est la plus intéressante) pour éviter de surcharger les temps d'exécution des trames avec les calculs liés aux temps de latence et de séjours dans ces files. Nous nous sommes pourtant permis de prévoir ces files pour un but d'évolutivité de notre intergiciel adaptable.

Les transformateurs (*XDRTranslator*, *Marshaller*, *Communicator*) sont des composants avec une entrée et une sortie. Chaque trame reçue sur l'entrée est transformée, puis délivrée sur la sortie. Les transformateurs peuvent être en push/push ou en pull/pull. Un exemple typique de transformation, pour l'interface *XDRTranslator*, consiste à traduire la représentation d'une trame entre un format dépendant de l'architecture matérielle de l'ECU et un format XDR (*External Data Representation Standard*) [WXDR]. Ceci permettra la portabilité de notre intergiciel sur différentes architectures matérielles d'ECUs tout en garantissant leur interopérabilité.

L'agrégateur/désagrégateur (*Dispatcher*) est un composant à plusieurs entrées et une sortie. Son rôle est de délivrer sur la sortie un agrégat des signaux (sous forme de trames) reçus en entrée et inversement.

Les composants de protocole (*MessageCommunicator*, *ByteCommunicator*, *SocketCommunicator*) sont en charge de faire respecter un protocole. Ils ont une entrée/sortie pour les trames entrantes, et une entrée/sortie pour les trames sortantes. Le composant *ActivationTable*, par le biais de *Sender*, peut être considéré comme un composant protocole car il assure l'ordonnancement causal des signaux en provenance de toutes les tâches applicatives. Ce composant possède une structure pour enregistrer (ou marquer l'envoi) tous les signaux devant être envoyés. Cette structure pourra renseigner le *Scheduler*, quand il sera activé par la tâche en charge de l'envoi, sur les différentes propriétés temporelles des signaux prêts (ou marqués à l'envoi) à être envoyés.

Les canaux (*CANCommunicator*, *LINCommunicator*, *ConcretByteCommunicator*) permettent l'échange des trames entre différents ECU par un accès au périphérique du bus de communication. Nous retiendrons pour la suite de la conception et pour notre implémentation du prototype le composant *CANCommunicator* qui fournira l'accès au bus de terrain et permettra l'envoi et la réception des trames à l'aide du protocole CAN [5].

Pour illustrer la dynamique du modèle de l'intergiciel, rien n'est plus explicite que les diagrammes de séquence.

Scénario du prélude

Commençons par un scénario de prélude impliquant une tâche applicative OSEK/VDX OS (*AnApplicative Task*).

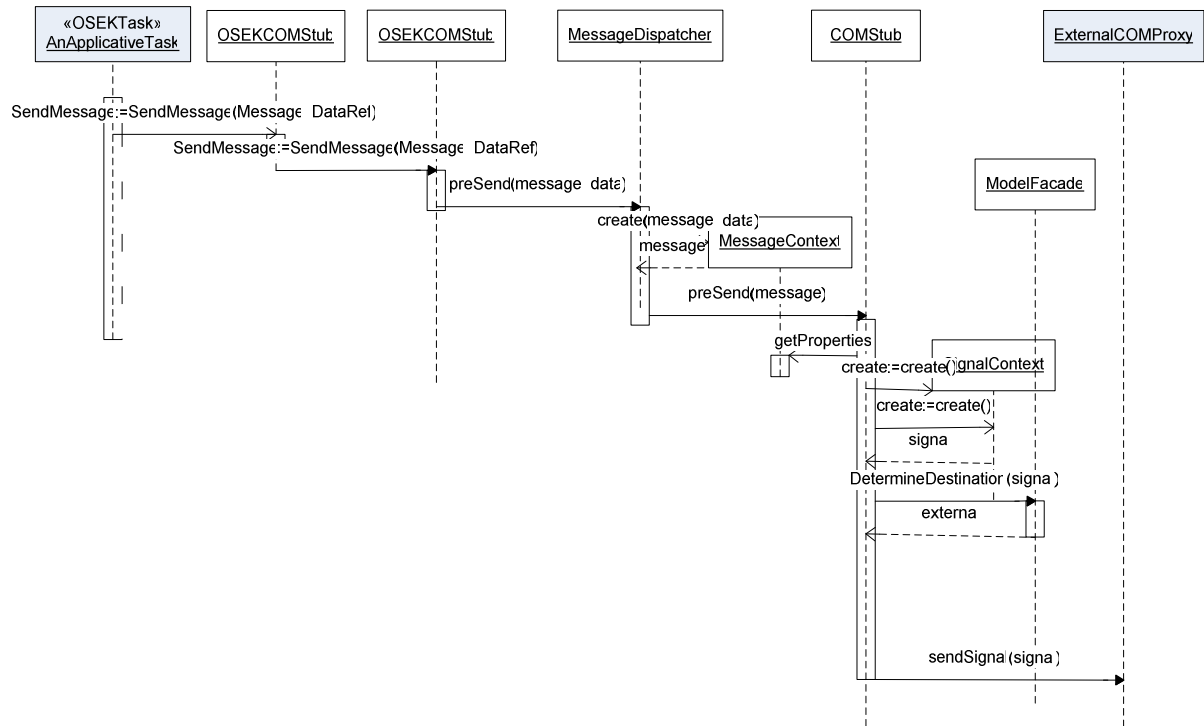


Figure 40 : Diagramme de séquence du mécanisme de prélude (1/3).

Cette tâche productrice, ayant construit un message à l'envoi, accède à l'API de l'intergiciel par appel de la primitive d'envoi asynchrone *SendMessage*, pour cela elle fournit le message et une référence sur la variable contenant la valeur de la donnée à envoyer. Ce message est alors transformé en signal compréhensible par le noyau de l'intergiciel (cf. Figure 32). Selon la nature du signal, qu'il soit à destination d'une tâche locale ou destiné à une tâche distante (*external*) se trouvant sur un autre ECU (cf. § III - 5 - 2), le *COMStub* le transmet alors à la passerelle correspondante. Dans notre cas de communication distante ce sera *ExternalComProxy*. Celui-ci aura le rôle de désynchroniser l'appel de la tâche applicative et de poster le signal en l'enregistrant comme étant prêt à l'envoi.

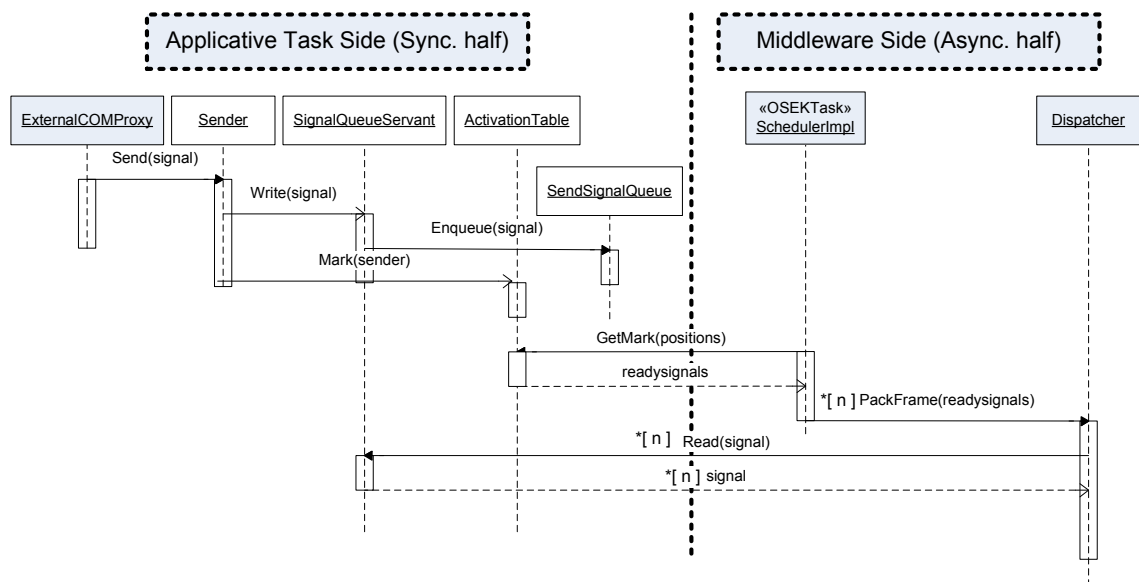


Figure 41 : Diagramme de séquence du mécanisme de prélude (2/3).

Cette désynchronisation est rendue possible par l'utilisation du *design pattern Half-Sync/Half-Async* [SCHH96] en séparant la couche d'interaction avec les tâches applicatives (cf. Figure 31) et le noyau de l'intergiciel. De ce fait, il utilise l'interface *Send* du *Sender* qui se charge de marquer la table des activations *ActivationTable* et d'insérer le signal dans la file d'envoi.

À l'activation d'une instance de la tâche en charge de l'envoi des signaux (représenté ici par la classe de son implémentation *SchedulerImpl*), celle-ci envoie un ordre de construction de trame avec les signaux (correspondants à cette fenêtre d'envoi) au *Dispatcher*. Le *Dispatcher* retrouve alors les signaux valués et en construit les trames sous une forme indépendante de tout détail inhérent au protocole réseau qu'il empile dans la *SendQueueFrame*.

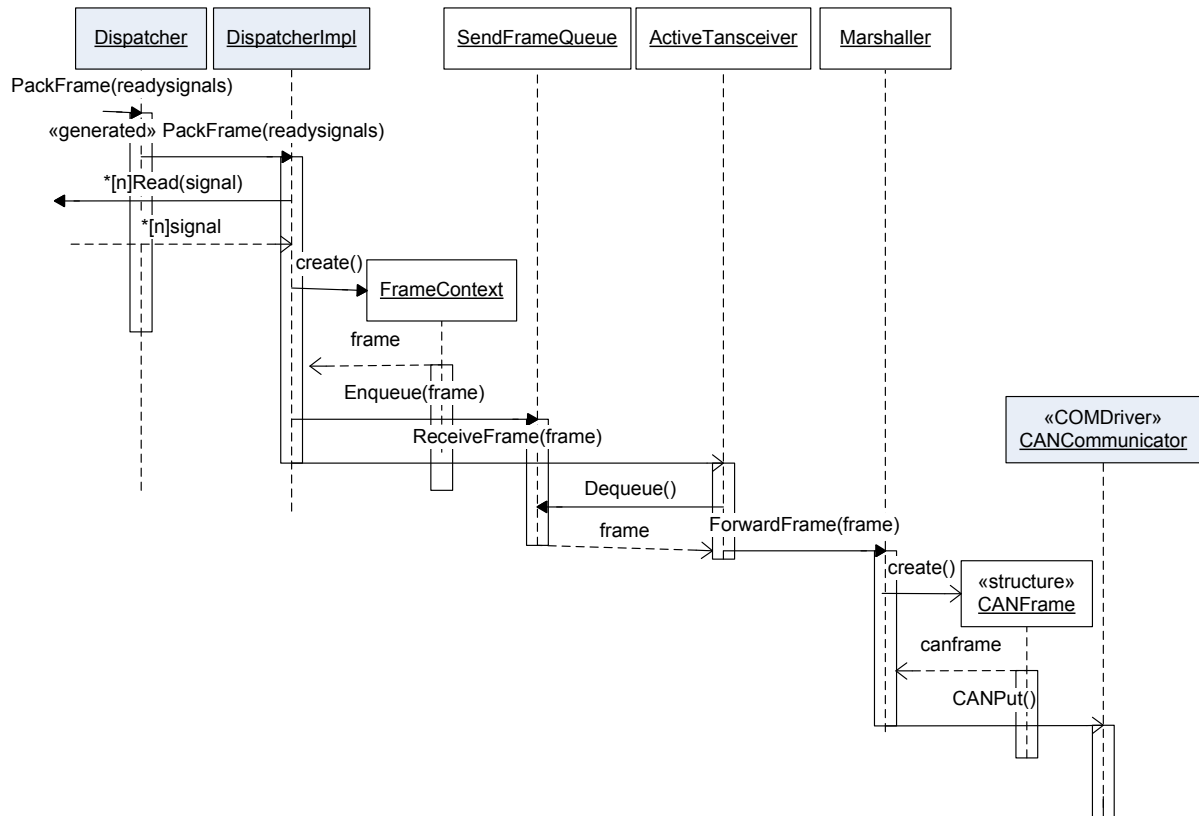


Figure 42 : Diagramme de séquence du mécanisme de prélude (3/3).

Conformément au design pattern *Active transceiver* [BASH], le *Dispatcher* place les trames ainsi construites dans une file et en notifie l'*ActiveTransceiver*, par son interface *SendFrame*, qui lui se chargera de récupérer la trame empilée et de la faire suivre au *Marshaller*. Ce transformateur conçoit une trame spécifique au protocole réseau qu'il fait suivre à un canal, et plus précisément à un gestionnaire de périphérique de bus pour un envoi concret de la trame sur le réseau. Dans notre cas c'est un bus CAN (*CANCommunicator*).

Scénario du postlude

Pour ce scénario, l'asynchronisme entre les tâches applicatives et celle de l'intergiciel veut que le mécanisme de délivrance d'un signal (implicitement d'un message) à une tâche applicative soit désynchronisée par rapport à la réception d'une trame en provenance du bus et contenant ce signal. Ce qui justifie le fait de retrouver, comme pour le prélude, la même hiérarchie des couches (cf. Figure 31). Cependant à l'inverse du prélude, ce service est considéré comme « bi-transactionnel », dans le sens où la réception d'une trame réseau constitue une première transaction totalement indépendante de celle du traitement de l'appel d'une tâche consommatrice d'un signal. Cela nous a amené à utiliser le mécanisme de

callback (l'interface *ReceiveCallback*, cf. Figure 38) pour assurer la délivrance d'un signal non encore reçu lors d'un appel de réception par une tâche consommatrice.

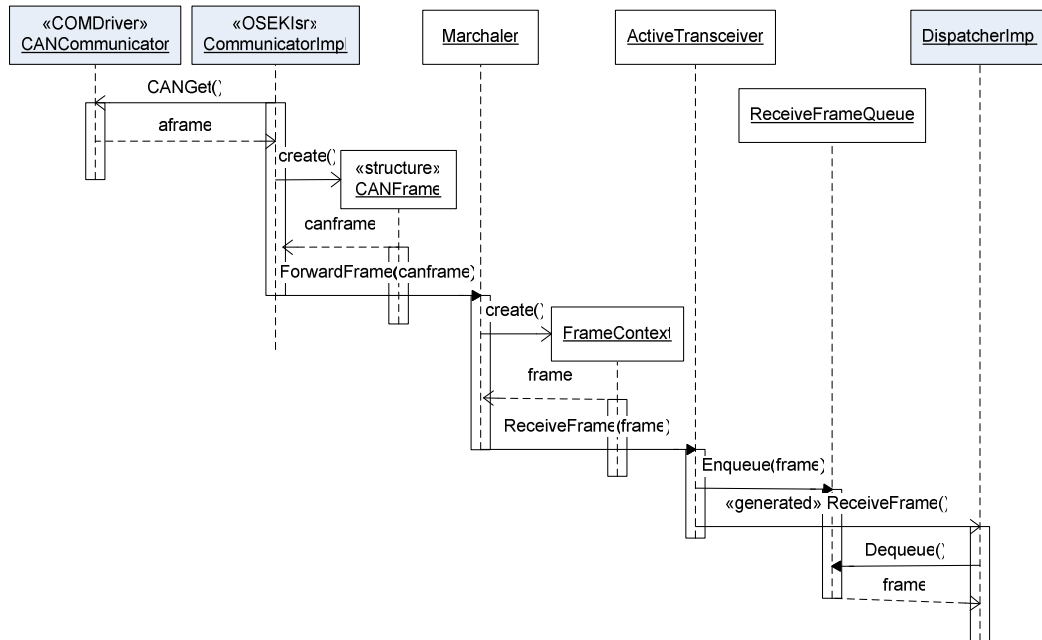


Figure 43 : Diagramme de séquence du mécanisme de postlude (1/3).

Ainsi la transaction, initialisée par une interruption du gestionnaire du bus de communication (*CANCommunicator*) au niveau de la couche *Data Link Layer*, est interceptée par le *Communicator*, dont l'implémentation est une routine d'interruption de service (voir ci-après). Cette ISR, inversement au mécanisme de prélude, remonte la trame par transformation jusqu'au *Dispatcher* qui en extrait les signaux et les enregistre auprès du *SignalQueueServant* (dans la file *ReceiveQueueSignal*).

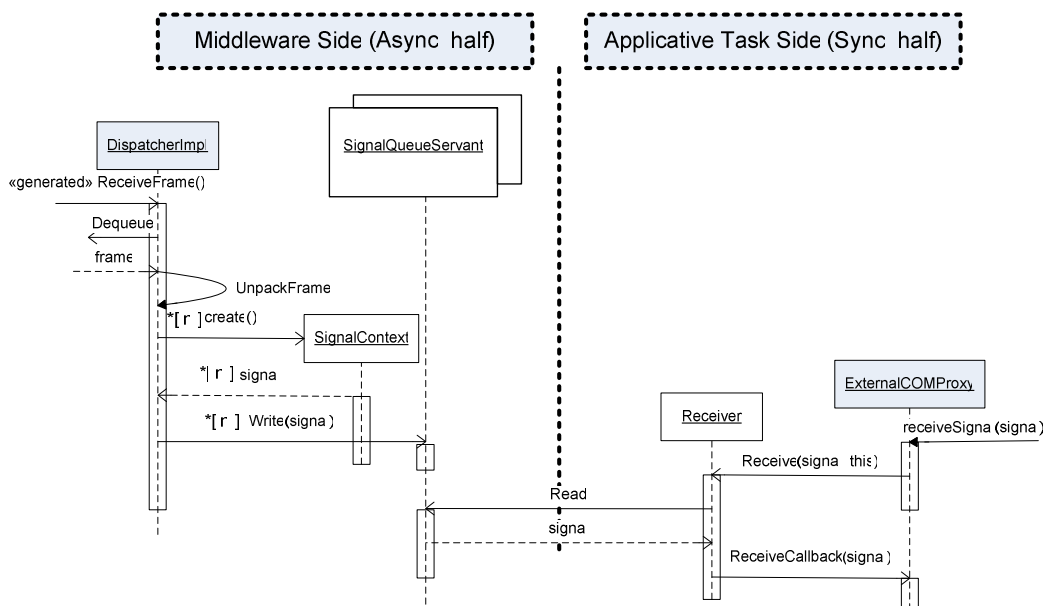


Figure 44 : Diagramme de séquence du mécanisme de postlude (2/3).

D'autre part, la transaction permettant de délivrer un signal à sa tâche consommatrice est décrit par la Figure 45 :

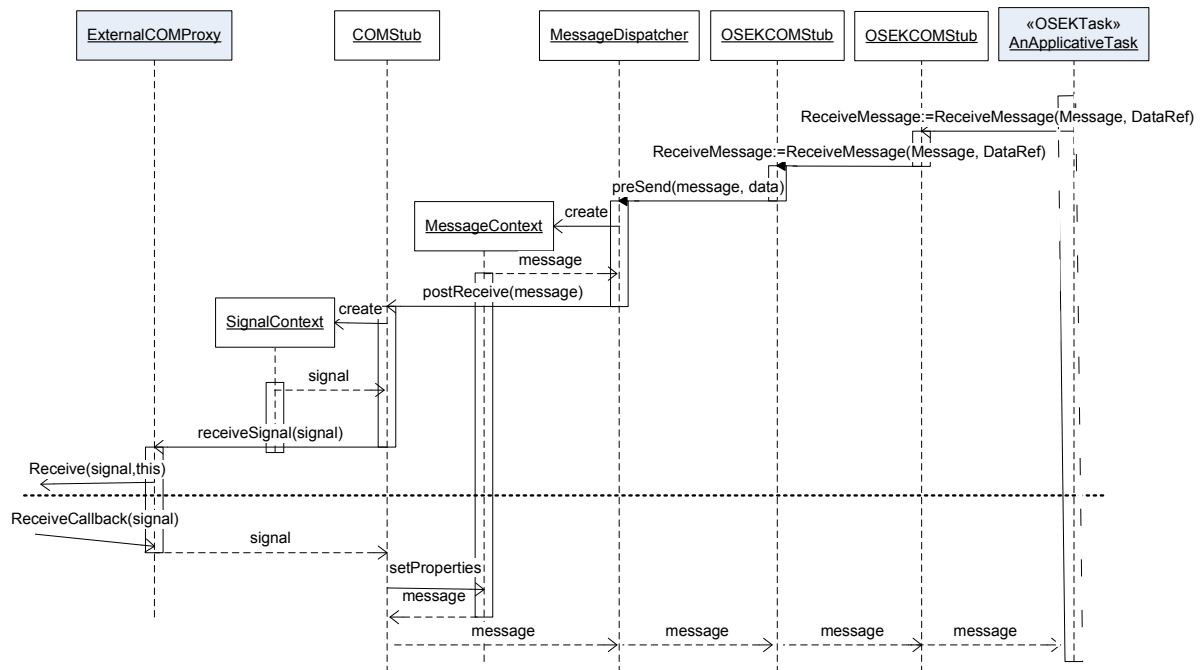


Figure 45 : Diagramme de séquence du mécanisme de postlude (3/3).

Inversement, la valeur reçue du signal est enregistrée dans le message qui a été fourni par la tâche consommatrice. Il est du ressort de la tâche applicative de veiller à l'isolation [WISOL] de cette valeur.

Communication locale

Pour le cas de la communication locale asynchrone et dans le cas optimiste où un envoi de signal précède la demande de consommation, le *SignalRequestListServant* (présenté par la Figure 46) désynchronise le service d'envoi en transformant les appels de fonction en requêtes qui sauvegarde le signal. Ce signal est alors délivré à la passerelle de communication locale *InternalCOMProxy* lorsqu'elle en fait la demande par la méthode *Receive* de l'interface *InternalCOM*.

Nous nous sommes servi aussi d'un mécanisme de *callback* pour enregistrer les appels de réception de message (implicitement de signal) qui ne peuvent pas être satisfaite (car le signal n'a pas encore été envoyé par la tâche productrice) au moment où la tâche consommatrice en fait la demande. Ainsi, dans le cas pessimiste (qu'il faudrait pouvoir valider le modèle temporel pour respecter les contraintes imposées sur les signaux), la requête est enregistrée comme un objet *SignalRequest* (ayant comme attribut un objet *InternalCOMProxy* qui a initié l'appel) dans la liste *SignalRequestList*. Permettant au *SignalListRequestServant*, lors d'une

invocation d'une méthode d'envoi de signal, de retrouver la requête en attente de ce signal et d'en pourvoir le demandeur par l'interface *InternalReceiveCallback*.

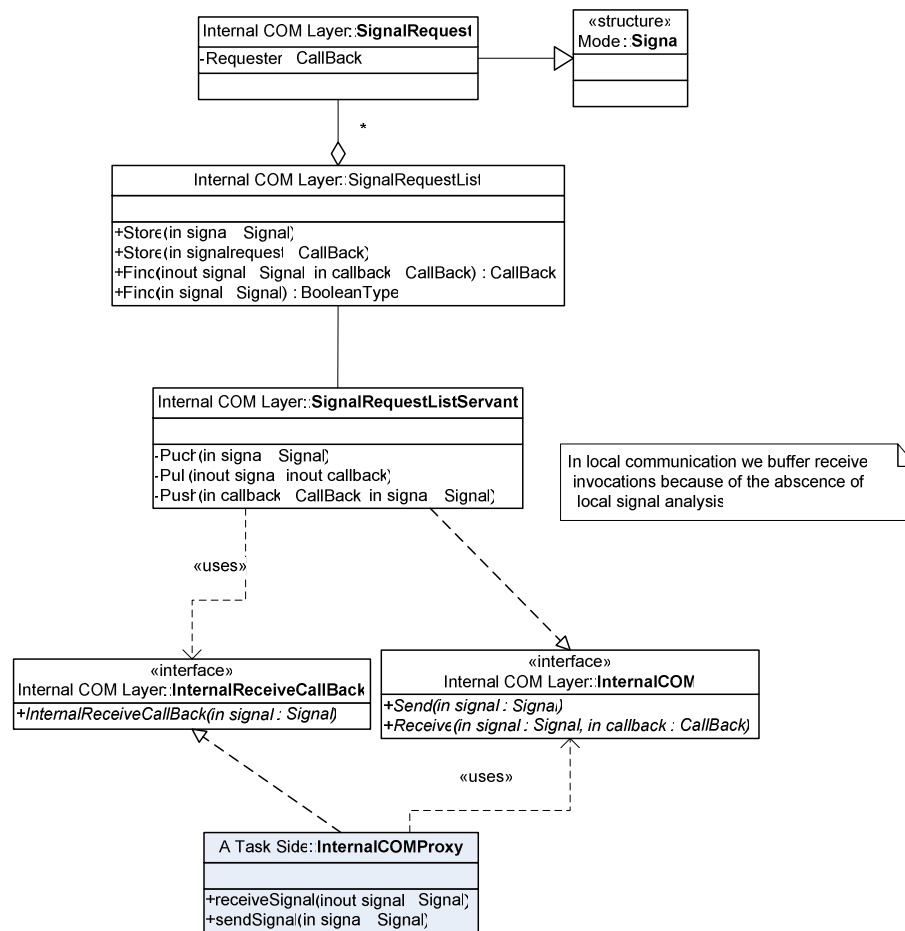


Figure 46 : Ebauche du modèle fonctionnel pour la communication locale.

IV - 5 - 2 Modèles de configurations pour la génération de l'intergiciel

Comme nous l'avons vu au § IV - 2 - 1, le générateur produit le corps de certaines méthodes qui permettent de concrétiser le *framework* et par conséquent d'adapter l'intergiciel aux spécifications temporelles de la messagerie.

Le diagramme de classe présenté dans la Figure 47 correspond au modèle de configuration des deux tâches caractérisant notre intergiciel. La conception générique d'un intergiciel a nécessité l'utilisation du *design pattern Bridge* [POSA1] pour découpler certains composants, dont le flot de contrôle doit être généré, de leurs implémentations. Ainsi les méthodes avec la

IV - 5 - 3 Les *design patterns* dans l'intergiciel

Les classes définies dans une conception ne sont pas toujours conformes aux possibilités du langage d'implémentation. Dans certains cas, comme le notre d'ailleurs, une conception orientée objet est implémentée dans un langage non objet. La transformation des classes en codage est alors particulièrement importante pour conserver la trace du passage de l'analyse au code. Concevoir des classes consiste tout d'abord à expliciter comment les concepts qu'elles introduisent devront être traduits dans le code. Mais aussi, introduire des responsabilités purement techniques. Les liens qui rattachent ces classes entre elles doivent le plus souvent correspondre à des *design patterns* (DP), c'est du moins ce qui est préconisé dans tout processus de développement.

Après avoir étudié les DPs qui pourraient convenir aux besoins de notre intergiciel, nous nous sommes tenus à ceux qui sont les plus pertinents. Voici une hiérarchisation de ces patrons d'architecture (cf. Figure 49) que nous avons pu utiliser, aux meilleurs des cas, pour notre intergiciel : *Gateway* [POSA1], *Interceptor* [POSA1], *Bridge* [POSA1], *Proxy* [GOF], *Adapter* [GOF], *Subscriber* [POSA1], *Wrapper Façade* [GOF], *Half-Sync/Half-Async*. [SCHH96], *Active transceiver* [BASH], *Strategy* [GOF], *Singleton* [GOF]. Cependant, nous avons renoncé à reprendre les DPs tels que les ORBs [SCHM00] régissant les intergiciels réflexifs (car notre middleware n'a pas cette vocation) et *Active object* [SCHH96], *Observer* [GOF], et *Asynchronous completion token* [WSCM98], car non adaptés à notre vision du système.

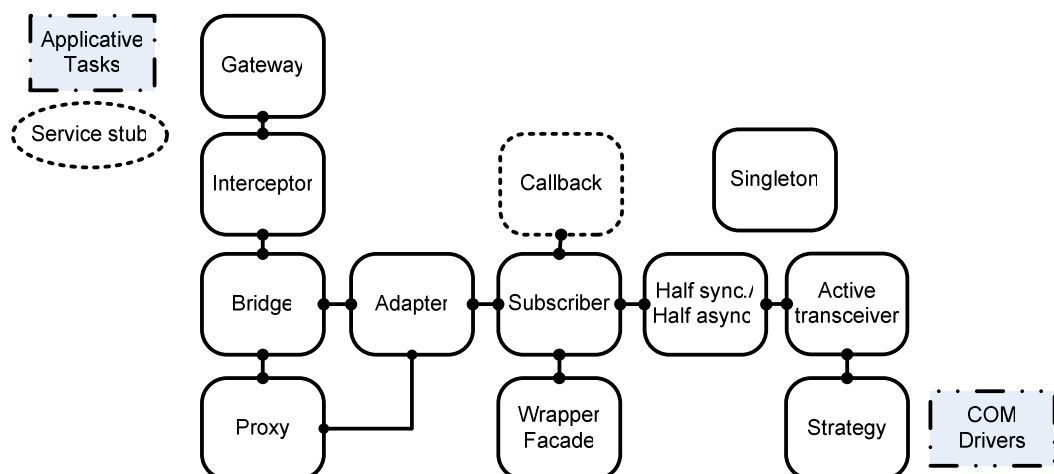


Figure 49 : Organisation des *design patterns* dans l'intergiciel.

IV - 6 Développement et déploiement

Ayant conçu ce *framework* technique qui nous permet de répondre aux spécificités techniques de notre système. Nous espérons que cette architecture technique ainsi obtenue garantit l'intégrité conceptuelle de tous développements présents et futurs. Conscient du fait que la conception évolue encore lors de son implémentation et que tout changement concernant la conception générique devient coûteux en phase d'implémentation, nous avons eu donc intérêt à développer le plus tôt possible un prototype d'architecture, afin de procéder le plus rapidement possible aux mises aux points qui s'imposent à notre conception générique.

La question que nous nous sommes posé est la suivante : « Que mettre dans un prototype ? » en sachant que le prototype doit répondre aux fonctionnalités techniques demandées par le système. Donc nous nous sommes afféré à ce que nos prototypes puissent au moins valider les fonctionnalités les plus importantes aussi bien du générateur que de l'intergiciel.

Pour le générateur, le prototype doit implémenter la construction de la tâche en charge de l'envoi des trames selon le modèle *generalized multiframe*. Et ce, en prenant en entrée un ensemble de trames comme définit au § IV - 4.

Par contre pour l'intergiciel, sur lequel nous avons concentré notre développement, le prototype doit :

- Offrir pour les tâches applicatives OSEK/VDX OS une interface conforme au standard OSEK/VDX COM pour leurs accès aux services de communication asynchrone distante ;
- Implémenter les principaux composants génériques définis dans la conception ;
- S'interfacer avec un gestionnaire de périphérique d'un bus CAN;
- Etre déployable sur un microcôntroleur ;

IV - 6 - 1 Un générateur en Java

Dans un premier temps, nous sommes servi du langage Java pour développer le prototype du générateur au sein de l'IDE *Oracle JDeveloper 10g*. Ceci est dû au fait qu'il fallait rapidement valider l'algorithme de construction du modèle temporelle de tâche en charge de l'émission. Les raisons de notre choix pour ce langage sont :

- Le générateur en lui-même n'est pas soumis à des contraintes temps réel donc il peut être développé en un langage de haut niveau ;
- Sa portabilité sur différents environnements (type Linux ou Windows) est un critère non fonctionnel à prendre en considération pour qu'il ne soit pas dépendant d'une

architecture particulière. Pour cela Java se révèle un bon compromis entre portabilité et performance. De plus, le langage d'implémentation du générateur n'influe en rien sur le langage utilisé pour l'intergiciel car le code injecté dans les méthodes des composants est totalement indépendant de l'exécution du générateur ;

- Notre connaissance de Java (rapidité de codage).

IV - 6 - 2 Langages et exécutifs temps réel pour l'intergiciel

Le domaine des systèmes temps réel embarqués dans l'automobile est très large, et beaucoup de langages de programmation peuvent être plus ou moins utilisés selon les besoins. Dans des systèmes temps réels souples tels que le multimédia, *Real-Time Java* [LEA00, WRTJAVA] peut jouer un rôle important. Cela est justifié par le fait que le langage Java est déjà bien installé dans la programmation d'applications liées à Internet. Plus récemment, des propositions ont été faites pour satisfaire des contraintes liées à la mise en œuvre d'applications mobiles, dont les PDA (assistants personnels portables), et les cartes à puce. Parallèlement, des requêtes émanant des milieux traditionnels développant des systèmes temps réel ont également émergé [BASI05]. Les propriétés principales exhibées par de tels systèmes sont, notamment, le respect des échéances temporelles et la sécurité. Dans ce but, des groupes d'experts ont été mandatés pour élaborer des spécifications temps réel pour Java [NIST99]. Des travaux sont également en cours pour l'utilisation de Java dans le développement de systèmes critiques. Il s'agit d'une proposition de profils de type Ravenscar [BUR04], à l'instar de celui proposé pour le langage Ada 95. Mais à ce jour, les machines virtuelles comme JamaicaVM, Espresso et PERC bourgeonnent encore.

Cependant, dans les systèmes temps réel stricts, où la sûreté et la stabilité sont des critères indispensables, le langage Ada reste un bon candidat du fait qu'il soit très ancré dans l'industrie. Nous pouvons mentionner d'autres langages de programmation d'applications temps réel tels que Pearl, Ltr (Langage Temps Réel) ou Chill (*Ccitt High Level Language*). Pour une étude comparative exhaustive de Ada et Real-Time Java, le lecteur pourra se référer à [BRO00].

Mais parmi les spécifications, ceux qui sont le plus utilisés dans la communauté des chercheurs ou dans le domaine automobile (et plus particulièrement l'équipe dans laquelle ce stage s'est déroulé) nous pouvant évoquer :

POSIX 1003.1 [POSIX]

Il s'agit surtout de l'extension temps réel du standard, appelé profile RT-POSIX. Elle propose un ensemble de services pour le développement de systèmes temps réel au sein

desquels l'ordonnancement des tâches repose sur des priorités fixes. Ces tâches peuvent être soit des processus ou des *threads* (processus légers). Les services proposés permettent :

- la synchronisation ;
- l'exclusion mutuelle avec utilisation de protocoles évitant l'inversion de priorité;
- la communication entre processus et *threads* à l'aide de files de messages ;
- la gestion du temps.

L'ordonnancement des tâches est basé sur diverses stratégies : *sched fifo* (politique en *First In First Out*), *sched rr* (politique en *Round Robin*), ou *sched other* (politique spécifique à une mise en oeuvre donnée). Des exemples d'exécutifs temps réel basés sur POSIX sont VxWorks, RTLinux ou LynxOS.

OSEK/VDX OS

Ce standard (cf. § Annexe 1 - 2) s'adresse à l'électronique embarquée dans le domaine de l'automobile. Il définit un ensemble de services de système d'exploitation, analogues à ceux proposés dans POSIX. On distingue deux types d'entités exécutives : les routines d'interruption et les tâches. Leur ordonnancement est basé sur des priorités statiques. Les routines d'interruption ont une priorité supérieure aux tâches. La gestion des routines d'interruption est prise en charge par le matériel (leur priorité dépend donc de la plateforme d'implantation). Les tâches quant à elles sont gérées à l'aide d'un ordonnanceur classique. Lors du développement d'un système, tous les objets (aussi bien de l'application que du système d'exploitation) doivent être alloués statiquement. Il existe aussi une autre spécification d'OSEK, qui est plutôt basée sur un schéma d'exécution de type *time-triggered*, compatible avec OSEK/VDX et appelée OSEKtime.

Dans nos hypothèses de la plateforme d'implémentation nous avons choisi de pouvoir disposer d'un exécutif temps réel basé sur le standard OSEK/VDX OS. Pour notre développement de l'intergiciel nous avons dû procéder à un choix d'un langage temps réel pour lequel il existe une implémentation de ce standard. Et plus particulièrement, une implémentation à destination des microcontrôleurs de faible coût (pouvant être acquis pour les besoins de simulation et de test). À défaut de trouver une solution plus appropriée que celle du noyau PICos18 [WPICOS18] (cf. Annexe 3) pour l'architecture matérielle des microcontrôleurs PIC18 [WPIC18] (cf. Annexe 2) qui plus est distribuée en *open source* sous licence GPL (*General Public Licence*). Cet exécutif est implémenté avec le langage C ANSI [RITC88] (mais adapté aux microcontrôleurs [WCPI]). N'ayant pas d'autre alternative, nous avons utilisé ce langage pour notre implémentation de l'intergiciel dans le but d'uniformiser le développement.

L'utilisation du langage C [WCMIC] qui est considérée de haut niveau (par rapport au langage assembleur, *Assembly Language*) permet de simplifier la programmation par l'utilisation de fonctions de haut niveau. Il est clair que l'écriture de programmes de plusieurs centaines de lignes impose quasiment l'emploi de langage de haut niveau (comme le C, JAVA, C++, BASIC, etc). L'utilisation de langages orientés objet n'étant pas encore commune dans le domaine des petits microcontrôleurs, même si certains compilateurs C++ commencent à faire leur apparition sur le marché. L'autre avantage du langage C est qu'il reste assez proche de la machine utilisée. Il faut prendre en compte que toutes les fonctions du langage C ANSI ne sont pas implémentées dans les bibliothèques offertes par les compilateurs (cf. § III - 1 - 8).

La majorité des exécutifs temps réel pour microcontrôleurs est écrite en langage C vu la prolifération des compilateurs (par exemple PICC-18 STD [WHITEC], CC5X [WCC5X], MediumC, IAR EW [WIAR], MPC CDS [WMPC], GRCC [WGRCC], pico-C [WPICC], Parsic [WPARS]) pour ce genre d'architectures mais des propositions non encore éprouvées en langage Java existent comme [STIL06]. Il est aussi possible de trouver des compilateurs pour le C++ qui semble très intéressant même si l'utilisation du C++ pour un microcontrôleur semble un peu luxueuse.

IV - 6 - 3 Le développement pour PIC

Pour la phase de codage du prototype du *framework* technique, nous avons décidé que l'implémentation de l'intergiciel sera déployée sur une architecture d'ECU de type microcontrôleur (cf. § III - 1 - 5). Un microcontrôleur est une unité de traitement de l'information de type microprocesseur RISC (*Reduced Instruction Set Computer*), à laquelle on a ajouté des périphériques internes permettant de réaliser des montages sans nécessiter l'ajout de composants annexes. Un microcontrôleur peut donc fonctionner de façon autonome après programmation.

Cet ECU est physiquement matérialisé par un PIC (ou *PICmicro* dans la terminologie du fabricant) qui appartient à la famille de microcontrôleurs de la société Microchip. En ce sens, les PICs sont particulièrement bien dotés et avec des fréquences particulièrement adaptés (aux alentours de 10Mhz et quelques 10MIPS), car ils intègrent mémoire de programme, mémoire de données, ports d'entrée-sortie, et même une horloge, bien que des bases de temps externes puissent être employées. Les PIC disposent de plusieurs technologies de mémoire de programme ROM (EPROM, EEPROM, UVPROM, flash).

Pour le développement d'applications pour PIC, il existe plusieurs choix de développement :

- Simulation puis programmation avec un programmeur puis insertion sur la maquette cible et essais finaux. ;
- Programmation et débogage in situ (ou in circuit) avec le PIC cible qui nécessite un module d'interfaçage ICD (*In Circuit Debugging*). Mais cela ne fonctionne que pour certains PICs particuliers ;
- Utilisation d'un émulateur puis programmation d'un PIC avec un programmeur, puis insertion sur la maquette cible et essais finaux (cf. Figure 50).

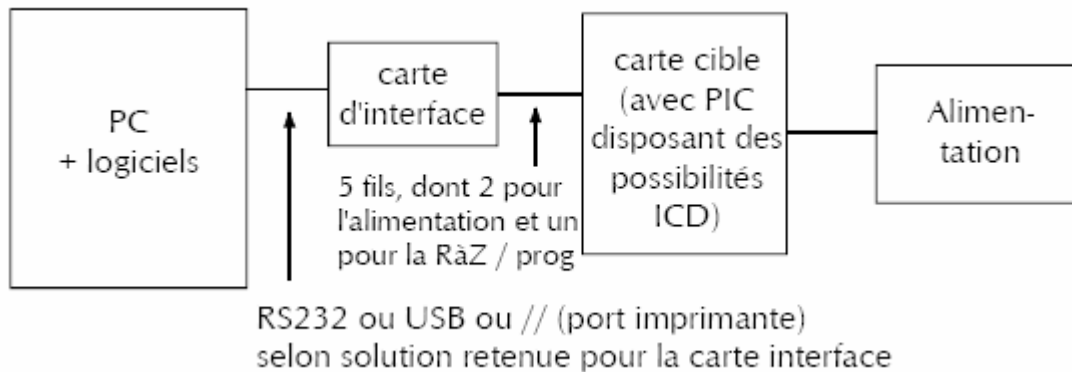


Figure 50 : Programmation et débogage in situ pour PIC. [MOREN]

IV - 6 - 4 Architecture logicielle pour le développement

Il existe cependant des environnements de développement graphique permettant la programmation et l'exécution contrôlée du programme in situ mais leur utilisation est uniquement adaptée pour des applications assez simples. C'est entre autre pour cette raison que nous avons préféré adopter une méthode de programmation en langage C et de débogage in situ car les fonctionnalités de notre intergiciel seront difficilement illustrables dans un langage graphique d'organigrammes.

Pour cette méthode, nous avons retenu (cf. Annexe 4) ici l'association de deux logiciels :

- L'Environnement de Développement Intégré (EDI ou IDE : *Integrated Development Environment*) MPLAB [WMPL] distribué actuellement par Microchip [WPIC18], et qui permet la gestion de projet, la saisie du programme source, la simulation, la programmation et le débogage avec différentes possibilités. Il offre pour cela un environnement émulé de l'architecture matérielle d'un PIC ;
- Le compilateur MCC18 [WMCC], également de Microship, dont il existe une version gratuite limitée s'est avérée suffisante pour la durée du stage et que l'on a intégré à MPLAB.

Pour synthétiser l'architecture logicielle de développement que nous avons utilisé, nous pouvons compléter ces logiciels par notre choix du noyau temps réel PICos18 (cf.

Annexe 3) et du module ECAN [WECAN]. Cette dernière bibliothèque écrite en langage C permet par des fonctions préimplémentés de :

- Emettre et recevoir des trames CAN Standard ou étendues ;
- Programmer des filtres de réception Standard ou étendus.

IV - 6 - 5 Architecture matérielle pour le développement

Avec une petite carte d'interface entre un PC et une carte cible (incluant des contrôleurs de protocole CAN), il est possible de disposer à faible coût d'un système de développement complet pour certains PIC de Microchip qui ont de la mémoire flash pour permettre des écritures multiples nécessaires lors de la phase de mise au point. On regrettera de n'avoir pas pu disposer au moment du développement d'une telle carte. Ce qui nous a contraint à uniquement à une simulation de programme et à un débogage avec l'IDE MPLAB pour une étude fine du comportement fonctionnel de l'intergiciel dans cet environnement simulé. Cependant nous présentons ici la solution matérielle cible que nous avons retenu pour le déploiement :

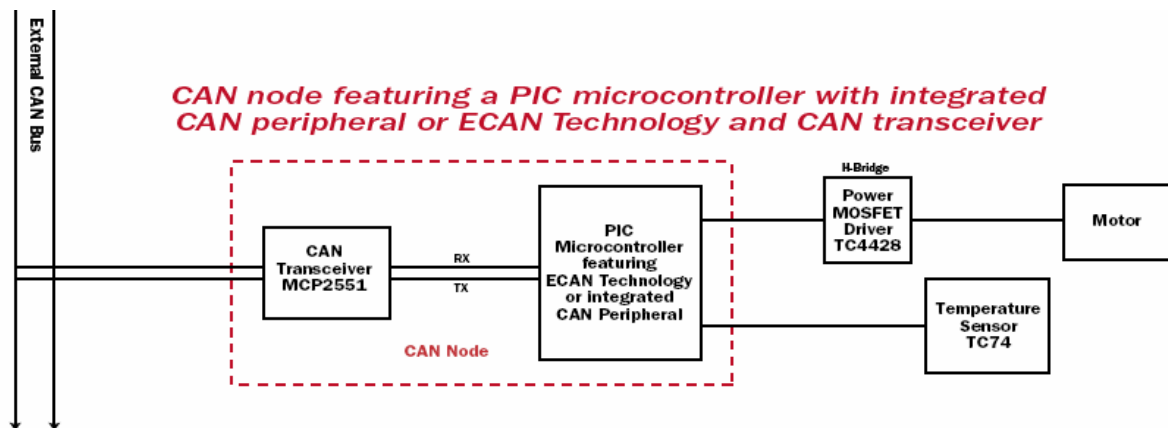


Figure 51 : Solution d'architecture pour le développement sur CAN. [WCANS]

En effet, une autre solution existe, et qui consiste à utiliser un contrôleur CAN (le MCP2551 de Microchip [WMMCP]) physiquement séparé du PIC. Ayant défini dans notre conception un composant en charge de la gestion de la communication avec un périphérique, nous utilisons pour cela le module logiciel ECAN [WECAN] qui est intégré au code de l'empreinte logicielle contenue dans le PIC. Pour le choix du microcontrôleur, nous avons retenu le PIC 18F458 (cf. Annexe 2) pour servir d'ECU. Ainsi lors de la compilation (l'édition de lien plus précisément) nous invoquons un fichier d'édition de lien définissant l'organisation mémoire du PIC et les paramètres d'initialisation de sa pile logicielle (taille et emplacement).

IV - 6 - 6 Environnement de travail

Ainsi nous pouvant présenter l'environnement de travail dans lequel s'est passé le stage :

Environnement matériel du développement

Le travail a été réalisé sur un PC de l'équipe TRIO, équipé d'un processeur Intel Pentium IV[®], de 256 Mo de mémoire vive, faisant office de station de déploiement et de test. Un second ordinateur personnel portable, équipé d'un processeur Intel Centrino DUO[®] et de 1 Go de mémoire vive, utilisé comme station de développement. Ces ordinateurs font partie du même sous domaine réseau, ce qui a permis de réaliser une plateforme de développement sécurisé pour une plus grande portabilité des codes sources.

Environnement logiciel du développement

L'ordinateur de bureau est équipé du système d'exploitation *Windows XP[™] Professional Edition (SP 2)*. Il contient :

- Un environnement d'exécution *JAVA[®] (JRE en version 6 update 1)* pour exécuter l'application du générateur,
- L'environnement intégré de développement *Microchip[™] MPLAB[®] IDE (version 7.51)* utilisé pour la simulation de programme de l'intergiciel.

L'ordinateur portable tourne sous *Windows XP[™] Professional Edition (SP2)* servant de station de développement contient :

- Un environnement intégré de développement *Microchip[™] MPLAB[®] IDE (version 7.51)* pour le développement de l'intergiciel générique.
- Le compilateur croisé *Microchip[™] MCC18[®] (MPLAB[®] C18 C compiler)*.
- *Oracle JDeveloper 10g[™] 10.0.3.1* pour le développement du générateur d'intergiciels.

Par ailleurs, les logiciels suivants ont également été utilisés :

- *Microsoft Project 2003[™]* pour le suivi du projet,
- *Microsoft Word 2003[™]* et *Texmaker 1.5* pour la rédaction des rapports et de la documentation,
- *Microsoft Visio 2003[™]* pour la modélisation des diagrammes UML,
- *Adobe Photoshop Element[™] 2.0* pour les réalisations graphiques.

IV - 6 - 7 La méthodologie de développement

C'est ici que s'achève la réalisation du projet, et nous pouvons donc présenter la Figure 52 qui correspond à toutes les phases de la méthodologie d'un intergiciel temps réel embarqué dans l'automobile que nous avons pu évoquer dans ce mémoire.

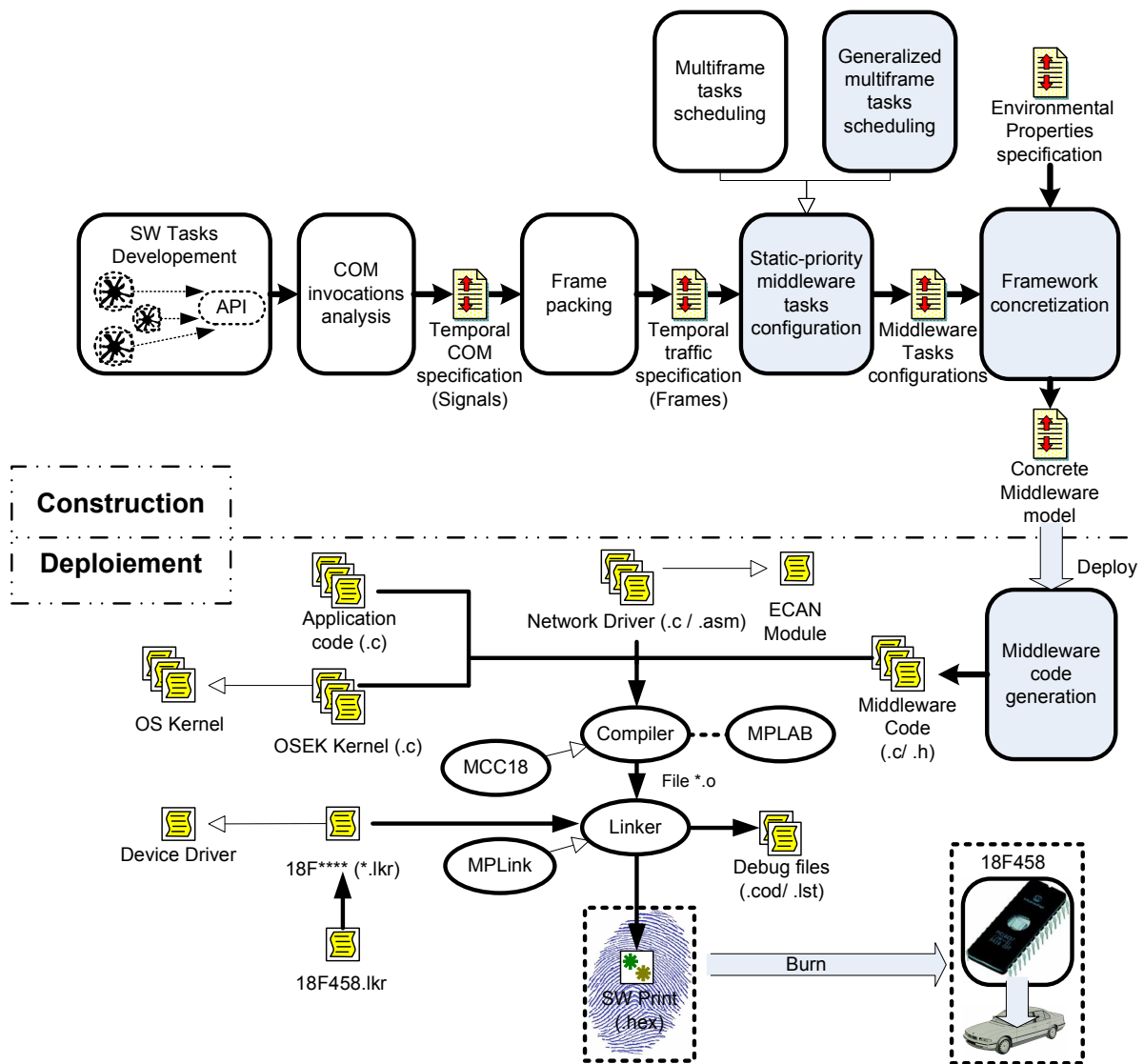


Figure 52 : La méthodologie de développement globale.

Ainsi nous pouvons concrètement présenter notre approche de réalisation d'une application distribué temps réel embarqué dans l'automobile. La phase de construction présentée par [RSM06] est alors complétée par la phase de déploiement qui consiste en une intégration des différentes couches logicielles composant l'application.

Nous avons vu que la concrétisation du *framework* technique se fait par l'adaptation du modèle générique de l'intergiciel. Ceci se fait par une génération du flot de contrôle de certains composants configurables de l'intergiciel produisant ainsi le code source de l'intergiciel sur chaque ECU composant le système. Ce code source spécifique au modèle de répartition d'une application temps réel peut alors être intégré au reste des couches logicielles, à savoir : le gestionnaire du périphérique d'accès au bus de communication, l'exécutif temps réel et les tâches concurrentes formant la couche applicative. Nous entendons par intégration le fait d'importer tous les codes sources dans un seul et unique projet d'un environnement de développement (ce sera MPLAB dans notre cas). Ce projet subit alors la chaîne de

compilation d'un compilateur croisé (MCC18) produisant au final l'empreinte logicielle de l'application qui pourra être gravée sur un microcontrôleur (PIC18F458) pouvant être monté dans un véhicule.

IV - 6 - 8 Les résultats du prototypage

Le prototype de notre conception a pu aboutir aux résultats suivants :

Adaptation du développement du frame packing:

- Dépouillage des résultats du codage ;
- Génération d'un fichier à plat contenant la spécification temporelle du trafic ainsi que le modèle de distribution (cf. § IV - 2).

Développements pour le générateur :

- Codage en Java des prémisses du modèle fonctionnel du générateur défini dans le (cf. § IV - 4) modèle de configuration de l'intergiciel et le modèle fonctionnel du générateur ;
- Codage de l'algorithme de configuration en *Generalized Multiframe Model* [BAR99] (prenant en entrée un ensemble de trames périodiques à départs simultanés) pour la tâche en charge de l'envoi de l'intergiciel ;
- Production d'une matrice contenant la configuration des fenêtres temporelles (les trames à envoyer), le vecteur des périodes d'activation et le vecteur des périodes des exécutions.

Développements pour l'intergiciel :

- Codage en ANSI C ([WCMIC] pour PIC) sous MPLAB [WMPL] et MCC18 [WMCC] au dessus du micronoyau PICos18 [WPICOS18] pour le PIC18F458 [WPIC458] ;
- Emulation du protocole CAN par un *driver* série RS232 [WRS232] ;
- Implémentation de la couche d'interaction (cf. Figure 31) selon la spécification OSEK/VDX COM comme le veut la conception (cf. § IV - 5 - 1) ainsi que les services liés à : l'initialisation, le démarrage, l'arrêt, l'envoi et la réception de messages (par *callback*);
- Codage des fonctionnalités d'interposition et d'interception de la couche *Task Side* (cf. Figure 38) du *framework* technique à savoir le talon de communication et la passerelle de communication distante ;
- Implémentations primaires du composant *Scheduler* comme tâche OSEK/VDX OS en charge de l'envoi (sélection des signaux et empaquetage des trames) et le *Dispatcher* comme ISR pour la réception.

IV - 7 Conclusion sur la réalisation

Le prototype valide et termine éventuellement la phase de conception générique. Nous sommes maintenant arrivés au terme de ce mémoire et par conséquent au terme de ce stage et allons en récapituler les phases de réalisation.

La conception générique avec UML s'appuie sur le développement de *framework* répondant aux spécifications techniques. Aujourd'hui, la conception orientée objet consiste à recourir aux *design patterns* ainsi qu'à schématiser les mécanismes les plus utilisés. L'organisation en architecture technique, puis en composants logiciels doit ensuite répondre à des objectifs de réutilisation, de fabrication et de déploiement. Nous avons donc suivi les étapes suivantes pour la conception générique qui sont :

Elaboration du modèle logique de conception et du framework technique :

- Synthétiser les besoins et les propositions existantes ;
- Schématiser le contexte d'utilisation de notre système ainsi que les séquences des cas d'utilisation de l'intergiciel;
- Représenter de la même façon les mécanismes de notre conception et leur identification avec des valeurs étiquetées ;
- Identifier et schématiser avec UML le modèle logique du système.

Elaboration de la conception du générateur :

- Identifier les composants logiciels correspondant aux fonctionnalités du générateur ;
- Schématiser par un diagramme de classes UML la structure logicielle du générateur ;
- Identifier les composants logiciels correspondant au *framework* technique de l'intergiciel;

Organisation du modèle fonctionnel de l'intergiciel ainsi que la répartition des services offerts :

- Schématiser à l'aide de diagrammes de classes le modèle générique de l'intergiciel ;
- Exhiber par des diagrammes de séquences les interactions entre l'intergiciel et la couche applicative ;
- Elaboration du modèle de configuration logicielle de l'intergiciel par une présentation des tâches le caractérisant ;
- Identifier les composants fonctionnels faisant l'objet d'une génération de code ;
- Organiser la configuration de l'intergiciel indépendamment de son modèle structurel ;
- Schématiser l'interaction entre les *designs patterns* que nous avons utilisé.

Ensuite la réalisation des prototypes a pu mettre l'accent sur :

- Les choix technologiques que nous avons pu prendre pour la réalisation aussi bien du générateur que de l'intergiciel ;
- Les fonctionnalités retenues pour le codage du générateur ;
- Le développement croisé des composants de l'intergiciel générique pour une architecture de PIC ;

Pour finir par le choix de l'architecture matérielle cible sur laquelle un déploiement de l'intergiciel pourra être fait.

V Conclusion et perspectives

Nous avons présenté, en premier lieu de ce mémoire, la spécificité des systèmes temps réel embarqués et des applications concurrentes vis à vis des systèmes informatiques classiques. Nous avons mis en évidence l'importance de la quantification du temps dans les applications temps réel en passant en revue leurs architectures et nous avons alors énoncé les différents modèles de tâches qui les composent ainsi que les différentes mesures pouvant y être associées.

Suite à cela, nous avons évoqué le contexte des systèmes temps réel embarqués dans les véhicules, les problématiques de maîtrise des coûts et de l'intégration logicielle auxquelles doivent faire face les constructeurs lors de la production. Nous avons ainsi pu mettre en évidence l'évolution des architectures électroniques et logicielles dans le domaine automobile vers la distribution des calculateurs et les difficultés de réutilisation du logiciel dans la répartition des fonctionnalités. Ceci nous a valu de procéder à une étude des architectures distribuées ainsi que la nécessité d'user de nouvelles techniques empruntées au génie du logiciel, et qui sont des architectures construites au dessus d'intergiciel.

La diversité des systèmes de communication et la spécificité des architectures des systèmes distribués dans la manufacture automobile, particulièrement voués à des contraintes temporelles strictes, nécessitent le recours à des méthodologies et architectures d'intergiciels innovants dont le souci est double. Premièrement, offrir des services de communication de haut niveau (envoi et réception de signaux) aux couches applicatives en masquant la disparité du modèle de répartition dont les exécutifs temps réel à micronoyau en sont dépourvus. Et deuxièmement, garantir sur chaque site de l'application distribuée le respect des contraintes de fraîcheur imposées aux interactions asynchrones inter-tâches. Ceci, nous a conduit à l'élaboration d'une architecture cible abstraite qui rend compte des modèles que nous avons retenu pour notre vision du système à développer.

Dans une seconde partie du mémoire, nous avons présenté les exigences du processus de développement qui nous a permis de réaliser une conception générique du système. Dans ce processus, nous avons défini le contexte d'utilisation et les exigences liées à l'utilisation du système, baptisé *GenIETeR*. En écumant les catalogues des *design patterns* et les modèles théoriques nous avons construis un archétype, pour notre contexte, d'un intergiciel adaptable par génération de code source et d'un générateur concrétisant un *framework* technique. Ayant aboutit à un *framework*, la conception technique sollicitait donc une validation des modèles utilisés par une phase de prototypage.

Cette phase de prototypage réunissant le développement conjoint du générateur et de l'intergiciel a justifié le choix de notre architecture cible pour les PICs, et valider le *framework* technique construit au niveau de la conception générique.

Et finalement, nous avons complété la méthodologie de construction des intergiciels temps réel embarqués dans l'automobile par une étape de déploiement. Partant d'une spécification donnée du modèle de répartition et de la genèse du trafic temporel des échanges asynchrones entre les tâches applicatives, un générateur produit l'empreinte logicielle d'un intergiciel configuré et optimisé (pour architecture matérielle concrète) accomplissant en ligne, et sur chaque ECU d'un véhicule, les mécanismes d'une communication distante asynchrone tout en garantissant le respect de la criticité des interactions.

V - 1 Les résultats du projet

Nous voudrions que les réalisations accomplies durant ce stage soient à l'image d'une contribution à la définition de [RSM06] d'une méthodologie de développement d'un intergiciel automobile centré sur les services liés à la communication. De ce fait, on pourrait mesurer le succès du projet en fonction des objectifs énoncés et qui nous ont guidé tout au long de notre travail. De ce point de vue, les objectifs ont été en grandes parties atteints.

Concrètement durant notre réalisation, nous avons donc essayé au mieux d'aborder :

- L'aspect méthodologique, qui concerne les techniques de mise en œuvre des applications aussi bien du domaine du temps réel embarqué dans l'automobile que celui du développement pour PIC ;
- L'aspect spécification fonctionnelle, qui concerne l'organisation des besoins fonctionnels exprimant l'utilisation de notre système, et exprimés par les cas d'utilisation et les contraintes dynamiques ;
- L'aspect logique relatif à l'organisation du modèle de solution élaboré en classes regroupées en catégories (ou stéréotypes), les interfaces, les associations, les généralisations, les réalisations, les attributs, les états, les opérations et leurs méthodes. Fournissant la vision « prêt à implémenter » de notre solution ;
- L'aspect conceptuel et structurel correspondant à l'organisation en canevas et en patrons de classes que nous avons pu mettre en œuvre, et qui regroupe temps réel et paradigme objets;
- L'aspect spécification logicielle et génie du logiciel concernant la répartition en couches des exigences techniques de notre solution afin de dissocier par nature les différentes responsabilités offertes en interface ;

- L'aspect technique développe selon un premier point de vue la structure physique des ECUs (architectures logicielles et matérielles) et des réseaux (protocoles et architectures) sur laquelle repose notre système applicatif. Et d'un autre point de vue, il concerne les éléments de séparation des techniques réalisées en « hors-ligne » (comme la phase de construction : *frame packing*, ordonnancement, faisabilité du système, génération de l'intergiciel) et ceux qui sont dits « en ligne » pour désigner leur exécution en environnement (plus précisément embarqué : intergiciel, exécutif temps réel, micronoyaux, tâches applicatives) ;
- L'aspect configuration retrace les paramètres de dimensionnement du système qui permettent à l'adaptation de l'intergiciel déployé (l'empreinte logicielle ou l'image de l'intergiciel), de s'exécuter tout en respectant les contraintes temporelles imposées sur les tâches, les signaux et les trames.
- L'aspect optimisation expose la manière dont sont construits les composants logiciels qui servent à l'élaboration de l'intergiciel, qui paradoxalement, minimise l'intervention humaine et augmente le déterminisme du comportement en environnement ;
- L'aspect technologique représente les choix que nous avons retenus pour les différentes phases du projet à savoir UML pour la modélisation objets, 2TUP pour le processus, OSEK/VDX (OS et COM) comme standard d'architecture, ANSI C et Java pour les langages de programmation et les différents produits aussi bien logiciels (PICos18, MPLAB, MCC18, JDeveloper, ECAN, etc.) que matériels (PIC18F458, CAN, etc.) ;
- L'aspect implémentation correspondant aux paradigmes et aux spécificités de la programmation des PICs d'une part. Et d'autre part, il concerne la définition d'une architecture optimale capable de grouper les spécifications de code de l'intergiciel, ainsi que son modèle de programmation concurrente en usant de *design patterns* ;
- L'aspect déploiement représente la structure de l'intergiciel (modèle de tâche concurrente) sur chaque ECU et localise la dispersion des composants sur le réseau physique et leur interopérabilité ;
- L'aspect exploitation concerne, d'une part, la dichotomie entre générateur et intergiciel générique. Et d'autre part, il désigne l'organisation des composants et identifie les interfaces et les fonctions qu'ils prennent en charge lors de l'exécution (sous une forme de tâches OSEK/VDX OS : périodique ou ISR) en environnement ;
- L'aspect non fonctionnel voulant paradoxalement, avec une stratégie prédictive de maintenance, minimiser la consommation par l'intergiciel de ressources matérielles

avec des configurations logicielles minimales (en le réduisant aux services réellement nécessaires pour l'application considérée), et par la même occasion en augmenter la réutilisation au travers sa généricité (qu'il soit indépendant vis-à-vis de l'hétérogénéité des supports d'exécution).

V - 2 Conclusion

Mon stage s'achevant au stade du prototypage, qui je l'espère ne représente pas une fin en soi, je vais dans ce qui suit dresser un bilan personnel des compétences acquises durant ce projet.

Au début du stage l'objectif, comme le présentait le sujet et comme je le prévoyais moi-même, revêtait la forme suivante :

*« L'objectif du stage est de mettre en oeuvre de telles techniques ... » [RSM05] « ...pour la génération d'intergiciels de communication embarqués pour l'automobile ... cette approche doit permettre de concevoir un générateur d'un ensemble de tâches implémentant cette couche de communication pour un type d'application donnée ... L'auditeur devra assurer le déploiement de l'intergiciel généré sur une plateforme temps réel. Le résultat du stage sera donc un **logiciel** qui, ... »*

Pour la mise en œuvre d'une solution logicielle, et comme le préconisent les meilleures pratiques des méthodes de développement, il faut commencer par l'analyse de l'existant et la capture des besoins fonctionnels. Vraisemblablement cela tient de l'ingénierie logicielle. Mais très rapidement, ce travail s'est transformé en une extraordinaire exploration bibliographique assez surprenante, je dirais, par rapport aux antérieurs projets informatiques (stage de technicien supérieur et ceux d'application d'ingénieur) auxquels j'ai pu participé.

La Figure 53, plus explicite à mon avis qu'un diagramme de Gantt, est l'illustration des efforts impartis à chaque phase du projet. Il est évident que dans ce cas, on ne peut parler ni de transitions ni de processus itératif. Vu la période limitée (moins de quatre mois) d'un tel projet de fin d'études, et même si les pics des courbes représentent mon activité principale, je ne puis m'empêcher de continuer à m'occuper parallèlement des autres tâches qui m'incombais.

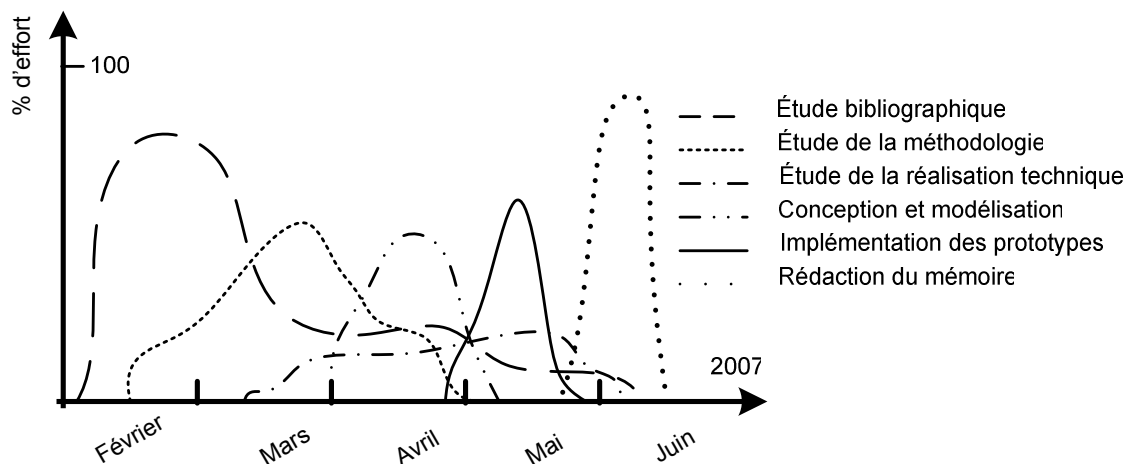


Figure 53 : Courbe d'efforts impartis au projet.

Il m'a fallu, durant ce stage au LORIA, m'impliquer pour la première fois dans une étude des modèles théoriques qui constituent le contexte de mon application dite « réactive », ainsi que dans l'analyse des exigences régissant un domaine professionnel (*business logic*) comme pour le cas des applications transformationnelles. Ceci m'a valu de :

- Approfondir mes connaissances des systèmes « temps réel » que ne je connaissais que partiellement, ainsi que celles relatives aux propriétés d'architectures, des modèles de ces systèmes réactifs, de l'ordonnancements, des algorithmes, etc. ;
- Comprendre les enjeux (d'intégration massive, de la réutilisation et la réduction aussi bien du coût que de l'encombrement électrique) du domaine de l'électronique et du logiciel embarqué dans l'automobile (les architectures matérielles, ceux s'apparentant à la spécification OSEK/VDX et les protocoles utilisés comme CAN, etc.) ;
- Assimiler les concepts liés aux modèles de répartition (ECUs, signaux, interactions, trames, hétérogénéité, localisation, etc.) et au dimensionnement temps réel dans les architectures distribuées pouvant être embarquées dans un véhicule aux travers des technologies utilisées ;
- Parfaire mes connaissances des intergiciels (réflexifs, configurables, adaptables, personnalisables), surtout ceux destinés et employés dans l'industrie auto ;
- Etudier les *design patterns* issus des mécanismes temps réel et pouvant y être utilisés;
- S'initier à la conception générique et la construction de *frameworks* techniques ;

- Etudier la faisabilité technique du développement pour microcontrôleurs et de déploiement sur PIC ;
- Apprendre la programmation spécifique et « méticuleuse » de ces PICs, même si le codage doit se faire en langage « de haut niveau » comme le C. Il n'en reste pas moins, que par exemple, dans la déclaration d'une variable, le choix du type (en terme d'encombrement mémoire) doit être optimisé pour éviter la consommation inutile des ressources ;
- Saisir les concepts du codage dans l'espace du micronoyau de l'exécutifs temps réel PICos18, pour lequel, il n'est pas possible de suivre la trace d'exécution (*logging*, car par exemple la fonction *printf* de la bibliothèque *stdio.h* est inhibée) que par l'unique moyen du débogage, des *breakpoints*, et des *watches*. Même si la plupart des bibliothèques standards du langage ANSI C sont implémentées, il n'est pas possible (même, prohibé) d'accéder à des mécanismes d'allocation dynamique de la mémoire. Forcément, cela est contraignant pour implémenter des structures de données évoluées, sauf en ayant recours aux macros ;

Et finalement, pour la première fois, il m'a été possible de travailler dans un laboratoire de recherche tel que le LORIA et de pouvoir découvrir l'Univers de la recherche française en Informatique. Le travail en équipe (même sur des thématiques différentes), la modestie, le dévouement et l'ouverture des Hommes m'a agréablement fait découvrir qu'il existait une vie de Chercheur passionné.

Regarder « loin », c'est regarder « tôt ».
[Hubert Reeves]

V - 3 Perspectives

Il est évident qu'un projet de cette nature ne se termine jamais. Le prototype que nous avons réalisé préfigure un intergiciel utilisable dans des applications réelles.

Techniquement, il est regrettable que le développement n'ait pas pu être réalisé sur une carte cible de démonstration (*Demonstration Board*), mais je crois personnellement que c'est ce qui est le plus urgent à faire à ce stade d'avancement du projet.

Cependant bien d'autres améliorations peuvent être faites dans *GeniETeR* pour qu'il puisse être réellement exploitable et industrialisé. En outre, l'architecture générique aussi bien du générateur que celle de l'intergiciel pourra, on espère, supporter des évolutions telles que :

Etudes des performances et optimisation

Il serait intéressant d'évaluer les comportements temporels par des bancs d'essai (*benchmark*) des services de l'intergiciel réalisé. L'IDE MPLAB permet de suivre l'évolution en temps réel des applications concurrentes en environnement émulé. Sauf que pour valider ces résultats, il est nécessaire d'observer le comportement de l'intergiciel en environnement réel pour prendre en considération tous les temps (de latence) intrinsèquement liés au matériel.

Intégration d'un profil UML temps réel

La formalisation de notre conception dans un profil UML pour le temps réel serait un fort avantage pour pouvoir intégrer les modèles de notre intergiciel dans les processus de conception des couches applicatives construites sur celui-ci.

Indépendance par rapport aux langages de programmation

La conception de l'intergiciel que nous avons proposé à ce stade du projet est fortement couplée à l'architecture d'OSEK/VDX OS. En conséquence, et dans le but d'harmoniser les développements et de capitaliser les compétences, le langage d'implémentation de cet intergiciel dépendra forcément du langage avec lequel est écrit l'exécutif temps réel répondant au standard OSEK/VDX OS. Il serait judicieux, d'introduire un langage de description des interfaces comme IDL (*Interface Definition Language*) pour réduire cette dépendance et en augmenter l'interopérabilité.

Portabilité de l'intergiciel sur différentes architectures matérielles

Dans notre cas nous avons choisi l'architecture de micronoyau pour des microcontrôleurs. Cependant, une extension peut être envisagée pour un portage de l'intergiciel sur des architectures de microprocesseur différentes (16-bits par exemple). Des améliorations peuvent aussi être faites sur la manière de charger les bibliothèques de composants statiques de l'intergiciel dans la mémoire étendue d'un PIC et d'en charger que les composants actifs dans la mémoire interne. Ceci permettra de laisser plus de mémoire principale aux tâches applicatives.

Prendre en considération la spécification OSEK/VDK OIL

La spécification d'un système d'exploitation temps réel OSEK/VDX OS a pour but de répondre aux contraintes strictes d'exécution en temps réel des logiciels embarqués utilisés par l'électronique automobile. Elle permet la portabilité des différents modules constituant une application logicielle. L'une des caractéristiques du projet OSEK/VDX est que le système est défini de façon statique lors de la compilation : tous les services utilisés (tâches, messages, etc.) sont définis de façon statique dans un langage OIL (*OSEK Implementation Language*). Comme notre intergiciel réside au dessus d'une implémentation d'OSEK/VDX OS, il serait intéressant d'ajouter une fonctionnalité au générateur permettant de générer une définition OIL de l'intergiciel et de facilement l'intégrer dans la méthodologie de développement OSEK/VDX (cf. Annexe 1 - 1).

Intergiciel pour la tolérance aux fautes Fonctionnement en mode dégradé (temporel)

La conception de l'intergiciel et celle du générateur que nous proposons ne prennent pas en considération la tolérance aux fautes. Plus précisément, on se place dans le contexte de « la prévention » des fautes temporelles (éviter qu'elles surviennent) en garantissant que toutes les contraintes de fraîcheur (des signaux et des trames) soient strictement respectées. Bien que des intergiciels tolérants aux fautes existent, il serait quand même possible de considérer le point de vue de la « guérison » des fautes temporelles (réagir à leurs manifestations). C'est-à-dire, qu'il faudrait prévoir des mécanismes de fonctionnement en mode dégradé de l'intergiciel lui permettant de recouvrir les erreurs survenues. De part la criticité des application temps réel, cela n'est malheureusement pas toujours fonctionnellement acceptable.

Fonctionnement en Gateway

L'architecture électrique dans l'automobile repose aujourd'hui sur les réseaux multiplexés. La question est : un ou plusieurs réseaux ? Un seul réseau, auquel tous les calculateurs seraient reliés et par lequel toutes les informations partagées passeraient, est une idée simple et séduisante. Mais il n'existe pas de réseau parfait. Un réseau est fait de compromis entre performance et coût. Le nombre de nœuds, la longueur du médium et la bande passante sont limités. Les besoins dans l'automobile sont hétérogènes : des informations événementielles et avec réveil par le bus (fonctions de l'habitacle), des informations à cycle d'émission rapide (fonctions moteurs et châssis) et « l'infotainment » embarqué [WINFO] à fort volume de données, ces deux dernières exigeant particulièrement une grande bande passante. Un système automobile est généralement compartimenté pour ne

pas compromettre les fonctions vitales en cas de défaillance d'un calculateur de confort et/ou en cas de sectionnement d'un réseau dans une zone exposée. Les réseaux sont ou ne sont pas de différents types (CAN, LIN, etc.) et communiquent entre eux au travers d'un calculateur communément appelé *Gateway* (Passerelle). Cette passerelle peut être centralisée (tous les réseaux sont connectés à une seule passerelle) ou bien décentralisée ou distribuée (la passerelle interconnecte deux à deux les réseaux), voir Figure 54. De ce fait nous espérons que l'intégriciel puisse prévoir les besoins de telles passerelles pour qu'il puisse être déployé en tant que gateway.

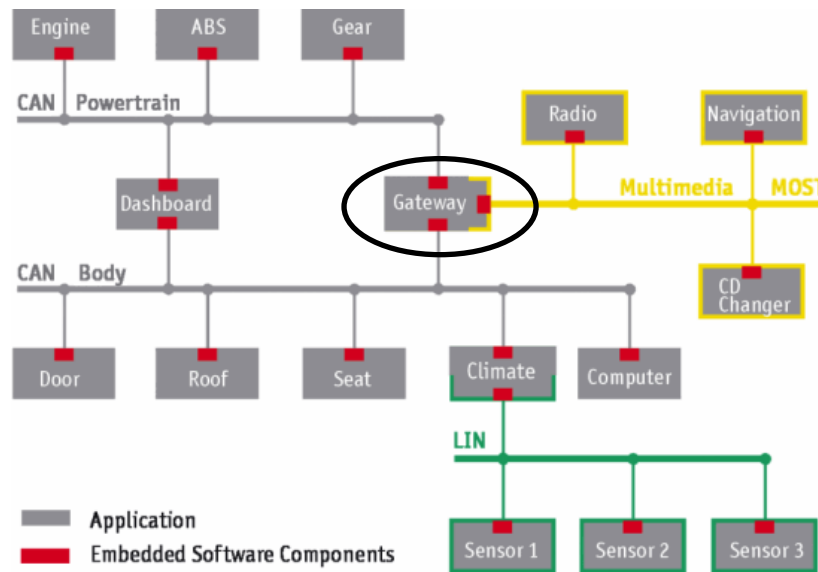


Figure 54 : Passerelle distribuée. [WVECT]

VI Annexes

Annexe 1	L'architecture OSEK/VDX	120
Annexe 2	Le PIC18 : un microcontrôleur à 8 bits	124
Annexe 3	Le micronoyau PICos18 : noyau temps réel pour PIC18	127
Annexe 4	L'IDE MPLAB et le compilateur MCC18 de Microchip™	129
Annexe 5	Le multiplexage dans l'automobile (source [WNET1])	130

Annexe 1 L'architecture OSEK/VDX

OSEK/VDX [WOSEK] (pour *open systems and their corresponding interfaces for automotive electronics/ Vehicle Distributed eXecutive*) est une proposition de standardisation de système des architectures pour les applications automobiles embarquées. Il regroupe le savoir-faire de constructeurs, d'équipementiers et d'universitaires allemands (BMW, Daimler-Benz/ Mercedes-Benz, Opel Volkswagen, Bosch, Siemens et le IIIT de l'université de Karlsruhe) pour OSEK, et de constructeurs automobiles français (Peugeot et Renault) qui ont apporté l'approche VDX. Cette approche définit un OS temps réel, ainsi que des services de communication et de gestion de tâches en réseau.

Annexe 1 - 1 Objectifs d'OSEK

Le but de la standardisation est d'assurer une portabilité du code sur différents ECUs et de permettre la réutilisation des applications afin de réduire leurs coûts de développement. OSEK/VDX cherche à définir une architecture logicielle modulaire et ouverte (cf. Figure 55) ainsi que leurs interfaces associées pour interfacier l'ensemble des ECUs d'un véhicule.

Les fonctionnalités décrites s'apparentent à trois domaines distincts :

- **OSEK/VDX COMmunication** : définit les services pour les mécanismes de communication entre les ECUs et leurs architectures dans les ECUs ;
- **OSEK/VDX Network Management** : définit des services de configuration du réseau d'ECU et de *monitoring* ;
- **OSEK/VDX Operating System** : définit les services d'un exécutif temps réel pour la gestion des tâches concurrentes.

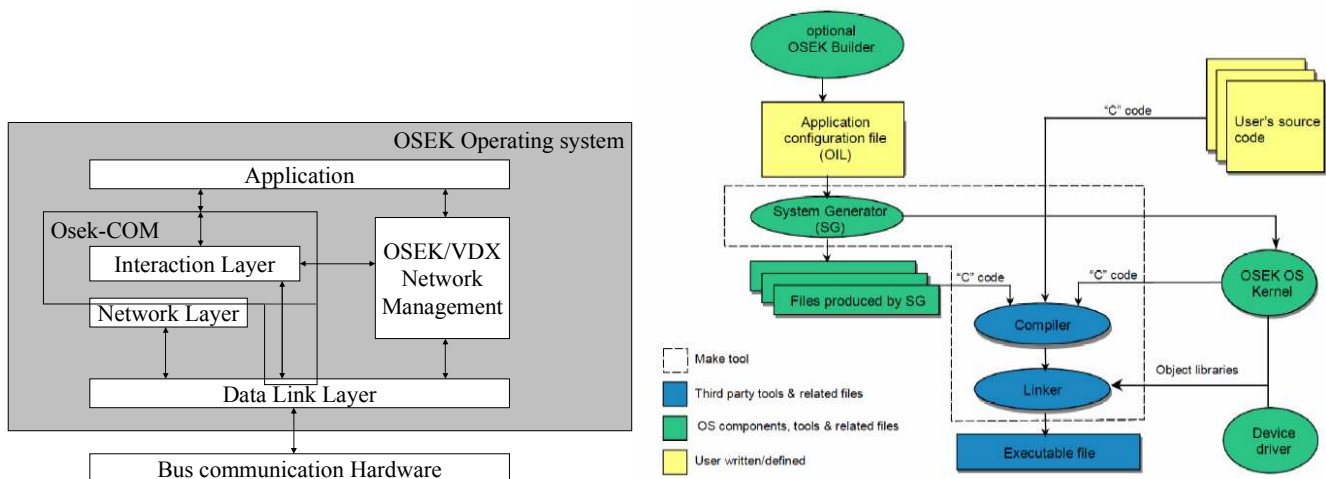


Figure 55 : Architecture et méthodologie de développement OSEK/VDX. [WOSEK2]

Annexe 1 - 2 OSEK/VDX OS

Le système d'exploitation défini par cette norme a été conçu de manière à répondre aux différentes contraintes propres aux applications embarquées et plus particulièrement dans l'automobile. Ces contraintes sont :

- Optimiser l'implémentation de l'OS pour les meilleures performances en terme de temps d'exécution processeur;
- Minimiser l'utilisation de ressources matérielles (RAM, ROM, temps d'exécution CPU) pour pouvoir implanter l'OS sur des microcontrôleurs 8 bits ;
- Concevoir une architecture modulaire pouvant être configurée et dimensionnée en fonction des besoins de chaque application au travers d'interfaces standardisées ;
- Supporter des implémentations en ROM pour que le code puisse être exécuté à partir d'une telle mémoire.

Pour répondre à ce besoin d'adaptabilité, des classes de conformité (*Conformance Classes*) définissent des niveaux d'implémentation portables. Selon le choix de l'utilisateur en besoins fonctionnels, une classe est définie pour permettre la portabilité de l'implémentation sur différentes plateformes souscrivant à cette même classe.

Annexe 1 - 2 - 1 Modèles de tâches

Dans OSEK/VDX OS il est possible d'utiliser un modèle de tâches simples ou de tâches étendues. Une tâche simple est réduite aux états : suspendue, prête à être exécutée ou active (en cours d'exécution). Alors qu'une tâche étendue est une tâche simple à laquelle un état d'attente a été rajouté. Dans cet état une tâche attend l'arrivée d'au moins un événement en provenance d'une autre tâche, d'un compteur ou d'une alarme.

Annexe 1 - 2 - 2 Politiques d'ordonnancement

Le choix de l'ordonnancement des tâches est réalisé à l'exécution par l'Ordonnanceur. Néanmoins, le développeur de l'application peut décider de la séquence d'exécution en configurant les priorités des tâches et en sélectionnant un mécanisme d'ordonnancement parmi les modes non préemptif, préemptif ou mixte. Une hiérarchie des priorités prévoit que le plus haut niveau de priorité d'exécution est réservé aux interruptions matérielles. Le niveau de priorité juste en dessous est réservé aux fonctions de l'OS. Les niveaux inférieurs peuvent être assignés aux tâches applicatives.

Annexe 1 - 2 - 3 Gestion des ressources

Afin de garantir une portabilité des applications, une interface unique permet d'accéder aux ressources matérielles gérées par des services de l'OS. Afin d'éviter les phénomènes d'inter-blocage l'Ordonnanceur utilise le *Priority Ceiling Protocol*. Pour accéder exclusivement à une ressource, une tâche se voit attribuer momentanément une priorité maximale pour qu'aucune autre tâche ne puisse la préempter et accéder à la même ressource.

Annexe 1 - 3 OSEK/VDX COM

OSEK/VDX Communication est un standard de communications orientées messages en réseaux local entre entités électroniques ou ECU. Ces communications inter-ECUs ou intra-ECU sont réalisées par des services communs, la position dans le réseau de l'émetteur et du récepteur d'une communication n'a pas d'importance. OSEK/VDX COM est construit sur un modèle composé de trois couches :

- Une couche réseau constituée de composants qui gèrent matériellement le protocole de communication du bus, ainsi que de drivers services de gestion de ces composants pour l'échange de message ;
- Une couche de liaison de données gère le transfert des trames entre les noeuds ;
- Une couche interaction qui est l'interface avec l'application. Cette couche rend les communications indépendantes du bus ou du protocole choisi.

Les communications sont réalisées par deux types de messages. Les messages de type « état » sont utilisés pour les routines de service du système. Les messages de type « événement » représentent les messages contenant les données en provenance de l'application. Les communications peuvent être selon le modèle point-à-point ou multipoints.

Annexe 1 - 4 OSEK/VDX NM

L'objectif de OSEK/VDX Network Management est de garantir la sûreté de démarrage et de l'arrêt de tous les ECUs, la vérification et la surveillance de la configuration et de l'aide au diagnostic.

Annexe 1 - 5 OSEK/VDX dans *GenIETeR*

Nous avons pu, grâce à la lecture des spécifications, traduire l'API d'OSEK/VDX en un diagramme de classes UML (cf. Figure 56), que nous avons pu utiliser comme un pseudo-profil de stéréotypes pour caractériser certaines facettes de notre modèle d'intergiciel générique (défini dans le *package* OSEKCore de la Figure 32). Cette utilisation du système

d'exploitation n'exprime pas la dépendance vis-à-vis d'OSEK/VDX OS. Mais bien au contraire, elle met en exergue l'interface normalisée que propose notre intergiciel pour une plus grande portabilité, en proposant à la couche applicative une API conforme à la classe de conformité CCCB d'OSEK/VDX COM. La place de OSEK/VDX COM dans l'architecture de notre système est la conséquence du fait que [RSM03] suppose que chaque ECU, de l'architecture distribuée étudiée, soit muni d'OSEK/VDX OS pour la validation de ses modèles. Il exclut en revanche l'utilisation d'OSEK/VDX COM pour son manque de directives d'implémentation et de configuration des services de communication à offrir. Sauf que pour le projet GenIETeR, nous avons décidé d'implémenter une interface d'OSEK/VDX COM qui malgré la complexité de la spécification, nous a parus intéressante à plus d'un niveau.

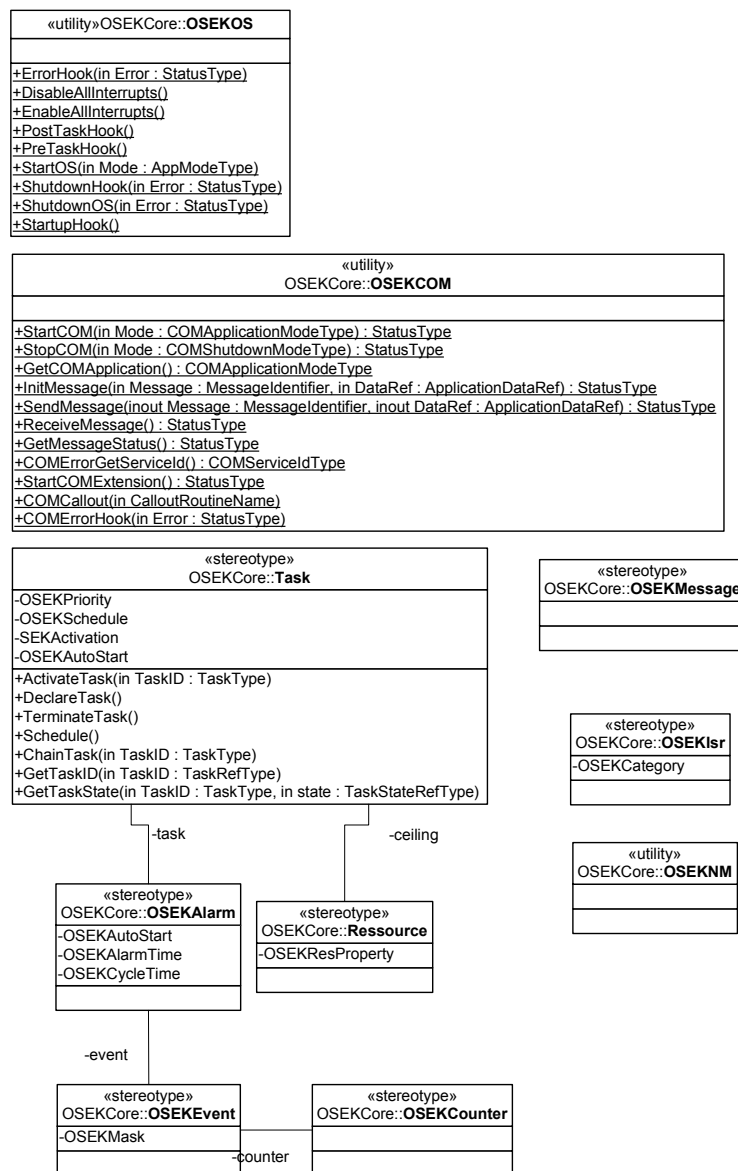


Figure 56 : Vue de l'API de OSEK/VDX.

Annexe 1 - 6 Implémentation d'OSEK/VDX

OSEK/VDX n'est pas en soit un OS temps réel, mais juste un ensemble de propositions de standardisation d'un OS temps réel pour le domaine automobile. Pour être utilisée dans une application, une implantation particulière d'OSEK/VDX doit être utilisée pour chaque type de processeurs utilisés (ou plateforme matérielle).

Les standards OIL (*OSEK Implementation Language*) et ORTI (*OSEK/VDX Real-Time Interface*) ont été créés pour une gestion améliorée et cohérente du système d'exploitation (classes de conformité utilisées, type d'ordonnancement, configuration des tâches ...utilisés par chacun des ECUs de l'application).

Différentes implantations existent aujourd'hui, chacune adaptée à une architecture cible on peut citer [WVECTOR, WPICOS18, WOOSEK, WTRAMP].

Annexe 2 Le PIC18 : un microcontrôleur à 8 bits

Annexe 2 - 1 - 1 Présentation de la famille PIC18

La famille de microcontrôleurs « PIC18 » (par analogie aux 16-bit PIC[®] Microcontrollers et les dsPIC[®] *Digital Signal Controllers*) est fabriquée par la société Microship [WPIC18]. Cette famille est caractérisée par les principaux avantages suivants :

- Architecture optimisée pour les compilateurs C ;
- Technologie *Nanowatt* pour certains modèles, permettant ainsi une faible consommation ;
- Une mémoire programme du type Flash (modèle F) possédant 100000 cycles d'effacement / écriture (programmation) ;
- Vitesse d'exécution pouvant atteindre 10 MIPS, *Méga Instructions Par Seconde* (horloge jusqu'à 10 MHz en mode PLL) ;
- Jeu d'instruction compatible avec la famille PIC16 ;
- Multiplication Hardware 8x8 bits;

- Programmation ICSP (In Circuit Serial Programming) et débogage ICD (*In Circuit Debugging*).

Les noms de ces PICs se composent de 2 chiffres, 1 ou 2 lettres, un nombre. Par exemple le 16F548 où 16 est le type de processeur, F est le type de mémoire flash, 54 est la version du PIC et 8 signifie qu'il intègre un périphérique supplémentaire, ici est une extension CAN 2.0B.

Annexe 2 - 1 - 2 Le PIC18F458

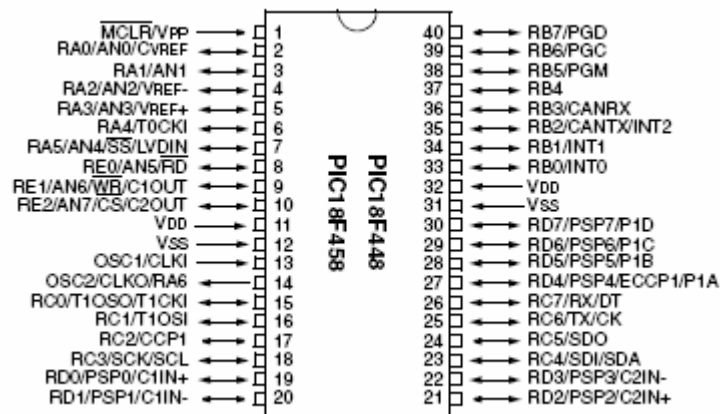


Figure 57 : Le PIC18458.

Ce microcontrôleur se présente sous la forme d'un boîtier 20 broches (DIP et SOIC). Sa structure est du type HARVARD et se compose d'un CPU à architecture RISC 8 bits.

Paramètres	Valeurs
Taille de la mémoire de programme	32 KOctets
Taille de la RAM	1536 Octets
Données EEPROM	256 Octets
I/O	34

Voici les ses caractéristiques au complet :

High-Performance RISC CPU:

- Linear program memory addressing up to 2 Mbytes
- Linear data memory addressing to 4 Kbytes
- Up to 10 MIPS operation
- DC – 40 MHz clock input
- 4 MHz-10 MHz oscillator/clock input with PLL active
- 16-bit wide instructions, 8-bit wide data path
- Priority levels for interrupts
- 8 x 8 Single-Cycle Hardware Multiplier

Peripheral Features:

- High current sink/source 25 mA/25 mA
- Three external interrupt pins
- Timer0 module: 8-bit/16-bit timer/counter with 8-bit programmable prescaler
- Timer1 module: 16-bit timer/counter
- Timer2 module: 8-bit timer/counter with 8-bit period register (time base for PWM)
- Timer3 module: 16-bit timer/counter
- Secondary oscillator clock option – Timer1/Timer3
- Capture/Compare/PWM (CCP) modules; CCP pins can be configured as:
 - Capture input: 16-bit, max resolution 6.25 ns
 - Compare: 16-bit, max resolution 100 ns (Tcy)
 - PWM output: PWM resolution is 1 to 10-bit
Max. PWM freq. @ :8-bit resolution = 156 kHz
10-bit resolution = 39 kHz
- Enhanced CCP module which has all the features of the standard CCP module, but also has the following features for advanced motor control:
 - 1, 2 or 4 PWM outputs
 - Selectable PWM polarity
 - Programmable PWM dead time
- Master Synchronous Serial Port (MSSP) with two modes of operation:
 - 3-wire SPI™ (Supports all 4 SPI modes)
 - I²C™ Master and Slave mode
- Addressable USART module:
 - Supports interrupt-on-address bit

Advanced Analog Features:

- 10-bit, up to 8-channel Analog-to-Digital Converter module (A/D) with:
 - Conversion available during Sleep
 - Up to 8 channels available
- Analog Comparator module:
 - Programmable input and output multiplexing
- Comparator Voltage Reference module
- Programmable Low-Voltage Detection (LVD) module:
 - Supports interrupt-on-Low-Voltage Detection
- Programmable Brown-out Reset (BOR)

CAN bus Module Features:

- Complies with ISO CAN Conformance Test
- Message bit rates up to 1 Mbps
- Conforms to CAN 2.0B Active Spec with:
 - 29-bit Identifier Fields
 - 8-byte message length
 - 3 Transmit Message Buffers with prioritization
 - 2 Receive Message Buffers
 - 6 full, 29-bit Acceptance Filters
 - Prioritization of Acceptance Filters
 - Multiple Receive Buffers for High Priority Messages to prevent loss due to overflow
 - Advanced Error Management Features

Special Microcontroller Features:

- Power-on Reset (POR), Power-up Timer (PWRT) and Oscillator Start-up Timer (OST)
- Watchdog Timer (WDT) with its own on-chip RC oscillator
- Programmable code protection
- Power-saving Sleep mode
- Selectable oscillator options, including:
 - 4x Phase Lock Loop (PLL) of primary oscillator
 - Secondary Oscillator (32 kHz) clock input
- In-Circuit Serial Programming™ (ICSP™) via two pins

Flash Technology:

- Low-power, high-speed Enhanced Flash technology
- Fully static design
- Wide operating voltage range (2.0V to 5.5V)
- Industrial and Extended temperature ranges

Des cavaliers de configuration permettent de choisir entre ces différentes options. Pour plus de détails se reporter au *datasheet* [WPIC458].

Annexe 3 Le micronoyau PICos18 : noyau temps réel pour PIC18

Le micronoyau PICos18 [WPICOS18], développé par la société Pragmatec [WPRAG], est un noyau temps réel destiné aux composants PICmicro de la famille PIC18 de Microchip [WPIC18]. Il est entièrement gratuit et est distribué sous licence GPL. La version actuelle, au jour où nous rédigeons ce mémoire, ne respecte que la norme OS de OSEK (et c'est le seul exécutif abouti pour PIC18, à notre connaissance, qui implémente cette norme). Mais des évolutions sont prévues pour intégrer les moyens de communication définis par OSEK/COM et les services réseaux conformément à OSEK/NM. PICos18 se veut donc être un noyau multi-tâches préemptif temps réel.

Annexe 3 - 1 Architecture du noyau

Le noyau PICos18 est architecturé de la manière suivante :

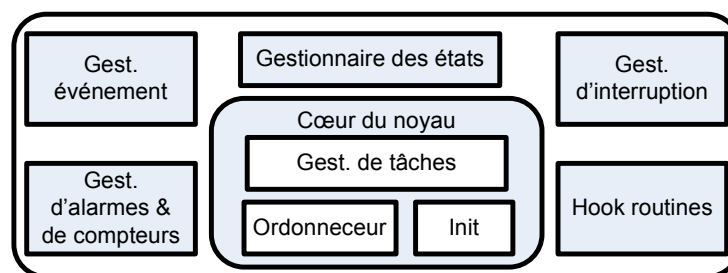


Figure 58 : Architecture de PICos18.

- Le coeur du noyau (Init, Ordonneur, Gestionnaire de tâches) qui a la responsabilité de gérer et d'ordonnancer les tâches applicatives.
- Le gestionnaire d'alarmes et de compteurs répond à l'interruption du TIMER0 afin de mettre à jour périodiquement les alarmes et compteurs associées aux tâches.
- Les Hook routines permettent à l'utilisateur de dérouter l'exécution du noyau de façon à prendre temporairement le contrôle du système.
- Le gestionnaire d'états offre à l'application les fonctions nécessaires à la gestion des états des tâches (changer l'état d'une tâche, chaîner des tâches, activer une tâche...).
- Le gestionnaire d'évènement offre à l'application les fonctions nécessaires à la gestion des évènements d'une tâche (mise en attente sur un évènement, effacer un évènement...).
- Le gestionnaire d'interruption permet à l'application d'activer et de désactiver les interruptions du système.

PICos18 est un noyau modulaire dans le sens où les accès spécifiques aux ressources (drivers, file system manager, etc.) peuvent être réalisés par des tâches dissociées du noyau. De cette façon il permet d'intégrer des modules externes sous forme de briques logicielles afin de constituer un projet, telles que les extensions proposées par la société PRAGMATEC.

Annexe 3 - 2 Performances et caractéristiques

Le noyau PICos présente les performances [WPICOS18] suivantes :

<i>Taille du noyau (ROM) :</i>	< 1 Ko
<i>Taille du noyau (RAM) :</i>	7 octets
<i>Taille des service (ROM) :</i>	4 Ko
<i>Taille des services (RAM) :</i>	121 octets
<i>Taille de la pile hardware :</i>	32 appels de fonctions pour toutes les tâches
<i>Taille de la pile software :</i>	128 octets
<i>Temps de latence de l'Ordonnanceur:</i>	24,5 μ s (Freq = 40 MHz)
<i>Nombre de tâches :</i>	16
<i>Nombre d'évènements par tâche :</i>	8
<i>Nombre de priorités :</i>	15
<i>Nombre de timers logiciels :</i>	Pas de limites
<i>Taille (contexte+pile) d'une tâche :</i>	entre 128 et 256 octets

Ces performances sont généralement suffisantes pour un développement d'applications sur PIC. Le noyau et les services prennent donc moins de 5 Ko de mémoire de code. PICos18 est livré avec le noyau seul ainsi qu'un tutorial. Toutefois le site web [WPICOS18] propose des drivers ou des bibliothèques disponibles gratuitement.

Annexe 4 L'IDE MPLAB et le compilateur MCC18 de Microchip™

Annexe 4 - 1 L'environnement MPLAB

Microchip fournit gratuitement un outil logiciel de développement MPLAB [WMPL] pouvant s'interfacer avec des compilateurs croisés, des programmeurs et des émulateurs. L'Environnement de Développement Intégré (EDI) MPLAB regroupe toutes les fonctionnalités nécessaires à :

- une chaîne d'assemblage / compilation ;
- et un débogueur.

Les logiciels de même type offrent souvent un EDI pour la compilation et un EDI pour le débogueur distincts. Il permet de créer le fichier exécutable et de déboguer des applications pour tous les PICs de Microchip. Pour le débogage, il offre le choix entre :

- un simulateur ;
- un émulateur (ICE 1000, ICE 2000, PICMASTER) dialoguant avec le logiciel par la liaison parallèle ;
- l'utilisation des ressources intégrées dans certains PICs (*In Circuit Debugger*, ICD), via une liaison série et une petite carte d'interfaçage entre le PC et le PIC cible.

Il est possible de choisir le mode de développement avec ou sans débogage suivant la construction du fichier exécutable final. Le mode édition seule permet de réaliser toutes les étapes jusqu'à la construction du fichier exécutable : édition du ou des fichiers source, assemblage/compilation, édition de liens. Dans un mode de développement avec débogage, après construction du fichier exécutable, ce dernier est automatiquement chargé dans la mémoire du simulateur ou de l'émulateur ou dans la mémoire de l'ordinateur pour recopie ou programmation selon le mode de développement (émulateur, ICD).

Plusieurs compilateurs / assembleurs peuvent être utilisés. Un assembleur est livré avec MPLAB : MPASM. Les autres peuvent être achetés et y être intégrés. Microchip a prévu de pouvoir choisir un des compilateurs suivants : Byte Craft, HiTech [WHITEC], IAR [WIAR], CCS, MCC18.

Le fichier final exécutable (lors du débogage ou après programmation du composant) s'appelle le fichier cible (*Target File*) qui un fichier hexadécimal pouvant être chargé sur un PIC. Les noeuds source (*Node Source*) sont les fichiers utilisés pour la création du fichier final. Ce sont aussi des fichiers du projet (*Project Files*). Ils comprennent :

- les fichiers source en langage d'assemblage ;

- les fichiers source en langage C ;
- les fichiers objets résultants d'assemblages ou de compilations précédents ;
- les fichiers bibliothèques utilisateur ;
- le fichier de commande de l'éditeur de liens (Linker Script) ;

Annexe 4 - 2 Le compilateur croisé MCC18

Le développement de programmes pour PIC peut se faire aussi en langage évolué comme par exemple avec le C. Dans notre projet on a souhaité utiliser le langage C. Ce dernier étant plutôt utilisé pour les systèmes temps réel.

Le *cross-compileur* MCC18 [WMCC] de Microship correspond aussi à ces critères. Il s'interface directement avec MPLAB avec lequel un débogage est possible. Il possède cependant les propriétés suivantes :

- Compatibilité C ANSI ;
- Génération de modules objet relogeables ;
- Compatible avec des modules objets générés par MPASM ;
- Bibliothèque étendue incluant des modules de gestion des périphériques ; PWM, SPI, etc. ;
- Contrôle total par l'utilisateur de l'allocation des données et du code en mémoire ;

Le langage C au standard ANSI définit un certain nombre de fonctions standards, qui sont regroupées dans les bibliothèques de fonctions, chacune définie par un fichier d'entête (*header*). Par conséquent on retrouve des fichiers spécifiques au PIC, ceci pour simplifier la gestion des diverses interfaces du microcontrôleur.

Annexe 5 Le multiplexage dans l'automobile (source [WNET1])

Annexe 5 - 1 Multiplexage carrosserie (Classe A) :

Le multiplexage carrosserie concerne le contrôle des éléments de carrosserie tels que rétroviseurs, vitres, sièges, éclairages. Le protocole utilisé est le VAN (Vehicle Area

Network) à faible débit (62.5 Kbits/s), la quantité d'informations qui circulent sur le réseau est faible. Les échanges sont de type maître-esclaves. Le temps de réponse demandé est de 200 ms (entre une commande et une activation).

Annexe 5 - 2 Multiplexage des fonctions Confort (Classe D) :

Le réseau confort permet aux équipements de se partager le système d'affichage. Le protocole utilisé est le VAN à débit moyen (125 Kbits/s). La quantité d'informations qui circulent est très importante. Les échanges sont du type multi-maître. Le temps de réponse demandé est de 500 ms (entre une commande et une activation). Ce type de multiplexage est déjà présent sur le véhicule Peugeot 206.

Annexe 5 - 3 Multiplexage inter-systèmes (Classes B/C) :

Le réseau inter-système permet aux équipements de se partager l'information de leurs capteurs, et à chaque équipement d'adapter son fonctionnement (agrément de conduite, antipollution). Le protocole utilisé est le CAN à débit moyen 250 Kbits/s ou haut débit 50 Kbits/s. La quantité d'informations qui circulent est faible. Les échanges sont du type multi-maître. Le temps de réponse demandé est de 50 ms (entre une commande et une activation).

Les futurs véhicules haut de gamme auront une architecture totalement multiplexés, avec les 3 types de multiplexage.

Pour les modèles de la gamme PSA (comme la Peugeot 607 ou la Xsara Picasso par exemple) l'architecture électrique et électronique s'articule autour du Boîtier de Servitude Intelligent (BSI), véritable "tête pensante" qui centralise et traite les informations issues d'un réseau inter-système mécanique (protocole CAN : contrôler area network) pour ce qui concerne l'ABS, le contrôle des moteurs, la boîte de vitesses automatique, l'ESP... Celui-ci est également relié à trois autres réseaux de protocole VAN dont l'un est dédié au confort (écrans multifonctions, climatisation, guidage embarqué, radio) l'autre à la carrosserie (les sièges et les portes) et le dernier aux fonctions de sécurité (airbags, feux de signalisation). Le BSI peut supporter plusieurs réseaux CAN ou VAN (La Peugeot 306 par exemple possède 3 réseaux VAN et 1 réseau CAN).

VII Bibliographie

- [AD92] M. Alabau and T. Dechaize. *Ordonnement temps réel par échéance*. In T.S.I., volume 11. n.3, 1992.
- [ALB02] Alberto Sangiovanni-Vincentelli, *Embedded System Design: Foundations, Methodologies and Applications*, Chess Seminar, Oct. 15th, 2002.
- [AUD91] N.C. Audsley. *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times*. Technical Report YCS-164, 31, nov 1991.
- [AUD97] N. C. Audsley, A. Grigg., *Timing analysis of the ARINC 629 data bus for real-time applications*. Microprocessors and Microsystems 1997.
- [BAR99] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, *Generalized multiframe tasks*. Real-Time Systems, 17(1) :5–22, 1999.
- [BASH] Omar Bashir, Mubashir Hayat, *Active Transceiver Design Pattern for Data Communication Applications*.
- [BASI05] E. Basilico, *Environnements Java pour applications embarquées et temps réel*. Document interne, projet RT Java / TI 08-01, version préliminaire et provisoire, Laboratoire d'informatique industrielle, EIG/HES-SO, fév. 2005.
- [BEL01] Belloir N. and Al., *Formalisation de la relation Tout-Partie : application a l'assemblage des composants logiciels*, Journées Composants, Besancon, Octobre 2001.
- [BOO99] G. Booch, J. Rumbaugh, I. Jacobson, *The unified Modeling Langage User Guide*, 1999, Addison-Wesley.
- [BRO00] Benjamin M. Brosgol, *A Comparison of Ada and Java™ as a Foundation Teaching Language*, Ada Core Technologies, updated version March 2000.
- [BUR04] Burns, B. Dobbing and T. Vardanega, *Guide for the use of the Ada Ravenscar Profile in high integrity systems*, ACM SIGAda Ada Letters, juin 2004.
- [BW90] A. Burns and A. Wellings. *Real-Time Systems and their Programming Languages*. Addison-Wesley, 1990.
- [CAN] ISO 11898, *Road vehicles - Interchange of digital information - Controller Area Network for high-speed Communication*, 1994, International Standard organization.
- [CHAR] L. Chaari, N. Masmoudi, L. Kammoun, *Etude comparative entre les protocoles du multiplexage de données dans le véhicule*, Laboratoire d'Electronique et Technologie de L'Information.
- [CNR88] G.D.R.T.R CNRS. *Le temps réel, technique et Science Informatiques*, 1988.
- [COFF96] E. Coffman, M. Garey, and D. Johnson, *Approximation algorithms for bin packing: a survey*. In Approximation algorithms for NP-hard problems, pages 46_93. PWS Publishing Co., Boston, MA, USA, 1996.
- [ELL97] Jean Pierre Elloy, *Systèmes réactifs synchrones et asynchrones*. In *Applications, Réseaux et Systèmes*, École d'été temps réel'99, pages 43–51, Futuroscope, Septembre 1997.

- [ETR05] J. De Oliveira, *Dimensionnement temps réel d'un véhicule : étude de cas et perspectives*, ETR 2005.
- [GOF] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [HAL93] N. Halbwachs. Kluwer, *Synchronous programming of reactive systems*, Academic Pub., 1993.
- [HAR04] B. Hardung, T. Kolzow, A. Kruger, *Reuse of software embedded automotive systems*, EMSOFT'04, 2004.
- [KOP94] Hermann Kopetz et Gunter Grunsteidl, *Protocol for fault tolerant real-time system*. IEEE Computer 14-23, January 1994.
- [LAY73] C. L. Liu and J. W. Layland, *Scheduling algorithms for multiprogramming in a hard-real-time environment*. J. ACM, 20(1) :46_61, 1973.
- [LEA00] D. Lea, *Concurrent Programming in JavaTM. Second Edition. Design Principles and Patterns*. Addison-Wesley, nov. 2000.
- [MER01] Mercer Management Consulting and Hypovereinsbank. *Studie, Automobiletechnologie 2010*, Munchen, Aug. 2001.
- [MOK83] A.K. Mok., *Fundamental design problems for the hard real-time environments*. PhD thesis, MIT, 1983.
- [MOK96] A. K. Mok , D. Chen, *A multiframe model for real-time tasks*, Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96), p.22, December 04-06, 1996 .
- [MOREN] Philippe Morenton, *Introduction au développement pour PIC*, LT PE Martin Bourge.
- [MRTC05] Ivica Crnkovic, *Component-Based Development of Safety-Critical Vehicular Systems*, MRTC Report,2005.
- [OMG02] OMG Object Management Group. *The Common Object Request Broker: Core Specification*, version 3.0.2 edition, December 2002.
- [PB00] P. Puschner and A. Burns, Guest editorial: *A review of worst-case execution time analysis*. Real-Time Systems, 18, 115–128, 2000.
- [POSA1] Buschmann F., Meunier R., Rohnert H., Sommerlad P., *«Pattern-Oriented Software Architecture – A System of Pattern*, John Wiley & Sons, 1996.
- [POSIX] (ISO/IEC). 9945-1:1996 (ISO/IEC) [IEEE/ANSI Std 1003.1 1996 Edition] *Information Technology, Portable Operating System Interface (POSIX) - Part 1, System Application, Program Interface*. IEEE Standards Press, ISBN 1-55937-573-6, 1996.
- [PSA07] Joseph Beretta, *les Systèmes Électroniques Embarqués un enjeu majeur pour l'automobile*, PSA PEUGEOT CITROEN, 2007.
- [PUA05] I. Puaut. *Méthodes de calcul de wcet (worst case execution time) état de l'art*. Ecole d'été Temps Réel, 4 :165–175, Septembre 2005.
- [RITC88] Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*,

- Second Edition*, Prentice Hall, Inc., 1988, ISBN 0-13-110362-8.
- [ROQ03] Pascal ROQUES, Franck VALLÉE, *UML en action, De l'analyse des besoins à la conception en Java*, Eyrolles, 2003, ISBN : 2-212-11213-0.
- [RSM03] R. Santos Marques, N. Navet, F. Simonot-Lion, *Frame-Packing under real-time constraints*, 5th IFAC International Conference on Fieldbus Systems and their Applications (FeT 2003), 7-9 July 2003, Aveiro (Portugal).
- [RSM05] R. Santos Marques, F. Simonot-Lion, *Guidelines for the development of communication middleware for automotive applications*. In Proceedings of the 3rd Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER3), Paderborn, Germany, October 2005.
- [RSM06] R. Santos Marques, Thèse, *Méthodologie de développement des services de communication temps réel d'un intergiciel embarqué dans l'automobile*, Doctorat de l'Institut National Polytechnique de Lorraine, Septembre 2006.
- [RSM06-2] R. Santos Marques, F. Simonot-Lion, and N. Navet, *Optimal configuration of an in-vehicle embedded middleware*. In the 3rd Taiwanese-French Conference on Information Technology (TFIT'06), pages 425-444, Nancy, France, 2006.
- [RV93] Rodrigues L. et Ver'issimo P., *Message Slotting : Ensuring Replica Determinism in Preemptive Real-Time Systems*, Rapport technique, RT/58-93, Portugal, INESC, 1993.
- [SAND00] K. Sandstrom, C. Norstrom, and M. Ahlmark, *Frame Packing in Real-Time Communication*, Department of Computer Engineering, Malardalen Real-Time Research Centre, 2000 IEEE.
- [SCHH96] Douglas C. Schmidt and Charles D. Cranor, *Half-Sync/Half-Async An Architectural Pattern for Efficient and Well-structured Concurrent I/O*, Pattern Languages of Program Design 2, ISBN 0-201-89527-7, Addison-Wesley, 1996.
- [SCHM00] Douglas C. Schmidt, Chris Cleeland, *Applying a Pattern Language to Develop Extensible ORB Middleware*, Design Patterns in Communications, (Linda Rising, ed.), Cambridge University Press, 2000.
- [SCHM98] D. Schmidt, D. Levine, and S. Mungee., *The design of the TAO real-time Object Request Broker*. Computer Communications, 21(4), April 1998.
- [STA88] J.A. Stankovic. *Misconception about real-time computing*. In IEEE Computer Magazine, volume 10, pages 0–19. 21, 1988.
- [STE01] Stefan Polendna, Wolfgang Ettlmaye, Markus Novak, *Communications bus for automotive architecture*, 2001.
- [STIL06] M. Stilkerich, C. Wawersich, A. Gal, W. Schroder-PreikschatM. Franz, *OSEK/VDX API for Java*, PLOS 2006, Oct. 22, 2006, San Jose, California, United States.
- [SZYP98] C. Szyperski, C. Pfister, *WCOP'96 Workshop Report*, Workshop Reader ECOOP'96, Juin 1996.
- [TAKA97] H. Takada K. Sakamura, *Schedulability of generalized multiframe task sets*

under static priority assignment, (RTCSA '97), 1997.

- [TBW95] K. Tindel, A. Burns, J. Wellings, *Analysis of real-time communications*, J. of Real-Time Systems 9, 147-171, 1995.
- [TIN93] Tindell K., J. Clark, *Holistic Schedulability Analysis for Distributed Hard Real-time Systems*, Technical Report YCS197, Real-Time Systems Research Group, Univ. of York, 1993.
- [TIN95] K. W. Tindell, A. Burns and A. J. Wellings., *Calculating Controller Area Network (CAN) message response times*, Control Engineering Practice 3(8):1163-1169, 1995.
- [TTT99] Time Trigered Technology TTTech, Computer Technick AG Vienna, Austria. *Specification of the TTP/C Protocol*, July 1999.
- [VOL98] A. Rajnak, K. Tindell, and L. Casparsson, *Volcano communications concept*. Technical report, Volcano Communications Technologies AB, 1998.
- [WASAR] *AUTOSAR, AUTomotive Open System Architecture*, <http://www.autosar.org/>
- [WCAN] *Le protocole CAN*, <http://www.oberle.org/can-can.html>
- [WCANS] Complete CAN Solutions for Diverse Embedded Applications, www.microchip.com/stellent/groups/picmicro_sg/documents/devicedoc/en021962.pdf
- [WCBM] *Le mécanisme des callbacks*, A. Dutot, D. Olivier, <http://www-lih.univ-lehavre.fr/~dutot/enseignement/CORBA/Cours/5Callbacks.pdf>
- [WCC5X] *Compileur CC5X*, <http://www.bknd.com/cc5x/>
- [WCCM] *CORBA™/IIOP™ Specification*, http://www.omg.org/technology/documents/formal/corba_iiop.htm
- [WCMIC] *Compileur Microchip C18*, <http://www.aix-mrs.iufm.fr/formations/filieres/ge/data/PIC/PICC/MCC18%20v14.pdf>
- [WCPIC] *Le langage C adapté aux microcôntroleurs*, http://sti.tice.ac-orleans-tours.fr/spip/IMG/pdf/C_micro_c_1_.pdf
- [WECAN] *Le module ECAN*, http://sti.tice.ac-orleans-tours.fr/spip/IMG/pdf/Fiche_PIC18_bus_CAN.pdf
- [WEURK] *EAST-EEA Embedded Electronic Architecture*, <http://www.east-eea.net>
- [WGRCC] *GRCC de Grich RC Inc*, <http://members.aol.com/piccompile/prod01.htm>
- [WHITEC] *Compileur C Hi-TECH pour PIC18*, <http://www.htsoft.com/products/compilers/pic18ccompiler.php>
- [WIAR] *IAR Embedded Workbench de IAR systems*,

- <http://www.iar.com/Products/>
- [WINFO] *Infotainment embarqué,*
<http://www.epn-online.fr/page/34413/philips-transforme-l-infotainment-embarque.html>
- [WISOL] *Isolation,*
[http://en.wikipedia.org/wiki/Isolation_\(computer_science\)](http://en.wikipedia.org/wiki/Isolation_(computer_science))
- [WJINI] *Jini Network Technology,*
<http://www.sun.com/software/jini/>
- [WJORAM] *JORAM. ObjectWeb,*
[http://www.objectweb.org/joram/.](http://www.objectweb.org/joram/)
- [WMCC] *MPLAB[®] C18, PICmicro(R) 18Cxx C Compiler v3.02,*
<http://www.microchip.com/>
- [WMMCP] *MCP2551, High-Speed CAN Transceiver,*
<http://ww1.microchip.com/downloads/en/DeviceDoc/21667d.pdf>
- [WMPC] *MPC Code Development System V1.40 de Byte Craft's,*
<http://www.bytecraft.com/impc.html>
- [WMPL] *Design, MPLAB[®] IDE,*
<http://www.microchip.com/>
- [WMSMQ] *Microsoft Message Queuing (MSMQ),*
[http://www.microsoft.com/msmq/.](http://www.microsoft.com/msmq/)
- [WNET1] *L'électronique dans l'automobile,*
<http://pboursin.club.fr/pdgmpr.htm#mpx01>
- [WOIL] *OIL, OSEK Implementation Language, Message,*
<http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf>
- [WOOSEK] *openOSEK,*
<http://www.openosek.org/>
- [WORLO02] *M. Orlov. Efficient generation of set partitions,*
[http://www.cs.bgu.ac.il/~orlovm/papers/partitions.pdf.](http://www.cs.bgu.ac.il/~orlovm/papers/partitions.pdf)
- [WOSEK] *OSEK/VDX Portal,*
<http://www.osek-vdx.org/>
- [WOSEK2] *OSEK/VDX Specifications,*
<http://portal.osek-vdx.org/files/pdf/specs/>
- [WOSEKISR] *Interrupt processing, OSEK/VDX, Operating System,*
<http://portal.osek-vdx.org/files/pdf/specs/os223.pdf>
- [WOSI] *Le modèle OSI,*
[http://fr.wikipedia.org/wiki/Mod%C3%A8le_OSI.](http://fr.wikipedia.org/wiki/Mod%C3%A8le_OSI)
- [WPARS] *Parsic v3.39 de Swen Gosh,*
<http://www.parsic.de/>
- [WPIC18] *8-bit PIC[®] Microcontrollers,*

- <http://www.microchip.com/>
- [WPIC458] *PIC18FXX8 Data Sheet*,
<http://www1.microchip.com/downloads/en/DeviceDoc/41159e.pdf>
- [WPICC] *pico-C free de jokinen*,
<http://personal.eunet.fi/pp/jokinen/>
- [WPICOS18] *PICos18*,
<http://www.picos18.com/>
- [WPOLY] *The PolyORB schizophrenic middleware*,
<http://polyorb.objectweb.org/>
- [WPRAG] *Pragmatec*,
<http://www.pragmatec.net/>
- [WREN] *L'électronique embarquée : l'analyse d'un expert indépendant*,
http://www.leblogauto.com/2005/04/renaut_scoop_su.html
- [WRTAI] *The RealTime Application Interface for Linux*,
<http://www.rtai.org/>
- [WRTJAVA] <http://www.onjava.com/pub/a/onjava/2006/05/10/real-time-java-introduction.html>
- [WSCHM] *Patterns for Concurrent, Parallel, and Distributed Systems*, Douglas C Schmidt, <http://www.cs.wustl.edu/~schmidt/patterns-ace.html>
- [WSCM98] *Asynchronous Completion Token*,
<http://www.cs.wustl.edu/~schmidt/PDF/ACT.pdf>, Douglas C. Schmidt 1998
- [WSMQ] *sonicMQ*, Sonic software,
<http://www.sonicsoftware.com/>.
- [WTRAMP] *Trampoline*,
<http://trampoline.rts-software.org/>
- [WTTP] *TTTech's solutions*,
<http://www.ttchip.com>
- [WUML] *Unified Modeling Language (UML), version 2.1.1*,
<http://www.omg.org/technology/documents/formal/uml.htm>
- [WUML] *Unified Modeling Language : Superstructure*
<http://www.omg.org/docs/formal/07-02-03.pdf>
- [WVECT] *CANbedded*,
http://www.vector-france.com/vf_canbedded_fr,,8212.html
- [WVECTOR] *Vector Informatik*,
http://www.vector-informatik.com/vi_index_en.html
- [WWIKI] *Automobile, Sécurité des véhicules*,
<http://fr.wikipedia.org/wiki/Automobile>
- [WXDR] *XDR : External Data Representation Standard*,
<http://www.ietf.org/rfc/rfc4506.txt>

Résumé

Les applications temps réel sont des applications dont la correction ne dépend pas seulement du résultat produit, mais aussi de la date à laquelle il est produit. Par exemple, dans le domaine de l'automobile, lors d'un freinage, la force à appliquer sur les freins est évaluée par une application (ABS : *AntiBlockier System*, système évitant de bloquer les roues au freinage, permettant ainsi de conserver la direction du véhicule). Il existe donc des contraintes de délai de réaction entre la demande de l'utilisateur et la consigne appliquée sur les freins.

Dans ce domaine, des applications de plus en plus complexes sont aujourd'hui développées. Elles sont souvent distribuées sur plusieurs calculateurs, parfois hétérogènes, et doivent coopérer avec d'autres applications, elles aussi distribuées. Afin de maîtriser cette complexité, on développe classiquement une couche logicielle appelée intergiciel (middleware). Cette couche a pour but de masquer l'hétérogénéité des supports d'exécution et la distribution des applications. Elle propose aux applications un ensemble de services standards pour communiquer et interagir avec le système d'exploitation, tout en garantissant une qualité de service temporelle.

Notre objectif est de mettre en oeuvre une technique pour la génération d'intergiciels de communication embarqués pour l'automobile. Le cadre d'utilisation de nos travaux est la conception de systèmes embarqués dans les véhicules. Plus précisément, cette approche permet de concevoir un générateur d'un ensemble de tâches implémentant cette couche de communication pour un type d'application donnée. Pour cela, elle prend en compte l'ensemble des événements requis et offerts par l'application tout en assurant la qualité de service requise par cette dernière (propriétés temps réel), et en minimisant la surcharge due à l'exécution de l'intergiciel. Les travaux présentés visent la réalisation d'un générateur d'intergiciels optimisés, et abordent deux aspects : la conception générique des architectures d'implémentation du générateur et de l'intergiciel, et le déploiement de l'intergiciel généré sur une plateforme temps réel.

Constituant un *framework* technique, l'architecture d'implémentation est optimisée dans le sens où l'intergiciel est adapté à l'environnement d'exécution (le système d'exploitation OSEK/VDX OS et le réseau CAN), et minimise son utilisation des ressources disponibles. Elle apporte une réponse, d'une part, au niveau de la spécification d'une architecture logicielle (construite à l'aide de *design patterns*), et, d'autre part, à la manière dont cette architecture est déployée sur une plateforme concrète (sous la forme d'un ensemble de tâches). Quant au déploiement de l'intergiciel, il est réalisé sur une architecture matérielle de microcontrôleurs (PIC).

Mots-clés : *Temps Réel, Systèmes Embarqués, Automobile, Générateur, Intergiciel, OSEK/VDX, CAN, PIC, Génie du Logiciel.*