HAL
open science

Méthodologie de développement des services de communication temps-réel d'un intergiciel embarqué dans l'automobile

Ricardo Santos Marques

To cite this version:

Ricardo Santos Marques. Méthodologie de développement des services de communication temps-réel d'un intergiciel embarqué dans l'automobile. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Lorraine - INPL, 2006. Français. NNT : 2006INPL037N . tel-01752748v2

HAL Id: tel-01752748

https://theses.hal.science/tel-01752748v2

Submitted on 19 Nov 2006

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.

Institut National
Polytechnique de Lorraine

Département de formation doctorale en informatique

École doctorale IAEM Lorraine

Méthodologie de développement des services de communication temps-réel d'un intergiciel embarqué dans l'automobile

THÈSE

présentée et soutenue publiquement le 14 Septembre de 2006

pour l'obtention du

Doctorat de l'Institut National Polytechnique de Lorraine

(spécialité informatique)

par

Ricardo Santos Marques

Composition du jury

Président :	Jeanine Souquières	Professeur (Université Nancy 2)
Rapporteurs :	Dieter Zöbel	Professeur (Université de Koblenz-Landau)
	Laurence Duchien	Professeur (Université de Lille)
Examinateurs :	Françoise Simonot-Lion	Professeur (École de Mines de Nancy - INPL)
	Nicolas Navet	Chargé de Recherche (INRIA)
	Charles André	Professeur (Université de Nice Sophia Antipolis)
Invité :	Jörn Migge	Ingénieur (PSA Peugeot Citroën)

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Remerciements

Tout d'abord, je dois remercier Françoise Simonot-Lion et Nicolas Navet, mes encadrants et guides tout au long de ces années de thèse. Sans eux, je n'aurais sûrement pas pu arriver au bout de mon objectif. Leur expertise, expérience et amitié m'ont permis de surmonter les difficultés liées à la réalisation d'une telle tâche.

Merci aussi à ceux qui ont partagé ces années de travail avec moi à l'ENSEM, à Saint-Fiacre, et au LORIA. Merci à Marcel, Xavier, Olivier, Michel, Jean-Pierre, et Bruno pour leurs conseils de chercheurs et enseignants expérimentés. Merci aux thésards, ingénieurs et stagiaires qui m'ont accompagné et aidé : Raul, Mathieu, Philippe, Jian, Mohamed, Liping, Ning, Najet, Fabrice, Emmanuel, Paolo (et Yo), Jörn et Tiziana, Paolo Spagnoletti, Rafaelle, Orazio, Cédric, Gerardo, Anis, Yann, et Miguel. Merci à Frédéric (et Nicolas aussi) pour les moments sportifs qui m'ont permis de garder la forme tout au long de ces années. Merci à Josette Contal et Laurence Benini, mes guides dans le monde complexe qu'est l'administration.

Finalement, je tiens aussi à remercier :

- Laurence Duchien, Professeur à l'Université de Lille,
- Dieter Zöbel, Professeur à l'Université de Koblenz-Landau,
- Jeanine Souquières, Professeur à l'Université Nancy 2,
- Charles André, Professeur à l'Université de Nice Sophia Antipolis, et
- Jörn Migge, Ingénieur PSA Peugeot Citroën

pour m'avoir fait l'honneur de participer à mon jury de thèse.

A Aude,
à ma famille,
et à mes amis.

À Aude,
à minha família,
e aos meus amigos.

Contents

List of Figures **xi**

List of Tables **xiii**

Partie I Introduction

Chapter 1
Contexte et problématique **3**

1.1 Introduction 3
1.2 Caractéristiques principales des systèmes embarqués dans l'au-tomobile 4
1.2.1 Caractéristiques de l'application 4
1.2.2 Les systèmes d'exploitation 6
1.2.3 Les systèmes de communication 6
1.2.4 Conclusion 7
1.3 Périmètre des travaux 8
1.3.1 Introduction du sujet - cahier des charges 8
1.3.2 Hypothèses et choix considérés 9
1.3.3 Contributions 9
1.4 Conclusion 11

Chapter 2
État de l'art **13**

2.1 Introduction 13
2.2 Intergiciels 13
2.2.1 Intergiciels généraux, industriels et académiques 13
2.2.1.1 CORBA 14
2.2.1.2 PolyORB 15

2.2.1.3 ACE et TAO . . . 15
2.2.2 Intergiciels embarqués dans l'automobile . . . 16
2.2.2.1 Volcano . . . 16
2.2.2.2 OSEK/VDX Communication . . . 18
2.2.2.3 Eureka ITEA EAST-EEA . . . 19
2.2.2.4 AUTOSAR . . . 19
2.2.3 Bilan sur les intergiciels et les systèmes embarqués . . . 20
2.3 Méthodes de développement d'un intergiciel . . . 21
2.3.1 Contexte automobile . . . 21
2.3.1.1 Volcano . . . 21
2.3.1.2 Titus . . . 22
2.3.1.3 AUTOSAR . . . 22
2.3.2 Contexte académique: approche objets . . . 23
2.3.2.1 Design patterns . . . 23
2.3.2.2 Utilisation des design patterns dans PolyORB . . . 24
2.3.2.3 Utilisation des design patterns dans TAO . . . 24
2.3.3 Synthèse sur les techniques de développement d'intergiciels . . . 25
2.4 Identification de tâches . . . 26
2.4.1 Stratégies proposées par Douglass . . . 26
2.4.2 Stratégies proposées par Saksena *et al.* . . . 26
2.4.3 Conclusion . . . 27
2.5 Configuration de tâches et trames . . . 27
2.5.1 Configuration de tâches : priorité . . . 27
2.5.2 Frame packing . . . 28
2.5.2.1 Algorithme proposé par Tindell et Burns . . . 28
2.5.2.2 Algorithmes proposés par Norström *et al.* . . . 28
2.5.2.3 Algorithmes proposés par Pop *et al.* . . . 28
2.5.3 Conclusions sur la configuration de tâches et trames . . . 29
2.6 Conclusions . . . 29

Chapter 3
Présentation des contributions **31**

3.1 Généralités sur les contributions . . . 31
3.2 Résumé du chapitre 4 : introduction . . . 31
3.2.1 Services de communication fournis par l'intergiciel . . . 31
3.2.2 Principes méthodologiques pour le développement de l'intergiciel . . . 32
3.3 Résumé du chapitre 5 : implementation model of the middleware . . . 34
3.3.1 Fonctions de l'intergiciel : bibliothèque de fonctions, tâches et événements d'activation . . . 35
3.3.2 Identification de tâches de l'intergiciel . . . 35
3.4 Résumé du chapitre 6 : software model of the middleware . . . 36
3.4.1 Les *design patterns* pour l'intergiciel . . . 36

3.4.2 Le diagramme de classes et le modèle d'implémentation 37
3.5 Résumé du chapitre 7 : configuration of the frames 37
3.5.1 Contraintes de fraîcheur et échéance relative d'une trame 38
3.5.2 Algorithmes de frame packing 39
3.6 Résumé du chapitre 8 : configuration of the applicative and middleware tasks 39
3.7 Résumé du chapitre 9 : quality of service monitoring through communication services 40
3.7.1 Services de notification à intégrer 40
3.7.2 Conséquences sur le modèle d'implémentation de l'intergiciel 40
3.7.3 Conséquences sur le modèle de composants logiciels 41
3.8 Conclusion 41

Partie II Contributions

Chapter 4
Introduction **45**

4.1 Introduction 45
4.2 Communication services provided by the middleware 45
4.3 Methodology for the development of the middleware 47
4.3.1 Generic architecture 48
4.3.1.1 Identification of tasks 48
4.3.1.2 Software components identification 49
4.3.2 Middleware configuration 49
4.3.2.1 Frame packing 49
4.3.2.2 Configuration of the middleware tasks 49
4.3.2.3 Priority allocation 50
4.4 Conclusion 50

Chapter 5
Implementation model of the middleware **51**

5.1 Introduction 51
5.2 Functionalities of the middleware and their activation events 52
5.3 Middleware tasks identification strategies 53
5.3.1 "One task for each event" strategy 53
5.3.2 "One task for each type of event" strategy 53
5.3.3 "One task for each signal's purpose" strategy 53
5.3.4 Strategy selected to identify the middleware tasks 54

5.4 Middleware tasks model . . . 55
5.4.1 Task in charge of the construction and the transmission request of frames . . 55
5.4.1.1 Multiframe task model . . . 55
5.4.1.2 Generalized multiframe task model . . . 56
5.4.1.3 Other solution . . . 57
5.4.2 Task in charge of receiving and handling frames . . . 57
5.5 Conclusion . . . 58

Chapter 6
Software model of the middleware 59

6.1 Introduction . . . 59
6.2 Design patterns for the middleware . . . 60
6.2.1 "Active object" pattern . . . 60
6.2.2 "Adapter" pattern . . . 61
6.2.3 "Observer" pattern . . . 63
6.2.4 "Asynchronous completion token" pattern . . . 66
6.3 Class diagram and the implementation model of the middleware . . . 68
6.3.1 Class diagram: composition of design patterns . . . 68
6.3.2 Sequences of code of the middleware tasks . . . 70
6.3.2.1 Storage of signal values . . . 70
6.3.2.2 Retrieval of signal values . . . 70
6.3.2.3 Construction and transmission request of frames . . . 71
6.3.2.4 Reception and handling of frames . . . 71
6.3.3 Complementary code . . . 73
6.4 Conclusion . . . 74

Chapter 7
Configuration of the frames 75

7.1 Introduction . . . 75
7.1.1 Formulation of the problem . . . 76
7.1.2 Guaranteeing freshness constraints . . . 78
7.1.2.1 The maximum age of a signal . . . 78
7.1.2.2 The relative deadline of a frame . . . 81
7.1.2.3 Assumptions on the worst-case behaviour of the middleware tasks . . 81
7.1.3 Complexity of the problem . . . 83
7.1.4 Heuristics proposed to solve the problem . . . 83
7.2 The "Bandwidth-Best-Fit decreasing" heuristic . . . 84
7.2.1 Insertion: function *insert()* . . . 84
7.2.2 Decomposition: function *decompose()* . . . 85
7.2.3 Local optimization algorithm . . . 86
7.3 The "Semi-Exhaustive" heuristic . . . 86
7.4 Conclusion . . . 87

Chapter 8
Configuration of the applicative and middleware tasks **89**

8.1 Introduction . . . 89
8.2 Configuration of the middleware tasks . . . 89
8.2.1 Task constructing and requesting transmission of frames . . . 91
8.2.1.1 Implementation as a multiframe task . . . 91
8.2.1.2 Implementation as a generalized multiframe task . . . 92
8.2.2 Task receiving and handling frames . . . 94
8.2.3 Priority assignment . . . 94
8.3 A feasible priority for the applicative tasks . . . 94
8.4 Conclusion . . . 95

Chapter 9
Quality of service monitoring through notification services **97**

9.1 Notification services . . . 97
9.1.1 Transmission state . . . 98
9.1.2 Reception state . . . 98
9.2 Consequences on the implementation model of the middleware . . . 98
9.2.1 Service returning the transmission state . . . 98
9.2.1.1 Implementation based on the frames sender task . . . 99
9.2.1.2 Implementation based on a separate task . . . 99
9.2.2 Conclusion on the transmission state . . . 100
9.2.3 Service returning the reception state . . . 101
9.2.3.1 Implementation based on the interrupt service routine . . . 101
9.2.3.2 Implementation based on a separate task . . . 102
9.2.4 Conclusion on the reception state . . . 104
9.2.5 Conclusion . . . 104
9.3 Consequences on the software model of the middleware . . . 105
9.4 Conclusion . . . 105

Partie III Conclusion

Chapter 10
Conclusions **109**

10.1 Contributions . . . 109
10.2 Perspectives . . . 110

Appendixs

Appendix A
The *Mediator* design pattern **113**

Appendix B
Performances evaluation **117**

B.1 Bandwidth-Best-Fit-decreasing performances evaluation 118
B.1.1 Bandwidth consumption 119
B.1.2 Configurations feasibility 120
B.1.3 Local optimization algorithm 120
B.2 Semi-Exhaustive performances evaluation 120

Index **123**

Bibliography **125**

List of Figures

1.1 Illustration d'un système électrique/électronique embarqué dans un véhicule. 4
1.2 Illustration d'une échéance relative sur une instance de tâche. 5
1.3 Contrainte de fraîcheur appliquée à la production et consommation d'un signal. 5

2.1 Schéma illustrant la place occupée par un intergiciel au sein d'un système distribué. . . . 14
2.2 Composants et organisation des design patterns dans TAO. 25

3.1 Processus de configuration des trames et des tâches à partir des contraintes temporelles. . 33
3.2 Identification d'une architecture de composants logiciels et de tâches de l'intergiciel. . . . 34

4.1 Asynchronous execution of the application level and the middleware: production and transmission of signals. 46
4.2 Asynchronous execution of the application level and the middleware: reception and consumption of signals. 46
4.3 Algorithm representing the part of the methodology whose purpose is to configure the frames and the tasks according to timing constraints. 47
4.4 Principles for the identification of the software components architecture and the task architecture of the middleware. 48

5.1 Input and output data of the activity in charge of identifying a set of middleware tasks. . 51
5.2 Set of middleware tasks identified with the "one task for each event" strategy. 53
5.3 Set of middleware tasks identified with the "one task for each type of event" strategy. . . . 54
5.4 Set of middleware tasks identified with the "one task for each signal's purpose" strategy. . 54
5.5 Example of multiframe task. 56
5.6 Example of generalized multiframe task. 56
5.7 This figure illustrates the problem of setting an OSEK/VDX Operating System timing alarm that respects all activation periods of a generalized multiframe task. 57

6.1 Input and output data of the activity in charge of the design of the middleware software. 59
6.2 Class diagram illustrating the *Integrated Scheduler* variant of the *Active Object* pattern applied on the middleware context. 61
6.3 Sequence diagram showing the *Integrated Scheduler* variant of the *Active Object* pattern in the middleware context. 62
6.4 Class diagram providing an example of utilization of the *Adapter* pattern in the middleware context. 62
6.5 The structure of the *Observer* pattern applied in the middleware context. 64
6.6 Sequence diagram showing the *Observer* pattern applied in the middleware context. . . . 65
6.7 Utilization of the *Asynchronous Completion Token* pattern in the middleware context. . . 66
6.8 Chain of methods induced by the *Asynchronous Completion Token* pattern in the middleware context. 67
6.9 UML class diagram representing the structure of the middleware software. 68
6.10 *Pattern-Level* diagram [75] for the middleware software. 69

6.11 Sequence diagram illustrating the production and storage of two signal values produced by two different applicative tasks. 70
6.12 Sequence diagram illustrating the retrieval of two signal values, and their consumption by two different applicative tasks. 71
6.13 Sequence diagram detailing the construction and transmission request of frames. 72
6.14 Sequence diagram detailing the reception and handling of frames. 73

7.1 Input and output data of the activity in charge of building the frame packing configuration. 75
7.2 Scenario in which the exchange of signal $s_{i',j'}$ is constrained by a freshness constraint. . . 77
7.3 Scenario where value $m1$ reaches the maximum age of signal $s_{i,j}$, and the production and transmission periods of $s_{i,j}$ are the same. 79
7.4 Scenario where value $m1$ reaches the maximum age of signal $s_{i,j}$, and the production period of $s_{i,j}$ is greater than its transmission rate. 80
7.5 Description of the BBFd heuristic. 84
7.6 Search through the solutions space in the Semi-Exhaustive heuristic. 88

8.1 Input and output data of the activities responsible of the configuration of the middleware and applicative tasks on each ECU. 90
8.2 Execution of the multiframe task characterized by set $Q_i = \{(3, 20), (6, 60), (4, 20)\}$. . . . 92
8.3 Execution of the GMF task characterized by set $Q_i = \{(3, 15), (6, 60), (4, 20)\}$. 94

9.1 Execution of the transmission handler task proposed to implement a notification service associated to the transmission of produced signals. 100
9.2 Representation of D_f^+, the longest time interval between two consecutive arrivals of frame f on a CAN network. 102
9.3 Example of cooperation between the middleware tasks implementing a service that returns the reception state of a signal. 103
9.4 Modifications introduced in classes *Proxy_Scheduler* and *Signals* in order to reflect the existence of notification services. 105

A.1 Class diagrams representing the structure of the *Mediator* design pattern. 114
A.2 Sequence diagram depicting the behaviour of objects participating in the *Mediator* design pattern. 115

B.1 Two signals with production periods equal to 10 and 14. 118
B.2 Total network load for a nominal load ranging from 15 to 35%. 119
B.3 Total network load for a number of stations varying from 4 to 10 with 10 signals by station. 121

List of Tables

1.1 Tableau résumant les hypothèses et les considérations faites dans le cadre de cette thèse. . 10

9.1 Comparison between the proposed implementation schemes for the transmission state. . . 101

B.1 Number of feasible configurations over 100 tests for a nominal load varying from 15 to 35%. 120
B.2 Average network bandwidth consumption over 100 feasible solutions for a nominal load varying from 15 to 35%. 120
B.3 Percentage of cases where BBFd with LO decreased the bandwidth consumption level starting from 100 feasible solutions for a nominal load varying from 15 to 35%. 120

Part I

Introduction

Chapter 1

Contexte et problématique

1.1 Introduction

Depuis les années 70, l'utilisation des systèmes électroniques dans les véhicules a fortement augmenté. Cette croissance a permis aux constructeurs d'implémenter des fonctions automobiles de plus en plus complexes, en améliorant le confort, la conduite, et la sécurité des utilisateurs. Par exemple, des fonctions telles que l'*Anti-lock Breaking System* (ABS) ou le contrôle du moteur, ont eu comme objectif d'assister le conducteur dans sa tâche de conduite du véhicule, tout en garantissant sa sécurité.

A l'aide de cette technologie, les constructeurs automobiles ont commencé à implémenter des nouvelles fonctions, de plus en plus distribuées dans le véhicule. Néanmoins, dans ce nouveau contexte, l'élimination des traditionnelles connexions point-à-point entre équipements électroniques est apparue comme un besoin de plus en plus croissant (problèmes de maintenance, de poids, et de coût). A ce stade, la communication "multiplexée" (utilisation d'un médium de transmission commun) a apporté une solution à ce besoin.

Le contexte des travaux présentés dans ce document est la conception de systèmes électroniques distribués embarqués dans l'automobile. Ces systèmes sont donc composés de logiciels implantés sur un ensemble de calculateurs, appelés *Electronic Control Units* (ECUs), inter-connectés par des réseaux de communication (voir figure 1.1). Ces ECUs sont attachés à des capteurs et actionneurs qui ont comme fonction principale d'interagir avec des dispositifs matériels, comme par exemple, les freins ou le moteur.

L'apparition des nouvelles fonctions automobiles, s'accompagne d'une forte augmentation du logiciel embarqué dans les ECUs. Le processus de développement des systèmes embarqués repose, au moins dans l'industrie automobile européenne, sur une relation contractuelle "constructeurs/équipementiers". Dans ce schéma, les constructeurs intègrent, au sein du système embarqué, les solutions fournies par les équipementiers en réponse à un cahier des charges. Ces solutions se présentent comme des composants logiciels communicants et implémentés sur un ou plusieurs ECUs munis de leurs capteurs, actionneurs et des pilotes correspondants. Cette façon de procéder a comme conséquence importante l'apparition d'un problème d'hétérogénéité au niveau des ECUs. Parfois, pour résoudre ce problème au moment de l'intégration des composants, les constructeurs sont obligés d'ajouter eux mêmes des logiciels, conduisant à une augmentation de la mémoire et de la charge du processeur.

Comme tous les systèmes qui en cas de défaillance peuvent entraîner des conséquences dangereuses sur des vies humaines, les systèmes électroniques embarqués dans l'automobile sont soumis à des niveaux d'exigence élevés en termes de performance et de sûreté. Par exemple, afin de répondre efficacement à la demande du conducteur, et ainsi de préserver la sécurité au niveau du véhicule, le système "contrôle moteur" doit recevoir périodiquement la valeur captée de la position de la pédale d'accélérateur. De plus, cette valeur doit être transmise et traitée dans un intervalle borné à partir de sa production par le capteur.

Dans un contexte où la taille du code embarqué croît constamment, et où l'hétérogénéité des ECUs devient inévitable, il est difficile de garantir l'interopérabilité entre composants logiciels par les techniques traditionnelles de développement : interopérabilité d'interfaces et surtout interopérabilité temporelle (respect des contraintes de temps). Les constructeurs et les équipementiers ont aujourd'hui besoin de

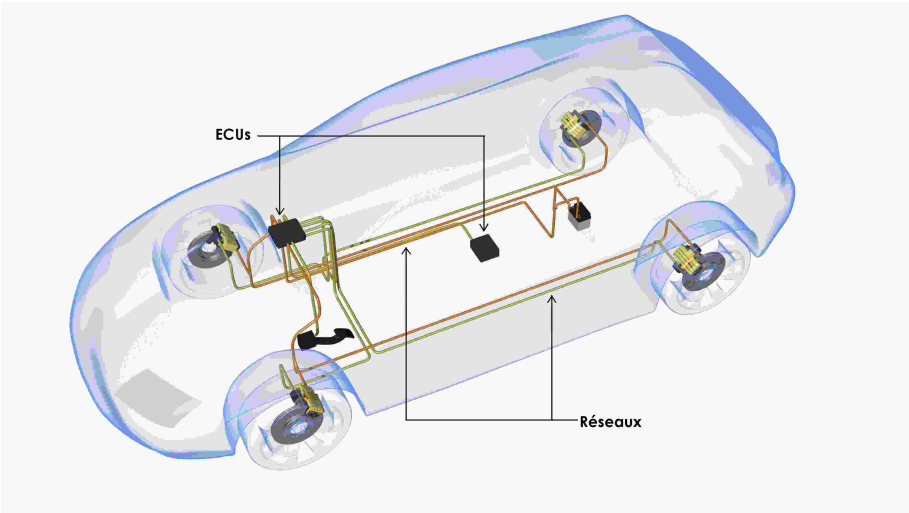


Figure 1.1: Illustration d'un système électrique/électronique embarqué dans un véhicule.

(Source de l'image : `http://www.usa.siemensvdo.com/`)

méthodologies de développement qui garantissent par construction le respect de toutes les propriétés. Comme nous le verrons dans le chapitre 2, une partie de la solution repose sur la notion de standardisation des architectures logicielles.

Pour mieux comprendre les objectifs qu'une telle méthodologie doit atteindre, regardons plus en détail comment un système électronique embarqué dans l'automobile est organisé.

1.2 Caractéristiques principales des systèmes embarqués dans l'automobile

Contribuer à la conception sûre de systèmes embarqués, c'est-à-dire à une méthode de conception garantissant les propriétés, en particulier temporelles, requises par le cahier des charges, passe par l'identification des constituants de ces systèmes et de certaines de leurs caractéristiques. Nous n'analysons pas ici les caractéristiques matérielles des micro-contrôleurs, des contrôleurs de communication ou des coupleurs d'Entrées/Sorties. Par contre, dans la suite de la section, nous identifions les composants logiciels et les caractéristiques propres à l'application, ainsi que les caractéristiques des composants logiciels gérant les ressources matérielles à savoir les systèmes d'exploitation et les protocoles de communication.

1.2.1 Caractéristiques de l'application

Sur chaque ECU, un ensemble de *tâches de niveau applicatif* exécute des algorithmes de contrôle (e.g. contrôle de l'injection de carburant, échantillonnage de la vitesse de rotation d'une roue, etc). Ces tâches représentent donc l'implémentation des fonctions automobiles sur chaque ECU. L'activation des tâches suit un modèle cyclique, périodique ou sporadique. Dans le premier cas, l'activation des tâches suit un taux (ou une *période*) stricte, alors que dans le deuxième cas, l'activation suit un taux variable dont, néanmoins, la valeur minimale (ou le plus petit intervalle d'inter-activation de deux instances de la tâche) est connue.

L'exécution des tâches applicatives est souvent soumise à une contrainte temporelle, appelée *échéance relative*. Cette contrainte indique l'intervalle de temps maximum tolérable entre la date d'activation de chacune des instances d'une tâche, et l'instant de fin d'exécution de l'instance. Cet intervalle est

illustré dans la figure 1.2. Elle montre une instance d'une tâche préemptée (son exécution est arrêtée et le processeur attribué à une autre tâche) en début et pendant son exécution, mais dont la date de fin d'exécution respecte bien la contrainte d'échéance relative imposée sur la tâche. Pour les fonctions critiques, le non respect de cette borne peut engendrer des conséquences graves au niveau de la sécurité du véhicule.

Pour pouvoir exécuter les algorithmes de contrôle, et en plus appliquer une consigne considérée correcte sur les actionneurs, les tâches applicatives ont besoin des données fournies par le capteurs, mais éventuellement aussi provenant de tâches distantes. Ce scénario se produit dans le cas où les fonctions automobiles sont implémentées par des tâches distribuées sur plusieurs ECUs. Par exemple, l'application qui démarre l'essuie-glace arrière dès que le conducteur enclenche la marche-en-arrière, a besoin d'être "réveillée" aussi rapidement que possible après que l'enclenchement de la marche arrière ait été détectée sur un autre ECU, et à ce moment, d'avoir promptement une information délivrée par le capteur de pluie situé à l'avant de la voiture.

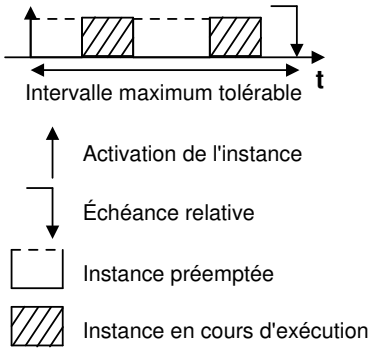


Figure 1.2: Illustration d'une échéance relative sur une instance de tâche.

Lorsqu'une fonction est distribuée, la communication entre les tâches distantes réalisant la fonction se fait alors par échange de données. Ces données appelées *signaux* (e.g. le nombre de rotations du moteur), sont produites et consommées par les tâches. La durée de vie d'un signal est limitée, ce qui veut dire que l'âge du signal, c'est-à-dire l'intervalle de temps entre la consommation distante d'un signal et l'activation de la tâche qui produit ce signal ne peut pas excéder une borne temporelle spécifiée. Cette borne sera appelée dans ce document contrainte de *fraîcheur*, et elle est expliquée graphiquement dans la figure 1.3.

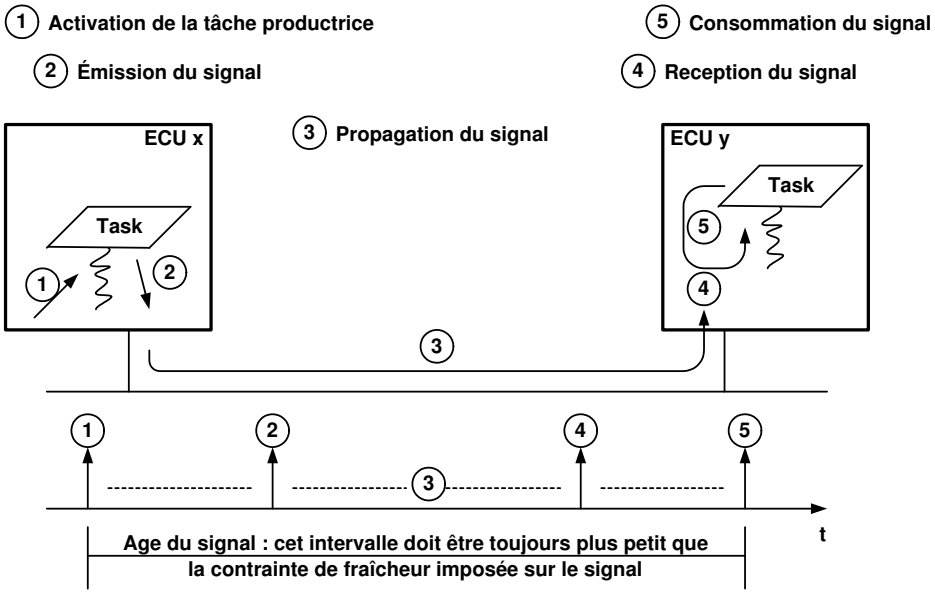


Figure 1.3: Contrainte de fraîcheur appliquée à la production et consommation d'un signal.

Enfin, les fonctions embarquées peuvent aussi avoir des contraintes de natures différentes. Une tâche qui vérifie si une fenêtre est ouverte ou non, ne relève pas des mêmes propriétés (vitesse de processeur, mémoire, communication, etc) qu'une tâche qui contrôle le moteur. L'industrie automobile divise traditionnellement le véhicule en *domaines*. En général, ceux-ci sont :

- le domaine moto-propulseur ou *powertrain*, qui englobe le contrôle du moteur et de la transmission,
- le châssis ou *chassis*, qui s'occupe du contrôle de la suspension, du guidage et du freinage,
- le domaine habitacle/carrosserie ou *body*, qui implémente les fonctions liées au confort du conducteur, et
- le domaine télématique/multimédia ou *telematic*, qui prend en charge la communication sans-fil, les systèmes de surveillance du véhicule, de navigation, de loisirs, etc.

Chaque domaine du véhicule implémente un ensemble différent de fonctionnalités ayant des besoins variés, et soumis à des contraintes temporelles dont le respect peut être plus ou moins strict. Ainsi, la plupart des fonctions des domaines moto-propulseur et châssis exigent des garanties déterministes de propriétés temporelles, tandis que celles du domaine habitable relèvent du temps-réel mou et celles du domaine télématique nécessitent des garanties de qualité de service (en moyenne, probabiliste, etc). Cette différence d'éxigences a donné lieu à la spécification de services support (protocoles, systèmes d'exploitation) différents. Ceux-ci sont rapidement présentés dans la suite.

1.2.2 Les systèmes d'exploitation

La plupart des systèmes d'exploitation utilisés au sein de l'industrie automobile sont des produits propriétaires (constructeurs ou équipementiers). Néanmoins, le consortium OSEK/VDX propose un standard qui spécifie une architecture ouverte pour les ECUs, dans lequel nous pouvons trouver un volet dédié aux systèmes d'exploitation. Dans ce volet, deux systèmes ont été introduits : le premier, *OSEK/VDX Operating System* [1] (OSEK/VDX OS), qui est focalisé surtout pour les systèmes événementiels, et le deuxième, *OSEK/VDX Time-Triggered Operating System* [2] (OSEKtime), dont le fonctionnement est basé sur le temps.

Il n'y a pas à notre connaissance actuellement, d'implémentations compatibles avec OSEKtime. Par contre, des produits conformes OSEK/VDX OS existent et sont actuellement utilisés. Les principaux services spécifiés par le standard sont la gestion de tâches, le traitement d'interruptions, et le mécanisme de gestion d'événements/alarmes. La gestion de tâches repose sur une politique d'ordonnancement à *priorités* fixes, où les tâches peuvent être préemptées ou non. Les priorités sont affectées aux tâches de l'application de façon à garantir la faisabilité de celles-ci (respect des échéances relatives). Le traitement d'interruptions permet la définition de routines de service, dans lesquelles le concepteur du système peut introduire le code qui se chargera du traitement des interruptions générées par les différents dispositifs attachés à un ECU. Finalement, le mécanisme d'événements/alarmes a comme objectif de permettre la synchronisation entre tâches. Pour cela, l'occurrence d'événements est déclenchée par une ou plusieurs tâches de façon à "réveiller", le cas échéant, une autre tâche, qui passera de l'état *waiting* (en attente d'un événement) à l'état *ready* (prête à s'exécuter mais en attente d'obtenir le processeur). Les alarmes sont des événements qui ont une occurrence régulière, et qui ne sont pas déclenchés par des tâches. Elles sont implémentées sur OSEK/VDX OS par le moyen de compteurs, à partir desquels des alarmes peuvent être établies. Une alarme s'active quand le compteur associé atteint une valeur pré-définie.

La spécification OSEK/VDX OS définit des classes de conformité. Chaque classe équivaut à une *version* du système d'exploitation offrant des niveaux spécifiques d'utilisation de ressources (processeur, mémoire). Selon les services et mécanismes nécessaires pour pouvoir exécuter un ensemble de tâches applicatives donné, le concepteur choisit la classe de conformité qui s'adapte au mieux aux besoins et qui utilise le moins de ressources possible.

1.2.3 Les systèmes de communication

Comme nous avons remarqué précédemment, les tâches applicatives peuvent avoir besoin de signaux produits par d'autres tâches, déployées dans des ECUs distants. Pour la transmission de ces signaux, les tâches utilisent une architecture de communication. Dans le contexte automobile plusieurs types de réseau existent : réseaux à priorités, réseaux maître-esclaves, réseaux basés sur le temps, et réseaux multimédia.

Dans les réseaux à priorités l'accès au bus de communication est contrôlé selon la priorité des trames. Les exemples de ce type de réseau sont CAN [3, 4], VAN [5], et J1850 [6]. Les réseaux maître-esclaves sont normalement de bas coût, et fonctionnent suivant le principe d'un noeud maître qui, à partir d'une configuration temporelle précise, autorise consécutivement les noeuds esclaves à transmettre leurs trames. LIN [7] et TTP/A [8] sont des exemples de réseaux maître-esclaves. Le fonctionnement des réseaux basés sur le temps, comme le nom l'indique, est guidé par le progression du temps : chaque noeud a l'accès au bus pendant des intervalles de temps réguliers. Trois exemples de ce type de réseau peuvent être donnés : TTP/C [9], FlexRay [10], et TTCAN [11]. Parmi ces exemples, seul TTCAN a été déployé dans des véhicules. Néanmoins, FlexRay semble être le prochain standard *de facto* pour l'automobile.

Un point commun parmi ces trois types de réseau est le fait que, sous certaines hypothèses sur les occurrences d'erreurs de transmission, ils assurent un temps de réponse (intervalle de temps entre la requête de transmission et la réception d'une trame réseau par tous les destinataires) dont il est possible d'évaluer une borne supérieure pour toutes les trames. Cette propriété rend ces types de réseau utilisables pour supporter la distribution de fonctions à contraintes temps réel. Les réseaux multimédia ne sont eux pas soumis à des contraintes de temps strictes; ils gèrent de grandes quantités de données nécessaires aux applications multimédia embarquées dans l'automobile. Deux exemples de réseaux multimédia automobile existent : MOST [12] et IDB-1394 (`http://www.idbforum.org` et `http://www.1394ta.org`).

Pour pouvoir transmettre leurs signaux, les tâches applicatives utilisent des trames, qui sont les structures offertes par les réseaux pour le transport de données. Plusieurs signaux émis par un même ECU peuvent être regroupés dans une seule trame. L'ensemble de trames émises sur les réseaux embarqués forme la messagerie automobile. Chaque trame qui compose cette messagerie est décrite à l'aide d'un groupe de caractéristiques, qui dépendent du type de réseau utilisé. Les principales caractéristiques sont la taille de la trame, la période de transmission ou l'intervalle attribué à la trame parmi l'ensemble d'intervalles réguliers (dans le cas d'un réseau basé sur le temps), l'échéance relative, et la priorité (selon le réseau). Les valeurs de ces caractéristiques doivent toutefois garantir la faisabilité de la messagerie (respect de l'échéance relative de chaque trame) et, évidemment, le respect de la fraîcheur des signaux transportés. Une contrainte forte de l'industrie est que les caractéristiques d'une messagerie doivent aussi minimiser la consommation de bande passante, ce qui permet l'utilisation de processeurs de moindre puissance, et des réseaux ayant des débits inférieurs et donc de plus bas coût.

La raison majeure pour laquelle il existe tant de types de réseaux embarqués dans un véhicule, est la différence entre les besoins exprimés par chaque domaine dans l'automobile (voir section 1.2.1). Un réseau utilisé dans le domaine *body*, pour s'occuper par exemple du système des portes (ouverture/fermeture des fenêtres et des portes), peut être du type maître-esclave et de bas coût, comme c'est le cas de LIN. Le domaine *powertrain* a besoin d'une grande puissance de calcul (contrôle moteur) et de communiquer de façon intense avec d'autres domaines (*chassis* à cause de ABS et de l'ESP - *Electronic Stability Program*, et le *body* pour la climatisation et le tableau de bord). Dans ce cas, un réseau comme CAN (modèle haut-débit - 500Kbits/s) est utilisé. A l'hétérogénéité déjà vue des ECUs (voir section 1.2.1), s'ajoute donc l'hétérogénéité des réseaux de communication cohabitant dans une automobile. Ceci accroît encore la difficulté d'implémentation des tâches applicatives communicantes sous garantie des propriétés temps réel.

1.2.4 Conclusion

L'hétérogénéité des plates-formes, des protocoles, et des systèmes, et la nécessité de concevoir des systèmes sûrs par garantie de l'interopérabilité temporelle des composants logiciels applicatifs, forment le contexte de fond de notre étude. Une solution à la première partie du problème est classiquement fournie par la spécification d'un intergiciel [13]. Résoudre la deuxième partie du problème demande, entre autres, de définir une méthode de développement/déploiement des composants logiciels de cet intergiciel qui préserve les propriétés temporelles requises au niveau applicatif.

Enfin, des impératifs économiques amènent à réduire de plus en plus les coûts et la durée de production des systèmes. Il devient important d'étudier comment prendre en compte cette exigence par le développement d'outils automatisant tout ou partie de la production d'un tel intergiciel.

1.3 Périmètre des travaux

Les caractéristiques des systèmes embarqués dans l'automobile énoncées précédemment forment le contexte de cette thèse. Le sujet traité est décrit par la suite. Plus particulièrement, nous proposons une méthode de développement d'un intergiciel embarqué dans l'automobile qui s'appuie sur les caractéristiques énoncées précédemment.

1.3.1 Introduction du sujet - cahier des charges

Le sujet traité dans cette thèse concerne donc le développement d'un intergiciel à embarquer dans l'automobile. Dans ce contexte, un intergiciel doit fournir des services de communication, des modules de diagnostique, des fonctions de calibration, ou des primitives d'abstraction des dispositifs attachés à l'ECU. Concrètement, dans cette thèse, nous nous sommes seulement intéressés aux services liés à la communication. En effet, ces services forment la fonction primordiale à offrir au niveau applicatif, en raison de son impact sur le bon fonctionnement des fonctions automobiles réparties, et en particulier sur le respect des contraintes temporelles imposées aux échanges entre ces fonctions.

Les objectifs de l'intergiciel traité dans ce document sont :

- Offrir un ensemble de services de communication sous la forme d'une interface standard. Cet interface doit permettre aux tâches applicatives d'effectuer des échanges de signaux entre elles. Certains de ces échanges impliquent une communication distante ; les services ont en charge l'insertion de 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 assurées).

- Cacher la distribution des tâches applicatives. Les services de communication fournis doivent être indépendants de la localisation des intervenants dans les échanges.

- Masquer l'hétérogénéité de la plate-forme de communication. En plus de la localisation des tâches, les services de communication doivent masquer les réseaux utilisés, de façon à diminuer le temps de développement des logiciels de niveau applicatif, et à favoriser leur qualité et leur ré-utilisation.

- Garantir une qualité de service temporelle, où le respect de toutes les contraintes temporelles est assuré. Cet objectif touche deux points importants. D'une part, l'intergiciel sera un logiciel de plus dans chaque ECU. Il est donc impératif de certifier que son interférence sur l'exécution des tâches applicatives n'empêchera pas celles-ci de respecter leurs propres échéances relatives. D'autre part, les instants dans le temps où l'intergiciel construit et demande la transmission des trames, doivent être précédemment déterminés de façon à garantir que la fraîcheur des signaux transmis sera toujours respectée.

- Finalement, présenter des propriétés qui favorisent son évolution. En effet, un intergiciel automobile sera une couche logicielle partagée entre constructeurs et équipementiers, et donc, soumise aux volontés de ces acteurs industriels. L'intergiciel doit être alors facile à comprendre, modifier, faire évoluer, et ré-utiliser, de façon à rendre plus rapide le processus d'intégration de logiciels embarqués dans l'automobile.

Pour résoudre ces problèmes et atteindre ces objectifs, cette thèse propose une méthodologie de conception des services de communication. Ceci demande de résoudre deux problèmes :

- 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)

1.3.2 Hypothèses et choix considérés

Comme annoncé, la méthodologie proposée dans ce document concerne le développement d'un intergiciel automobile centré sur les services de communication. Pour cadrer notre démarche, nous décrivons ci-dessous les hypothèses faites sur l'exécution de l'intergiciel, ainsi que les choix faits au niveau de son fonctionnement.

Nous commençons par décrire les hypothèses faites sur les tâches applicatives. Nous supposons que celles-ci sont périodiques, avec départ simultané. Néanmoins, les tâches sporadiques peuvent être inclues, leur période d'activation étant égale au plus petit intervalle de temps entre deux activations consécutives. Chaque instance d'une tâche applicative en cours d'exécution peut consommer plusieurs signaux, mais n'en produit qu'un seul. De plus, nous supposons que les consommations et la production peuvent avoir lieu à tout instant pendant l'exécution.

En ce qui concerne l'environnement d'exécution de l'intergiciel, nous faisons l'hypothèse que le système d'exploitation embarqué dans chaque ECU est conforme au standard OSEK/VDX OS (voir section 1.2.2). Notre intergiciel doit donc pouvoir s'exécuter sur ce système. De plus, nous considérons que la politique d'ordonnancement des tâches repose sur des priorités fixes avec préemption. Nous faisons aussi l'hypothèse que le réseau de communication utilisé est CAN, et que celui-ci est le seul réseau pour lequel l'intergiciel sera configuré (nous ne traitons pas le cas d'un intergiciel pour des ECUs passerelle). Néanmoins, la méthodologie de développement doit prévoir l'utilisation d'autres réseaux, toujours en mode isolé.

Concernant l'intergiciel, nous considérons que la méthodologie est centrée seulement sur la communication distante. Ceci veut dire que l'implémentation et la configuration de l'intergiciel ne prend pas en compte l'échange de données entre tâches applicatives qui s'exécutent dans un même ECU. Pour les échanges nous considérons seulement des signaux dont le stockage en réception est fait selon le mode à écrasement : quand une nouvelle valeur d'un signal arrive, l'ancienne est détruite pour donner place à la plus récente (nous ne traitons pas le mode *buffers* de stockage).

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.

Finalement, au niveau des services de communication, nous supposons que la taille des signaux est inférieure à la taille maximale des données utiles imposée par le protocole de communication. Concrètement, l'intergiciel n'effectuera pas de la segmentation des signaux applicatifs. Enfin, tous les signaux reçus seront mis à disposition des tâches consommatrices, ce qui implique que notre intergiciel ne fournit pas de mécanismes de filtrage de signaux.

L'ensemble de ces considérations est résumé dans le tableau 1.1.

1.3.3 Contributions

La méthodologie proposée dans ce document pour le développement d'un intergiciel automobile optimal traite les deux volets introduits précédemment : l'implémentation et la configuration.

En ce qui concerne l'implémentation de l'intergiciel, nous avons travaillé sur deux aspects. Le premier vise la définition d'un ensemble de tâches capables de représenter l'intergiciel sur chaque ECU. Cet ensemble est optimal dans le sens où il est adapté aux propriétés du système d'exploitation utilisé (OSEK/VDX OS), et cible une moindre utilisation de ressources. Dans le deuxième aspect nous définissons une architecture de composants logiciels à l'aide de modèles objets ré-utilisables (*design patterns*). Cette architecture spécifie les séquences de code qui implémentent les services de communication, et qui seront exécutées par chaque tâche intergicielle. Ces deux aspects de l'implémentation ensemble doivent contribuer au déploiement de l'intergiciel indépendamment des plates-formes cibles.

Le deuxième volet, la configuration de l'intergiciel, essaye d'attribuer des paramètres aux tâches applicatives et intergicielles, et aux trames, de manière à ce que les contraintes temporelles imposées sur

Choix stratégiques dans la conception de l'intergiciel : • mémorisation des signaux reçus dans un *buffer* à écrasement • asynchronisme entre les tâches applicatives et les tâches de l'intergiciel
Hypothèses sur la plate-forme support : • chaque ECU est muni d'OSEK/VDX OS • l'ordonnancement est FPP • les ECUs sont connectés sur un seul réseau
Hypothèses sur les composants logiciels applicatifs : • tâches périodiques à départ simultané • une tâche peut consommer plusieurs signaux à tout l'instant d'exécution de ses instances • une tâche ne peut produire qu'un seul signal; ce signal peut être produit à tout instant d'exécution de ses instances • la longueur d'un signal est toujours inférieure à la taille maximale des données utiles d'une trame telle qu'imposée par le protocole
Hypothèses supplémentaires pour illustrer les principes de la méthodologie proposée : • le réseau de communication est CAN

Table 1.1: Tableau résumant les hypothèses et les considérations faites dans le cadre de cette thèse.

les tâches et les signaux soient respectées. Dans ce but, nous avons commencé par la configuration des trames, ce que nous a permis ensuite d'attribuer des paramètres aux tâches intergicielles selon les besoins de transmission et réception de trames. En plus de devoir atteindre l'optimalité temporelle (respect des propriétés sur les tâches et les signaux), ce volet a été soumis à l'objectif de déterminer les paramètres des trames de façon à minimiser la consommation de bande passante.

1.4 Conclusion

Nous avons présenté le contexte actuel relatif au développement de logiciels embarqués dans l'automobile, et, en particulier, nous avons vu quels sont les problèmes qui justifient la définition d'une méthodologie pour le développement d'un intergiciel. En adoptant cette problématique comme étant le sujet de cette thèse, nous avons listé les objectifs de l'intergiciel proposé, en remarquant que celui-ci sera centré sur les services liés à la communication. Nous avons aussi détaillé les objectifs de la méthodologie proposée pour le développement de l'intergiciel. Ensuite, l'ensemble des hypothèses faites dans le but de cadrer notre démarche a été présenté. Nous constatons principalement qu'une implémentation asynchrone de l'intergiciel a été assumée.

Le reste de ce document est organisé de la façon suivante. Cette partie I est complétée par deux chapitres. Le chapitre 2 parcourt les travaux existants sur le problème que nous voulons traiter dans cette thèse. En particulier, il donne un aperçu des résultats académiques et industriels concernant les intergiciels et les méthodes pour leur développement. Le chapitre 3 survole les contributions apportées par cette thèse, en résumant chacun des chapitres contenus dans la partie II.

La partie II présente la contribution de la thèse. Elle est composée de six chapitres. Le premier, chapitre 4, introduit les services offerts par l'intergiciel traité dans ce travail, et donne les grands principes de la méthodologie proposée pour son développement. Le chapitre 5 détaille le modèle d'implémentation de l'intergiciel, qui est le moyen choisi pour son déploiement sur un ECU. Le chapitre 6 traite le modèle de composants logiciels de l'intergiciel, qui spécifie le code à exécuter par le modèle d'implémentation. Le chapitre 7 apporte des algorithmes de configuration de trames de communication. Le chapitre 8 montre comment, à partir du résultat de la configuration de trames, il est possible de caractériser le modèle d'implémentation de l'intergiciel au niveau des paramètres temporels. Le dernier chapitre, chapitre 9, étudie la possibilité d'intégration de nouveaux services à l'intergiciel, en particulier, des services de notification aux tâches applicatives. Finalement, la partie III présente les conclusions de nos contributions, et les perspectives ouvertes par ce travail.

Chapter 2

État de l'art

2.1 Introduction

L'objectif de ce chapitre est de présenter des travaux portant sur des domaines connexes à celui traité dans cette thèse, et d'évaluer certaines solutions proposées. Nous commençons par détailler quelques exemples d'intergiciels, parmi lesquels ceux du domaine automobile ; puis, nous regardons les méthodes proposées pour le développement d'intergiciels. Nous nous intéressons ensuite plus en détail à une de ces méthodes, qui repose sur une approche de modèles objet ré-utilisables. Cette approche, bien qu'adaptée au développement de logiciels en général, est insuffisante pour être directement applicable au domaine automobile et à la validation dans un contexte temps-réel. Enfin, nous présentons des travaux ponctuels, qui peuvent contribuer à résoudre des sous-problèmes bien identifiés dans la problématique générale de cette thèse.

2.2 Intergiciels

Un *intergiciel* est une classe de technologies logicielles développée pour gérer la complexité et l'hétérogénéité des systèmes distribués [14, 15, 16]. Un intergiciel est implémenté comme une couche logicielle située entre le niveau applicatif, et les systèmes d'exploitation et de communication (figure 2.1). Il offre une abstraction de programmation qui masque la distribution grâce à une interface standard (API - *Application Programming Interface*).

L'API réduit considérablement le travail des programmeurs des applications, étant donné qu'elle cache les détails inhérents à la programmation distribuée. En particulier, elle offre une transparence de la distribution en plusieurs dimensions : localisation, concurrence, réplication, fautes et mobilité. Les intergiciels masquent certains types d'hétérogénéité : l'hétérogénéité des réseaux et des composants physiques, des systèmes d'exploitation ou des langages de programmation.

Par la suite, nous présentons quelques exemples d'intergiciels utilisés dans des contextes généraux, industriel et académique, mais surtout dans un contexte bien plus précis, comme celui de l'industrie automobile.

2.2.1 Intergiciels généraux, industriels et académiques

Dans les sections suivantes nous détaillons trois intergiciels utilisés dans des domaines comme, par exemple, les télécommunications et le multimédia : CORBA (et Real-Time CORBA), PolyORB, et ACE/TAO. Nous n'incluons pas .NET [17] dans ce panorama car, malgré sa grande diffusion actuelle, il n'intègre aucune caractéristique pour le temps-réel. Les seules propriétés prises en compte sont celles liées à la sécurité et à la confidentialité.

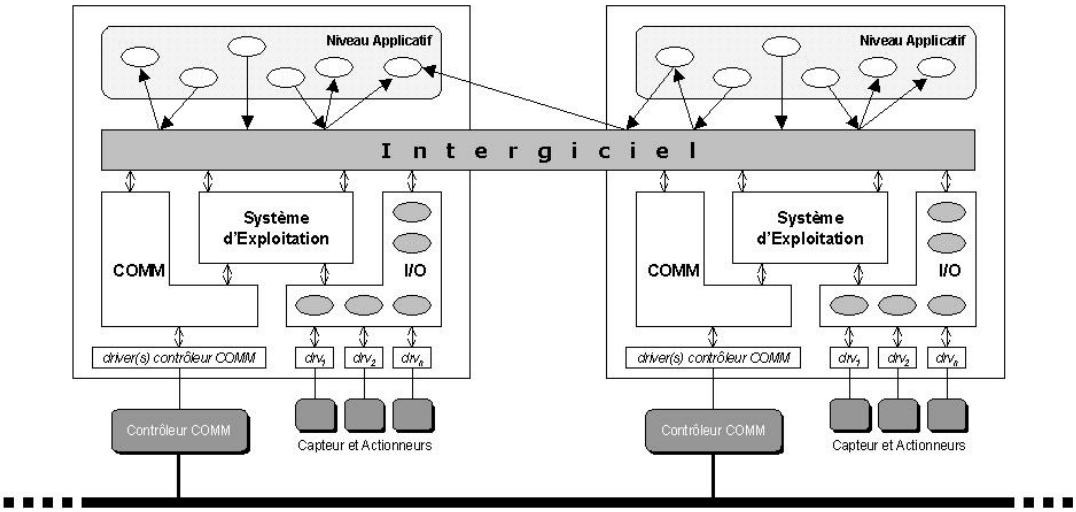


Figure 2.1: Schéma illustrant la place occupée par un intergiciel au sein d'un système distribué.

2.2.1.1 CORBA

Un des exemples d'intergiciel le plus connu est CORBA (*Common Object Request Broker Architecture* [18]), développé par l'*Object Management Group*. CORBA est un standard pour le traitement d'ORBs (*Object Request Brokers*) distribués. Un ORB est un composant de l'intergiciel CORBA qui permet aux différents objets d'un système de pouvoir communiquer de façon distante. Il fournit les mécanismes par lesquels des objets font des requêtes et reçoivent des réponses, et ce de manière transparente. Il fournit également l'interopérabilité entre des applications sur différentes machines dans des environnements distribués hétérogènes. L'interopérabilité entre objets est obtenue grâce à deux facteurs clés de la spécification : l'IDL (*Interface Definition Language*) et les protocoles GIOP et IIOP. Ils permettent à des applications CORBA construites par des concepteurs différents, et en utilisant des systèmes d'exploitation, réseaux et langages de programmation différents, d'être interopérables entre elles. Les principes de base de CORBA sont de permettre une séparation entre l'interface et l'implémentation des logiciels distribués, et de fournir une abstraction de la localisation et de l'accès aux objets distribués.

Parmi les extensions CORBA existantes, celles qui sont plus proches du contexte de cette thèse sont deux extensions *Real-Time* CORBA, qui reposent, pour la première, sur l'ordonnancement à priorités statiques [19] et, pour la deuxième, sur l'ordonnancement à priorités dynamiques [20]. Globalement, *Real-Time* CORBA est un ensemble optionnel d'extensions à CORBA de base, à utiliser pour enrichir les ORBs qui seront utilisés comme composants des systèmes temps-réel. D'autres composants et mécanismes ont été spécifiés, comme l'ordonnancement et la gestion de *threads*, le schéma de priorités globales et leur traduction en priorités locales (propres à un système d'exploitation donné), et les modèles de priorités des objets serveurs. Le but est de créer une architecture logicielle distribuée capable d'être analysée temporellement de bout-en-bout, où les priorités sont respectées et la durée des invocations de service et des inversions de priorité sont bornées.

CORBA et les besoins de l'automobile

Real-Time CORBA (RTCORBA) est un intergiciel adapté aux environnements distribués où l'utilisation de ressources ne doit pas être limitée. Or, ceci n'est pas le cas dans le contexte automobile, où il faut minimiser l'utilisation des ressources et ainsi minimiser le coût des micro-contrôleurs et l'énergie qu'ils nécessitent.

RTCORBA présente un ensemble de services qui dépasse les besoins d'un middleware automobile. Par exemple, il implémente en particulier un modèle de référence client/serveur, alors que la spécification des échanges entre composants logiciels dans l'automobile se fait selon un modèle producteur/consommateurs.

Passer du deuxième modèle au premier imposerait une surcharge à l'exécution. D'autre part, RTCORBA oblige le concepteur à ajouter des services si un mécanisme additionnel est souhaité, comme par exemple, l'*EventService* qui doit être ajouté si l'application a besoin d'opérations asynchrones.

Finalement, le dynamisme au niveau des services et mécanismes offert par RTCORBA (construction dynamique de requêtes, découverte en-ligne de serveurs et de services) implique des délais indéterministes qui ne sont pas acceptables dans le domaine des systèmes embarqués dans l'automobile.

2.2.1.2 PolyORB

PolyORB [21] est un intergiciel modulaire, dont le but est de résoudre le problème de l'interopérabilité entre différents modèles de distribution objet (e.g. CORBA, RMI, MPI, Ada 95 DSA). Sachant que chaque modèle est aussi nommé personnalité, l'objectif est de pouvoir supporter plusieurs personnalités simultanément dans une seule instance de l'intergiciel. Pour cela, il a été observé que tout intergiciel d'objets distribués rend disponible un ensemble commun de services. Ces services sont l'adressage d'objets, le transport (communication distante), le *marshaling* (conversion entre les représentations machine et réseau des données), le protocole, l'activation d'objets, et l'expédition (attribution de requêtes à des tâches).

PolyORB définit une architecture en couches, qui découple les personnalités de niveau applicatif de celles du niveau protocole au moyen d'une couche intergiciel générique. Cette couche générique implémente l'ensemble de services listés ci-dessus, et selon les auteurs, respecte les propriétés de généricité, *patterns* (modèles objets ré-utilisables), configuration, et interopérabilité. Les personnalités de niveau applicatif sont les interfaces présentées par l'intergiciel aux composants de l'application : objets client et serveur. Les personnalités de niveau protocole sont responsables de la transmission des requêtes de l'application précédemment transformées par la couche générique. Grâce à cette dernière, les personnalités de l'application et de protocole peuvent alors être remplacées indépendamment les unes des autres. En plus, l'architecture proposée évite la création de passerelles d'interopérabilité spécifiques à deux personnalités de l'application et de protocole données.

La conception de l'implémentation de PolyORB est faite en utilisant des diagrammes de classe UML [22]. L'implémentation suit un modèle d'activation pour PolyORB basé sur événements, c'est à dire que l'intergiciel s'exécute seulement quand nécessaire (réception de données d'origine distante ou invocation locale de méthodes distantes). Les auteurs ont plus tard travaillé sur la vérification de l'intergiciel à l'aide de réseaux de Petri [23]. Pour cela, le *pattern* (modèle objet ré-utilisable) de niveau architectural *Broker* [24] a été redéfini et réduit au niveau conception (suppression des fonctionnalités déjà fournies par PolyORB). Le modèle résultant a été ajouté à PolyORB sous la forme d'une personnalité, et le tout a été modélisé par des réseaux de Petri.

PolyORB et les besoins de l'automobile

L'intergiciel PolyORB essaye de découpler le niveau applicatif du niveau protocole, ce qui dans le contexte automobile permettrait de masquer l'hétérogénéité des plates-formes de communication. De plus, il présente la propriété importante d'être construit à l'aide de modèles objets ré-utilisables, et dont l'interopérabilité de comportement logique a été vérifiée en utilisant une méthode formelle (réseaux de Petri). Cette caractéristique fait que PolyORB est un intergiciel modulaire qui peut être facilement modifié. Néanmoins, PolyORB ne traite pas une question importante pour le domaine automobile : le temps. Il n'y a aucune garantie sur les délais associés à l'exécution de l'intergiciel, vu que, aucune indication sur son implémentation (nombre de tâches) n'est donnée. De plus, des services comme l'adressage d'objets, présentent des caractéristiques qui ne seraient pas souhaitables pour un intergiciel embarqué dans un véhicule.

2.2.1.3 ACE et TAO

ACE [25, 26] (*Adaptative Communication Environment*) est un cadre de travail (lire *framework*) qui implémente des *patterns* pour la concurrence et pour la distribution. Cet intergiciel fournit un ensemble de *wrappers* (types primitifs de données qui offrent des méthodes permettant leur manipulation) et de composants en C++, qui sont ciblés vers le développement d'applications et de services de haute-performance

et de temps-réel, en recouvrant un vaste éventail de plates-formes. ACE simplifie le développement d'applications distribuées orientées objets en offrant des services de :

- établissement de connexions et initialisation de services,
- démultiplexage et expédition d'événements,
- communication inter-processus et gestion de mémoire partagée,
- configuration statique et dynamique de services de communication,
- concurrence et synchronisation,
- communication distribuée (nommage, *logging*, synchronisation d'horloges, ...), et
- composants distribués de haut-niveau (ORBs, serveurs *web*).

TAO (*The Ace Orb* [27]) est un ORB ciblé pour des applications ayant des besoins de qualité de service temps-réel (traitement de missions aéronautiques, applications multimédia, et simulations interactives distribuées). Cet intergiciel objet présente une conception extensible car son développement est aussi guidé par un langage de *patterns* [28]. Cette caractéristique permet à TAO d'être dynamiquement configuré de façon à respecter, d'une part, les besoins de qualité de service des applications, et, d'autre part, les caractéristiques de la plate-forme de communication utilisée.

TAO est développé en utilisant ACE. En effet, ACE fournit un cadre de travail et des composants ré-utilisables qui peuvent supporter les besoins de qualité de service, les applications temps-réel, et donc les intergiciels de haut-niveau tels que TAO. Celui-ci n'apporte donc que les solutions manquantes pour que l'on puisse parler d'un intergiciel temps-réel objet :

- possibilité pour les applications de pouvoir exprimer leurs besoins de qualité de service, et
- détermination des aspects nécessaires pour construire des ORBs qui établissent des garanties de qualité de service déterministes (protocoles, traitement et expédition de requêtes, et gestion de la mémoire efficace et prédictible).

TAO et les besoins de l'automobile

TAO présente une caractéristique essentielle au niveau de son développement : les *patterns*, ce que lui permet d'être extensible et modulaire. De plus, il semble contenir des services qui lui permettent de garantir des délais déterministes aux applications. Nous verrons dans la section 2.3.2.3 que TAO ne nous donne aucune indication relativement à son implémentation (le nombre de tâches essentiellement), ce qu'empêche toute analyse d'ordonnançabilité. Enfin, les services de communication en utilisant des mécanismes de nommage, et la configuration dynamique de ces services font de lui un candidat peu adapté à être un intergiciel automobile.

2.2.2 Intergiciels embarqués dans l'automobile

La spécification et l'implémentation d'intergiciels a fait l'objet de travaux de la part des acteurs de l'industrie automobile. Nous listons par la suite quelques propositions, telles que Volcano, le volet lié à la communication du projet OSEK/VDX, le travail développé dans le projet Eureka ITEA EAST-EEA, et, finalement, les propositions engagées par le consortium AUTOSAR.

2.2.2.1 Volcano

Les objectifs de l'intergiciel Volcano [29, 30] sont de fournir une assistance à la conception des systèmes de multiplexage véhicule (pour l'instant, autour des réseaux CAN [3] et LIN [7]), indépendamment du protocole de communication sous-jacent. Volcano est caractérisé par :

- la garantie *a priori* des propriétés temporelles de la messagerie par la méthode de conception du système (garantie de respect des échéances sur les signaux),
- la flexibilité de la méthodologie donnant au constructeur la possibilité de mise à jour et re-configuration (*upgrade*) de la messagerie en phase de pré-production, et même après la phase de mise en marché du véhicule, et
- l'utilisation efficace des ressources.

L'implémentation actuelle de Volcano repose sur deux parties majeures :

- un ensemble d'outils à utiliser hors-ligne pour la conception automatisée de la messagerie, et
- un outil logiciel qui s'occupe de générer le code Volcano optimisé pour un ECU cible. Cet outil fournit une API basée sur les signaux, supporte plusieurs protocoles réseaux, et permet la re-configuration de la messagerie même après la compilation de l'application.

Ces deux parties reposent sur le concept que Volcano est capable de prouver la garantie d'échéance de toutes les trames envoyées sur une architecture de réseaux hétérogènes. Pour cela, Volcano construit *a priori* la messagerie réseau (*frame packing*). Les paramètres de configuration de cette messagerie sont : les signaux qui composent chaque trame, la taille, la période de transmission, l'échéance, et la priorité. L'ensemble de ces paramètres doit assurer que :

- la consommation de bande passante du réseau est minimisée, et
- les échéances sur la transmission des signaux transportés par les trames sont respectées.

Le respect des échéances est vérifié en utilisant la méthode d'analyse proposée par K. Tindell dans [31].

Du point de vue de l'application, l'intergicel Volcano offre une API basée sur les signaux, la donnée considérée par cet intergicel comme le moyen utilisé par l'application pour communiquer. Le modèle de cette communication repose sur le modèle producteur/consommateur (ou *éditeur/souscripteur* pour Volcano). L'API cache tous les détails concernant le comportement du réseau, et ne doit permettre que la manipulation des signaux produits et consommés. Elle est composée des primitives suivantes :

- *read* : lit la valeur courante d'un signal (les données ne sont pas lues dans le contrôleur réseau mais dans la couche Volcano) ; notez que pour recevoir les dernières valeurs, il faut précédemment appeler la primitive *v_input* qui ramène les valeurs reçues du réseau vers la couche Volcano,
- *write* : met à jour la valeur d'un signal (les données ne sont pas écrites dans le contrôleur réseau mais dans la couche Volcano),
- *v_input* : transfert les trames (arrivées depuis le dernier appel) du contrôleur réseau vers la couche Volcano, et rend les valeurs disponibles pour être lues par des appels à la primitive *read*,
- *v_output* : transfert les trames de la couche Volcano au contrôleur réseau (requête(s) de transfert de données), et
- *v_gateway* : copie tout ou partie d'une trame reçue vers une trame à émettre (ex : interface réseau habitacle réseau moteur).

Volcano offre aussi une API dite de basse latence pour *trames immédiates*. Les primitives en question,*v_imf_rx* et *v_imf_tx*, permettent la transmission et réception de trames de/vers la couche Volcano, sans avoir besoin d'utiliser les primitives normales *v_input* et *v_output*. Une troisième primitive, *v_imf_queued*, permet de vérifier si une trame immédiate a été effectivement transmise sur le réseau.

Dans [32] le lecteur pourra trouver un résumé actuel de Volcano, où, en particulier, il est possible de trouver quelques détails sur la méthodologie de conception du logiciel Volcano (travail conjoint entre constructeurs et équipementiers) et sa configuration.

Avantages/Inconvénients de Volcano

Volcano présente certaines caractéristiques qui peuvent faire de lui un intergiciel standard pour les applications des systèmes automobiles : une API qui masque l'interface réseau, la garantie d'une performance temps-réel, et le mécanisme de *frame packing*. Le problème principal de Volcano est dans le fait d'être un produit commercial, et donc, les détails concernant son implémentation et ses algorithmes de configuration ne sont pas publiques. Le principal inconvénient de Volcano est qu'il présente une API qui ne cache pas à l'application l'existence d'un réseau de communication. Aussi, avant de pouvoir consommer des signaux, l'application doit appeler une primitive qui fait le transfert des trames du contrôleur réseau vers la couche Volcano. Cette contrainte oblige l'application à appeler cette primitive à chaque fois qu'un signal est consommé.

2.2.2.2 OSEK/VDX Communication

OSEK/VDX est un projet qui vise à spécifier des standards qui supportent le développement d'applications automobiles ré-utilisables et dont la portabilité se fait facilement. Plusieurs standards ont été spécifiés : les systèmes d'exploitation OSEK/VDX OS et OSEK/VDX OStime, les services de communication par OSEK/VDX COM et OSEK/VDX FTCom, et la gestion de réseau avec OSEK/VDX NM. Concrètement, le but est de spécifier des interfaces les plus abstraites et indépendantes possible de l'application, de la plate-forme, et du réseau utilisés. Les standards qui se rapprochent d'un intergiciel sont OSEK/VDX COM [33] et OSEK/VDX FTCom [34], qui s'occupent de la communication dans et entre les ECUs embarqués dans le véhicule.

Grâce à son API, OSEK/VDX COM apporte des avantages au niveau de la ré-utilisation de code applicatif. Pour les applications qui n'utilisent qu'un sous-ensemble de services, OSEK/VDX COM définit cinq classes de conformité, où chacune assure l'implémentation d'une partie des fonctionnalités fournies par OSEK/VDX COM.

OSEK/VDX COM est divisé en couches, où chaque couche a un rôle spécifique et bien défini dans l'architecture de cet intergiciel :

- couche *Interaction* - elle offre l'API de programmation d'OSEK/VDX COM qui définit les services de communication pour le transfert des signaux de l'application. Pour la communication réseau, la couche "interaction" utilise les services fournis par les couches inférieures. La communication interne à l'ECU est manipulée directement par cette couche sans la participation des couches inférieures.

- couche *Network* - elle offre des services pour le transfert sans acquittement et segmenté des messages de l'application et, fournit des mécanismes de contrôle de flux qui permettent la communication entre hôtes avec différents niveaux de performance et de capacité. Cette couche utilise les services fournis par la couche inférieure.

- couche *Data Link* - elle offre des services aux couches supérieures pour le transfert sans acquittement des paquets individuels de données sur le réseau. La taille des paquets dépend du réseau utilisé.

OSEK/VDX COM fournit une interface basée sur les signaux, dont la taille de ceux-ci peut être statique ou dynamique (mais toujours bornée). Les signaux à envoyer entre ECUs peuvent avoir un parmi deux modes de transfert. *Triggered*, ce qui indique que le signal est mis en trame et sa transmission demandée dès qu'il est produit, et *pending* qui établit que la production d'un signal n'est pas immédiatement suivie de sa mise en trame ni de son émission sur le réseau. Les trames suivent un de trois modes possibles de transmission : *direct*, où la transmission d'une trame est initiée par la production d'un signal avec la propriété *triggered*, *periodic* où les trames sont transmises en suivant une période pré-définie, et *mixed* qui est une combinaison des deux autres.

OSEK/VDX COM fournit aussi un mécanisme de surveillance des échéances. D'une part, cette fonctionnalité vérifie que, du côté émetteur, la couche sous-jacente confirme la transmission des trames dans un intervalle de temps pré-défini. D'autre part, elle vérifie que du côté récepteur, les trames périodiques sont reçues dans un intervalle de temps pré-défini. En complément de ce mécanisme, OSEK/VDX COM offre des services de notification qui donnent à l'application la possibilité de connaître l'état d'une requête d'envoi ou de réception de signaux faite antérieurement.

OSEK/VDX COM a été défini pour s'exécuter dans un environnement où l'activation des applications est basée soit sur le temps, soit sur le déclenchement d'événements. Dans un système où toute activation se base sur le temps (*time-triggered*), le standard OSEK/VDX FTCom a été spécifié. Dans ce type de système, OSEK/VDX OStime [2] peut être utilisé comme système d'exploitation. Il se chargera en-ligne de la gestion des ressources CPU, du temps, et de l'ordonnancement des tâches. OSEK/VDX FTCom est responsable de la communication entre les noeuds, la détection d'erreurs, et de la tolérance aux fautes liée à la communication. Concrètement, ses services sont le filtrage des trames, la gestion de la réplication des trames, le temps, et, en option, la synchronisation d'horloges avec l'extérieur. Il n'existe pas, à l'heure actuelle, d'implémentation d'OSEK/VDX OStime ; sa spécification et celle d'OSEK/VDX FTCom n'ont pas évolué depuis 2001.

Conclusions sur OSEK/VDX Communication

Le standard de communication d'OSEK/VDX spécifie les services à offrir à la couche application mais ne propose aucune directive pour leur implémentation et configuration. Nous pouvons aussi ajouter que l'API semble trop confuse, et qu'elle ne cache pas aux applications certains détails liés à la communication. Par exemple, l'application doit indiquer si un signal est *pending* ou *triggered*, ce qui ramène à la responsabilité de l'application de s'assurer que les productions des signaux *pending* sont complétées par la production d'au moins un signal *triggered*. Il faut noter également que l'existence conjointe de trames périodiques et trames directes et mixtes, peut rendre difficile une analyse d'ordonnançabilité au niveau du réseau.

2.2.2.3 Eureka ITEA EAST-EEA

Le projet européen EAST-EEA [35] a eu comme objectif de permettre l'intégration de composants électroniques dans un véhicule par la définition d'une architecture ouverte. Celle-ci devrait supporter l'interopérabilité et la ré-utilisation de composants matériels et logiciels. De plus, des objectifs sous-jacents étaient la réduction du temps de développement et de mise en marché des véhicules, et l'amélioration de la qualité du logiciel embarqué.

L'approche utilisée dans le projet a été, d'une part, de spécifier une architecture d'intergiciel et un modèle de communication pour une architecture embarquée dans un véhicule. D'autre part, de créer un ADL (*Architecture Description Language*) pour le développement et la validation d'architectures et modules fonctionnelles. Cette dernière partie du projet s'est focalisée sur l'aspect structurel, mais aussi sur la modélisation des descriptions du système à différents niveaux d'abstraction, dans un objectif de documentation, conception, analyse, et synthèse du système. Même si l'ADL EAST fournit un moyen capable de modéliser le comportement (basé sur les machines à états UML), elle suppose surtout que le comportement est modélisé à l'aide d'outils externes.

La partie dédiée à l'intergiciel n'a pas abouti à des résultats aussi avancés que la partie ADL. Néanmoins, des propositions existent qui spécifient l'intergiciel pour chacun des domaines véhicule. En particulier, le document relatif au domaine *Powertrain* détaille l'API proposée, l'architecture et la méthode de configuration de l'intergiciel, et les interactions intra-intergiciel et entre celui-ci et l'application.

Conclusions sur EAST-EEA

La proposition provenant du projet EAST-EEA a été la première tentative du domaine automobile pour la définition d'une architecture d'intergiciel masquant la communication (intra et inter-ECU) et la plateforme matérielle. Mais, encore une fois, cette proposition ne donne aucune indication sur l'implémentation de l'intergiciel, ni sur les méthodes de configuration.

2.2.2.4 AUTOSAR

Le consortium AUTOSAR (*AUTmotive Open System ARchitecure* [36]) a produit la spécification d'un intergiciel automobile la plus récente. Ce consortium est un partenariat entre les principaux acteurs mondiaux de l'automobile (constructeurs, équipementiers, éditeurs de logiciels, et fournisseurs de composants électroniques). Son objectif est de développer et établir un standard industriel ouvert pour les architectures électriques/électroniques dans l'automobile. Les objectifs de ce partenariat sont :

- prise en compte des besoins de disponibilité, sûreté et redondance des applications embarquées,
- passage à l'échelle et variabilité des véhicules et des plates-formes,
- standardisation des fonctions systèmes de base,
- possibilité de déploiements différents sur une architecture distribuée,
- intégration de modules de plusieurs équipementiers,
- maintenance pendant le cycle de vie des produits,
- augmentation de l'utilisation des produits matériels COTS (*Components Of The Shelf*), et
- mises à jour pour les produits logiciel pendant toute leur vie.

Pour AUTOSAR, une application automobile consiste en un ensemble de composants logiciels (*Software Components* - SWC) interconnectés. Un SWC peut être une petite fonction ré-utilisable, ou un grand bloc englobant une fonctionnalité automobile entière. Le but est de séparer l'application de l'infrastructure. Chaque SWC vient accompagné de sa *Software Component Description*, qui contient des informations telles que les opérations et les données que le SWC fournit ou demande, les besoins du SWC au niveau de l'infrastructure, et les ressources exigées par le SWC.

Le projet AUTOSAR définit également plusieurs concepts. Le premier est la notion de *Virtual Functional Bus* (VFB), qui est l'abstraction des interconnexions entre les SWCs dans le véhicule (abstraction de la communication, *hardware*, système d'exploitation, services). Au niveau de la communication, le VFB fournit des patrons de communication bien connus et indépendants du réseau utilisé : client-serveur (bloquant et non-bloquant) et émetteur-récepteur. Avec le VFB il est possible de faire une intégration virtuelle des SWCs, de façon à ce qu'une partie du processus d'intégration des logiciels embarqués se passent plutôt dans la phase de conception. Un deuxième concept est l'intergiciel AUTOSAR lui-même, qui s'appelle *Runtime Environment* (RTE). Pour le créer sur chaque ECU (voir section 2.3.1.3), l'image du RTE représentant le VFB est générée.

Conclusions sur AUTOSAR

AUTOSAR est le standard le plus adapté aux besoins des acteurs du domaine automobile. Les concepts introduits sont en effet nécessaires pour la validation *a priori* des composants logiciels embarqués dans les véhicules, et pour la construction d'une image intergicielle locale à chaque ECU. Cette image a l'avantage de correspondre et être strictement adaptée aux besoins des applications qui s'exécutent dans un ECU.

Cependant, il faut noter qu'AUTOSAR ne fournit qu'une spécification, dont les détails de l'implémentation ne concernent que les partenaires du consortium. En particulier, nous n'avons pas accès à la définition de l'interface du RTE, au contraire de ce qui se passe avec EAST-EEA. De plus, aucune référence est faite aux propriétés temporelles ; la procédure utilisée pour configurer le RTE de manière à ce que les contraintes temporelles soient respectées n'est pas connue. En effet, le VFB ne sert qu'à la vérification des interfaces des SWCs (par exemple, il vérifie qu'il existe au moins un consommateur pour chaque signal produit), il ne fait aucune validation temporelle des échanges. Dans tous les cas, il est clair que le standard AUTOSAR est, en ce moment, la proposition qui doit guider toute tentative de construction d'un intergiciel automobile.

2.2.3 Bilan sur les intergiciels et les systèmes embarqués

Nous venons de détailler quelques exemples d'intergiciels, spécifications ou implémentations, tirés du contexte général et du contexte automobile. D'une part, il est clair que dans un contexte général, tous les efforts suivent une direction vers la construction d'intergiciels basés sur une approche objet. Par contre, les solutions actuelles ne sont pas compatibles avec les contraintes des systèmes embarqués dans l'automobile. D'autre part, les acteurs du secteur automobile sont impliqués dans la définition de standards. Néanmoins, cette spécification n'est pas publique, la prise en compte des aspects temporels n'est pas assurée à l'heure actuelle, et aucun support pour l'implantation de l'intergiciel n'est fourni. En

conclusion, il est nécessaire de trouver une méthode ouverte supportant le développement d'un intergiciel automobile lui même aussi ouvert.

Le standard AUTOSAR est actuellement une référence ; ce standard doit donc être pris en compte pour la définition de cette méthode. En particulier, nous nous intéressons à la proposition faite par le consortium pour le déploiement et la configuration d'empreintes d'intergiciel adaptées aux besoins de chaque ECU (voir section 2.3.1.3). Néanmoins, la jeunesse de cette proposition, et l'absence de prise en compte des aspects temps-réel, montrent qu'un certain nombre de travaux doivent être menés pour accompagner ce standard.

Dans la suite, nous détaillerons la proposition de méthodologie du consortium AUTOSAR, ainsi que d'autres issues du domaine automobile et du domaine académique, pour le développement d'intergiciels. Dans le domaine académique, nous nous concentrons sur l'approche objet, et plus particulièrement sur le développement de logiciels à l'aide de modèles objets ré-utilisables (*patterns*), utilisés par exemple dans TAO et PolyORB.

2.3 Méthodes de développement d'un intergiciel

Soit au sein de l'industrie automobile, soit dans un milieu académique, plusieurs tentatives de spécification de méthodes pour le développement d'intergiciels ont eu lieu. Les principales tentatives sont décrites par la suite, d'abord celles qui concernent le contexte automobile, et ensuite celles qui sont nées des recherches en milieu universitaire. Parmi ces dernières, nous nous intéressons plus particulièrement aux approches basées sur les modèles objets ré-utilisables.

Nous pourrions aussi référencer une autre approche qui concerne l'utilisation d'aspects (*Aspect Oriented Programming* [37]). Ce paradigme de programmation permet de réduire les entrecroisements entre les différents aspects techniques (*e.g.* persistance, authentification, trace, sécurité, ...) d'un logiciel. Ainsi, pour éviter d'avoir un logiciel où tous ces aspects sont fortement liés, la solution passe par développer le code qui est focalisé sur l'objectif du logiciel, et insérer dans le code des aspects qui appelleront les modules techniques requis. Toutefois, le point de départ de cette approche est situé au niveau du code du logiciel. De son côté, l'approche modèles objets ré-utilisables aide à la modélisation de haut-niveau en utilisant des diagrammes de classe UML. Une fois le code généré à partir des diagrammes, l'approche aspects peut éventuellement se rendre utile. Nous considérons donc que l'approche aspects peut être complémentaire à celle des modèles objets ré-utilisables [38].

2.3.1 Contexte automobile

Si dans ce contexte, des spécifications d'intergiciels ont été fournies (voir section 2.2.2), peu se sont accompagnées d'une méthode d'implémentation finale. Dans les sections suivantes, nous montrons les quelques méthodologies proposées pour la construction et configuration, soit explicitement d'intergiciels, soit d'architectures de composants logiciels distribués et interopérables. De plus, nous identifions les aspects couverts par ces méthodologies.

2.3.1.1 Volcano

La méthode proposée par Volcano pour développer l'intergiciel de même nom est basée sur une approche à deux phases. La première construit les fichiers de configuration réseau (configuration des trames) à partir d'une base de données de signaux. La deuxième phase consiste à générer les fichiers binaires pour chaque ECU à partir des fichiers de configuration. Du fait que Volcano est un produit commercial, les algorithmes permettant d'accomplir ces objectifs ne sont pas publiques.

Un intérêt particulier a été donné à la deuxième phase en ce qui concerne l'échange de fichiers entre les constructeurs automobiles et les équipementiers. Un flux de travail définissant le passage de fichiers entre ces acteurs a été défini. Plusieurs types de fichiers ont été spécifiés, certains privés à un des acteurs, d'autres publiques et accessibles à tous. Une procédure de génération d'image binaires de Volcano est fournie, qui prend en entrée tous les fichiers, soit privés soit publiques.

Conclusions sur Volcano

Volcano présente une tentative de découplage entre la configuration globale (paramètres réseau, messagerie), et la configuration locale à chaque ECU (binaire Volcano dans chaque station). Néanmoins, comme il a été dit précédemment, les algorithmes de Volcano ne sont pas publiques, ce qui nous empêche d'avoir une connaissance sur son modèle d'implémentation, et rend difficile son évaluation et sa comparaison avec d'autres propositions de stratégies.

2.3.1.2 Titus

Titus [39] est une méthodologie pour le développement de logiciels distribués sur des ECUs. Cette méthodologie essaye de montrer que les fonctions automobiles peuvent être ré-utilisées sur un ensemble de réseaux d'ECUs. L'objectif est de séparer le développement des logiciels automobiles de la plate-forme sur laquelle les logiciels s'exécuteront. Plus en détail, le but de Titus est de :

- détecter au plus tôt les problèmes d'intégration,
- permettre la ré-utilisation de composants et fonctions,
- fournir des moyens pour que l'on puisse ré-utiliser des tests et des procédures de validation,
- pouvoir générer du code automatiquement, et
- combiner des technologies de production bien rodées.

Pour cela, la méthodologie repose sur un ensemble d'outils, proposant une interface graphique, qui permettent la conception de l'architecture fonctionnelle du véhicule, la définition des ECUs et du réseau, et la projection (*mapping*) des composants sur les ECUs (fonctions sur les tâches, des tâches sur les ECUs, et des signaux sur les trames). La méthodologie est peu explicite sur les algorithmes utilisés pendant l'étape de projection, mais en phase de conception de l'architecture fonctionnelle du véhicule, Titus propose les diagrammes de bloc pour la description structurelle et les diagrammes de machine à états finis pour le côté comportemental.

En particulier, Titus définit un ensemble d'éléments à utiliser pour la conception des diagrammes de bloc. L'élément principal de base pour modéliser le logiciel distribué dans un véhicule est la *Process Class* (PC). Une PC est un objet actif (qui peut s'exécuter en concurrence avec d'autres objets actifs) qui cache son implémentation et qui inter-agit avec d'autres PCs au moyen d'interfaces. Une interface, appelée *Service Access Point* (SAP), représente un aspect d'une PC et permet à celle-ci d'inter-agir avec d'autres PCs. Chaque PC peut avoir plusieurs SAPs, et la communication se fait par appels de méthode, dont un ensemble compose un *Protocol*. Un *Protocol* est une agrégation de méthodes fournies par un SAP serveur ou requises par un SAP client.

Conclusions sur Titus

La méthodologie Titus est essentiellement tournée vers la spécification d'architectures fonctionnelles de niveau applicatif. Tous les détails sur les aspects intergiciel, particulièrement ceux relatifs à la projection des signaux sur les trames (*frame packing*), ne sont pas publiques. Titus reste néanmoins étant une bonne méthodologie pour la construction d'architectures fonctionnelles.

2.3.1.3 AUTOSAR

AUTOSAR [36] définit la méthodologie nécessaire pour, à partir de l'information des différents éléments de description (voir section 2.2.2.4), pouvoir développer un système concret d'ECUs. Ceci inclut particulièrement la configuration et génération de l'intergiciel AUTOSAR, appelé de *Runtime Environment* (RTE), et du *Basic Software* (couche contenant des composants complémentaires au RTE) sur chaque ECU.

Pour le déroulement initial de la méthodologie, un ensemble de paramètres d'entrée doivent exister. Le premier est l'ensemble des *Software Component Descriptions* qui contiennent des informations telles

que les opérations et les données que les SWCs (*Software Components*, présentés dans la section 2.2.2.4) fournissent ou demandent, les besoins des SWCs au niveau de l'infrastructure, et ressources exigées par les SWCs. Le deuxième paramètre est formé par les *ECU Resource Descriptions* qui décrivent les ressources matérielles disponibles (périphériques, mémoire, CPU). Le troisième et dernier paramètre est la *System Constraint Description* qui inclut des contraintes données par une architecture réseau déjà existante.

Tous ces paramètres sont utilisés par le premier générateur de l'intergiciel, appelé de *System Configuration Generator*. Ce générateur fait le placement des SWCs parmi les ECUs, et construit la matrice de communication. Le placement est fait en prenant en compte les besoins des SWCs et leurs liens de communication, les ressources disponibles, et les contraintes de l'architecture. La matrice de communication décrit la composition des trames envoyées sur le réseau, ainsi que leurs caractéristiques temporelles.

A partir de cette configuration de niveau système, le deuxième générateur appelé *ECU Configuration Generator*, extrait les informations relatives à chaque ECU, prend des décisions sur la conception locale à chaque station, et configure et génère le RTE et le *Basic Software* qui seront intégrés aux SWCs de l'application de chaque ECU.

Conclusions sur AUTOSAR

La méthodologie proposée par le consortium AUTOSAR définit un enchaînement d'activités pour la configuration système (trames et placement des composants application), et la configuration et implémentation ECU (image du RTE sur chaque station). De plus, le fait de commencer par les aspects système pour après configurer les aspects ECU est une stratégie qui apparaît comme la mieux adaptée. Notons que cette approche est celle utilisée par TTTech pour configurer les stations utilisant le plate-forme réseau TTP [40].

Cependant, cette méthodologie est spécifiée à un haut niveau d'abstraction et les activités restent à spécifier. Quelles sont les stratégies utilisées pour le placement des composants application sur chaque ECU ? Ensuite, quels sont les algorithmes qui permettent déterminer la matrice de communication ? Finalement, quelle est la procédure à utiliser pour créer les images du RTE et du *Basic Software* à partir de la configuration système et ECU ?

2.3.2 Contexte académique: approche objets

Dans cette section nous analysons une méthode pour le développement d'intergiciels, qui a été appliquée dans le contexte de TAO et PolyORB (voir section 2.2.1). Cette méthode repose sur le paradigme orienté objets, ce qui favorise le passage à l'échelle, la ré-utilisation, et la maintenance du logiciel. L'existence d'outils graphiques pour la modélisation, lié à la possibilité de transformer les modèles en code pour des langages orientés objets, a été un atout qui a amené à une utilisation croissante de ce paradigme.

En particulier, cette méthode utilise la notion de patrons de conception orientés objets (nous conservons par la suite le terme anglais de *design patterns*). Nous allons donc commencer par présenter les design patterns et leur utilisation dans le domaine du temps-réel, pour ensuite pouvoir décrire leur application au développement des intergiciels PolyORB et TAO.

2.3.2.1 Design patterns

Un design pattern est une solution orientée objets pour un problème identifié et récurrent de conception logicielle dans un contexte bien précis. Il identifie les composants d'un système logiciel (structure de classes ou objets), le rôle de chaque composant, et les relations (comportement) entre eux. Dans l'annexe A (Appendix A - The Mediator design pattern) le lecteur pourra trouver un exemple de design pattern. L'objectif est de résoudre des problèmes de conception de logiciels, de rendre ces logiciels plus flexibles et ré-utilisables, et d'améliorer la documentation et la maintenance des systèmes logiciels existants.

Les design patterns essayent de résoudre plusieurs types de problèmes : structuraux (organisation de classes), comportementaux (traitement d'événements, synchronisation, concurrence), création d'objets, etc. Dans [41, 24, 42] le lecteur pourra trouver des catalogues de design patterns classés selon le type de problème traité.

En dehors des intergiciels dont les méthodes de développement sont présentées (PolyORB et TAO) ci-dessous, d'autres essais d'utilisation de design patterns dans le domaine du temps-réel existent. Dans [43] l'auteur propose un catalogue de design patterns qui traitent un ensemble de problèmes courants dans le contexte du temps-réel : distribution, sûreté, et fiabilité. D'autres propositions plus spécifiques ont aussi été faites. Dans [44] le *real-time constraints as strategies* design pattern est présenté. Il permet la séparation entre contraintes et comportement spécifiques au temps-réel (implémentés comme des routines crochets (*hooks*)), et les services auxquels ils sont appliqués (codés comme étant des stratégies). Le *multithreaded rendezvous* pattern [45] propose un serveur multi-*thread* (une *thread* pour chaque client), et un objet expéditeur qui fait le transfert des requêtes vers la *thread* serveur correspondante. De cette façon, les requêtes provenant de différents clients n'interfèrent pas entre elles. Finalement, [46] décrit le *recursive control* pattern qui sépare les aspects contrôle (ensemble de services secondaires) et les aspects service (services primaires) dans un programme temps-réel, permettant que chaque aspect soit défini et modifié indépendamment. Plus de détails sur l'utilisation de tels patterns pour la conception de logiciels temps-réel peuvent être trouvés dans [47].

Conclusions sur les design patterns

Nous avons discuté sur les avantages liées à l'utilisation des design patterns. Nous avons aussi listé quelques exemples de leur utilisation au sein du domaine du temps-réel. Cependant, les design patterns proposés pour ce domaine ne traitent pas la question du temps et de l'ordonnançabilité. Spécifiés tels quels, ces patterns pourraient être appliqués au développement de n'importe quel type de logiciel, et non seulement du temps-réel. Les patterns ne nous donnent que la spécification de code qui sera exécuté. Il manque donc des méthodes de configuration et vérification de ces modèles objets ré-utilisables dans un contexte temps-réel. Des méthodes qui, plus qu'une simple vérification d'interopérabilité entre interface de composants logiciels, vérifient l'interopérabilité temporelle.

2.3.2.2 Utilisation des design patterns dans PolyORB

Les auteurs de PolyORB déclarent que cet intergiciel est implémenté à l'aide de design patterns [48]. Cependant, peu d'informations existent sur leur rôle au sein de l'intergiciel. Les design patterns utilisés sont *Facade* [41], *Reactor* [42], *Annotation* [49], et *Component* [49].

2.3.2.3 Utilisation des design patterns dans TAO

L'intergiciel TAO a été présenté dans la section 2.2.1.3. Comme mentionné précédemment, cet intergiciel est développé à l'aide de design patterns [28] (voir figure 2.2). Nous détaillerons ci-dessous de façon succincte le rôle joué par chacun de ces patterns au sein de l'intergiciel.

Le pattern *Wrapper Facade* [42] est utilisé pour encapsuler les fonctions et les données non orienté objets (interface des *sockets*, POSIX *threads*, ...) dans des classes ré-utilisables. Le pattern *Reactor* [42] a comme fonction d'implémenter la boucle principale de l'intergiciel qui accepte des connexions, et reçoit/envoie les requêtes/réponses des clients. Le pattern *Acceptor-Connector* [42] est responsable de l'établissement des connexions indépendantes du mécanisme de transport sous-jacent. Le pattern *Leader/Followers* [42] a comme but de faciliter l'utilisation de multiples stratégies de concurrence, qui peuvent être configurées en-ligne dans l'intergicel. *Thread-Specific Storage* [42] est utilisé pour minimiser les situations de contention de verrous et d'inversion de priorités. *Strategy* [41] a comme objectif de permettre à TAO d'être configuré au niveau des stratégies de concurrence, communication, ordonnancement, et traitement/expédition de requêtes de manière extensible. Le pattern *Abstract Factory* [41] a la fonction de permettre aux composants logiciels de TAO d'accéder aux stratégies sans spécifier explicitement de nom de la classe. Finalement, le pattern *Component Configurator* [42] supporte le changement dynamique d'implémentations de façon à ce que TAO puisse configurer ses composants en-ligne.

Le résultat est un intergiciel dont l'implémentation présente la propriété d'être facilement extensible ou configurable, une capacité à être ré-utilisable et portable sur une autre plate-forme, et une conception claire et expressive. Spécifiquement, les auteurs de TAO ont remarqué une diminution dans le nombre de méthodes nécessaires pour implémenter les tâches de l'intergiciel, et aussi dans le total de lignes de code de ces méthodes [28].

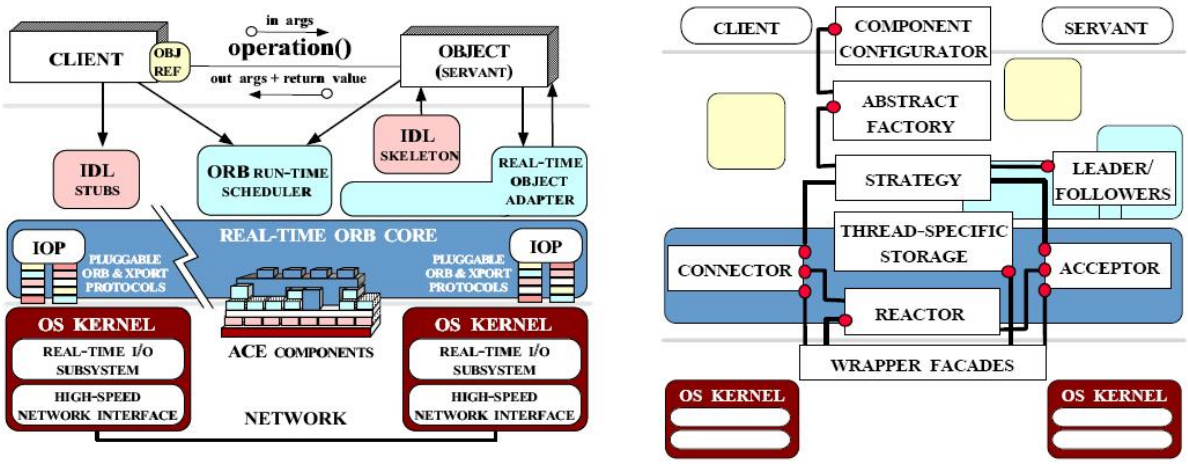


Figure 2.2: Composants et organisation des design patterns dans TAO.

L'image à gauche illustre les composants de l'intergiciel TAO. La figure de droite liste tous les design patterns utilisés pour le développement de TAO. Ces images ont été prises de [28].

Utilisation des design patterns dans le contexte automobile

La conception et l'implémentation de PolyORB et TAO utilisent les design patterns, accroissant ainsi la flexibilité lors de leur conception, leur ré-utilisation et leur maintenance. L'utilisation d'une telle approche pour développer les intergiciels embarqués dans l'automobile permettrait donc d'augmenter la qualité du logiciel embarqué, ainsi que sa documentation et sa maintenance. Elle offrirait aux acteurs de l'automobile une adaptation plus rapide et plus efficace à l'intégration de nouvelles fonctions embarquées dans les véhicules. De plus, les design patterns sont la solution idéale pour fournir une portabilité et une interopérabilité entre composants logiciels développés séparément dans un processus multi-partenaires. Cette caractéristique est importante dans un contexte où les constructeurs automobiles achètent des composants développés par équipementiers, et donc, l'intergiciel est de fait un composant dont le développement est partagé entre ces acteurs.

2.3.3 Synthèse sur les techniques de développement d'intergiciels

Ainsi que nous l'avons déjà souligné dans la section 2.2.3, les résultats du consortium AUTOSAR sont incontournables actuellement dans l'industrie automobile, et apparaissent comme des standards de fait. Dans le contexte de cette thèse, la spécification structurelle, fonctionnelle, et d'interface de l'intergiciel AUTOSAR est une entrée pour nos travaux. De plus, le consortium a assorti cette spécification d'une méthodologie de développement donnée à un haut niveau d'abstraction (identification des activités, mais absence de spécification des activités elles-même). Cette vision abstraite et l'absence, à l'heure actuelle, de la prise en compte des aspects temporels (temps métrique) nécessitent donc de mener des travaux complémentaires sur ces propositions.

Nous avons vu l'intérêt des design patterns et leurs avantages, et présenté certaines propositions dans le domaine du temps-réel. Néanmoins, nous avons montré que celles-ci ne traitent pas réellement le problème du temps ni celui de l'ordonnançabilité, et fournissent essentiellement une spécification de code. Les design patterns sont décrits à l'aide du formalisme UML : un diagramme de classe introduit la structure du pattern, et un diagramme de séquence illustre le comportement. Dans le but d'effectuer une analyse d'ordonnançabilité aux patterns décrits à l'aide d'UML, il existe des méthodes comme le profil UML *Schedulability, Performance and Time* [50], la méthodologie ACCORD [51, 52], ou même l'outil MAST [53]. Cependant, pour pouvoir utiliser ces méthodes, il faut définir quels sont les objets actifs de

l'intergiciel (objets qui résident dans des tâches), ainsi que les tâches, au sens système d'exploitation, qui les incluront. Avec cette information, et après une étape de configuration des objets actifs et des tâches correspondantes à partir des besoins de l'application, les méthodes référencées ci-dessus pourraient être utilisées afin de vérifier que l'intergiciel préserve les propriétés temporelles requises par l'application.

Une autre possibilité, est d'identifier un ensemble de tâches capables d'exécuter le code exprimé par les design patterns. Dans ce cas, nous devons nous poser les questions de quelles sont les tâches qui implémentent l'intergiciel sur chaque ECU, quelles sont les caractéristiques temporelles (périodes d'activation, temps d'exécution, échéances, et priorités) de chaque tâche, et, par conséquence, à quel modèle d'activation (périodique, sporadique, ou autre) ces tâches appartiennent. Pour répondre à ces questions nous devons donc nous concentrer sur des thèmes précis et analyser pour chacun d'eux ce que propose actuellement la littérature : identification des tâches nécessaires à l'implémentation de l'intergiciel, et caractérisation des tâches et des trames (vu que les caractéristiques des trames peuvent influencer celles des tâches). Dans la section suivante, nous allons détailler quelques travaux qui concernent les questions posées.

2.4 Identification de tâches

Pour qu'une configuration des composants d'un intergiciel permette une analyse d'ordonnançabilité il est nécessaire d'identifier quels objets actifs ou quelles tâches vont exécuter cet intergiciel. Certains design patterns guident cette identification. D'autres ne donne aucune indication à ce sujet. Deux types de travaux sur l'identification de tâches à partir d'une spécification orientée objets existent. Le premier a été développé par Douglass [54], tandis que le deuxième appartient à Saksena *et al.* [55].

2.4.1 Stratégies proposées par Douglass

Un des sujets traités par Douglass dans [54] est la représentation de *threads* au sein du formalisme UML, et en particulier la relation entre celles-ci et les objets. La représentation est faite à l'aide de diagrammes illustrant les *threads* qui contiendront les objets du logiciel, et dont lesquels il est possible de trouver un objet actif. Celui-ci est responsable de la réception d'événements à destination des objets contenus dans le *thread*. L'auteur déclare alors qu'au moment d'effectuer le diagramme de *threads*, il est nécessaire d'identifier un ensemble de *threads*, de définir les objets du logiciel qui s'exécuteront dans chacun, et enfin, d'attribuer des valeurs aux caractéristiques de chaque *thread*.

Dans ce but, des stratégies d'identification de *threads*, basées sur les événements traités par le logiciel, ont été proposées, dont les principales sont :

- création d'un *thread* pour chaque événement traité par le logiciel,
- d'un *thread* pour traiter tous les d'événements originaires d'une même source,
- un *thread* pour la réception de tous les événements qui arrivent suivant une même période, et
- un *thread* pour traiter tous les événements qui ont le même objectif ou la même sémantique au sein du logiciel (par exemple, un *thread* pourra recevoir toutes les alarmes reçues, ou pourra traiter toutes les messages venant du même émetteur).

2.4.2 Stratégies proposées par Saksena *et al.*

Saksena *et al.* proposent une approche qui consiste à dériver automatiquement des implémentations sous la forme de tâches, à partir de modèles de conception temps-réel orientés objets [55]. Les modèles utilisés suivent le formalisme ROOM [56], où les composants principaux (appelés capsules) inter-agissent au moyen d'émission et réception d'événements.

Une des étapes de la procédure de dérivation proposée, consiste à associer les événements à des tâches. Cette étape détermine le nombre de celles-ci, ainsi que les capsules qui s'exécuteront dans chacune. Pour cela, un ensemble de stratégies sont suggérées :

- attribution de tous les événements destinés à une capsule à une seule tâche (une tâche par capsule),
- association de tous les événements de la même transaction (suite d'exécutions chaînées de capsules, activée par un événement) à une tâche ; le nombre de tâches identifiées par cette stratégie dépend du total de transactions dans le logiciel, et
- groupement de tous les événements ayant la même priorité à une tâche ; le nombre de tâches identifiées par cette stratégie dépend du nombre de priorités associées aux événements.

2.4.3 Conclusion

Les stratégies présentées ci-dessus, permettent au concepteur d'un logiciel orienté objets d'identifier un ensemble de tâches capables d'implémenter le logiciel, et en prenant en compte l'environnement où ce logiciel s'exécutera. Ces stratégies sont donc des sérieuses candidates à être utilisées pour résoudre notre problème d'identification des tâches de l'intergiciel, à partir de la structure de composants logiciels créée à l'aide de design patterns.

Ceci nous permettra de résoudre une partie du problème de l'implémentation de l'intergiciel. Il restera alors le problème de la configuration au sens de la garantie des propriétés temps-réel. Dans notre cas, nous allons particulièrement analyser les travaux existants relatifs au paramétrage de tâches, c'est à dire, à la définition des caractéristiques permettant d'atteindre la faisabilité de celles-ci. Toutefois, nous ne pouvons pas oublier le fait que la configuration des aspects locaux à chaque ECU (dont la configuration de tâches) est dépendante des aspects globaux, tels que les paramètres de la matrice de communication. Aussi, nous étudions également les travaux existants à ce sujet, couramment appelé *frame packing*.

2.5 Configuration de tâches et trames

Dans un système temps-réel, la configuration des tâches détermine leur faisabilité, c'est à dire, leur capacité à respecter les échéances sur une plate-forme donnée. La même chose peut être dite à propos d'un ensemble de trames. En ce qui concerne les tâches, la configuration consiste à attribuer des valeurs à certaines caractéristiques, comme le temps d'exécution, la période d'activation, l'échéance relative, et la priorité (selon le système d'exploitation utilisé).

La période d'activation d'une tâche dépend souvent de son environnement : l'arrivée des événements à traiter, ou la règle de récurrence des lois de contrôle/commande. L'échéance relative est fixée par le concepteur du système (dans la plupart des cas, elle est égale à la période d'activation). Les caractéristiques qui ont été le plus souvent la cible des chercheurs sont le temps d'exécution et la priorité. Pour des résultats concernant le calcul du temps d'exécution d'une tâche, le lecteur peut se procurer des travaux de Puaut [57].

Dans cet état de l'art, nous allons tout d'abord nous focaliser sur l'attribution de priorités, du fait qu'il s'agit pratiquement de la seule caractéristique modifiable par le concepteur dans le but d'essayer de rendre une configuration de tâches faisable. Il n'en est pas de même pour le temps d'exécution qui est constant et mesurable pour une plate-forme donnée. Ensuite, nous traitons la configuration de trames. Par la suite, nous utiliserons le terme en anglais *frame packing*, pour faire référence à la configuration d'une matrice de communication.

2.5.1 Configuration de tâches : priorité

Nous nous intéressons à l'attribution de priorités aux tâches dans le cas d'un ordonnancement préemptif à priorités fixes (contrainte du système d'exploitation OSEK/VDX OS [1]). Concrètement, nous nous sommes intéressés à l'algorithme d'attribution de priorités proposé par Audsley [58]. Cet algorithme est optimal dans le sens où, s'il existe une solution qui rend les tâches faisables, alors elle sera nécessairement trouvée. L'idée est de partir du niveau de priorité le plus faible (m) et de chercher une tâche qui soit faisable à ce niveau de priorité. Si il n'y a aucune tâche faisable au niveau de priorité m, alors l'ensemble des tâches est nécessairement non-faisable. La première tâche faisable au niveau m se voit affecter cette priorité. Il est clair que cette tâche aurait été faisable à tous les autres niveaux de priorité. Une fois

le niveau *m* attribué, l'algorithme cherche une tâche faisable au niveau *m-1* et ainsi de suite jusqu'à la priorité *1,* qui est la plus forte priorité du système.

Il existe des politiques d'attribution de priorités fixes aux tâches, comme *Rate Monotonic* (RM) ou *Deadline Monotonic* (DM) [59]. Ces politiques nous donnent une seule solution, c'est-à-dire une seule allocation de priorités qui doit être soumise à une vérification de la faisabilité des tâches. En cas d'échec, le concepteur doit utiliser une autre politique. De son côté, en cas de non faisabilité des tâches vis-à-vis d'une allocation de priorités donnée, l'algorithme d'Audsley présente l'avantage d'essayer de trouver une autre allocation. En particulier, une allocation non faisable de priorités déterminée par RM ou DM, peut être donnée à l'algorithme d'Audsley qui se chargera de vérifier s'il existe une autre allocation qui rend les tâches faisables.

2.5.2 Frame packing

Dans cette section nous allons présenter quelques algorithmes de frame packing. L'objectif de ce type d'algorithme est de construire chaque trame (à partir d'un ensemble de signaux), et pour chacune d'elles de calculer leurs paramètres. Ceux-ci sont : la taille (somme des tailles des signaux composant la trame), la période de transmission, l'échéance relative, et la priorité (si on utilise un réseau à priorités). Le but de tous les algorithmes présentés ci-dessous est de minimiser la consommation de bande passante. Toutefois, la faisabilité des trames n'est pas traitée par tous.

Nous avons décidé de ne pas présenter Volcano de nouveau. Nous avons vu dans la section 2.2.2.1 que cet intergiciel propose et exécute un algorithme de frame packing. Néanmoins, cet algorithme n'est pas connu à cause du caractère commercial de Volcano. Les algorithmes présentés ci-dessous sont le fruit de la recherche universitaire et sont donc publiques.

2.5.2.1 Algorithme proposé par Tindell et Burns

La première référence à un algorithme de frame packing, appelé *piggyback*, a été faite dans [31]. Le but des auteurs de ce travail était bien celui de minimiser la consommation de bande passante d'un ensemble de trames émises sur le réseau CAN. La stratégie utilisée était d'insérer des signaux ayant la même période de production dans une trame (celle-ci aurait la même période que les signaux). En plus, les auteurs ont prévu le cas où il y ait des signaux dont la production ne soit pas périodique. Dans cette situation, ils ont proposé l'utilisation d'une trame serveur périodique qui, au moment d'être construite, prendrait les signaux dont une occurrence ait été enregistrée.

2.5.2.2 Algorithmes proposés par Norström *et al.*

Dans [60], deux heuristiques pour la construction et configuration de trames qui minimisent la consommation de bande passante sur les réseaux CAN et LIN ont été introduites. La première heuristique insère des signaux dans une trame jusqu'à ce que celle-ci ne puisse plus accepter de signaux. A ce moment là, une nouvelle trame est créée. La deuxième heuristique considère que les trames peuvent avoir différentes tailles. L'algorithme démarre avec une trame ayant la plus petite taille, et y insère des signaux jusqu'à que la trame n'ait plus de place libre pour recevoir un signal donné. A ce stade, un choix est fait : l'augmentation de consommation de bande passante causée par l'accroissement de la trame de façon à pouvoir accepter le nouveau signal, est comparée à celle causée par la création d'une nouvelle trame ayant la plus petite taille capable d'accepter le nouveau signal. L'alternative qui augmente le moins la consommation de bande passante est choisie. A partir de la création d'une deuxième trame, chaque signal à insérer est essayé dans toutes les trames existantes qui ont encore de la place libre.

Ce travail est accompagné d'une évaluation de performance, qui nous montre l'efficacité des heuristiques sur les deux réseaux. Néanmoins, ce travail ne considère pas explicitement la faisabilité de l'ensemble de trames créées. Aucune vérification du respect de leurs échéances relatives n'est proposée.

2.5.2.3 Algorithmes proposés par Pop *et al.*

Le travail présenté dans [61] a aussi traité le problème du frame packing. En particulier, ces chercheurs ont travaillé sur la construction et la configuration faisable de trames dans un système distribué, composé

d'un réseau du type *time-triggered*, d'un autre du type *event-triggered*, et d'une passerelle qui lie les deux réseaux. Ce travail dépasse notre contexte, vu que nous allons nous concentrer sur un seul réseau du deuxième type.

L'autre différence dans ce travail par rapport à ceux présentés précédemment est dans le modèle de l'application. En effet, le modèle de l'application utilisé est basé sur les graphes acycliques de processus. Une application est représentée par un ensemble de graphes de processus, où chaque processus (ou noeud) est une séquence de calculs qui démarre quand toutes ses valeurs d'entrée sont disponibles. Les arcs dans un graphe représentent les dépendances entre les processus qu'ils inter-connectent, et prennent la forme de signaux si les processus inter-connectés sont distribués. De plus, tous les processus et signaux présents dans un graphe ont la même période d'activation ou production. Finalement, les échéances sont attribuées au graphe, et non à chaque processus. Le contexte de nos travaux n'observe pas ces propriétés et cette stratégie ne peut donc y être appliquée.

2.5.3 Conclusions sur la configuration de tâches et trames

Nous avons vu que l'algorithme d'Audsley est parfaitement adapté au calcul de priorités fixes pour qu'un ensemble de tâches soit faisable. Pour pouvoir faire des testes de faisabilité, cet algorithme a besoin des caractéristiques des tâches : temps d'exécution, période d'activation et échéance relative. Dans le cas d'un intergiciel automobile qui offre des services de communication, les caractéristiques des tâches de l'intergiciel peuvent dépendre des paramètres des trames. Nous avons donc analysé des travaux sur la construction et configuration des trames. Ces travaux essayent de minimiser la consommation de bande passante, et certains tentent de configurer les trames de façon à ce qu'une messagerie soit faisable.

Néanmoins, aucun des travaux présentés ne considère simultanément les propriétés suivantes :

- indépendance entre les caractéristiques des tâches applicatives,
- caractérisation des trames en prenant en compte la contrainte de fraîcheur des signaux, et
- vérification de la faisabilité de trames.

Le travail de Tindell [31] ne considère pas la deuxième propriété. D'ailleurs, aucune explication n'est donnée concernant le calcul des échéances relatives des trames, seulement la période (égale à la période des signaux). Ensuite, le travail de Norström [60] ne prend pas en compte la troisième propriété, la vérification de la faisabilité de trames. Finalement, le travail de Pop [61] considère que la période d'activation des tâches (ou processus) est fruit de leur relation de producteur et consommateur. Plus spécifiquement, les tâches qui coopère pour la production et la consommation d'un signal donné ont la même période d'activation. Ensuite, il y a le problème du modèle de l'application concernant la dépendance cyclique entre processus. Dans notre contexte, il peut y avoir une tâche x qui produit un signal qui est consommé par une tâche y, et celle-ci à son tour produit un signal qui sera ensuite consommé par la tâche x.

2.6 Conclusions

Dans ce chapitre nous avons présenté un état de l'art relatif au sujet traité dans cette thèse. Tout d'abord, nous avons détaillé quelques exemples d'intergiciels, certains issus d'un cadre général (académique, industriel, ...) et d'autres étant le résultat d'efforts faits au sein du domaine automobile. A ce sujet, nous avons vérifié que les propositions les plus adaptées à notre problème sont issues du domaine automobile, mais sont des spécifications qui ne traitent pas l'implémentation de l'intergiciel. Parmi ces propositions, nous nous sommes intéressés plus particulièrement à celle du consortium AUTOSAR.

Ensuite, nous avons regardé quelques méthodologies de développement d'intergiciels. Dans le contexte automobile nous remarquons la méthode proposée par le consortium AUTOSAR. Cette méthode prévoit une construction de l'intergiciel à deux étapes. Une première phase qui s'occupe de la génération des aspects système, dont il est possible de trouver la matrice de communication (paramètres du frame packing), et une deuxième étape qui génère l'implémentation (l'image binaire) et la configuration de l'intergiciel.

AUTOSAR n'offrant actuellement qu'une vue abstraite du développement de son intergiciel, nous avons décidé de regarder quelques travaux concernant, d'une part, l'implémentation logicielle d'un intergiciel, et, d'autre part, la configuration de celui-ci. Pour l'implémentation de l'intergiciel nous avons analysé la méthode basée sur les design patterns. Cette méthode permet la création d'un logiciel orienté objets qui présente des propriétés favorables à son utilisation dans un contexte automobile. Étant une méthode dont le résultat est spécifié à l'aide du formalisme UML, nous avons dû considérer le problème de configuration de l'intergiciel à partir du formalisme UML. Cette considération nous a amené à regarder des travaux qui pourraient nous permettre de trouver à partir d'UML, une forme d'expression de l'intergiciel qui soit plus facile à configurer, et dans le but aussi d'appliquer des méthodes de vérification. La forme d'expression choisie a été la tâche.

Nous avons donc alors étudié des travaux capables d'identifier des tâches à partir de modèles UML. Nous avons remarqué que les travaux perpétrés par Douglass et Saksena sont des solutions à ce problème. Ils proposent des stratégies qui à partir de l'ensemble des événements traités par le logiciel, identifient un ensemble de tâches. Finalement, ayant cet ensemble, il nous restait le point relatif à la configuration.

En ce qui concerne le paramétrage des tâches, et en particulier l'attribution de priorités, nous avons discuté l'algorithme optimal d'Audsley. Il est adapté à la tentative de calcul de priorités fixes faisables aux tâches. Nous avons aussi remarqué que cet algorithme effectue pendant son déroulement une vérification de la faisabilité des tâches. Pour cela, l'algorithme a besoin des caractéristiques temporelles de toutes les tâches, ce qui dans notre contexte veut dire les caractéristiques des tâches applicatives et de l'intergiciel. Néanmoins, les paramètres de celles de l'intergiciel peuvent être dépendantes de la configuration de la matrice de communication.

Pour finaliser cet état de l'art, nous avons listé trois travaux importants sur le sujet du frame packing. Leur problème réside dans le fait qu'aucun ne propose des heuristiques vérifiant trois propriétés importantes dans notre contexte : tâches applicatives indépendantes, caractérisation des trames à partir de la contrainte de fraîcheur des signaux, et vérification de la faisabilité des trames.

Ce chapitre a montré qu'une méthodologie de développement d'un intergiciel compatible avec celle proposée par le consortium AUTOSAR est nécessaire. Nous avons détaillé les besoins pour accomplir chacune des étapes composant la méthodologie, et nous pouvons déclarer que, même si quelques travaux existants peuvent nous aider pour l'implémentation de l'intergiciel, du travail reste à faire à ce sujet et surtout au niveau de la configuration.

Chapter 3

Présentation des contributions

3.1 Généralités sur les contributions

Le problème traité dans cette thèse concerne la définition d'une méthodologie de développement d'un intergiciel automobile centré sur les services liés à la communication. Nous avons remarqué que les propositions du consortium AUTOSAR (voir chapitre 2) sont celles qui guident la proposition faite dans cette thèse. Deux aspects importants pour le développement de l'intergiciel doivent être abordés :

- l'aspect implémentation, qui sera traité en définissant une architecture logicielle optimale capable de grouper les spécifications de code de l'intergiciel, ainsi que son déploiement, et
- l'aspect configuration, qui se chargera de déterminer les paramètres qui permettront à l'image de l'intergiciel déployé, de s'exécuter tout en respectant les contraintes temporelles imposées sur les tâches et les signaux.

Le méthodologie présentée dans ce document doit permettre la création d'un intergiciel qui atteint les objectifs énoncés dans le chapitre 1. La méthodologie que nous proposons fait l'objet de la partie II. Cette partie est rédigée en anglais ; aussi, nous en résumons ci-dessous chaque chapitre en français.

3.2 Résumé du chapitre 4 : introduction

Le chapitre 4 a comme objectif de présenter l'ensemble des services de communication fournis par l'intergiciel, et les principes méthodologiques pour son développement.

3.2.1 Services de communication fournis par l'intergiciel

L'intergiciel doit offrir des services permettant les échanges de signaux entre tâches applicatives distantes. Néanmoins, selon ce qui a été établi dans le chapitre 1, ces services doivent d'une part, masquer le protocole de communication utilisé, et, d'autre part, cacher la localisation des tâches applicatives qui reçoivent les signaux. L'ensemble de services de communication est composé de deux primitives accessibles aux tâches applicatives :

1. une primitive qui permet à une tâche applicative la soumission de la valeur d'un signal produit, afin que l'intergiciel se charge de sa transmission, et
2. une primitive qui permet à une tâche applicative d'obtenir la dernière valeur d'un signal reçue par l'intergiciel.

Quand la primitive 1 est utilisée, la tâche applicative fournit la nouvelle valeur produite et une identification du signal correspondant. Cette nouvelle valeur est sauvegardée dans l'intergiciel (en écrasant

l'ancienne), et elle sera mise en trame et transmise par l'intergiciel aux instants déterminés par la configuration de trames (voir chapitre *Configuration of the frames*, résumé en section 3.5).

Pour utiliser la primitive 2, la tâche applicative fournit un identificateur de signal et reçoit la valeur du signal stockée dans l'intergiciel, donc la dernière valeur reçue par celui-ci. Quand une nouvelle valeur est reçue par l'intergiciel, elle écrase l'ancienne valeur. Ainsi, les tâches applicatives peuvent ou non consommer toutes les valeurs reçues, tout dépend de leur période de consommation et des intervalles de temps entre les arrivées des valeurs.

La méthodologie proposée ne considère que ces deux services, qui sont toutefois conformes aux objectifs énoncés : ils sont indépendants vis-à-vis du protocole de communication et de la localisation des tâches. D'autres services pourraient être offerts par l'intergiciel. Le chapitre *Quality of Service monitoring through communication services* (résumé en section 3.7) analyse, par exemple, les conséquences liées à l'intégration à l'intergiciel, de services de notification aux tâches applicatives.

3.2.2 Principes méthodologiques pour le développement de l'intergiciel

L'objectif sous-jacent final de la méthodologie est le développement d'un système qui respecte toutes les propriétés qui lui sont imposées, en particulier les propriétés de performances. Ceci passe par une caractérisation (activation, priorité) des tâches applicatives, des tâches de l'intergiciel et des trames échangées entre les ECUs. Dans ce problème, les tâches applicatives sont données (charge CPU, règles d'activation) mais leur priorité n'est pas définie. Nous considérons, dans ce travail, que pour leur attribuer des priorités, il faut au préalable définir les caractéristiques des tâches qui exécutent l'intergiciel sur chaque ECU. Ces caractéristiques permettent ainsi une quantification exacte de l'interférence provoquée par l'intergiciel sur les tâches applicatives. Cependant, les caractéristiques des tâches de l'intergiciel dépendent du résultat de la configuration de trames. L'algorithme qui détermine cette configuration repose sur un calcul du pire temps de réponse pour toutes les tâches (intervalle de temps maximal entre l'activation d'une tâche et sa fin d'exécution), c'est-à-dire sur un modèle de tâche applicative qui exhibe, déjà, les priorités de celles-ci. Ce raisonnement fait apparaître le fait que chaque activité élémentaire (caractérisation de tâches applicatives, caractérisation de tâches de l'intergiciel, caractérisation de trames) dépend du résultat des autres activités. Pour surmonter cette boucle d'interdépendance, nous proposons un processus de configuration, illustré à la figure 3.1et détaillé rapidement par la suite. Les principes méthodologiques de notre proposition reposent sur deux points principaux :

- l'identification d'une architecture générique de l'intergiciel (nombre de tâches, règles d'activation de chaque tâche et fonctions de celles-ci),
- l'instanciation de l'architecture générique précédente pour une application et une distribution données (configuration des tâches de l'intergiciel, des tâches applicatives et des trames).

La figure 3.1 représente la partie de la méthodologie qui concerne l'instanciation de l'architecture générique, c'est-à-dire le développement de l'intergiciel pour une application donnée ; le rôle de ce processus est la génération des paramètres de configuration qui sont dépendants des contraintes imposées par l'application (tâches applicatives et signaux).

- La première étape de ce processus est l'activité qui configure les trames (composition, priorité, règles d'émission). Pour pouvoir attribuer des caractéristiques temporelles aux trames tout en s'assurant du respect de la fraîcheur des signaux, cette activité est conditionnée par l'architecture générique de l'intergiciel, appelée par la suite modèle d'implémentation de l'intergiciel. Ses données d'entrées sont les caractéristiques des signaux (taille, contraintes de fraîcheur) et certaines caractéristiques des tâches applicatives (règles d'activation et charge CPU). Ce problème est développé au chapitre 7 et résumé à la section 3.5.
- La deuxième étape consiste en la caractérisation complète, dans un premier temps, des tâches de l'intergiciel puis, des tâches applicatives. La caractérisation des tâches de l'intergiciel est également conditionnée par le modèle générique d'implémentation de celui-ci ; ses données d'entrée sont l'ensemble des trames configurées et faisables ainsi que les composants logiciels à déployer dans

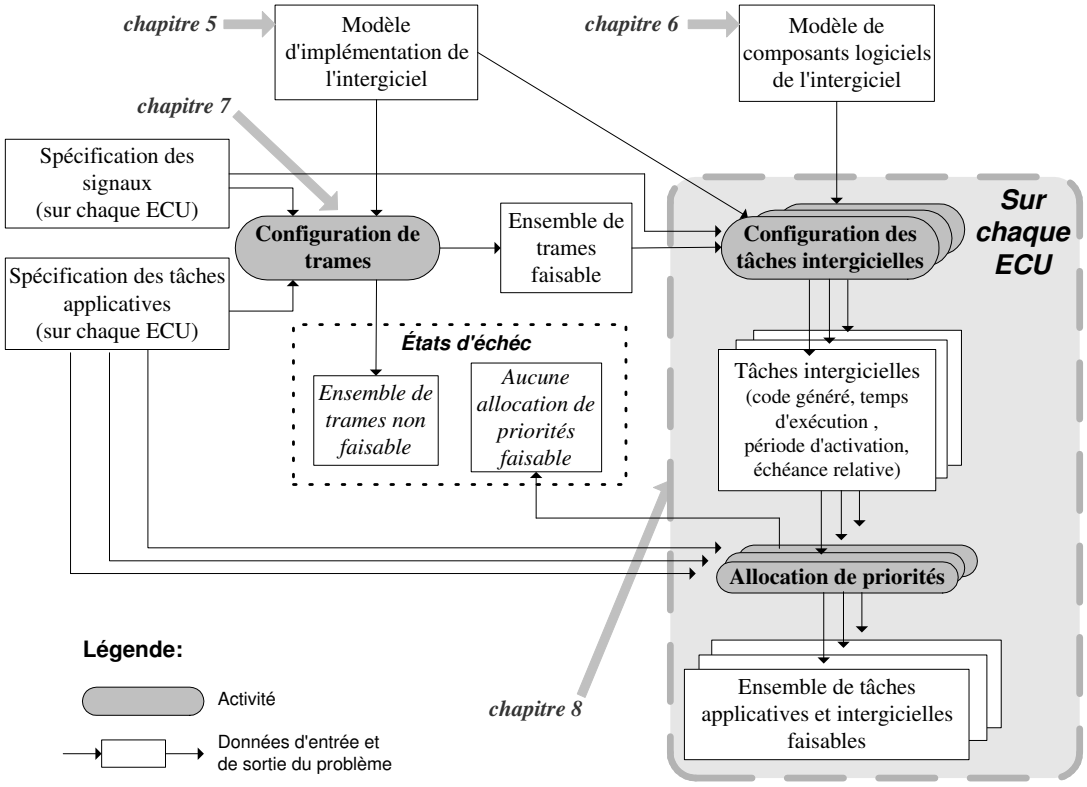


Figure 3.1: Processus de configuration des trames et des tâches à partir des contraintes temporelles.

ces tâches (méthodes, séquences de code). Le résultat est l'empreinte de l'intergiciel à savoir, un ensemble partiellement configuré de tâches (règles d'activation) ainsi que le code exécuté par ces tâches. Dans un deuxième temps, sont déterminées les priorités des tâches de l'intergiciel et des tâches applicatives. Le processus complet de cette étape est détaillé au chapitre 8 et résumé en section 3.6.

Le développement d'un intergiciel pour une application et une distribution données est conditionné, ainsi que nous l'avons vu ci-dessus, sur le modèle générique d'architecture de l'intergiciel (modèle d'implémentation et sur et utilise, comme donnée d'entrée, le modèle des composants logiciels de cet intergiciel tels que définis, par exemple, par un diagramme de classes UML. Ces modèles permettent l'identification des tâches, des composants logiciels génériques (méthodes ou séquences de code), et l'algorithme de déploiement des composants logiciels dans les tâches. La démarche, pour obtenir ces modèles est illustrée dans la figure 3.2. Cette démarche prend en compte les hypothèses préalables à notre étude et les contraintes imposées par le contexte d'exécution de l'intergiciel (système d'exploitation, plate-forme de communication, ...). Comment définir une architecture générique d'un intergiciel embarqué dans l'automobile fait l'objet du chapitre 5 (résumé en section 3.3) tandis que l'identification des composants logiciels génériques est traitée au chapitre 6 (résumé en section 3.4).

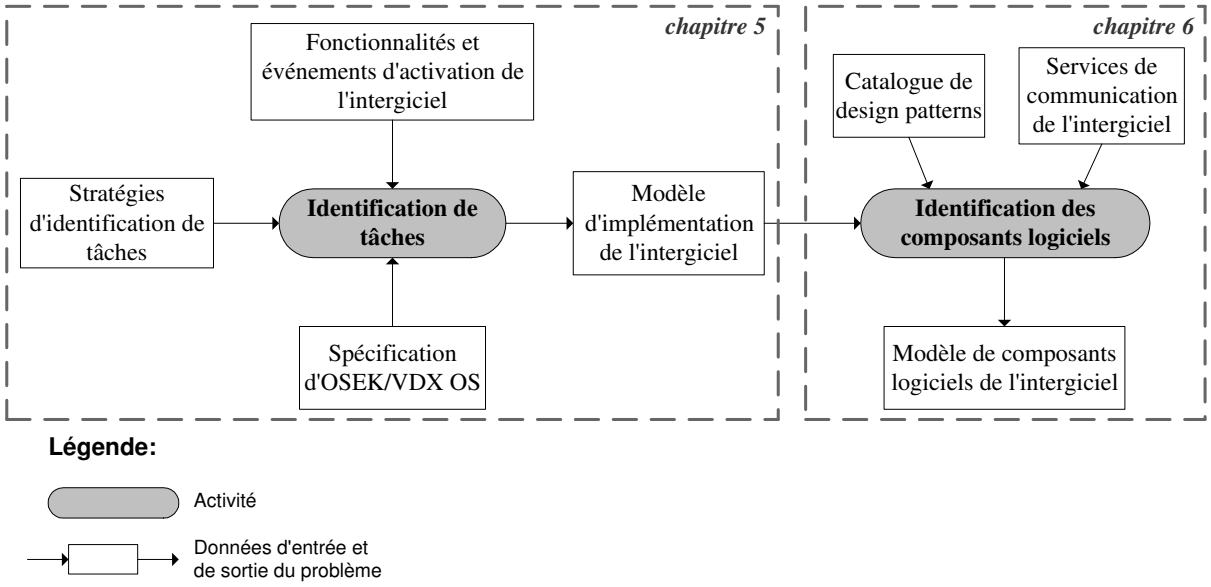


Figure 3.2: Identification d'une architecture de composants logiciels et de tâches de l'intergiciel.

3.3 Résumé du chapitre 5 : implementation model of the middleware

Le chapitre 5 montre comment construire le modèle d'implémentation de l'intergiciel en termes de tâches. Le problème est l'identification des tâches chargées d'exécuter les services de communication. Comme énoncé dans le chapitre 1, l'intergiciel s'exécute de façon asynchrone par rapport aux tâches applicatives ce qui signifie qu'il est réalisé par des tâches indépendantes de celles-ci. Résoudre le problème posé demande de se poser, alors, les questions suivantes :

- combien de tâches représentent l'intergiciel sur chaque ECU,
- quel est le rôle de chaque tâche dans la réalisation des services de communication, et

- en ce qui concerne la stratégie d'activation, à quel modèle de tâches appartient chacune des tâches de l' intergiciel.

Pour répondre à ces questions, il s'agit d'analyser l'ensemble de fonctionnalités que l'intergiciel doit accomplir, d'envisager et comparer des stratégies d'identification de tâches, et, finalement, de prendre en compte, d'une part, les spécificités du système d'exploitation utilisé, OSEK/VDX OS et, d'autre part, les événements qui activent les fonctions de l'intergiciel.

3.3.1 Fonctions de l'intergiciel : bibliothèque de fonctions, tâches et événements d'activation

Pour pouvoir exécuter les services de communication présentés dans la section 3.2, l'intergiciel doit accomplir quatre fonctions élémentaires :

1. Production de signal : recevoir et mémoriser la valeur d'un signal produit par une tâche applicative,
2. Construction et transmission d'une trame : lecture des valeurs mémorisées des signaux, construire la trame avec ces valeurs, et demander la transmission de la trame,
3. Réception et traitement de trame : recevoir la trame, la dépaqueter et mémoriser les valeurs dans les signaux, et
4. Consommation de signal : lire la valeur mémorisée d'un signal reçu et la délivrer à la tâche applicative.

Les fonctions 1 et 4 sont activées par simple demande des tâches applicatives. Pour des raisons de performance, nous pouvons considérer que ces fonctions sont réalisées par une bibliothèque intergicielle, et sont exécutées par les tâches applicatives. Comme l'intergiciel s'exécute de façon asynchrone par rapport aux tâches applicatives, les fonctions de transmission et réception de trames (fonctions 2 et 3) ne sont pas synchronisées avec les fonctions de production et consommation de signaux. Ainsi, les deux fonctionnalités, construction et transmission de trames (fonction 2), et réception et traitement de trames (fonction 3) sont réalisées de manière asynchrone avec les tâches applicatives par des tâches de l'intergiciel.

Les fonctions de l'intergiciel sont activées par des événements clairement identifiés : l'expiration des périodes de transmission de trames, et l'arrivée des trames. Il s'agit, alors, d'identifier le mécanisme OSEK/VDX OS qui permet d'activer la ou les tâches responsables de l'exécution de ces fonctions à l'occurrence d'un des ces événements. Parmi les mécanismes offerts par ce système, ceux qui peuvent être directement associés aux événements se produisant dans le contexte de l'intergiciel sont les interruptions matérielles (contrôleur réseau) et les alarmes temporelles. D'une part, l'expiration d'une période de transmission de trame déclenche une alarme temporelle qui doit lancer l'exécution de la fonction de construction et transmission de trames. D'autre part, l'arrivée d'un trame déclenche une interruption du contrôleur réseau qui doit lancer l'exécution de la fonction de réception et traitement de trames.

3.3.2 Identification de tâches de l'intergiciel

Après avoir sélectionné les fonctions qui sont exécutées par des tâches et les types d'événements déclencheur de ces fonctions, la question qui se pose est combien de tâches doivent-elles être construites pour une application donnée. Partant des travaux de Douglass [54] et Saksena [55] introduits dans le chapitre 2, nous avons analysé plusieurs stratégies d'identification de tâches possibles :

- une tâche pour chaque événement - cette stratégie attribue une tâche pour chaque événement d'activation différent, ce qui dans notre contexte signifie qu'il y aura sur chaque ECU une tâche pour chaque trame reçue, et une tâche pour chaque période de transmission de trames différente,
- une tâche pour chaque type d'événement - cette stratégie identifie une seule tâche activable à toutes les échéances de périodes de transmission de trames, et une autre tâche activable par toutes les arrivées de trames reçues par cet ECU,

- une tâche pour chaque objectif différent attribué au signal - l'idée de cette stratégie est d'identifier des tâches selon la sémantique des signaux (*e.g.* une tâche qui traite tous les signaux relatifs à la vitesse des roues, une tâche qui s'occupe des signaux concernant le mode de marche, ...).

Le choix de la meilleure stratégie doit prendre en compte les propriétés d'OSEK/VDX OS. Ce système fournit plusieurs classes de conformité qui impliquent des besoins en ressources différents. Par souci de minimisation de la consommation de ressources et pour assurer au maximum la conformité à OSEK/VDX OS, nous proposons de minimiser le nombre de tâches réalisant les services de communication de l'intergiciel, de façon à permettre l'exécution du plus grand nombre possible de tâches applicatives. De ce fait, nous avons sélectionné la deuxième stratégie, c'est-à-dire celle qui identifie une tâche pour chaque type d'événement. Cette stratégie reste valide même quand des nouvelles trames sont ajoutées à la configuration de trames. De plus, elle permet de ne mettre en oeuvre qu'une tâche OSEK/VDX OS. En effet, ce système fournit la possibilité d'utilisation de routines de service d'interruptions. Une telle solution, et non une tâche OS sera donc associée à la fonction en charge de la réception et traitement de trames.

3.4 Résumé du chapitre 6 : software model of the middleware

Ce chapitre apporte une solution à la conception des composants logiciels de l'intergiciel, c'est-à-dire du code qui sera exécuté par les tâches identifiées précédemment (chapitre 5 résumé en section 3.3). Il s'agit de déterminer :

- quelles sont les classes et les objets qui vont composer l'architecture logicielle de l'intergiciel, et
- à partir de la spécification de ces classes, quelles sont les séquences de code (méthodes) que chaque tâche intergicielle doit exécuter (déploiement).

Le processus de conception repose sur des *design patterns* orientés objets de façon à augmenter la réutilisation des composants logiciels, et faciliter leur maintenance et leur modification. Ce processus est présenté au chapitre *Software model of the middleware* (résumé en section 3.4).

L'objectif est, tout d'abord, de sélectionner les meilleurs *design patterns* pour ce problème, puis de les composer afin d'obtenir un diagramme de classes représentant la structure logicielle. Ensuite, le but est de considérer le modèle d'implémentation proposé au chapitre précédent (5) et d'identifier, à partir du diagramme de classes, les séquences de code qui seront exécutées par chaque tâche.

3.4.1 Les *design patterns* pour l'intergiciel

Les design patterns que nous avons identifiés comme pertinent pour la conception de l'intergiciel sont :

- *Active object* [42] : ce pattern permet de découpler l'invocation des services intergiciels de son exécution. D'une manière concrète, l'invocation du service est faite dans une tâche qui référence une bibliothèque intergicielle chargée de recevoir les requêtes, pendant que l'exécution de ce service est réalisée par une tâche séparée. Ce pattern permet de gérer l'asynchronisme entre l'application et l'intergiciel.
- *Adapter* [41] : l'intergiciel est conçu de façon à être compatible avec n'importe quel protocole d'accès au médium (*Medium Access Control*) de communication embarqué dans un véhicule. Le fait que chaque protocole fourni un ensemble différent de services et interfaces conduit à une inter-connexion laborieuse entre eux et l'intergiciel. Le pattern Adapter permet à une classe d'utiliser une autre classe, même si l'interface de cette dernière n'est pas celle attendue par la première. Dans notre contexte, nous définissons une classe abstraite qui fournit une interface de communication standard et la plus générale possible. Une classe concrète pour chaque protocole d'accès au médium se chargera de traduire l'interface du protocole vers celle de la classe abstraite. Cette solution permet à l'intergiciel de s'abstraire de l'hétérogénéité des plates-formes de communication.

- *Observer* [41] : la classe concrète pour l'accès au médium de communication n'est pas responsable de la construction ni du traitement de trames. Ce travail est le rôle d'une classe distincte, que nous appelons classe principale de l'intergiciel. De ce fait, quand une trame est construite par cette classe, celle-ci doit être récupérée par la classe d'accès au médium de communication afin de demander sa transmission. De la même façon, quand une trame est reçue par la classe d'accès au médium, la classe principale doit se la procurer afin de la traiter. Ces deux classes sont donc fortement liées à cause des échanges de trames. Néanmoins, elles doivent pouvoir évoluer indépendamment. Ce pattern aide à atteindre cet objectif : il est utilisé quand nous voulons créer une dépendance faible entre classes mais de manière à que quand l'état d'une d'elles change, l'autre (l'observatrice) soit immédiatement notifiée.

- *Asynchronous completion token* [42] (ACT) : quand la classe principale construit une trame et notifie la classe d'accès au médium de communication, normalement elle donne plusieurs données : la trame, la taille, et un identificateur. Le pattern ACT aide à encapsuler les données échangées entre ces deux classes. Quand une trame est construite ou reçue, un objet *token* encapsule les données liées à la trame.

3.4.2 Le diagramme de classes et le modèle d'implémentation

A ce stade, nous pouvons construire un diagramme de classes comme résultat d'un processus de composition des design patterns sélectionnés. A partir de ce diagramme de classe, nous identifierons les séquences de code exécutées par chaque tâche de l'intergiciel ainsi que celles de la bibliothèque intergicielle, exécutées par les tâches applicatives.

Diagramme de classes : composition des design patterns

Pour effectuer la composition des design patterns, nous avons choisi dans la description structurelle de chaque pattern la classe qui représente le noyau de l'intergiciel. Cette classe est présente dans chaque pattern utilisé, et donc, ces exemplaires de la même classe ont été fusionnés dans une seule classe.

Séquences de code des tâches intergicielles

Le code à exécuter par les tâches est spécifié dans le diagramme de classes (méthodes). Il est donc nécessaire de sélectionner le code à attribuer à chaque tâche définies dans la section précédente, et d'identifier les objets qui doivent être créés par chacune d'elles. Comme une partie du diagramme est allouée à la bibliothèque de l'intergiciel, nous devons aussi identifier les objets que celle-ci devra créer. Ainsi, prenant en compte toutes les fonctions de l'intergiciel (voir section 3.3), l'identification d'objets est faite pour les fonctions suivantes :

- la sauvegarde des valeurs des signaux produits par les tâches applicatives,
- la remise aux tâches applicatives des valeurs des signaux demandés,
- la construction et transmission des trames, et
- la réception et traitement des trames.

Pour chacune des ces fonctions, un diagramme de séquence aide à déterminer le code qui doit être exécuté. Ces diagrammes de séquence sont également présentés et commentés dans le chapitre 6.

3.5 Résumé du chapitre 7 : configuration of the frames

Après avoir déterminé les objets à créer, les séquences de code à exécuter (diagrammes de classes et de séquence), le problème consiste à configurer précisément les différents acteurs (trames et tâches) de l'intergiciel. Ainsi que nous l'avons vu dans la présentation de la méthodologie, ceci passe par deux étapes :

- configurer les trames à partir des signaux et du comportement des tâches applicatives (chapitre 7)
- configurer les tâches de l'intergiciel et les tâches applicatives (chapitre 8).

Ces configurations se font sous contraintes de respect des propriétés temporelles issues de l'application et reposent donc sur des analyses d'ordonnançabilité.

L'objectif du chapitre 7 est de proposer deux algorithmes pour traiter la première étape, c'est-à-dire pour la configuration de trames, appelée aussi *frame packing*. Concrètement, le problème à résoudre est :

- quelles sont les trames à envoyer par chaque ECU,
- quels sont les signaux qui composent chacune des trames,
- quelles sont les caractéristiques des trames (taille, période de transmission, et échéance relative), et
- quelle est la priorité de chaque trame pour le protocole d'accès au médium du réseau.

La fonctionnalité de l'intergiciel responsable de l'exécution en ligne du frame packing agit selon une configuration générée par un algorithme hors-ligne. Les caractéristiques des trames générées par l'algorithme sont déterminées de manière à remplir trois objectifs :

- minimiser de la consommation de bande passante,
- garantir la faisabilité de la messagerie (ensemble des trames sur le réseau), et
- assurer le respect des contraintes de fraîcheur associées aux signaux.

3.5.1 Contraintes de fraîcheur et échéance relative d'une trame

L'intergiciel doit s'assurer que n'importe quel signal respecte ses contraintes de fraîcheur à tout instant où il peut être consommé. Une valeur d'un signal qui respecte ses contraintes de fraîcheur à l'instant t, est une valeur dont l'âge à cet instant est inférieur, ou égal, à un âge donné (fraîcheur limite). L'âge d'une valeur d'un signal à l'instant t, est défini comme étant l'intervalle de temps entre t et l'instant d'activation de l'instance de la tâche qui a produit la valeur. Ainsi, l'âge maximal qu'un signal peut atteindre, est la plus longue durée entre l'activation d'une instance de la tâche produisant une valeur, et l'instant où le signal est mis à jour par cette valeur du côté consommateur (mise à disposition de la valeur suivante). L'approche utilisée par les algorithmes de frame packing proposés est d'assurer que l'âge maximal de tout signal, sera toujours inférieur ou égal à sa fraîcheur limite.

L'âge maximal concret d'un signal dépend de l'exécution de la tâche productrice, des tâches intergicielles des côtés producteur et consommateur, et du déroulement de la transmission sur le réseau du signal inséré dans une trame. L'échéance à attribuer à une trame qui contient un signal, est donc égale à l'intervalle de temps qui reste quand de la fraîcheur du signal nous soustrayons la quantité maximale totale de temps que le signal passe sur les ECUs producteur et consommateur. Quand une trame contient plusieurs signaux, l'échéance est égale au plus petit intervalle de temps calculé pour chaque signal. Enfin, pour chaque ECU, il suffit de considérer la fraîcheur la plus stricte imposée par tous les consommateurs de chacun des signaux de la trame (dans le cas où plusieurs tâches dans un même ECU consomment le même signal mais en imposant des fraîcheurs différentes).

Néanmoins, à ce stade nous n'avons toutes les caractéristiques des tâches applicatives et intergicielles, qui sont nécessaires pour le calcul de la quantité maximale totale de temps qu'un signal passe sur les ECUs producteur et consommateur. Ces caractéristiques dépendent de la configuration de trames, ce qui donne origine à la boucle d'interdépendance énoncée précédemment (voir section 3.2). Nous considérons alors le comportement "pire cas" de la part des tâches applicatives et intergicielles. L'expression de ces comportements est utilisée pour vérifier la faisabilité d'une messagerie dans les algorithmes de frame packing proposés.

3.5.2 Algorithmes de frame packing

Nous avons proposé deux algorithmes de frame packing. L'annexe B présente des évaluations de performances de ces algorithmes (en termes de minimisation de la bande passante et d'impact des contraintes de fraîcheur sur la capacité à trouver des solutions). Les algorithmes proposés sont décrits rapidement ci-dessous.

Algorithme *Bandwidth-Best-Fit decreasing*

L'algorithme *Bandwidth-Best-Fit decreasing* (BBFd) repose sur les approches de résolution hors-ligne des problèmes de *bin packing*. Dans ce genre de problèmes, l'objectif est de minimiser le nombre de boîtes, étant donné un ensemble d'objets de tailles différentes qui doivent être placés dans ces boîtes. Dans notre contexte, le but est de minimiser la consommation de bande passante sans tenir compte du nombre de trames. L'idée clé de cet algorithme est, à chaque étape, de placer un signal dans la trame qui minimise au plus la consommation de bande passante supplémentaire causée par le signal.

L'algorithme construit une seule solution dont les trames ont des caractéristiques qui assure le respect des contraintes de fraîcheurs des signaux transportés. La faisabilité des trames est vérifiée à l'aide de l'algorithme d'Audsley, introduit dans le chapitre 2. Si la solution est faisable, alors nous appliquons une procédure d'optimisation locale. Dans le cas d'échec au niveau de la faisabilité, l'algorithme BBFd exécute quelques transformations sur les trames de façon à isoler les signaux les plus exigeants (ceux qui ont les contraintes de fraîcheurs les plus strictes).

Algorithme *Semi-Exhaustive*

L'algorithme *Semi-Exhaustive* (SE) effectue un parcours exhaustif de l'espace des solutions. Il commence par construire sur chaque ECU la liste complète de tous les ensembles de trames possibles. Au contraire de l'algorithme BBFd qui peut être utilisé sur tous les problèmes, l'algorithme SE est seulement applicable sur des problèmes de moindre taille (moins de 12 signaux par ECU). Ensuite, l'algorithme trie la liste sur chaque ECU dans l'ordre croissant de la consommation de bande passante. Puis, il trie les ECUs dans l'ordre croissant du premier ensemble de trames, celui qui minimise le plus la consommation de bande passante sur chaque ECU. Finalement, l'algorithme SE construit des solutions possibles avec un ensemble de trames de chaque ECU, et arrête dès qu'une solution est faisable (algorithme d'Audsley).

3.6 Résumé du chapitre 8 : configuration of the applicative and middleware tasks

La configuration des tâches fait l'objet du chapitre 8, *Configuration of the applicative and middleware tasks*.

Dans un premier temps, nous traitons les **tâches de l'intergiciel**. Il s'agit de la deuxième étape du processus de configuration de l'intergiciel, c'est-à-dire la caractérisation des tâches de l'intergiciel sur chaque ECU. Plus précisément, nous devons déterminer, pour chacune d'elles, le temps d'exécution, la période d'activation, l'échéance relative, et la priorité de celles-ci. Ces paramètres sont calculés en utilisant la spécification des signaux et la configuration de trames. Cependant, pour déterminer le temps d'exécution de chaque tâche, l'ensemble des signaux et des trames que l'intergiciel doit manipuler n'est pas suffisant. Il faut connaître également les séquences de code qui seront exécutées par chacune des tâches. A cette fin, cette activité utilise le modèle de composants logiciels de l'intergiciel, établi dans la partie générique de la méthodologie. La caractérisation des tâches de l'intergiciel est différente pour chacune des deux tâches :

- **Tâche exécutant la fonction de construction et transmission de trames**. Les trames peuvent avoir des périodes de transmission différentes, et, dans ce cas, cette tâche, qui est une tâche OSEK/VDX OS, doit avoir un profil d'activation qui respecte chaque période. En conséquence, nous proposons d'utiliser des modèles de tâches comme le modèle *multiframe* [62] et le modèle *generalized multiframe* (GMF) [63].

- **Tâche exécutant la fonction de réception et traitement de trames.** Dans un contexte où le réseau CAN est utilisé, les ECUs connectés au réseau ne sont pas synchronisés. Ceci implique que l'intervalle de temps entre les arrivées de deux trames quelconques n'est pas constant. Cette tâche, implémentée sur OSEK/VDX OS sous la forme d'une routine de service d'interruption, est donc considérée comme étant sporadique [64]. Dans ce type de modèle, chaque tâche est caractérisée par un temps d'exécution, un intervalle minimum entre activations successives (assimilé, dans les calculs à une période d'activation, par la suite), et une échéance relative unique. Vu qu'une tâche sporadique ne suit pas un taux d'activation strict, sa période d'activation est fixée au plus petit intervalle de temps entre deux activations consécutives.

Dans un deuxième temps, nous montrons, dans ce chapitre, comment spécifier une allocation de priorités faisable pour les **tâches applicatives**. La méthode proposée afin d'essayer de déterminer une allocation de priorités faisables pour les tâches applicatives est l'algorithme d'Audsley [58], introduit dans le chapitre 2. Durant son déroulement, cet algorithme effectue des analyses de faisabilité, et procède donc à des calculs du pire temps de réponse. Pour cela, il faut connaître les valeurs des caractéristiques de toutes les tâches présentes sur l'ECU. En ayant maintenant toutes les tâches intergicielles bien caractérisées, toutes les conditions nécessaires à l'utilisation de l'algorithme d'Audsley sont réunies. Plus de détails sur l'utilisation de cet algorithme peuvent être trouvés dans le chapitre 8.

3.7 Résumé du chapitre 9 : quality of service monitoring through communication services

L'objectif de ce chapitre est d'étudier les conséquences de l'intégration de services de notification sur les modèles d'implémentation et de composants logiciels. En particulier, ces services doivent fournir aux tâches applicatives des informations concernant l'état de transmission et réception des signaux produits et consommés. Concrètement, nous nous sommes intéressés à la validation de notre méthodologie, en vérifiant que les modèles d'implémentation et de composants logiciels restent valides dans un nouveau contexte en termes de l'ensemble de services fournis par l'intergiciel. De plus, du fait que les services proposés sont les mêmes que ceux considérés par le standard OSEK/VDX COM, nous avons voulu vérifier que leur implémentation était possible dans notre contexte.

3.7.1 Services de notification à intégrer

Les services de notification intégrés à l'intergiciel sont :

- un service qui indique à une tâche applicative si la transmission des valeurs d'un signal produites précédemment a respecté les contraintes de fraîcheur ou non, et
- un service qui informe une tâche applicative si une valeur d'un signal disponible pour la consommation respecte la contrainte de fraîcheur ou non.

Ces services sont appelés d'état de transmission et réception respectivement. Les services fournis initialement par l'intergiciel (voir section 3.2) ont été modifiés de manière à retourner ces états.

3.7.2 Conséquences sur le modèle d'implémentation de l'intergiciel

Les nouveaux services sont fortement liés aux services de communication introduits dans la section 3.2. Pour cette raison, nous avons pondéré la possibilité d'implémenter les services de notification en utilisant la tâche et la routine de service d'interruption (ISR) qui s'occupent de la communication. Des nouveaux types d'alarmes temporelles devraient être ajoutées au système. Néanmoins, la spécification d'OSEK/VDX OS ne permet pas l'activation de l'ISR à partir d'une alarme temporelle. En plus, l'autre tâche de l'intergiciel perdrait son schéma d'activation périodique, rendant difficile l'analyse d'ordonnançabilité.

Nous avons alors essayé d'implémenter les services de notification en utilisant un ensemble de tâches différent, toujours activées par des alarmes temporelles. Cette solution, dont l'implémentation serait possible sur OSEK/VDX OS, avait l'avantage de valider notre méthodologie, vu que chaque nouvelle tâche serait activée par un type d'événements différent. Cependant, si la nouvelle tâche responsable par l'état de transmission serait préemptée en début d'exécution, alors la vérification du respect des contraintes de fraîcheur ne serait pas précise.

Il nous a semblé évident que la solution serait de diminuer le niveau de qualité de service requis. Ainsi, nous avons transformé l'ensemble de services de notification présentés dans la section 3.7.1 dans celui-ci :

- un service qui informe une tâche applicative si les valeurs d'un signal produites précédemment ont été déjà utilisées pour construire une trame ou non, et
- un service qui notifie une tâche applicative si la valeur d'un signal a été mise à jour depuis la dernière consommation ou non.

Notez que cette solution valide aussi notre méthodologie. Du fait que l'ensemble de tâches de l'intergiciel sur chaque ECU n'a pas changé, le modèle d'implémentation détaillé en section 3.3 reste valable. Remarquez toutefois que les services proposés par OSEK/VDX COM n'ont pas pu être implémentés, à cause d'une possible perte importante au niveau de la précision de la réponse offerte.

3.7.3 Conséquences sur le modèle de composants logiciels

Le modèle de composants logiciels de l'intergiciel n'est pas modifié profondément. L'ensemble de classes est le même, mais certaines méthodes doivent être changées de façon à exprimer l'existence des services de notification. Ces méthodes sont facilement identifiables à cause de l'utilisation des design patterns pour la construction du modèle. Les méthodes à modifier appartiennent aux classes *Proxy_Scheduler*, *Signals*, et *Core*. La liste des méthodes modifiés, bien comme les changements à effectuer, sont présentés de manière détaillée dans le chapitre 9.

3.8 Conclusion

Dans ce chapitre, nous avons résumé les contributions de notre travail qui sont présentées en détail dans la deuxième partie de ce document. Nous avons donné les grandes lignes de la méthodologie de développement des services de communication d'un intergiciel embarqué dans l'automobile. Nous avons montré que celle-ci est divisée en deux parties. La première s'occupe de spécifier les modèles génériques des composants logiciels et de leur déploiement sur une plate-forme donnée. La deuxième partie présente les solutions que nous proposons pour instancier ces modèles génériques pour une application spécifique. En particulier, nous montrons qu'il faut générer les paramètres temporels nécessaires à la configuration de l'intergiciel, des trames, et des tâches applicatives, de façon que toutes les contraintes sur les tâches et sur les signaux soient respectées. Nous allons ensuite entrer dans la deuxième partie de cette thèse, où cette méthodologie est présentée et chaque activité qui la compose est détaillée.

Part II

Contributions

Chapter 4

Introduction

4.1 Introduction

In chapter 1, the middleware that is proposed to be developed was introduced. In particular, some specific characteristics that the middleware must present were listed. Among them, two are enhanced in the following:

- the middleware must provide a standard set of communication services, and
- the middleware executes asynchronously from applicative tasks

The goal of this chapter is to present to the reader the set of communication services that are going to be provided by the middleware, and the methodology proposed for its development.

The communication services must allow signals to be exchanged between applicative tasks independently from their location. The methodology must firstly, permit the development of a middleware that implements the communication services in an asynchronous way. It handles signals, produced and consumed by applicative tasks, and handles frames, sent and received by the network controller. Secondly, it must construct the set of frames that are going to be transmitted over the network, such that, the bandwidth consumption is minimized. Finally, it has to determine configuration parameters that allow applicative tasks, the middleware itself, and the network frames to meet their deadline constraints, and signals freshness constraints to be respected.

The objective, for what concerns the implementation and deployment of the middleware, is:

- to identify a set of tasks capable of implementing the communication services,
- to express the sequences of code that will be executed by each task on each ECU, and
- to calculate the configuration parameters that will permit to attain the feasibility of the tasks while respecting the timing constraints of the signals.

At the end, one should have a middleware whose code is established and able to be generated, and whose configuration guarantees feasibility on each ECU.

4.2 Communication services provided by the middleware

The distribution of automotive functions through the vehicle requires the exchange of signals between the applicative tasks implementing those functions. Hence, the middleware must provide a set of communication services allowing the transmission of signals over the network, and, if possible, masking the communication protocol. Moreover, even if current in-vehicle network buses use a frame transmission mode based on broadcast, for portability reasons the middleware should hide the distribution of applicative tasks. Therefore, the set of communication services must also be fully independent of the location of the tasks. The focus is then made on a set of two basic communication services:

1. a service allowing applicative tasks to provide produced signal values for transmission, and
2. a service allowing applicative tasks to obtain the last value received of a signal to be consumed.

When using service 1, application level tasks provide a new value and the corresponding signal's identification, as illustrated in figure 4.1. The new value is stored (deleting the previous one), and will be packed into a frame and transmitted by the middleware at the time instants determined by the frames configuration parameters (instants t_a and t_b in figure 4.1).

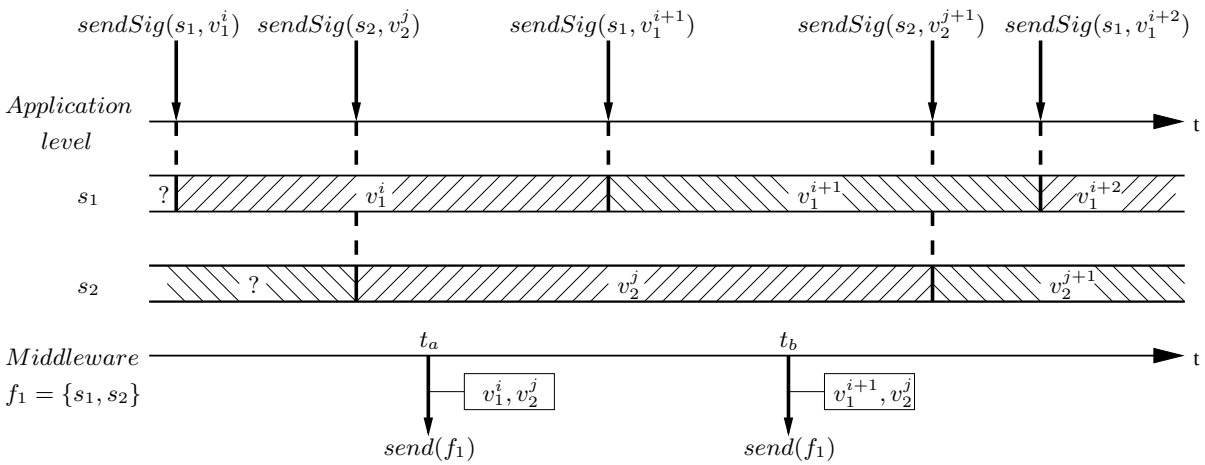


Figure 4.1: Asynchronous execution of the application level and the middleware: production and transmission of signals.

To use service 2, application level tasks provide a signal's identification and receive the latest value received, as exemplified in figure 4.2. When a new value is received and stored, the previous one is erased. Thus, applicative tasks may consume or not all received values, depending on their consumption rate and on the time intervals between the arrivals of the signal. For instance, in figure 4.2, value v_1^{i+1} of signal s_1 is never consumed.

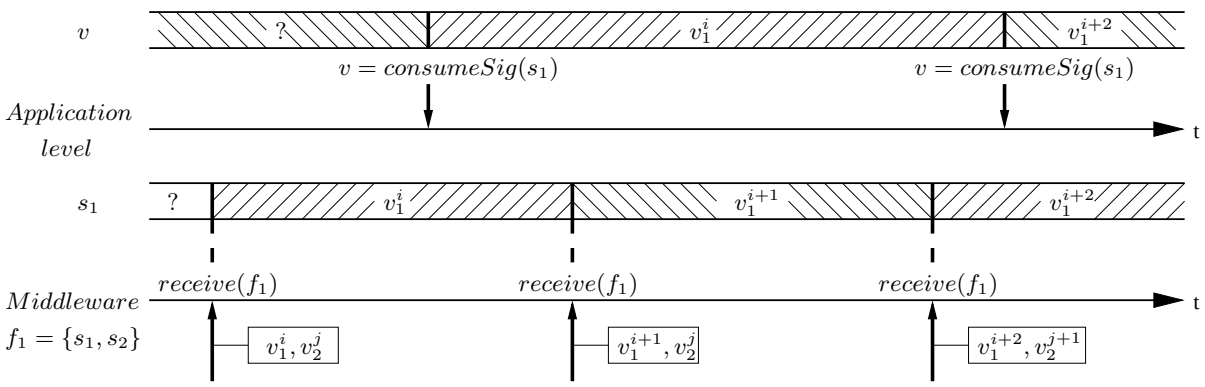


Figure 4.2: Asynchronous execution of the application level and the middleware: reception and consumption of signals.

The methodology proposed in this document considers only these two services. They are compliant with the requirements previously stated: they are independent of the communication protocol and the location of the tasks. However, other services could be proposed. For instance, the middleware could offer a service allowing a producer task to be notified if the last produced value was transmitted at a

time instant that will violate the signal's freshness constraint. Or, a service that notifies a consumer task that the value available no longer respects the demanded freshness constraint (frame carrying next value did not arrived due to an electro magnetic interference). Although the methodology does not take into account these services, chapter 9 will discuss the consequences of adding them to the middleware.

4.3 Methodology for the development of the middleware

One of the goals of the methodology is to assign feasible priorities to applicative tasks. In order to achieve this goal, one needs to have the timing characteristics of the middleware executing on each ECU. With these characteristics it is possible to precisely quantify the interference of the middleware on the applicative tasks. However, the characteristics of the middleware depend on the result of the frames configuration parameters. But, the algorithm that calculates these parameters must have the knowledge of the worst-case response times of applicative tasks and middleware, in order to assign timing characteristics to each frame (the relative deadline in particular) that guarantee the signals freshness constraints. To overcome this dependency cycle, the algorithm presented in figure 4.3 is proposed.

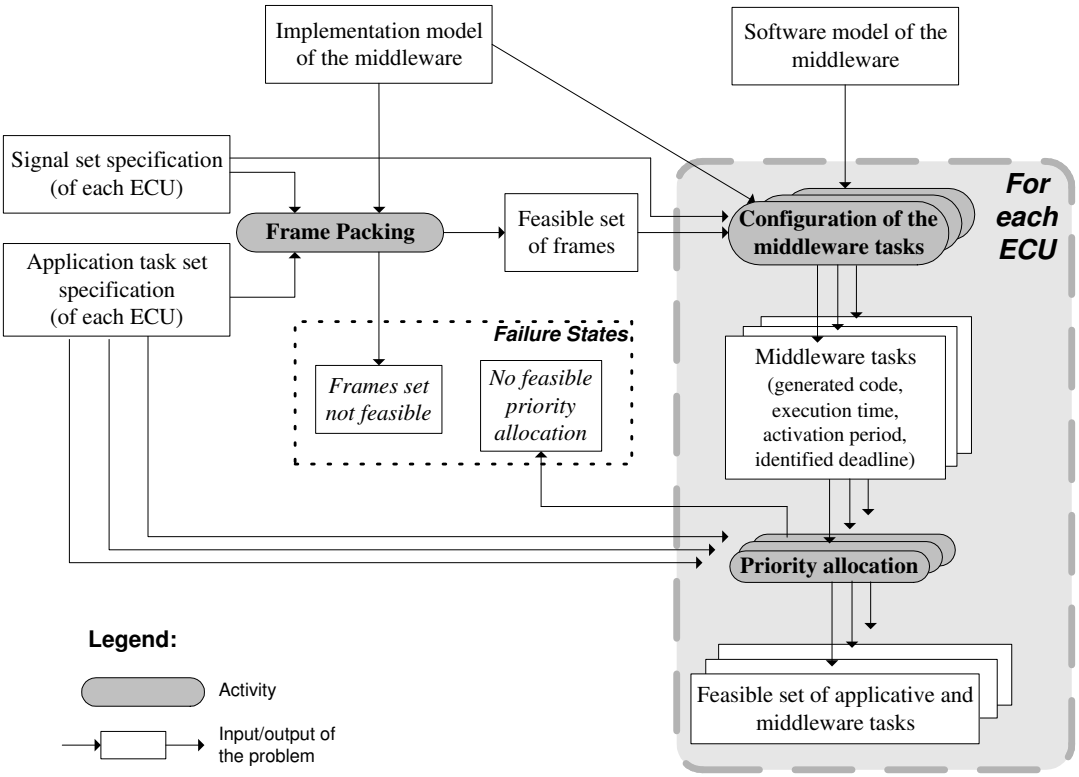


Figure 4.3: Algorithm representing the part of the methodology whose purpose is to configure the frames and the tasks according to timing constraints.

This algorithm represents one of the parts of the methodology that is proposed for the development of the middleware. As one can see, this part is composed of the activities that generate data dependent of the timing constraints imposed on each ECU by the applicative tasks and the signals. The algorithm begins by the frame packing activity, which determines the frames configuration parameters. This is the activity of the algorithm where the dependency cycle above mentioned will be broken. For this purpose, the frame packing uses the specification of the applicative tasks and signals. However, to be able to assign timing characteristics to each frame that guarantee the signals freshness constraints, this activity must be aware of the implementation model of the middleware.

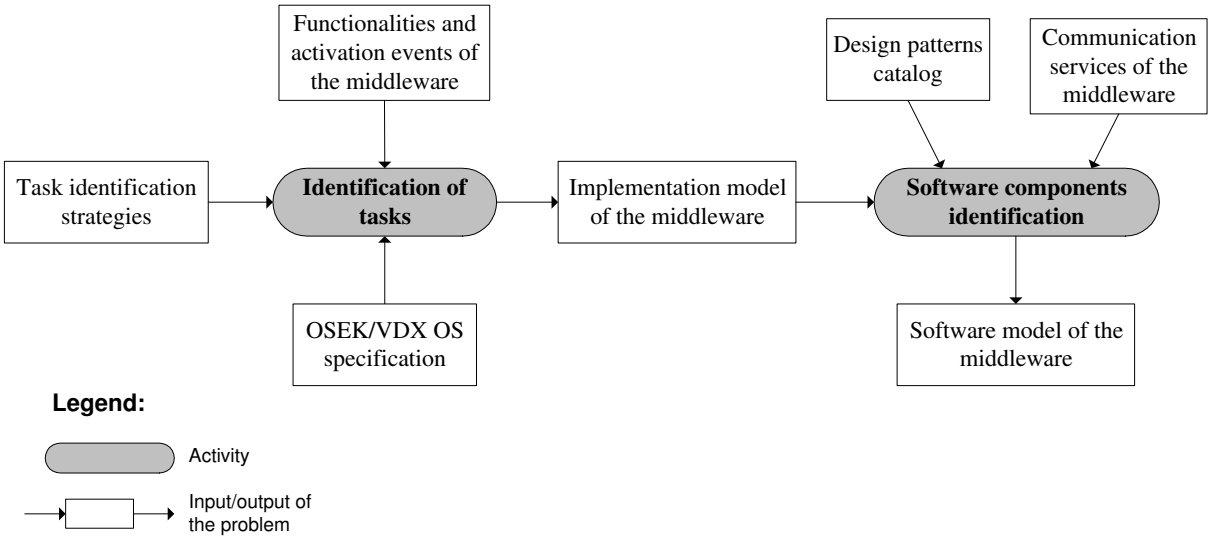


Figure 4.4: Principles for the identification of the software components architecture and the task architecture of the middleware.

The implementation model, as well as the software model of the middleware, are used by the activities presented above. These models aim to identify the tasks, the generic software components, and the deployment algorithm of the software onto the tasks. The underlying part of the methodology is illustrated in figure 4.4. It is generic and independent of any specific real-time distributed application. Nevertheless, it takes into account the assumptions and constraints coming from the context (operating system, communication platform, ...). Both parts of the methodology are going to be briefly detailed in the following.

4.3.1 Generic architecture

The generic part of the methodology is composed of activities that identify middleware components that constitute a generic architecture valid on each ECU. These activities are the identification of tasks, and the identification of software components that is done by the selection and composition of design patterns.

4.3.1.1 Identification of tasks

This activity generates the implementation model of the middleware. As stated in chapter 1, the middleware executes asynchronously from applicative tasks. The implementation model of the middleware answers then the following questions:

- how many tasks are going to represent the middleware's implementation on each ECU,
- what is the role and contribution of each task for the accomplishment of the communication services, and
- to which task model, with respect to the activation strategy, belongs each middleware task.

In order to determine responses for these questions, the activity needs some input information. The first is the functionalities that the middleware must fulfill, together with their activation events. This input is important for the task identification strategies that are the second type of information needed by this activity. Finally, the specification of the used operating system, OSEK/VDX OS in this case, is essential since the identified set of tasks must be adapted and prepared to execute on top of it. More details concerning this activity will be given in chapter 5.

4.3.1.2 Software components identification

This activity is in charge of building the software model of the middleware in terms of code sequence. Since an object-oriented design strategy is going to be used, this activity must determine:

- what are the classes/objects that are going to compose the software architecture of the middleware, and
- what are the sequences of code (methods) that each middleware task will execute (deployment).

To accomplish these goals, we need to, firstly, have the list of communication services that the middleware must provide. Secondly, we must have access to a design patterns catalog, from where we can select those that are appropriate for the middleware's implementation model. This model is the third input information for this activity, which will be presented in chapter 6.

4.3.2 Middleware configuration

This part of the methodology is responsible for constructing and configuring the participants of the system (applicative and middleware tasks, and frames) such that, all timing constraints (relative deadlines and freshness constraints) are respected. It cannot then be abstracted of the timing requirements imposed by applicative tasks and signals. The activities are the frame packing, the configuration of the middleware tasks, and the priority allocation for applicative tasks.

4.3.2.1 Frame packing

The input data of this activity is the characteristics of the applicative tasks and of the signals. Basically, the idea is to construct and configure the frames (signals composing the frame, transmission period, relative deadline, and priority) that are going to be transmitted by the ECUs over a CAN bus. As formerly mentioned, the parameters of this configuration must minimize the bandwidth consumption, and ensure that all deadlines and freshness constraints are met.

In order to break the dependency cycle, the frame packing should consider the delays induced by applicative and middleware tasks. However, at this point of the methodology, neither the middleware tasks have been characterized, nor a feasible priority has been assigned to each applicative task. The proposed solution is based on worst-case tolerated behaviours of tasks. If deadlines and freshness constraints are met under these assumptions, they will be necessarily met with the actual behaviours. The worst-case tolerated behaviours of the middleware tasks are achieved using its implementation model.

The successful output of this step is a feasible global frame configuration. Global in the sense where this configuration contains all frames that are going to be transmitted over the CAN bus. When no feasible configuration is found, the freshness constraints have to be relaxed, the applicative tasks re-designed or their placement allocation modified. The frame packing activity is detailed in chapter 7.

4.3.2.2 Configuration of the middleware tasks

This activity of the methodology consists in the characterization of each middleware task on each ECU. Precisely, one has to determine the execution time, the activation period, the relative deadline and the priority. These parameters are calculated using the signals specification and the frame packing configuration. The frames and the signals compose the data with which the middleware tasks are going to deal.

To determine the execution time of each task, the data that the middleware is going to handle is not sufficient. One must know the code sequences that will be executed by each task. For this purpose, the middleware tasks configuration activity uses the software model of the middleware, determined in the generic part of the methodology. For a thorough explanation on the configuration of the middleware tasks, the reader can refer to chapter 8.

4.3.2.3 Priority allocation

Finally, this last activity tries to determine a feasible priority allocation for the set of applicative tasks executing on each ECU. To achieve its goal, this activity uses the specification of the tasks, and the Audsley priority assignment algorithm [58] presented in chapter 2. Moreover, the activity also takes into account the fact that the middleware might be considered as belonging to system level, and thus, with a higher priority than applicative tasks. This consideration, as well as other aspects related to this activity, is going to be introduced in chapter 8.

If however no feasible priority allocation is obtained, either the applicative tasks set and their constraints have to be re-worked, or the tasks placement changed.

4.4 Conclusion

This chapter presented, on the one hand, the set of communication services that are going to be provided by the middleware and considered during its development. On the other hand, it introduced the methodology proposed for the construction of the middleware.

Two basic communication services are provided. One service allowing applicative tasks to make available produced signal values, and another that permits applicative tasks to retrieve signal values for consumption. These two services are in conformity with the requirements imposed on the middleware: they are independent of the communication protocol and they make abstraction of the location of the tasks.

The methodology that allows the development of the middleware is divided in two parts. The first part is concerned with the aspects of the middleware that are not dependent on the timing constraints established by the applicative tasks and the signals. The result of this part is valid for all ECUS. The second part of the methodology deals with the details related to the timing constraints of each ECU. It is responsible for the construction and configuration of the applicative and middleware tasks and network frames, such that, all timing constraints are respected. This methodology was presented in two articles [65, 66].

The next two chapters are going to present the generic part of the methodology. This way, one has all the information necessary to introduce in the following chapters the part of the methodology that is dependent on the timing constraints established by the applicative tasks and the signals.

Chapter 5

Implementation model of the middleware

5.1 Introduction

This chapter presents how to construct the implementation model of the middleware in terms of tasks. The result is a generic set of tasks executing the communication services (task architecture), independent of any timing constraints. As declared in chapter 1, the middleware executes asynchronously from applicative tasks. Thus, the goal of this chapter is to identify the number of tasks that are going to implement the middleware on each ECU, the role of each task, and the task model to which they belong.

The advantage of identifying a set of middleware tasks is in the fact of having a reference model at implementation level of the middleware during the frame packing activity (see chapter 7). To be able to break the dependency cycle introduced in chapter 4, and to assign a relative deadline to each frame such that the signals freshness constraints are respected, the frame packing needs this kind of information. The knowledge of the task model to which middleware tasks belong is a benefit for the activity where middleware tasks are configured (refer to chapter 8). The configuration method must suite the task model.

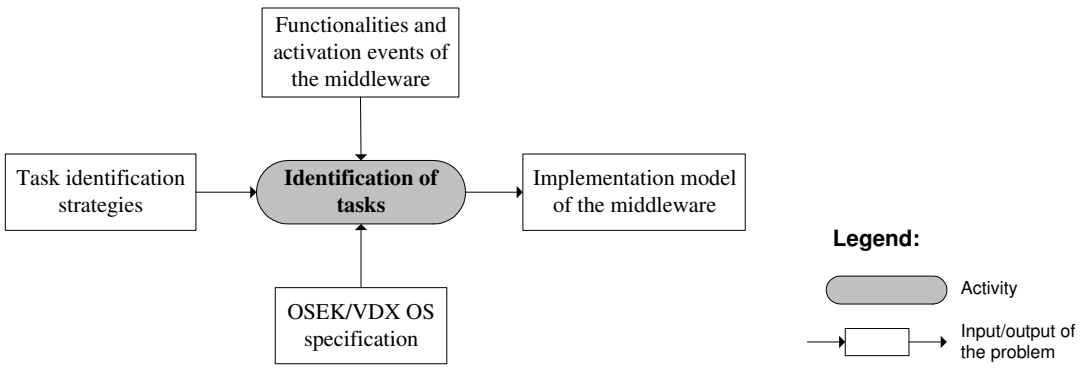


Figure 5.1: Input and output data of the activity in charge of identifying a set of middleware tasks.

The activity in charge of constructing the generic task architecture consists of an identification of tasks; its input and output information are illustrated in figure 5.1. This activity is conditioned by the operating system running on the ECUs. In the context of event-triggered automotive applications, OSEK/VDX OS [1] is becoming the standard operating system. OSEK/VDX OS is a uniform execution environment that provides for automotive applications an efficient utilization of ECU resources. The main goal is to reduce costs by enhancing portability and re-usability of applicative level software.

OSEK/VDX OS specifies a standardized interface of system services, which cover several key issues of the operating system: task and resource management, interrupt processing, alarm and event mechanisms, and error handling. In terms of scheduling, two policies are provided: Fixed Priority Preemptive (FPP) and Non-Preemptive Fixed Priority (NPFP). Contention situations are solved by the resource manager using the Priority Ceiling Protocol [67] (PCP). Moreover, different conformance classes of the OSEK/VDX OS exist, each one providing different features and capabilities in order to satisfy various applicative software requirements (e.g. processor, memory).

The identification of middleware tasks will consider the existence on each ECU of the OSEK/VDX OS, and will be thus constrained to identify a set of tasks adapted to the properties of this operating system. Besides, the task identification process must also take into account other characteristics of the environment in which the middleware will execute, such as, the events that might trigger the middleware functionalities.

5.2 Functionalities of the middleware and their activation events

The middleware must accomplish four main functions:

1. to receive signal values produced from applicative tasks, and store them in a signals repository,
2. to retrieve signal values from a repository, construct frames and request their transmission,
3. to receive frames, unpack them and save the signal values in a repository, and
4. to retrieve signal values from a repository and deliver them to applicative tasks.

Recall that the middleware executes asynchronously from the applicative tasks. Hence, the functions related to the transmission and reception of frames (functions 2 and 3) are not synchronous with the production and consumption of signals values made by the applicative tasks (functions 1 and 4). From this fact, one can also assume that, for performance reasons, the storage and retrieval of signal values is performed by a middleware library executed under the responsibility of applicative tasks. Therefore, the asynchronous part of the middleware performs two functionalities: construction and transmission of frames (function 2), and reception and handling of frames (function 3).

The functionalities of the middleware are activated through a set of well defined events: expiration of frames transmission periods, and arrival of frames. The former event should trigger the middleware functionality responsible for the construction and transmission request of a frame. The later event, the arrival of frames, should activate the functionality in charge of receiving and handling a frame. But, between the occurrence of one of these events and the subsequent activation of the corresponding functionality, there must be a task activation mechanism that, in this context, is provided by OSEK/VDX OS.

OSEK/VDX OS offers several task activation mechanisms: explicit calls (function invoked from another task or from an interrupt service routine), hardware interrupts (activating an interrupt service routine), timing alarms (cyclic or not), and events (in the context of OSEK/VDX OS). Those that can be directly related with the events arising in the middleware context are the hardware (network controller) interrupts and the timing alarms. On the one hand, the expiration of a frame's transmission period releases a timing alarm that triggers a task in charge of constructing and requesting the transmission of the frame. On the other hand, the arrival of a frame triggers a network controller interrupt that activates an interrupt service routine (which can then activates a task) responsible for the reception and handling of the frame.

Another possibility for the activation of the reception and handling of frames could also be a timing alarm. This way, the arrival of a frame would not trigger any task activation mechanism, and the functionality would be released by a cyclic alarm based on a polling period. However, this possibility degrades the middleware performance by increasing the time delay between the arrival of a frame and its handling. It is thus ruled out.

5.3 Middleware tasks identification strategies

With the set of events arising in the middleware context, and with the work of [54] and [55] introduced in chapter 2, one can construct a list of task identification strategies. From this list, a strategy can be selected to identify the set of middleware tasks on each ECU.

5.3.1 "One task for each event" strategy

This strategy assigns one task for each different activation event. There would be on each ECU a task for each different frame that is received, and one task for each different frame transmission period, as illustrated in figure 5.2. The number of middleware tasks identified by this strategy depends on the total of different frames that are received and on the amount of distinct transmission periods. Moreover, it is possible to derive an activation period for each task since frame transmissions obey a specific rate, and the minimum time interval of inter-arrival of each frame can be determined.

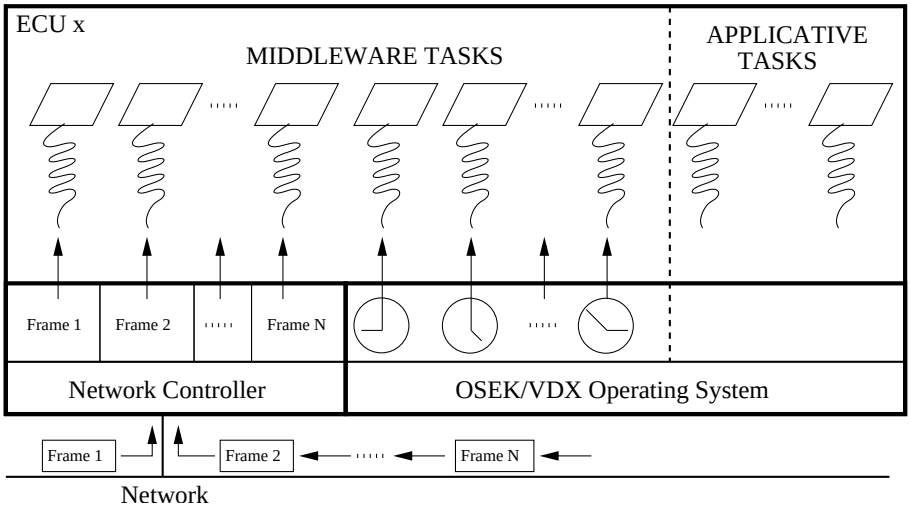


Figure 5.2: Set of middleware tasks identified with the "one task for each event" strategy.

5.3.2 "One task for each type of event" strategy

This strategy identifies one task to handle all frames transmission periods, and one task to manage all arrival of frames (see figure 5.3). The amount of middleware tasks is dependent on the number of different types of events. In this case, there are only two types and thus, two tasks. The activation period of each task is more difficult to obtain because tasks must either handle all frames, with different inter-arrival time intervals, or respect all the different frames transmission periods.

5.3.3 "One task for each signal's purpose" strategy

The idea of this strategy is to assign tasks according to signal semantics. One example in the middleware context is the set of signals that the ECUs exchange for operation mode management (e.g. Pre-Run-Mode for node testing and network initialization, Run-Mode for full functionality of the in-vehicle system, ...). For instance, one task can be activated by a cyclic timing alarm in order to send a frame containing the signal indicating the current mode, and can also be released by a network controller interrupt caused by the arrival of the frame carrying the signal informing on the new mode. Another example is the engine ECU that would have one task responsible for receiving and handling the frames that carry the speed signal from each of the wheels.

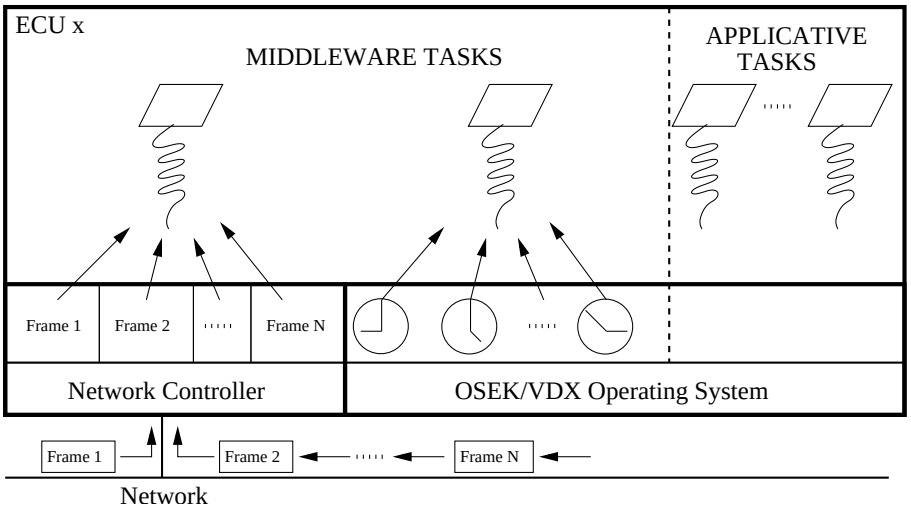


Figure 5.3: Set of middleware tasks identified with the "one task for each type of event" strategy.

The number of middleware tasks identified by this strategy depends then on the purpose of the signals, as shown in figure 5.4. The activation period of the tasks can be difficult to derive or not, depending on the fact of being based on both types of events or just one of them. Note that this strategy may result in the utilization of a mixture of strategies. Since some signals may have a semantics independent of all the others, we can obtain tasks handling several signals, as well as tasks dealing with one signal ("one task for each event" strategy).

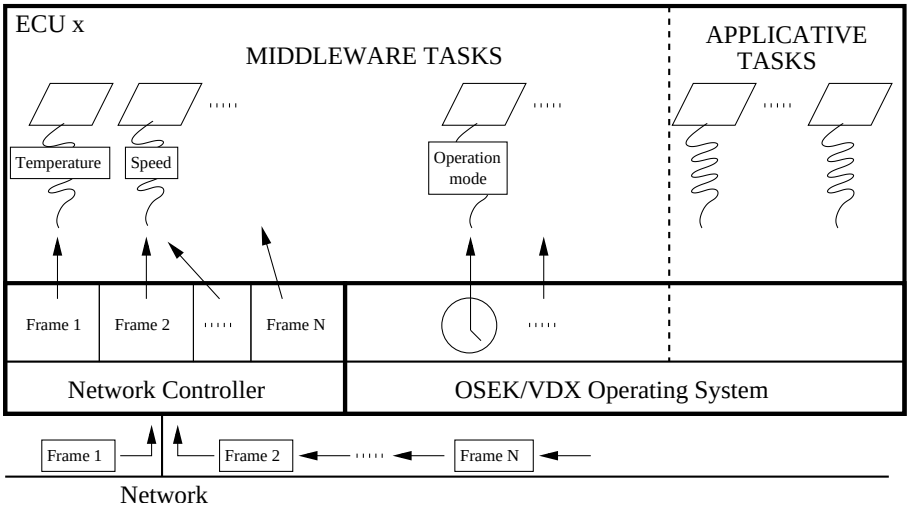


Figure 5.4: Set of middleware tasks identified with the "one task for each signal's purpose" strategy.

5.3.4 Strategy selected to identify the middleware tasks

In the specification of OSEK/VDX OS one can find the minimum requirements for the OS's implementation for each conformance class. The specification also declares that *the portability of applications can only be assumed if the minimum requirements are not exceeded* [1, page 14]. This declaration gave birth to a series of contradicting statements. In [68, page 15] the author considers that according to the con-

formance class, the number of priorities is limited to 8 or 16. However, the work of [69] considers that OSEK/VDX OS requires that any implementation must provide at least 8 priority levels.

Since different readings of the specification exist, the choice made during this work was to minimize the amount of middleware tasks, allowing the execution of a maximum number of applicative tasks. One can therefore exclude the utilization of the strategies "one task for each event" and "one task for each signal's purpose". The chosen strategy is the one that assigns one task to each type of event. There is then on each ECU one task responsible for the construction and transmission request of frames, activated by the expiration of cyclic timing alarms, and another in charge of receiving and handling the newly arrived frames, triggered by network controller interrupts.

This strategy remains valid even when new frames are added to the set of frames to be sent over the network. Moreover, if a new service is added to the middleware, this set of tasks is still valid if the functionality implementing the service is activated by a network controller interrupt or a timing alarm. Finally, in the OSEK/VDX OS, the task in charge of receiving frames would be most efficiently implemented as an interrupt service routine activated by network controller interrupts, decreasing even more the number of tasks necessary to implement the middleware.

5.4 Middleware tasks model

Recall that as stated in section 5.3.2, the chosen strategy identifies tasks for which, in the middleware context, is difficult to derive an activation period. Nevertheless, each task must belong to a task model whose characteristics allow the calculation of an activation period that respects all frames transmission periods and inter-arrival time intervals. The activation period must be necessarily determined since it is one of the characteristics of tasks that permits to calculate their worst-case response time, and to perform the subsequent feasibility analysis.

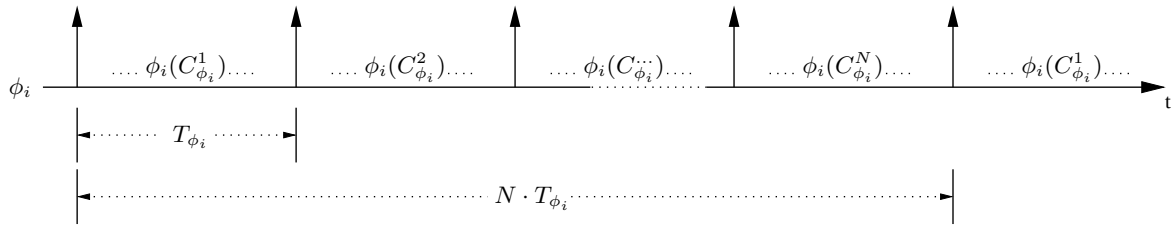
5.4.1 Task in charge of the construction and the transmission request of frames

The characteristics of this task depend on the result of the frame packing algorithm, which will be presented in chapter 7. The frames to send over the network can however have different emission periods, and therefore, the activation rate of the task is obliged to respect all those periods. Consequently, one cannot use the task models where tasks have a unique activation period and execution time [70, 64]. One has to study extended task models as multiframe [62] and generalized multiframe (GMF) [63]. The assignment of values to the characteristics of multiframe or GMF tasks in the middleware context will be detailed in chapter 8. In the remaining of this document, this middleware task will be denoted by ϕ_i, standing for the middleware task responsible for constructing and requesting the transmission of frames on ECU i.

5.4.1.1 Multiframe task model

Assuming that the first emission request of all frames is issued by the first instance of the task, the multiframe task model can be used. A multiframe task is characterized by a set of N execution times, and by a unique activation period and relative deadline (see figure 5.5 for an example of multiframe task). The activation period is calculated in order to respect all different frame emission periods. Thus, different instances of this task may construct and request transmission of a different set of frames.

The worst-case response time calculation method for this task model was introduced in [71]. However, this method does not take into account the time interval during which a multiframe task is blocked by a lower priority one, which uses a shared resource managed by the PCP protocol. Recall that this middleware task is going to use the signals repository (shared with other tasks), and hence, is submitted to the OSEK/VDX OS's PCP protocol. According to [67], the maximum time interval during which this situation occurs is equal to the longest time duration where a lower priority task uses the shared resource. This longest time duration must then be added to each iteration of the worst-case response time calculation method for multiframe tasks.



$\phi_i(C^n_{\phi_i})$ = *n-th* instance of task ϕ_i performing the *n-th* execution time
T_{ϕ_i} = activation period of task ϕ_i

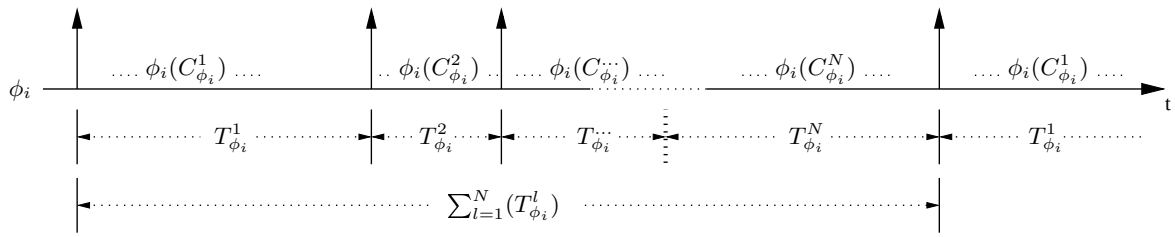
Figure 5.5: Example of multiframe task.

This example supposes that the relative deadline of the task is equal to the activation period.

5.4.1.2 Generalized multiframe task model

When trying to characterize a multiframe task (see chapter 8) that respects all different frame emission periods, it may happen that one of the several execution times is greater than the activation period or than the relative deadline. Imagine for instance that in figure 5.5 there is one execution time greater than the activation period ($\exists n \in [1..N]$, $C^n_{\phi_i} > T_{\phi_i}$). If this scenario occurs, then one can try to overcome this problem by implementing this task as GMF.

Again, it is assumed that the first emission request of all frames is issued by the first instance of the task. The main difference from the multiframe task model is that the activation period and relative deadline also become a vector composed of N elements (see figure 5.6). Once more, the several activation periods are calculated in order to respect all different frame transmission rates. To determine the worst-case response time of GMF tasks one can also use the algorithm presented in [71]. The problem of the blocking time induced by using the PCP protocol is solved as previously explained for multiframe tasks.



$\phi_i(C^n_{\phi_i})$ = *n-th* instance of task ϕ_i performing the *n-th* execution time
$T^n_{\phi_i}$ = *n-th* activation period of task ϕ_i

Figure 5.6: Example of generalized multiframe task.

This example supposes that the vector of relative deadlines of the task is equal to the vector of activation periods.

Furthermore, GMF tasks do not have a unique activation period. So, their implementation on top of OSEK/VDX OS is not trivial since the activation period can only be assigned dynamically. Each instance of a GMF task, just after its beginning of execution, cancels the previous alarm, and sets a new one equivalent to the next activation period, using the system services interface provided by OSEK/VDX OS. This procedure however, does not guarantee the respect of the set of activation periods. Figure 5.7 describes this problem.

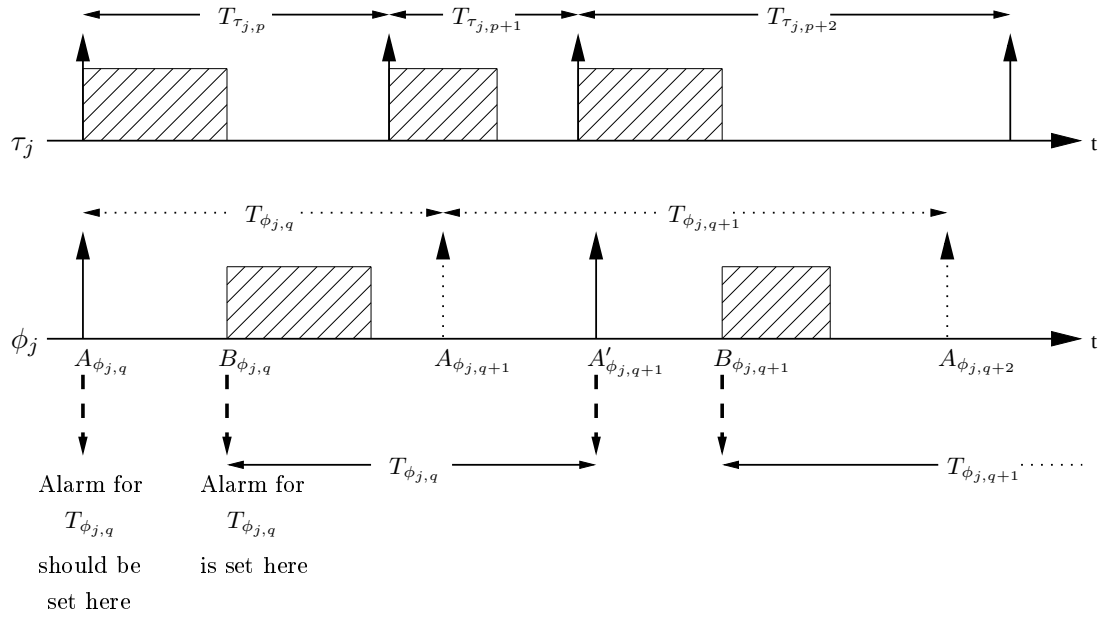


Figure 5.7: This figure illustrates the problem of setting an OSEK/VDX Operating System timing alarm that respects all activation periods of a generalized multiframe task.

Task ϕ_j, whose q-th activation takes place at instant $A_{\phi_{j,q}}$, cannot begin its execution since an instance of an higher priority task τ_j is executing. The setting of the new alarm that should take place as sooner as possible after instant $A_{\phi_{j,q}}$, is effectively set at $B_{\phi_{j,q}}$ the beginning of execution of the q-th instance of ϕ_j. Future activations of ϕ_j are now delayed of $B_{\phi_{j,q}} - A_{\phi_{j,q}}$ units of time. Note that this delay is increased each time an instance of ϕ_j cannot begin its execution at its activation. To overcome this problem, when the q-th instance of task ϕ_j begins its execution, it must calculate the value of $B_{\phi_{j,q}} - A_{\phi_{j,q}}$. Instead of setting the alarm with $T_{\phi_{j,q}}$, it sets with $T_{\phi_{j,q}} - (B_{\phi_{j,q}} - A_{\phi_{j,q}})$. Value $T_{\phi_{j,q}}$ is the $(q \bmod N)$-th element of the activation periods vector. It stands for the time interval between the activations of the q-th and $(q+1)$-th instances of ϕ_j.

5.4.1.3 Other solution

If however, during the characterization of the GMF task, one verifies that there is still an execution time greater than its corresponding activation period or relative deadline, then the solution is to split the work in two tasks (either multiframe or GMF). One task, having a higher priority, would be responsible for the transmission of the frames with smaller relative deadline, while the other task would be in charge of sending the frames with larger relative deadline.

This solution splits the work among two tasks, and increases the probability of respect of the frames relative deadline by delegating the transmission of those with a stricter timing constraint to the higher priority task. Nevertheless, throughout the rest of this document, the methodology considers that this middleware task is implemented as either multiframe or GMF.

5.4.2 Task in charge of receiving and handling frames

On event-triggered networks, such as CAN, where an ECU can receive frames sent from different and unsynchronized stations, the time interval between the arrival of any two frames is not constant. This task, implemented on OSEK/VDX OS as an interrupt service routine, is thus considered as sporadic [64]. In this task model, each task is characterized by a unique execution time, activation period and relative deadline. Since a sporadic task does not follow a strict releasing rate, its activation period is set to the

minimum inter-activation time interval. The assignment of values to these characteristics will be detailed in chapter 8, where the characterization of the middleware tasks is presented.

The worst-case response time calculation for this task model is detailed in [72]. It considers a set of periodic or sporadic tasks, scheduled according to the FPP policy, and submitted to the PCP protocol as access contention manager on shared resources. Indeed, since the interrupt service routine is going to use the signals repository (shared with other tasks), it is subjected to the OSEK/VDX OS's resource manager.

In the rest of this document, the interrupt service routine will be denoted by ω_i, standing for the middleware task responsible for receiving and handling frames on ECU i.

5.5 Conclusion

Before this chapter, the only information available on the implementation model of the middleware was that it executes asynchronously from applicative tasks. This chapter has given more details on this model. Specifically, one is now aware of how many tasks are going to execute on each ECU as part of the middleware, what is the goal of each task, and to which task model belongs each of the tasks.

The tasks were identified according to the events capable of activating the middleware functionalities. Moreover, it has been verified that OSEK/VDX OS, the used operating system, does provide task activation mechanisms that allow the events to trigger tasks executing the functionalities.

Two tasks have been identified: one task responsible for the construction and transmission request of frames, activated by the expiration of cyclic timing alarms, and another in charge of receiving and handling frames, triggered by network controller interrupts. The number of middleware tasks on each ECU is minimized in order to permit the execution of as many as possible applicative tasks.

Finally, the task model of each task has been determined according to the activation model of the triggering events. The task responsible for the construction of frames is either multiframe or generalized multiframe. Its activation period (in the multiframe scenario) or activation periods (in case of generalized multiframe) respect the transmission rate of all frames sent by a specific ECU. The task in charge of handling received frames (implemented as an interrupt service routine on OSEK/VDX OS) is sporadic. Its activation rate is set in order to meet the inter-arrival time intervals of frames on a CAN network.

The details on the implementation model of the middleware were presented in [73, 74]. With these details, the step of the methodology in charge of the construction and configuration of frames has the knowledge of how the middleware executes on each ECU during the exchange of signals. Thus, it will be able to break the dependency cycle introduced in chapter 4, by predicting more easily the middleware worst-case delays when assigning a relative deadline to each frame, such that, the signals freshness constraints are respected.

Moreover, the knowledge of the task model to which middleware tasks belong is a benefit for the middleware configuration activity. The configuration method must be adapted to the used task models. The task configuration activity also requires the software model of the middleware, so that it might be able to identify the sequences of code that will be carried out by each task. This model, built by an activity that is member of the generic part of the methodology, is introduced in the next chapter.

Chapter 6

Software model of the middleware

6.1 Introduction

This chapter is focused on the design of the middleware software. The result of this design is called the software model, and expresses the software components (code) and how to deploy them in the tasks composing the implementation model defined in chapter 5.

The construction of the software model of the middleware will be based on the object-oriented paradigm. This paradigm, already used in the real-time systems domain [56, 54, 19, 27], is well known to increase the portability, reusability and maintainability of software. The utilization of graphical notations [22] providing different and consistent model views, and improved mapping to problems of different domains, help to achieve the reusability and maintainability of software. Moreover, the direct translation of design artifacts to features of object-oriented programming languages promotes a gain in productivity.

Just like for the design of the middleware TAO (see chapter 2), the middleware presented in this document is built using the object-oriented design patterns approach [41, 24, 42, 43, 44, 45, 46, 47] (see appendix A for an example of design pattern). The goal of this approach is to propose object-oriented solutions for design problems arising in a certain context. With the middleware implementation model, a set of design patterns and the communication services that the middleware must offer to the application level, one constructs the software model as depicted in figure 6.1.

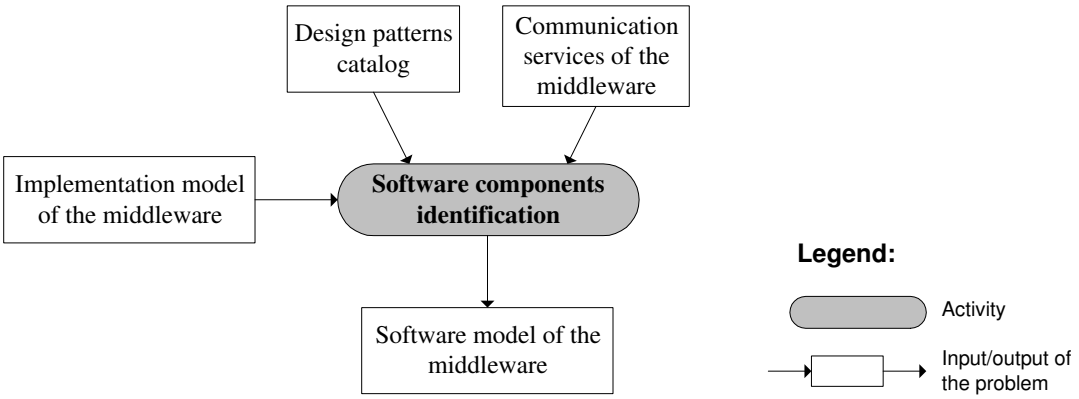


Figure 6.1: Input and output data of the activity in charge of the design of the middleware software.

The goal is, first, to select the design patterns that are the most adapted to solve the design problems arising in the middleware context. The selection of design patterns has also in mind the set of communication services the middleware provides. Second, we have to compose these patterns in order to form a class diagram [22] that represents the structure of the middleware software. This class diagram makes an

abstraction of the implementation model (set of middleware tasks). Finally, the third goal, is to consider the implementation model, and thus, to identify the objects that must be instantiated and executed in each middleware task. At the end of this chapter, the following questions should be answered:

- which classes compose the middleware class diagram,
- which methods and attributes must be present in each class,
- which objects are going to be instantiated in each middleware task, and then
- what are the sequences of code that will be carried out by each task.

The last point explains why the software model is important for the timing constraints dependent part of the methodology. The configuration of the execution time of each task does not depend only on the data (signals and frames) the middleware will handle. Another important information is the code that is executed on behalf of the middleware, and which handles the data that has just been mentioned. Moreover, the sequences of code executed by each task are ECU independent. Thus, with the information contained in the middleware software model, a procedure of automatic code generation can be applied.

The construction of the middleware software model begins with the selection of design patterns. These patterns are introduced in the following paragraph.

6.2 Design patterns for the middleware

The first goal of this chapter is to select a collection of design patterns, which must be adapted to the communication services provided by the middleware to the application level (see chapter 4). In the following, each design pattern selected to form the structure of the middleware software is presented, and the details of its utilization are given.

6.2.1 "Active object" pattern

Applicative tasks invoke, usually in a periodical manner, the service of the middleware allowing to make produced signal values available for transmission. In the middleware context, such an invocation should not block the calling task during a time interval that cannot be precisely known. This situation could even also degrade the quality of service of other tasks.

The *Active Object* pattern [42] can help to avoid this unwanted scenario. It "*decouples method invocation from method execution to enhance concurrency and simplify synchronized access to objects that reside in their own threads of control*". In other words, the method invocation occurs in a task while the execution happens in a separate task. Besides, in order to reduce the number of classes necessary to implement the pattern, the *Integrated Scheduler* variant [42, page 389] was chosen.

Figure 6.2 illustrates the structure of the pattern while displaying the distribution of the roles among classes composing the middleware software:

- The roles of *scheduler* and *proxy* are played by class *Proxy_Scheduler*. This class resides in the applicative tasks, and acts as a service requests receiver. Thus, it provides the communication interface proposed by the middleware to the application level. At run-time, class *Proxy_Scheduler* receives services invocations, and according to the type of service (production or consumption of a signal), stores or retrieves signal values from class *Signals*.
  When the service invoked concerns a signal consumption, then class *Proxy_Scheduler* returns the corresponding value to the applicative task. Values to be consumed are stored in class *Signals* by class *Core*, which will be presented later. If however the service allowing to make produced signal values available is called, then the value is simply stored in class *Signals*. In spite of playing the role of *scheduler*, class *Proxy_Scheduler* will not be responsible for deciding when to pack the produced signals into frames. This decision will be taken by class *Core*.

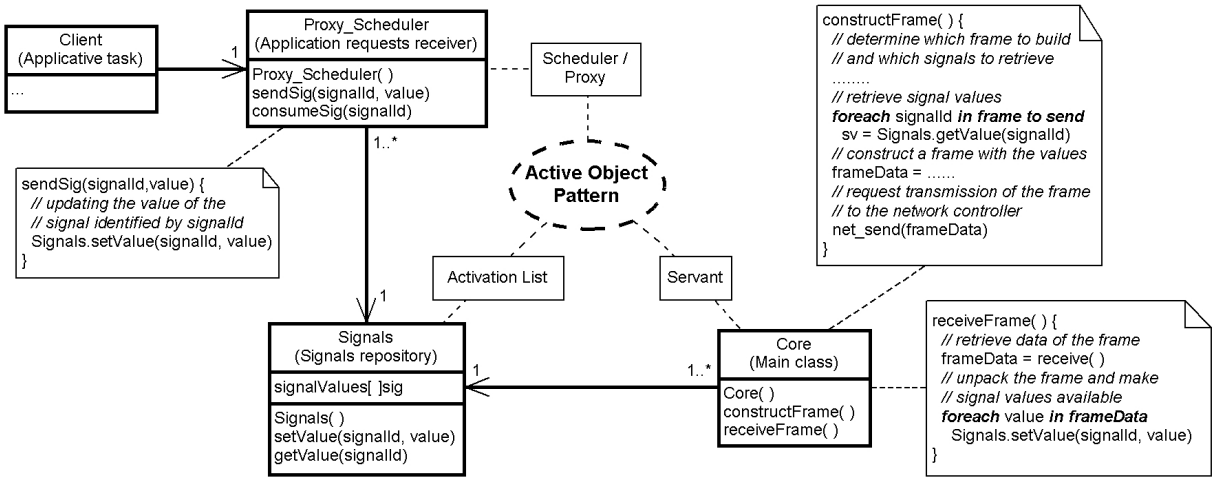


Figure 6.2: Class diagram illustrating the *Integrated Scheduler* variant of the *Active Object* pattern applied on the middleware context.

This pattern allows to decouple the invocation of the send signal service from its execution, by making them occur in separate tasks. The role played by each class in this pattern is illustrated by rectangular boxes over the dashed lines.

- Class *Signals* plays the role of *activation list*. It serves as a service requests repository, or, in other words, maintains information related to the signals produced and to be consumed. This class decouples the applicative tasks (where class *Proxy_Scheduler* resides) from the task where the role of *servant* is executed.

- The role of *servant* is assigned to class *Core*, the main class of the middleware software. This class executes in a separate task and implements the functionalities contributing to the accomplishment of the services provided by class *Proxy_Scheduler*. These functionalities are resumed to the packing and transmission request, and to the reception and unpacking of frames. For this purpose, it uses class *Signals* either to retrieve the signal values that must be packed and sent, or to store those that have been unpacked from a frame.

The behaviour of the pattern in the middleware context is shown in figure 6.3. The sequence diagram demonstrates the chain of methods executed when a signal value is made available, and while is being packed and requested for transmission. Recall that, as established in chapter 5, the functionality responsible for constructing and requesting transmission of frames is executed by a multiframe or generalized multiframe task, activated by an OSEK/VDX OS "timing" alarm (that is, "connected" to a timer). Thus, in figure 6.3, the instant t that activates the method in charge of packing the signal value, represents an expiration of the timing alarm (whose settings depend on the middleware tasks characterization presented in chapter 8).

6.2.2 "Adapter" pattern

The middleware must be conceived and implemented in such a way as being, as far as possible, compatible with any of the Medium Access Control protocols that can be used inside a vehicle (e.g., CAN [3], LIN [7], MOST [12], soon FlexRay [10], etc). The fact that each network provides different services and interfaces makes the interconnection difficult between them and the middleware.

A solution to this problem is the *Adapter* pattern [41] that can be used when one "*wants to use an existing class and its interface does not match the one that one needs*". It is composed of an abstract

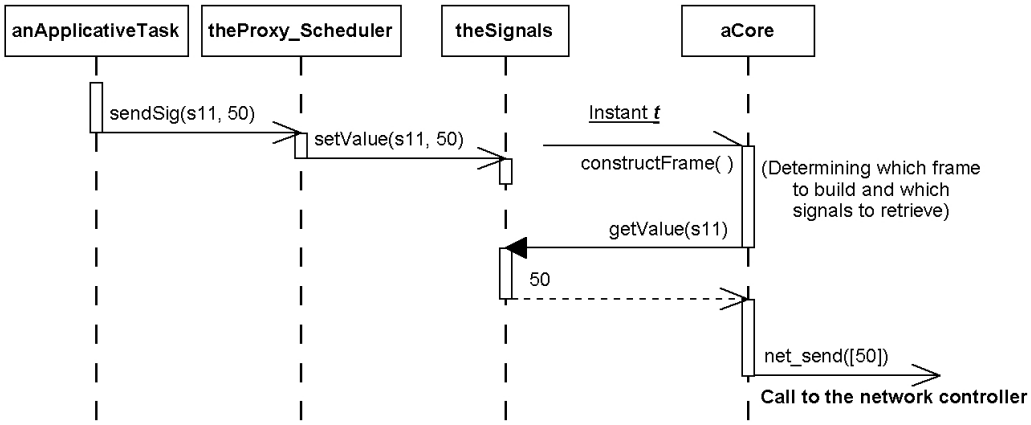


Figure 6.3: Sequence diagram showing the *Integrated Scheduler* variant of the *Active Object* pattern in the middleware context.

class defining a standard interface to be used by client classes, and of an adapter class that makes the translation between the standard interface and the incompatible one.

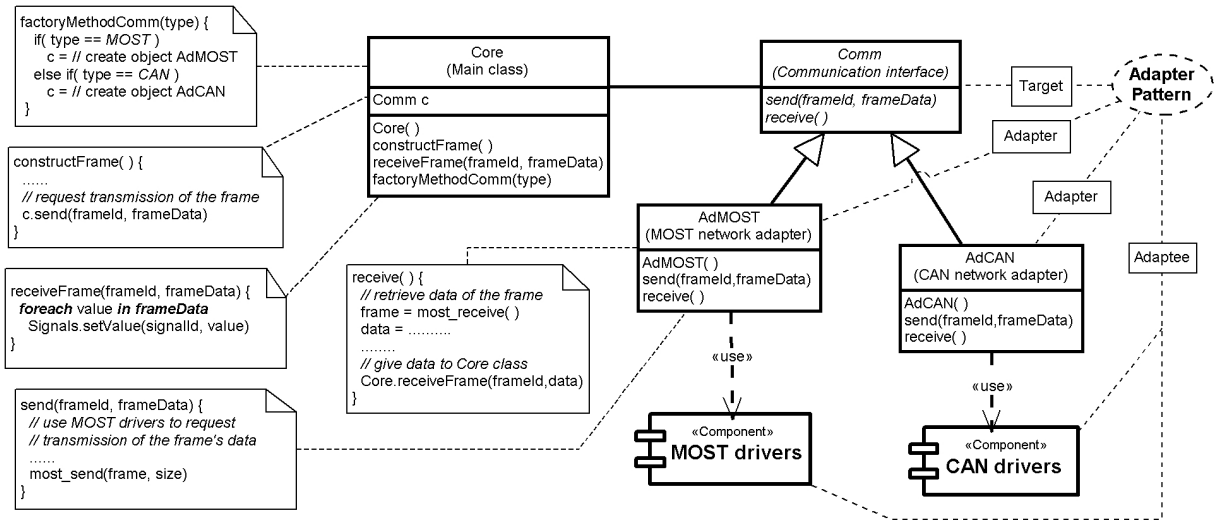


Figure 6.4: Class diagram providing an example of utilization of the *Adapter* pattern in the middleware context.

This pattern provides a unique communication interface inside the middleware software, and hides the heterogeneity of communication platforms. The role played by each class in this pattern is illustrated by rectangular boxes over the dashed lines.

In figure 6.4, the assignment of the roles of the pattern in the middleware context is illustrated. Class *Core*, introduced in section 6.2.1, becomes the client of this pattern since it is the main user of network services inside the middleware software structure:

- The role of *target* is played by an abstract class named *Comm*. This class defines a standard set of

communication services that will be used by class *Core*. The set of communication services must be as rich and abstract as possible, in order to allow the middleware main class to be independent of the used network protocol.

- Classes *AdMOST* and *AdCAN* serve as *adapter*. In this context, they adjust the interface of the used network drivers (MOST and CAN in this case) to the standard set of communication services defined in class *Comm*. These adapter classes must translate each service defined in class *Comm* to the services provided by the network drivers.

- Finally, the role of *adaptee* has been assigned to the drivers of the used network controller (drivers of the MOST and CAN networks in the example of figure 6.4).

The *Adapter* pattern helps the middleware to handle the heterogeneity of communication platforms, and allows its development and modification independently of the underlying communication network. The drawback is that for each new communication protocol, one must create an adapter class. However, in this situation, the effort needed merely consists in implementing a new adapter class.

6.2.3 "Observer" pattern

An instance of the network adapter class (see section 6.2.2) represents the communication protocol inside the middleware software. Any object of class *Core* wishing to use network services must pass by this instance. Hence, when a network frame is constructed by a *Core* object, the adapter instance must obtain it in order to issue a transmission request. Moreover, when a network frame is received, the adapter instance must route it to a *Core* object that will handle it.

The relation between the adapter instance and its client *Core* objects, can be modeled by the *Observer* pattern [41]. This pattern is used when an object changes its state, and must notify other objects without making assumptions about which these objects are. It also creates a loose dependency between objects, such that, when the state of an object changes, all its dependents (or "observers") are immediately notified.

The utilization of the *Observer* pattern in the middleware software is represented in figure 6.5:

- The role of *subject* is assigned to the class having the same name. Class *Subject* represents the components whose state changes and thus, must notify its "observers". References to these "observers" are stored in a list, defined as a class attribute. The interface of class *Subject* is composed of a method (*notify*) that is executed when its state is modified.

- Class *Observer* plays obviously the role of *observer*. It defines the interface of all classes that wish to be notified when the state of their subjects changes. This notification is made through the method *update*.

- Class *Core* must be immediately notified when a new frame is received by class *Comm*. The inverse situation occurs when class *Core* constructs a frame, where class *Comm* must be notified straight away. Thus, both classes play the role of *concrete observer* and *subject*. They must implement the interfaces of classes *Observer* and *Subject*.

This pattern permits classes *Core* and *Comm* to evolve independently without hindering the possibility of passing data between them. Figure 6.6 demonstrates how the exchange of frames is done between these classes using the *Observer* pattern. The sequence diagram demonstrates the chain of methods executed when a *Core* object constructs a frame, and when a new frame arrives at the network controller. Recall that, as determined in chapter 5, the two main functionalities of the middleware are activated by an OSEK/VDX OS timing alarm and by network controller interrupts. Thus, in figure 6.6, the instant t represents an expiration of the timing alarm, and interrupt symbolizes the triggering of a network controller interrupt.

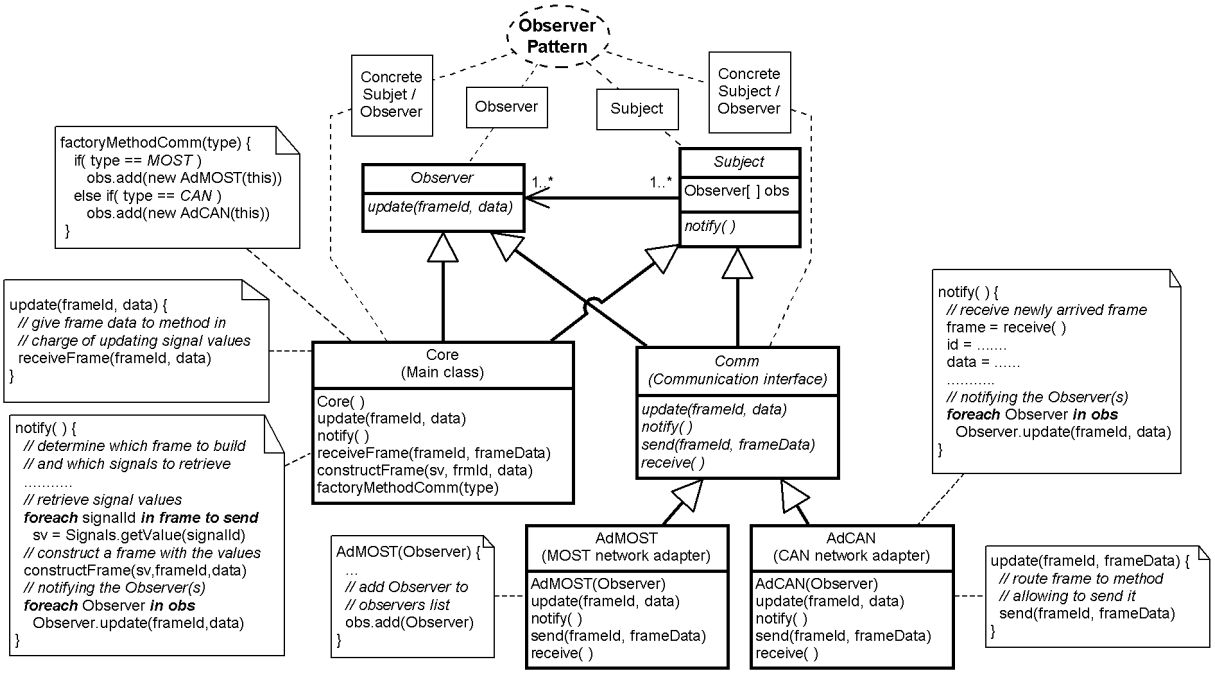


Figure 6.5: The structure of the *Observer* pattern applied in the middleware context.

This pattern creates a loose dependency between the class providing communication services and the middleware main class. However, this loose dependency do not hinder the possibility of passing data between them. The role played by each class in this pattern is illustrated by rectangular boxes over the dashed lines.

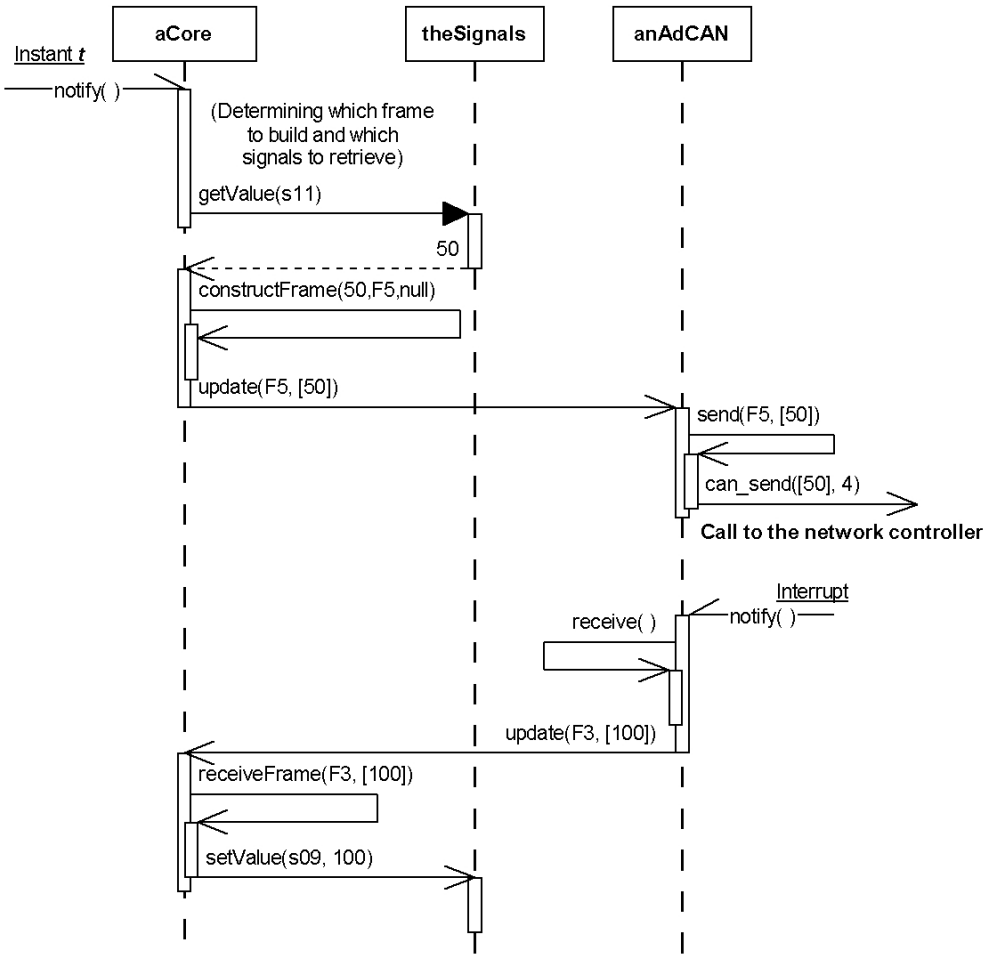


Figure 6.6: Sequence diagram showing the *Observer* pattern applied in the middleware context.

6.2.4 "Asynchronous completion token" pattern

The purpose of the *Asynchronous Completion Token* (ACT) pattern [42] is to "*allow an object to demultiplex and process efficiently the responses of asynchronous services it invokes on other objects*". For this purpose, when an asynchronous service is invoked, the invoker passes a token containing information that identifies its method that will be responsible for processing the service response. When the service terminates, the response contains the token and thus, the invoker object can identify its method that will process the response.

In the middleware context, one could assume a scenario where the service provided by the adapter class allowing to request transmission of frames would be implemented as asynchronous. More, in this scenario, it could be assumed that the used network protocol would return frame transmission completion events, indicating that a frame as been successfully transmitted[1]. In this situation, the ACT pattern would be useful because *Core* objects request the transmission of several frames, and thus, the several frame transmission completion events received could be efficiently handled.

However, not all in-vehicle communication platforms provide mechanisms that allow a network adapter object to return frame transmission completion events. Moreover, the service allowing to request transmission of frames may not always be implemented as asynchronous[2]. In spite of these negative conditions, the ACT pattern can still be used with the purpose of encapsulating the information exchanged between the network adapter and the *Core* objects.

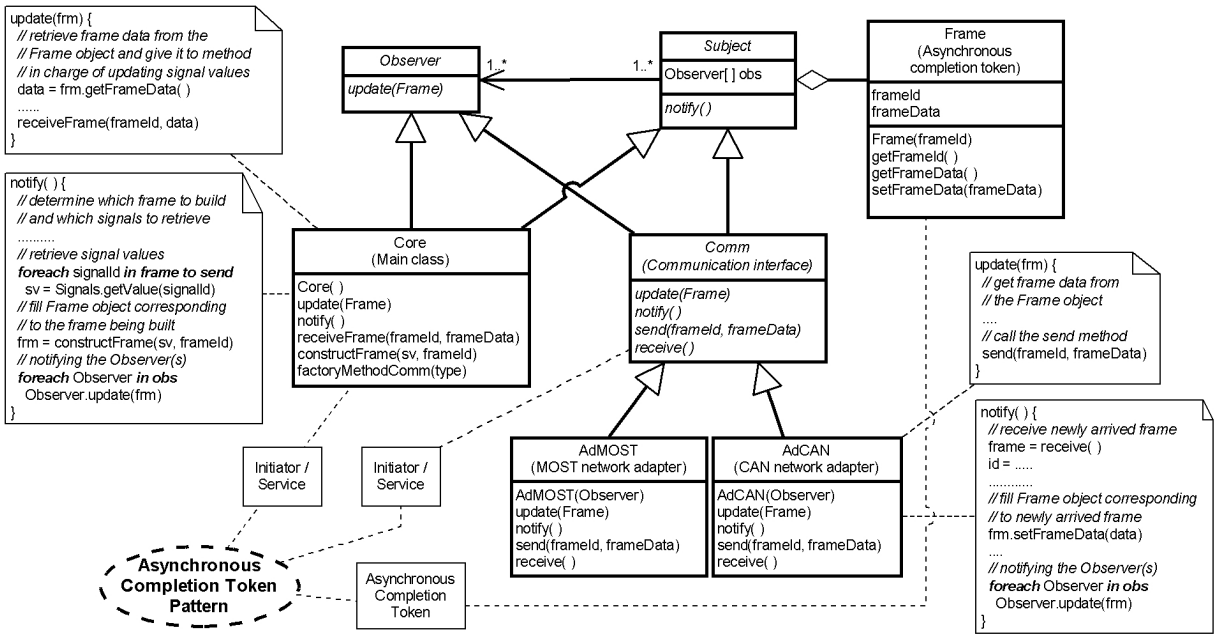


Figure 6.7: Utilization of the *Asynchronous Completion Token* pattern in the middleware context.

This pattern is used to encapsulate the information exchanged between the network adapter object and the objects of class *Core*. The role played by each class in this pattern is illustrated by rectangular boxes over the dashed lines.

Figure 6.7 presents the ACT pattern in the middleware context:

- The information exchanged between the network adapter and the *Core* objects concerns the frames

[1] A frame successfully transmitted means that all data bits have been sent. It does not mean that the frame has been received with success by all ECUs on the network.

[2] In chapter 5, it has been proposed and justified that the functionality responsible for constructing and requesting transmission of frames is operated in one single task.

either constructed or received. Therefore, the role of *initiator* is performed by both classes. They create an asynchronous completion token encapsulating the information related to the constructed or received frame. Moreover, both classes play the role of *service*, meaning that they provide a method receiving the token.

- The *Frame* class encapsulates the data exchanged between the network adapter and the *Core* objects. Thus, it plays the role of *asynchronous completion token*. This class depends on the abstract class *Subject* (described in section 6.2.3) because the classes that will inherit from *Subject* are going to construct or receive frames, and thus, need to create token objects.

Note that the role of *completion handler* (the last role proposed to compose the pattern structure) is not assigned because, as previously mentioned, this pattern is used with the purpose of encapsulating the information exchanged between the network adapter and the *Core* objects. Since there is no service response, then there is no need to implement a completion handler method. In figure 6.8 one can observe the behaviour of the pattern in the middleware context.

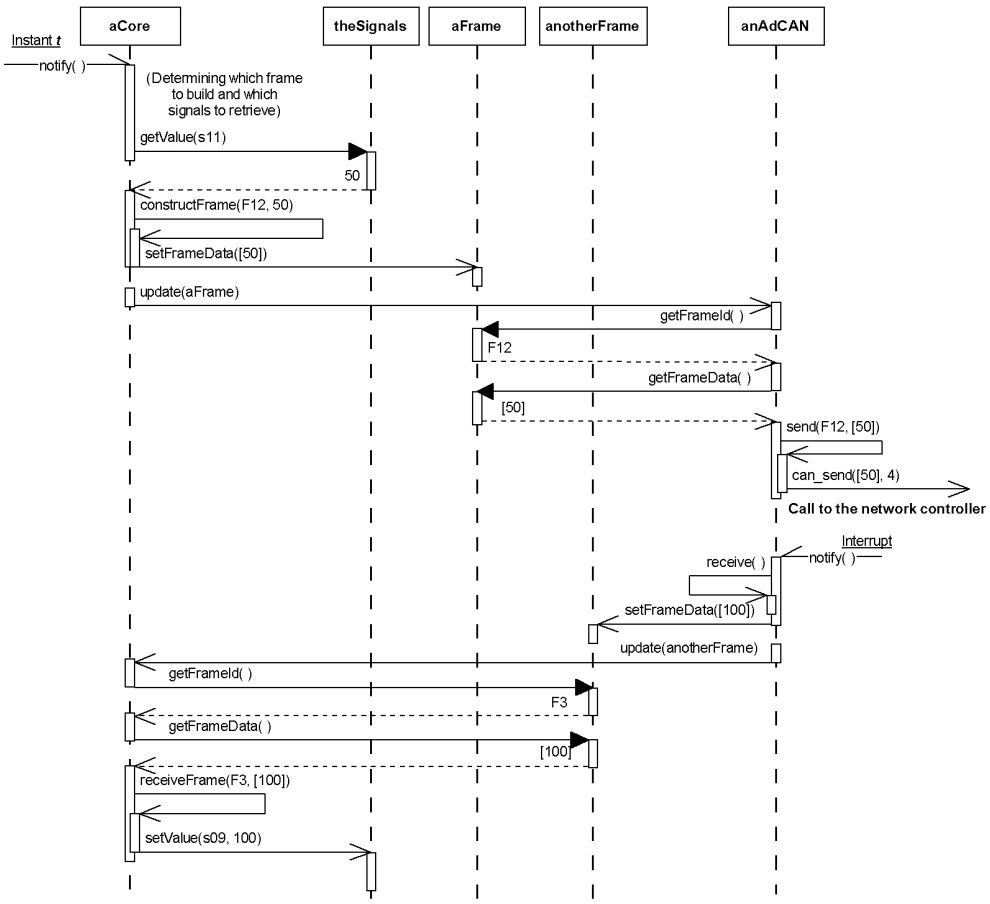


Figure 6.8: Chain of methods induced by the *Asynchronous Completion Token* pattern in the middleware context.

6.3 Class diagram and the implementation model of the middleware

Now that the selected design patterns have been presented, it is time to construct a class diagram as the result of the composition of the selected design patterns. Then, it remains to identify the sequences of code executed by each middleware task, in order to establish the software model of the middleware.

This section begins with the presentation of the construction of the class diagram, and ends with the process determining the code that the middleware tasks are going to execute. During this process, two aspects are handled: the code extracted from the methods specified in the class diagram, and the complementary code that must be added in order to overcome the specificities of the OS.

6.3.1 Class diagram: composition of design patterns

The class diagram representing the structure of the middleware software is illustrated in figure 6.9. This structural diagram is obtained from the composition of the previously presented patterns. Note that there is, for the present, no formal technique allowing to accomplish this *composition* activity. Nevertheless, some research as been published on this subject.

The work presented in [75] describes an approach that uses UML modeling capabilities to compose design patterns at several levels of abstraction. This approach takes advantage of the benefits of two strategies: the stringing design (patterns are glued together using UML relationships like association or dependency) for the higher level of abstraction, and the overlapping technique (a class is a participant in a pattern, and, at the same time, participant in another pattern) in low stages of abstraction. For this purpose, three hierarchical views based on UML models and with different levels of abstraction were defined. However, in a design where few patterns are used, some of these views are unnecessary. Moreover, the transition between the views are sometimes influenced by the designer skills.

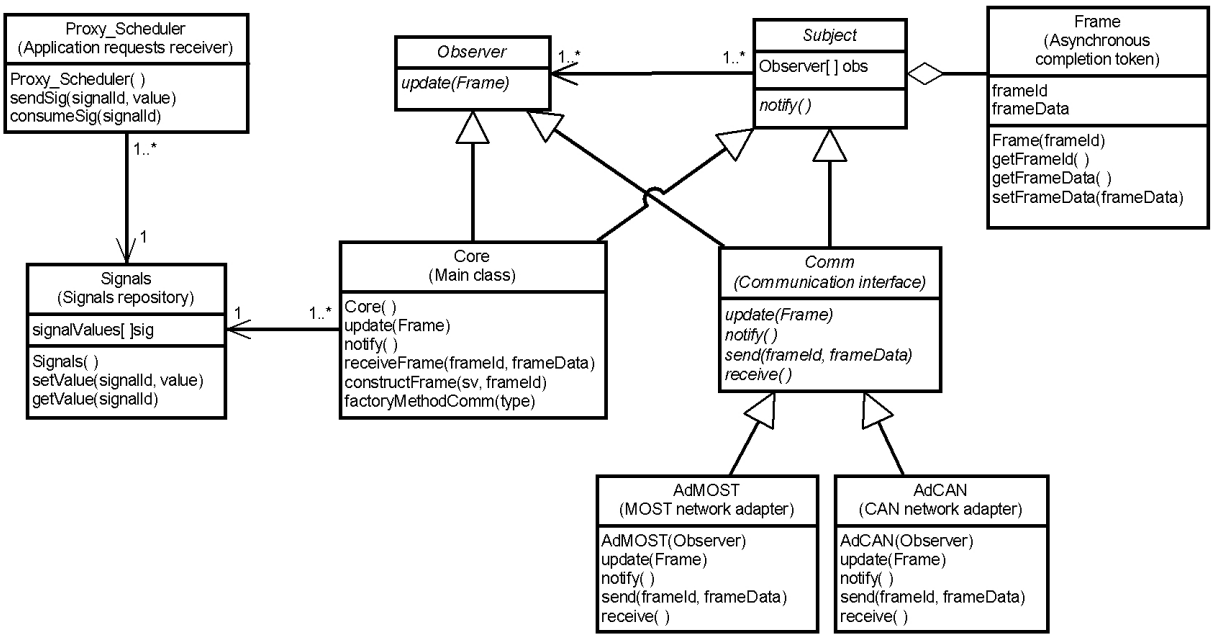


Figure 6.9: UML class diagram representing the structure of the middleware software.

The classes are the participants in the selected design patterns.

The approach used in this work for the composition of the patterns followed an intuitive and more

simpler rule. It was selected in the structural description of each design pattern the class that represents the core functionality of the middleware. Such a class is present in each used design pattern, and hence, these several samples of the same class were *merged*, or overlapped, in a unique class termed *Core* in figure 6.9. Obviously, the role of this class is different in each design pattern, as the reader can verify in the previous sections. Moreover, the merging of class *Core* was done step by step, each time a new pattern was selected for composing the middleware software.

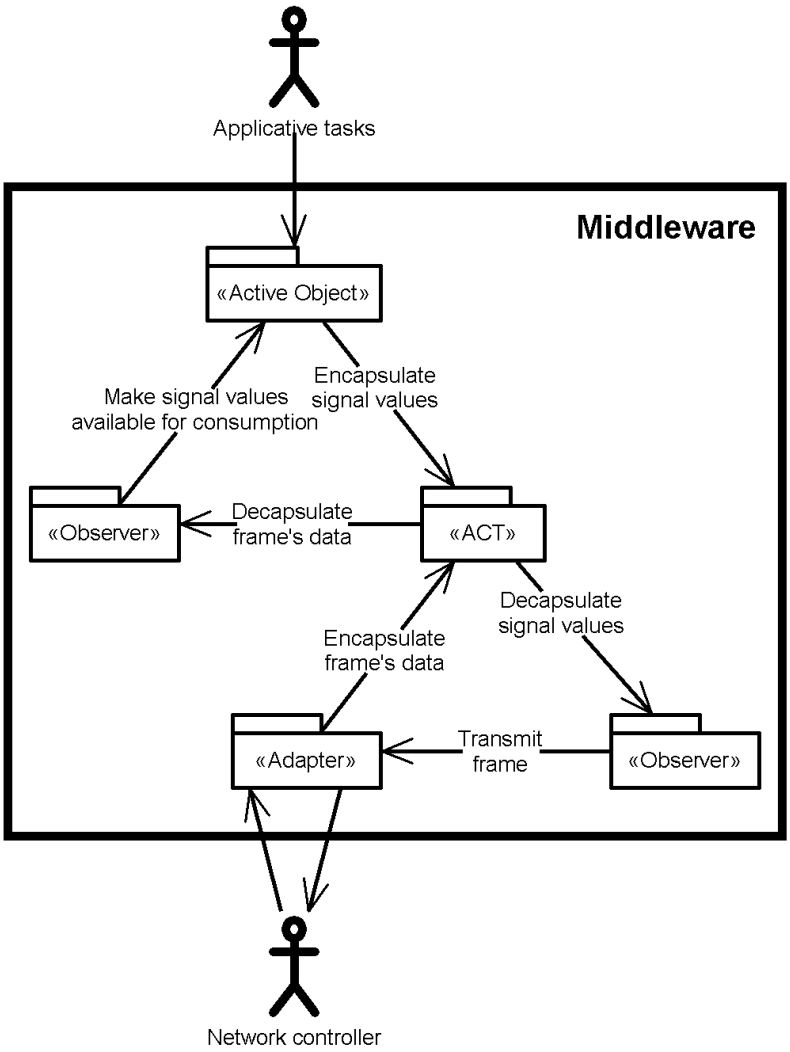


Figure 6.10: *Pattern-Level* diagram [75] for the middleware software.

The result of the composition of the selected patterns can also be illustrated using a *Pattern-Level* diagram, as proposed in [75]. This type of diagram displays instances of the selected patterns, and identifies the relationships between those instances. The *Pattern-Level* diagram for the middleware software is shown in figure 6.10. UML stereotypes are used to represent pattern instances. Each of the selected patterns is displayed, as well as, their dependency relationships.

The resulting class diagram of figure 6.9 symbolizes the code that is going to implement the middleware on each ECU. However, chapter 5 has established that the middleware is also represented on each ECU under the form of a set of tasks. The next step is then to relate the class diagram with the set of middleware tasks.

6.3.2 Sequences of code of the middleware tasks

The code defined in the previously presented class diagram is going to be executed by a set of tasks. Thus, it is necessary to *split* the class diagram among the tasks, and identify the objects that must be instantiated. Since the class diagram defines that class *Proxy_Scheduler* is to be run on each applicative task, one must also identify the middleware objects that will execute on behalf of those tasks. Hence, taking into account all the middleware functionalities (identified in chapter 5), the identification of objects is going to be presented in association with:

- the storage in the signals repository of values obtained from the applicative tasks,
- the retrieval of values from the signals repository to return to applicative tasks,
- the construction and transmission request of frames, and
- the reception and handling of frames.

For each of these scenarios, a sequence diagram is going to determine the code to be executed by each middleware task, and by the *library* (class *Proxy_Scheduler*) that must be compiled with each applicative task.

6.3.2.1 Storage of signal values

The storage of signal values in the repository is to be performed by an object of class *Proxy_Scheduler*. This class is executed on behalf of the applicative tasks, and thus, represents a library that is compiled with each one of those tasks. While one of these tasks exists, one corresponding *Proxy_Scheduler* object is active, and provides the middleware interface for the communication services.

In this scenario, there is then one *Proxy_Scheduler* object associated to one applicative task, and logically, there should be one object of class *Signals* (the repository). However, this class is going to be used by all the middleware functionalities, and thus, by several tasks. Class *Signal* represents then a shared resource, which will be managed by the used operating system, OSEK/VDX OS as established in chapter 5. In the sequence diagram of figure 6.11, this class is however displayed as an normal object. This figure illustrates the production and consequent storage of two signal values produced by two different applicative tasks.

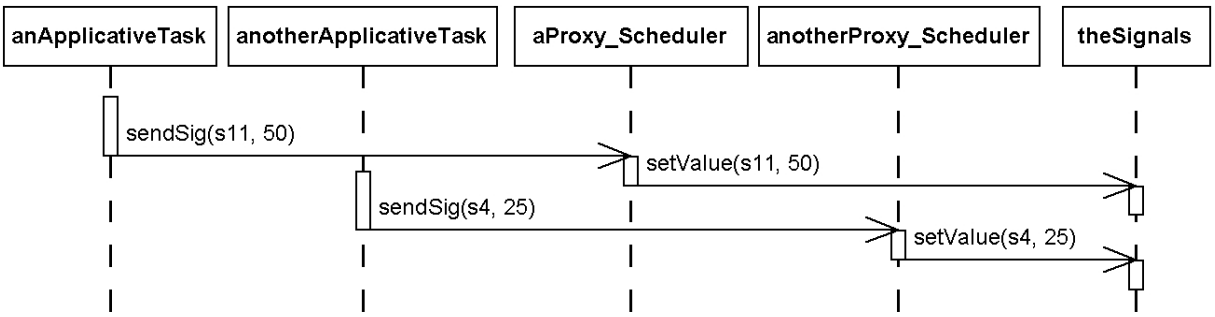


Figure 6.11: Sequence diagram illustrating the production and storage of two signal values produced by two different applicative tasks.

6.3.2.2 Retrieval of signal values

The retrieval of signal values from the repository is also performed by an object of class *Proxy_Scheduler*, executing on behalf of the applicative tasks. This object is the same that stores signal values in the

repository. Nevertheless, no concurrent access will be made on this object since it is associated to one applicative task, which cannot request both functionalities at the same time. The sequence diagram illustrating the list of methods executed to accomplish this functionality is presented in figure 6.12. It demonstrates the retrieval of two signal values and their subsequent consumption by two different applicative tasks. Again, class *Signals* (shared resource managed by OSEK/VDX OS) is displayed as a normal object.

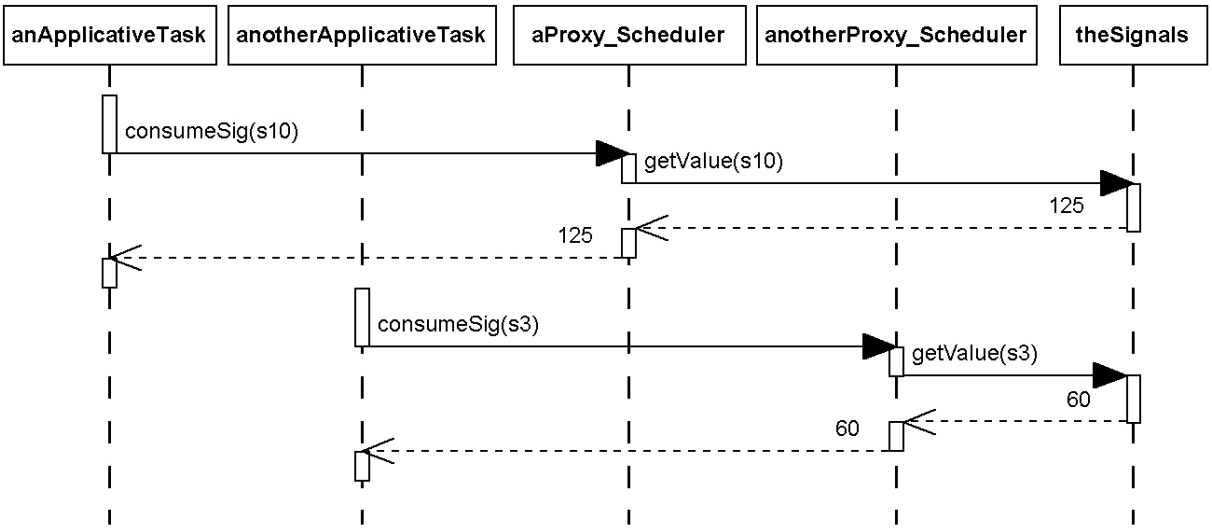


Figure 6.12: Sequence diagram illustrating the retrieval of two signal values, and their consumption by two different applicative tasks.

6.3.2.3 Construction and transmission request of frames

As defined in chapter 5, this functionality is going to be executed in a middleware task, belonging either to the multiframe or to the generalized multiframe task model. Thus, instances of the task will not have the same execution time. This variation depends on the number of frames to construct, and on the amount of signals carried by each frame. This information will be set during the timing constraints dependent part of the methodology, specifically, in the chapter where middleware tasks are characterized. Nevertheless, the code that this task is going to execute can be abstracted from the characterization parameters.

Figure 6.13 demonstrates the sequence of methods that cooperate to construct and request transmission of two frames. The first frame carries one signal, while the second frame is composed of two. There will be then one object from class *Core*, the main class, responsible for the construction of the frames. This object uses one *Frame* object for each frame that is constructed. For the construction of the sequence diagram it was assumed that these *Frame* objects were created previously, during system start-up time for instance. Finally, there is one object instantiated from a network adapter class (concrete class inheriting from the *Comm* abstract class). This object receives the *Frame* objects, and obtains all the necessary information to issue a frame transmission request on the underlying network controller. In figure 6.13, it is assumed that the used network is CAN, hence, the adapter object is of type *AdCAN*.

6.3.2.4 Reception and handling of frames

The functionality in charge of receiving and handling frames is executed by a sporadic task activated by network controller interrupts (see chapter 5). Once more, the characterization of this task (performed in chapter 8) does not influence the set of methods called during the reception and handling of frames. In figure 6.14 is shown the sequence diagram illustrating this set of methods.

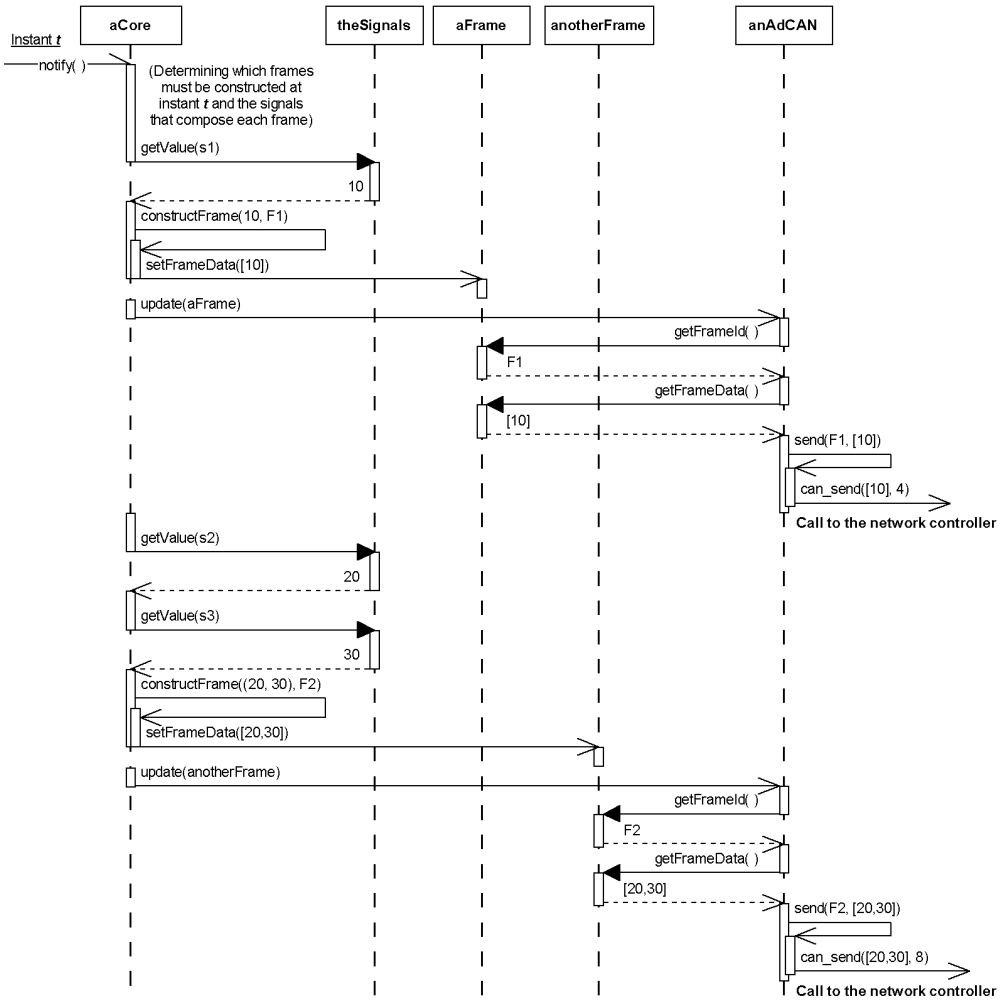


Figure 6.13: Sequence diagram detailing the construction and transmission request of frames.

This diagram lists the methods that participate in the construction of two frames. The first frame carries one signal, while the second frame is composed of two.

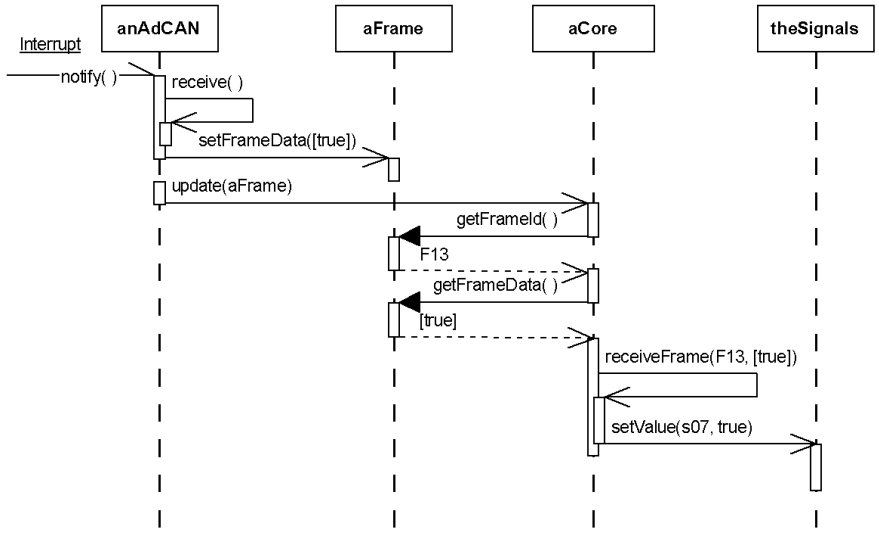


Figure 6.14: Sequence diagram detailing the reception and handling of frames.

This diagram lists the methods that participate in the handling of one frame carrying one signal.

In this figure one can perceive the set of objects that must be instantiated in order to perform this functionality. There is one network adapter object, created from class AdCAN in this scenario, which is responsible for gathering the data of the frame that has just arrived. This adapter object uses one *Frame* object for each frame that is received, and since this information will be part of the task configuration, all these *Frame* objects can be created beforehand. At last, one *Core* object will receive the *Frame* objects, will retrieve the frame data, and will be able to make the new signal values available to applicative tasks, by storing them in the signals repository (an object of class *Signals* in this example).

6.3.3 Complementary code

The code expressed in the methods declared in the class diagram (see figure 6.9), and assigned to the library and to the tasks of the middleware, does not take into account the operating system that is used, in our context OSEK/VDX OS. In order to tune the tasks to the OS, some code that must be added or modified.

The first specificity to overcome is the utilization of a shared resource, represented by class *Signals* in the class diagram. The access to the signals repository must use the service functions provided by OSEK/VDX OS. Calls to these functions must replace the code specified in the class diagram for the utilization of class *Signals*. As demonstrated in the sequence diagrams presented above, all functionalities of the middleware are concerned by this modification of the code.

The characterisation of the middleware tasks (see chapter 5) can lead to an implementation using a generalized multiframe task for the construction and transmission request of frames. If this situation occurs, then the code of the task must be changed, in order to cancel and set a new timing alarm each time the task is activated. Once again, some code must be added; instead of beginning its execution by determining which frames must be constructed, each instance of the task begins by canceling the previous alarm and setting the new one.

All these changes in the code concern the need to add calls to OSEK/VDX OS functions. Recall that the goal of the middleware presented in this work is to hide the heterogeneity of communication platforms. However, just like for the communication network, the middleware could be independent of the used OS. For this purpose, one could apply the adapter pattern [41] by creating an abstract class specifying a

standard set of OS functions, and establishing concrete classes translating each standard function into a service of a particular OS. In the context of the middleware, one would establish a OSEK/VDX OS adapter class.

Finally, the last piece of code that must be added is related to the configuration of the multiframe (or generalized multiframe) task. As stated in chapter 5, the activation period(s) of this task must respect all the frames transmission rate. To do so, each instance of this task will construct and request transmission of a different set of frames. The data informing the task about which frames should be built at which instance, may be part of a configuration file read at system start-up for example. The piece of code that performs such reading must also be added to the task software.

6.4 Conclusion

At the end of the previous chapter the set of tasks implementing the middleware was defined, as well as the role of the tasks and the task model to which they belong. This kind of information, independent of any timing constraint, is important for the characterization of the middleware tasks. To perform this characterization, it is necessary to have the data that will be handled by the middleware (frames and signals), but also the code that will execute and allow the middleware to handle the data. In particular, with this code it will be possible to define a worst-case execution time for each task, in order to perform a task schedulability analysis.

The goal of this chapter was then to construct the software model of the middleware. Specifically, it was focused in the design of the middleware software, and in the expression of the code executed by each middleware task. This step of the methodology is also independent of any timing constraint because the code executed by each middleware is the same on each ECU. The difference will be in the configuration parameters of each task.

The construction of the software model of the middleware was based on the object-oriented paradigm, and, in particular, built using object-oriented design patterns. These artifacts are reusable practices and design units, which have been proven useful in the development of software. To take advantage of these solutions in the construction of the middleware software, some design patterns were selected: the *Active Object*, the *Adapter*, the *Observer*, and the *Asynchronous Completion Token*.

Then, by composing these patterns together, a class diagram representing the structure of the middleware software was established. The main consequence is that the middleware code was divided in two parts. One part is executed on behalf of the applicative tasks, under the form of a library. The other part is processed by a separate set of tasks. This subdivision has the advantage to be well-suited with the development process of the automotive software.

From the class diagram, the next step was to identify the code executed by the library and by each middleware task. For this intention, the first aspect to be handled was the list of objects to instantiate, and the chain of methods that would be executed. The second aspect was the complementary code that should be added to make the middleware tasks and the library more suited to execute on top of OSEK/VDX OS.

At the end of this chapter, one is able to understand how the middleware software has been built. Moreover, one knows how this software is related with the set of tasks implementing the middleware. This information was part of a work that has been presented in [65, 73].

Now that all the methodology steps independent of any timing constraint have been presented, we shall discuss the construction and configuration of the elements that are guided by local timing requirements. As pointed out in chapter 5, the first step of the timing constraints dependent part of the methodology is the construction and configuration of a feasible set of frames.

Chapter 7

Configuration of the frames

7.1 Introduction

A major communication service provided by the middleware is to make produced signals available for consumer applicative tasks, and this wherever these tasks may be located. If the producer and the consumers tasks of a signal execute in different ECUs, then the sender middleware task (located in the producer ECU) requests the transmission of the frame that contains the signal. The problem is to decide:

- which frames are to be transmitted by this ECU,
- which signals integrate each frame,
- what is the transmission rule of each frame, and
- how to best configure the communication, for instance, on a priority based Medium Access Control (MAC) such as CAN, where one has to choose the priority of each frame so that each signal meets its freshness contraints.

At this step, the traffic generated by all ECUs that are connected to the network must be taken into account.

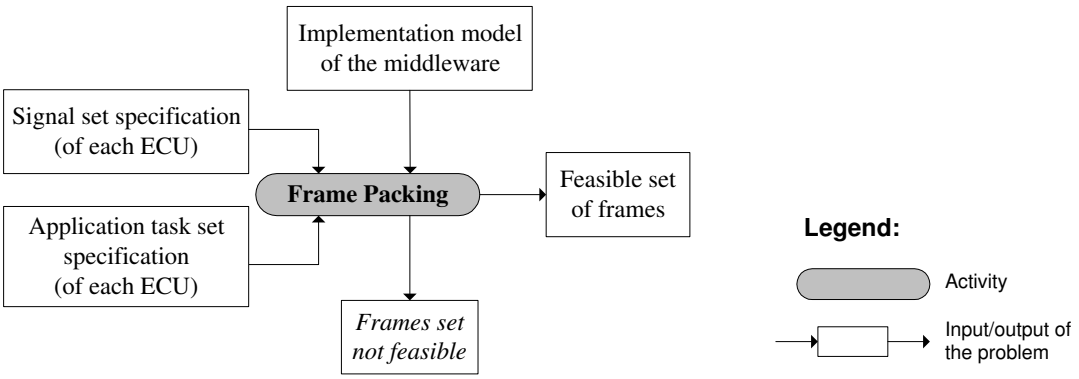


Figure 7.1: Input and output data of the activity in charge of building the frame packing configuration.

The middleware functionality that addresses this problem is called frame packing. It is responsible for constructing frames and requesting their transmission, and acts according to an off-line generated configuration. The objective of this chapter is to present two frame packing algorithms capable of generating and configuring off-line a set of frames for a CAN in-vehicle network. These algorithms can be used to implement the first activity of the timing constraints dependent part of the methodology.

This activity is illustrated in figure 7.1. The parameters composing the frame packing configuration are determined in order to fulfill three objectives:

- minimization of the network bandwidth consumption,
- feasibility of the frames, and
- respect of the freshness constraints associated to the signals.

The respect of the signals freshness constraints must be guaranteed because automotive functions are subjected to stringent timing restrictions, and, for this reason, signals have usually a limited lifetime. For instance, the engine controller function is executed with periods of milliseconds (due to the rotation speed of the engine), and each time it consumes the signal coming from the accelerator pedal, the value must report the current state of the accelerating angle. Therefore, the frame packing configuration must guarantee that frames are constructed and transmitted at the right points in time, leading to the respect of the freshness constraints of the signals.

The relative deadline of the frames is calculated according to the freshness constraints of the signals. The violation of a frame relative deadline can result in the disrespect of the signals timing constraints. Thus, the frame packing configuration is also obliged to guarantee that the MAC level priorities assigned to the frames will always ensure the respect of their relative deadline.

The minimization of the network bandwidth consumption is an important criteria for two reasons. The first reason lies on the fact that the possibility to add new automotive functions under the form of one or several ECUs must be preserved. Adding more ECUs implies that more signals will be exchanged on the bus. Such an incremental design is a standard procedure in the automotive industry. The second reason is to allow the use of low power processors, which are less expensive. The lower the number of frames an ECU transmits and receives, the higher the probability of being able to use devices of this type.

7.1.1 Formulation of the problem

The input data of the frame packing algorithms is constituted by the set of timing characteristics of the applicative tasks and signals, and by the implementation model of the middleware (see figure 7.1). Moreover, in order to build a set of frames adapted to a CAN bus, this activity also needs to have the number of ECUs attached to the bus, as well as, its transmission rate.

Specifically, the set of timing characteristics of the applicative tasks and signals is characterized as follows. Each ECU i contains a finite set $\Delta_i = \{\delta_{i,1}, \delta_{i,2}, ..., \delta_{i,j}\}$ of applicative tasks, where $\delta_{i,j}$ symbolizes applicative task j executing on ECU i. Each $\delta_{i,j}$ is characterized by a tuple $(C_{\delta_{i,j}}, T_{\delta_{i,j}}, \bar{D}_{\delta_{i,j}}, s_{i,j}, S_{i,j})$ where:

- $C_{\delta_{i,j}}$ is the worst-case execution time for task $\delta_{i,j}$,
- $T_{\delta_{i,j}}$ is the minimum inter-arrival time of two successive instances of $\delta_{i,j}$ (termed activation period in the following),
- $\bar{D}_{\delta_{i,j}}$ is the relative deadline, which is the maximum tolerable time interval between the activation of an instance of the task and the end of execution of this instance,
- $s_{i,j}$ is the output signal produced by task $\delta_{i,j}$ that is characterized by its size $C_{s_{i,j}}$ in bits. Its production period is equal to the period of task $\delta_{i,j}$,
- and $S_{i,j}$ is the set of signals consumed by $\delta_{i,j}$ that are characterized by pairs $(s_{i',j'}, \mathcal{F}_{\delta_{i,j}}^{s_{i',j'}})$ where
  - $s_{i',j'}$ is a signal produced in ECU i' by applicative task $\delta_{i',j'}$, and
  - $\mathcal{F}_{\delta_{i,j}}^{s_{i',j'}}$ is the freshness constraint requested by $\delta_{i,j}$ on signal $s_{i',j'}$.

This freshness constraint, briefly introduced in chapter 1, imposes a limit on the maximum age of the signal value. Precisely, the time interval between the consumption of a signal value and the activation of the instance of the task that produced that value, must be less than, or equal to, the freshness constraint. For instance, the number of RPM (Revolutions Per Minute) needed to compute the speed of the vehicle must have been read at the engine controller no more than 20ms before the speed is used by the active suspension. Figure 7.2 illustrates this constraint with respect to signal $s_{i',j'}$: when an instance of applicative task $\delta_{i,j}$ consumes the value α, its age must be less than, or equal to, the freshness constraint $\mathcal{F}_{\delta_{i,j}}^{s_{i',j'}}$.

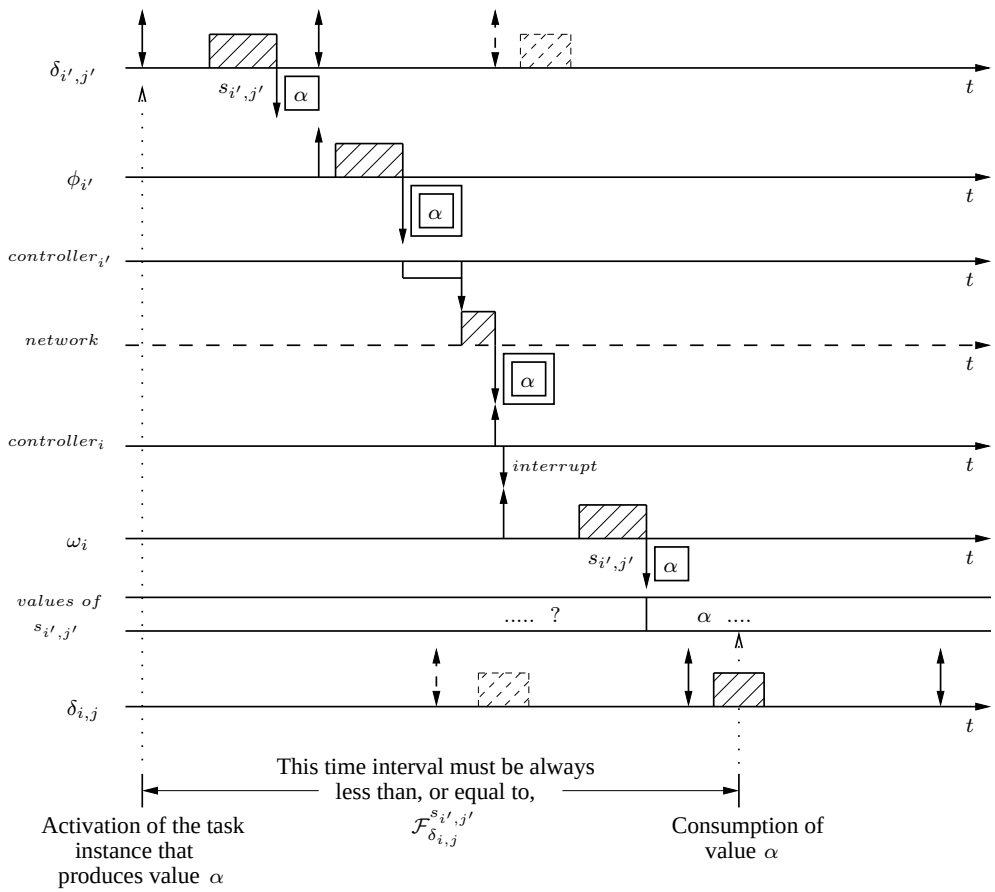


Figure 7.2: Scenario in which the exchange of signal $s_{i',j'}$ is constrained by a freshness constraint.

The configuration of the applicative and middleware tasks and the network frame must guarantee the respect of the freshness constraint $\mathcal{F}_{\delta_{i,j}}^{s_{i',j'}}$ imposed on $s_{i',j'}$. Double square boxes represent a network frame that, in this case, carries value α.

The problem addressed by the frame packing activity is to constitute a set of frames that minimizes the bandwidth consumption and respects the timing constraints of the frames and signals. Each frame $f_{i,k}$, belonging to the set of frames F_i transmitted from ECU i, is characterized by a tuple $(C_{f_{i,k}}, T_{f_{i,k}}, \bar{D}_{f_{i,k}}, S_{f_{i,k}}, P_{f_{i,k}})$ where:

- $C_{f_{i,k}}$ is the size of $f_{i,k}$ in bits,
- $T_{f_{i,k}}$ is the transmission period defining the time interval between any two consecutive transmissions,

- $\bar{D}_{f_{i,k}}$ is the relative deadline defining the maximum time interval tolerated between the transmission request and the reception of the frame at the network controller of all nodes,
- $S_{f_{i,k}}$ is the set of signals composing $f_{i,k}$, and
- $P_{f_{i,k}}$ is the priority of $f_{i,k}$, unique on the whole system, for the MAC protocol of the CAN network.

The priorities of the frames are assigned with the Audsley algorithm [58] (presented in chapter 2). However, in the general case, this algorithm cannot be applied to message scheduling under the Non-Preemptive Fixed Priority policy, which is the medium access strategy for priority buses. Indeed, on a network, the response time of a frame not only depends on the higher priority frames but also on the set of lower priority frames because of the blocking factor (maximal time interval during which one frame can be delayed by another with a lower priority). Since it will be assumed the existence of non real-time frames (e.g. diagnostic frames) having a priority lower than all real-time ones, the blocking factor will be equal for all frames. Without any other assumptions on this traffic, one must consider the blocking factor as being the size of the largest frame compliant with the CAN protocol: 128 bits (8 bytes of data plus the overhead). In this case, the conditions are fulfilled to apply the Audsley algorithm for assessing the feasibility of a set of frames (see [76] for more details).

The relative deadline is the characteristic that, if respected, will allow the feasibility of the frames, and, in particular, will meet the signals freshness constraints. Its definition is then one of the most important steps of the frame packing algorithms. In the following, it will be presented the strategy used by the algorithms to determine a frame relative deadline that takes into account the signals freshness constraints.

7.1.2 Guaranteeing freshness constraints

The aim of the middleware is to guarantee that any signal value meets its freshness constraints at the time it is consumed. However, nodes attached to a CAN bus are not synchronized. Hence, there is no constant time interval between the activation of the producer task, and the consumption instant of the signal value at a consumer node. Moreover, since tasks may be preempted during their execution, almost all consumption times are possible between their activation and the expiration of their relative deadline. The aim of the middleware becomes thus, to ensure that at any time, the value of a signal that is available to applicative tasks respects its freshness constraints.

A signal value that respects its freshness constraints at time t, is a value whose age at that time is less than, or equal to, its freshness constraints. The age of a signal value at time t is defined as the time interval between t, and the activation instant of the instance of the task that produced the value. Hence, the maximum age that a signal can reach, is the longest time duration between the activation of the task instance producing a value, and the moment where this value is updated at the receiver end (availability of the next instance of the signal). The approach is to ensure that the maximum age that any signal can reach will always be less than, or equal to, its freshness constraints.

7.1.2.1 The maximum age of a signal

To calculate the longest time duration between the activation of a task instance producing a signal value, and the moment where this value is updated at the receiver end, it is necessary to consider the execution of applicative and middleware tasks. However, at this point, one cannot predict how the middleware is going to execute since it will react according to the frame packing configuration (not yet determined). Thus, three different scenarios must be studied:

1. $T_{\delta_{i,j}} < T_{f_{i,k}}$: the activation period of the producer task (production period of the signal) is less than the transmission period of the frame carrying the signal;
2. $T_{\delta_{i,j}} = T_{f_{i,k}}$: the activation period of the producer task is equal to the transmission period of the frame carrying the signal;
3. $T_{\delta_{i,j}} > T_{f_{i,k}}$: the activation period of the producer task is greater than the transmission period of the frame carrying the signal.

Scenario 1 implies that some produced signals will not be handled by the middleware and will be lost. This scenario is thus ruled out since a frame packing algorithm must avoid this case. In scenarios 2 and 3 the maximum age of a signal depends on the values of $T_{\delta_{i,j}}$ (activation period of the producer task) and $T_{f_{i,k}}$ (transmission period of the frame). Figure 7.3 illustrates the exchange of value m_1 of signal $s_{i,j}$ that reaches its maximum age (worst-case scenario), and where $T_{\delta_{i,j}} = T_{f_{i,k}}$. Figure 7.4 shows the exchange of the same value (m_1) attaining the maximum age of the signal (worst-case scenario), but this time where $T_{\delta_{i,j}} > T_{f_{i,k}}$. These scenarios consider the middleware implementation model presented in chapter 5, and are based on the assumption that middleware and applicative tasks begin their execution simultaneously.

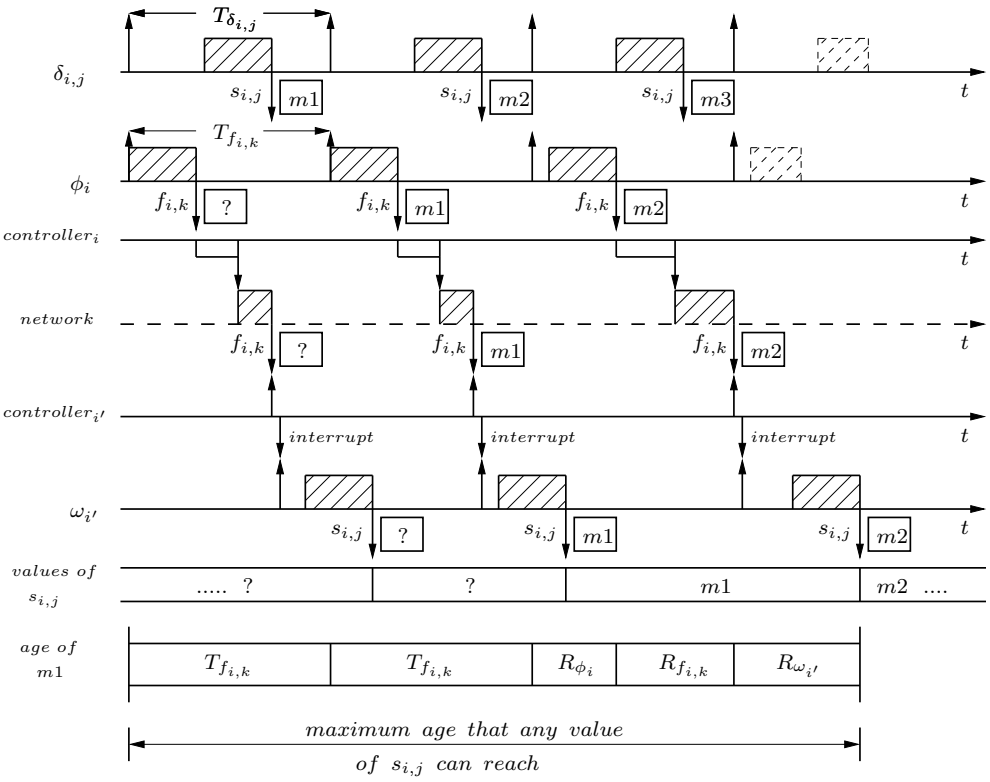


Figure 7.3: Scenario where value $m1$ reaches the maximum age of signal $s_{i,j}$, and the production and transmission periods of $s_{i,j}$ are the same.

As one can verify in both pictures, in the maximum age of a signal there is always at least a transmission period of the frame. The remaining time intervals are dependent on:

- $R_{f_{i,k}}$ - the worst-case response time of the frame (time spent waiting to obtain access to the CAN bus, plus the time necessary to transmit the frame),
- R_{ϕ_i} and $R_{\omega_{i'}}$ - the worst-case response times of the middleware tasks (sender and receiver side), and
- $\lceil \frac{T_{\delta_{i,j}}}{T_{f_{i,k}}} \rceil \cdot T_{f,i,k}$ - the number of times the frame can be transmitted during an activation period of the applicative task.

The maximum age of a signal $s_{i,j}$ that is transmitted in frame $f_{i,k}$ ($s_{i,j} \in S_{f_{i,k}}$) and consumed in ECU i' can be defined by the following expression:

$$\text{MaxAge}(s_{i,j}, i') = T_{f_{i,k}} + \lceil \frac{T_{\delta_{i,j}}}{T_{f_{i,k}}} \rceil \cdot T_{f,i,k} + R_{\phi_i} + R_{f_{i,k}} + R_{\omega_{i'}} \tag{7.1}$$

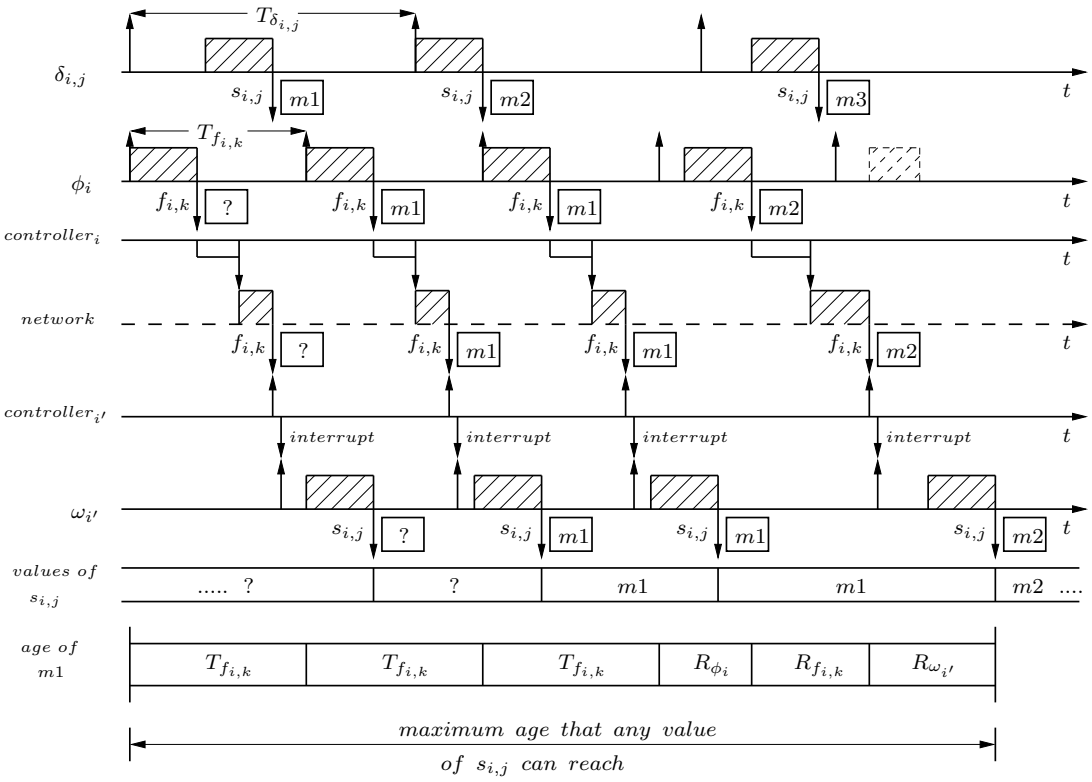


Figure 7.4: Scenario where value $m1$ reaches the maximum age of signal $s_{i,j}$, and the production period of $s_{i,j}$ is greater than its transmission rate.

The frame packing algorithms must construct a set of frames whose characteristics help the maximum age of each signal to respect their freshness constraints. However, if the ECUs have different processor speeds, then the value of the worst-case response time of the receiver middleware ($R_{\omega_{i'}}$ in expression 7.1) may vary. Hence, one must consider the maximum age of a signal for each of its consumer ECUs. Moreover, a signal can be consumed by different applicative tasks in one ECU, where each consumer task demands a different freshness constraint. Hence, one denotes $\mathcal{F}_x^{s_{i,j}}$ as the most stringent freshness constraint demanded on signal $s_{i,j}$ by its consumer tasks residing on ECU x. To guarantee that every signal $s_{i,j}$ respects $\mathcal{F}_x^{s_{i,j}}$, the frame packing algorithms must then construct a set of frames whose properties can help to verify the following:

$$\forall\, s_{i,j},\, x, \qquad \text{MaxAge}(s_{i,j},\, x) \leq \mathcal{F}_x^{s_{i,j}} \tag{7.2}$$

where x in an ECU that contains at least one applicative task consuming signal $s_{i,j}$.

7.1.2.2 The relative deadline of a frame

As one can see in equation 7.1, the time spent for the transmission of the frame (in component $R_{f_{i,k}}$) is one of the components of the maximum age of a signal, and contributes for the respect, or not, of the freshness constraints. In this inequality, only the value of the activation period of the applicative task is known in advance. The remaining values are dependent on the implementation model of the middleware (activation model of the tasks), and on the result of the frame packing algorithms, leading to the dependency cycle mentioned in chapter 4.

Nevertheless, while calculating the relative deadline of a frame with a specific signals composition, the algorithms are aware of the current frame transmission period (the smallest production period of the signals composing the frame). Moreover, at that point they can also assume values for the worst-case response time for the middleware tasks (sender and receiver side). These assumptions, which help breaking the dependency cycle, will be detailed in section 7.1.2.3. Thus, to determine a relative deadline of a frame with a given signals composition, one uses inequality 7.2 and the result of these assumptions. The following expression illustrates the approach:

$$\forall s_{i,j} \in S_{f_{i,k}},\, x = \{p \mid \exists q,\, s_{i,j} \in S_{p,q}\}, \text{ (for each signal } s_{i,j} \text{ and each ECU } x \text{ where } s_{i,j} \text{ is consumed)}$$

$$\bar{D}_{f_{i,k}} = \min_{\forall s_{i,j}}(\mathcal{F}_x^{s_{i,j}} - T_{f_{i,k}} - \lceil \frac{T_{\delta_{i,j}}}{T_{f_{i,k}}} \rceil \cdot T_{f,i,k} - R_{\phi_i} - R_{\omega_x}) \tag{7.3}$$

The result of this expression for one signal, is the remaining time interval when we subtract from the signal freshness the maximum duration the signal spends inside the producer and consumer ECUs. This maximum duration is determined using the assumptions made on the worst-case behaviour of the middleware tasks (see section 7.1.2.3). Since a frame might contain several signals, the relative deadline of the frame is the minimum value after applying each signal on the expression.

This expression is evaluated each time the frame packing algorithm determines the relative deadline for the current signals composition of a frame. A frame can then contain the current set of signals if the result of equation 7.3 is a positive value. But, to be more accurate, the current set of signals is valid if the result of equation 7.3 is greater than, or equal to, the time necessary to transmit the frame, that is, if $\bar{D}_{f_{i,k}} \geq \frac{C_{f_{i,k}}}{network\,bandwidth}$. Otherwise, this frame will definitely not be feasible. Note that although a frame signals composition is considered valid when $\bar{D}_{f_{i,k}} = \frac{C_{f_{i,k}}}{network\,bandwidth}$, it is clear that if more than one frame are in this situation, then a feasible priority allocation for the set of frames to be sent over CAN will definitely not be found. While constructing the set of frames, the algorithms must avoid this scenario.

7.1.2.3 Assumptions on the worst-case behaviour of the middleware tasks

The executions of the middlewares on the sender and receiver side are dependent on the results of the frame packing algorithms. Since the worst-case response times of the middlewares are used by the frame

packing algorithms in order to determine a relative deadline for the frames, there is a dependency cycle. To overcome this problem, we are going to assume worst-case behaviours for the middleware tasks. If under these conditions the relative deadlines assigned to frames are feasible and the freshness constraints are met, then they will also be with the actual executions. Note that these worst-case behaviours are always related to the signal that is being applied in expression 7.3. With these worst-case scenarios for the behaviour of the middleware, all the conditions necessary are gathered to use the relative deadline calculation method introduced in section 7.1.2.2.

Receiver side

According to the implementation model defined in chapter 5, the middleware task in charge of receiving and handling frames ($\omega_{i'}$ in figures 7.3 and 7.4) is implemented by an OSEK/VDX OS interrupt service routine (ISR). The specification of OSEK/VDX OS declares that ISRs have always a higher priority than any task. Thus, the worst-case behaviour of the ISR is simply:

- the worst-case execution time associated to the handling of the signal (being applied in expression 7.3),
- plus the maximum blocking time due to shared resources (signals repository) managed by the PCP protocol [67].

However, since at this stage we do not have the complete set of frames that each ECU is going to receive, we cannot determine the worst-case execution time of the ISR. To overcome this situation, we assume that the worst-case execution time of the ISR associated to a signal, occurs when it handles a frame containing the signal plus as many others as possible (produced in the same ECU) without violating the frame maximum size. Moreover, without a tasks priority allocation one cannot determine the maximum blocking time due to shared resources. Let us then assume that this blocking time duration is equal to the longest time interval during which an applicative task locks the signals repository for consumption.

Sender side

The task responsible for the construction and transmission request of frames (ϕ_i in figures 7.3 and 7.4) adheres to the multiframe task model. Hence, the longest execution time (from those that compose the execution times vector) occurs when the task has to transmit all the frames that are constructed in the ECU (for example, execution of the task at instant 0). Again, we do not have neither the set of frames that must be transmitted by each ECU, nor the tasks priority allocation. Thus, the worst-case behaviour of the task associated to a signal becomes to:

- the time necessary to construct and request the transmission of the frame where this signal is being tested in expression 7.3,
- plus the time needed to construct and request the transmission of the remaining signals produced in the ECU, and where each signal is inserted in one frame,
- plus the maximum blocking time due to shared resources (signals repository); we are going to consider the longest time interval during which an applicative task locks the signals repository for production.

To complete this worst-case scenario, one must integrate the interference caused by the ISR, and by the applicative tasks having a higher priority. We are going to assume that middleware tasks belong to system level, and consequently, they always have a higher priority than any applicative task (the reader can refer to chapter 8 for more details). Nevertheless, it is still necessary to consider a worst-case scenario for the ISR for the purpose of determining its interference on the middleware task constructing frames.

The worst-case behaviour of the ISR must consider its execution time, as well as its activation period, which is helpful to determine the number of times the ISR will interfere during the middleware task execution. The worst-case behaviour in terms of execution time is:

- the time duration necessary to handle the largest frame that can be constructed with signals produced in the same ECU an consumed in this station,

- plus the blocking time, which is equal to the longest time interval during which an applicative task locks the signals repository for consumption.

The worst-case behaviour with respect to the activation period is simply the time interval needed to transmit over the CAN network the smallest frame that can be constructed with the signals consumed in this station.

7.1.3 Complexity of the problem

The problem of building frames from signals is similar to the *bin-packing* problem and it was proven to be NP-complete in [60]. Nevertheless on small size problems an exhaustive approach could be envisaged. To assess whether such an exhaustive search is possible, one has to determine the exact complexity of this specific problem.

To group a set of n signals in k non-empty frames comes to create all the partitions of size k from the set of signals. The complexity of this problem is known, it is the Stirling number of the second kind (see [77] page 824):

$$\frac{1}{k!}\sum_{i=0}^{k}(-1)^{(k-i)}\binom{k}{i}i^n$$

The number k of frames per ECU can vary from 1 to n where n is the number of signals produced by the ECU. The number of frames to envisage on an ECU i that produces n signals is thus:

$$\mathcal{S}_i = \sum_{k=1}^{n}\frac{1}{k!}\sum_{i=0}^{k}(-1)^{(k-i)}\binom{k}{i}i^n$$

For example, three signals a, b, and c on a same ECU lead to five different partitions : $[(a,b,c)]$, $[(a),(b,c)]$, $[(a,b),(c)]$, $[(a,c),(b)]$ and $[(a),(b),(c)]$. If we consider a set of m ECUs, the solutions space becomes $\prod_{i=1..m}\mathcal{S}_i$. The solutions space grows very quickly as soon as n and m increase. For instance, with 10 signals per ECU ($\mathcal{S}_i = 115\,975$) and 5 ECUs, an exhaustive search would need to consider about $2\cdot 10^{25}$ cases. Moreover, this evaluation did not take account of the determination of frame priority. One can see that for real industrial cases an exhaustive approach cannot be applied. This justifies the design of specific heuristics.

7.1.4 Heuristics proposed to solve the problem

Two frame packing algorithms, under the form of heuristics, are going to be presented. The first heuristic is based on known bin-packing heuristics, and can be used to solve problems of any size. The second heuristic is an exhaustive approach, and hence, can only be applied in limited size problems. Both heuristics construct sets of frames whose feasibility is assessed afterwards. They stop their execution when a feasible solution is found.

The evaluation of the performance of the heuristics was presented in [78], and is detailled in appendix B. These results were obtained with previous versions of the heuristics, where the execution time induced by the middleware was not considered and the notion of freshness constraint was different. We intend to present in the future the results of a performance evaluation that is being conducted. We want to verify that the execution time of the middleware, and the new notion of freshness constraint, do not invalidate our previous results.

7.2 The "Bandwidth-Best-Fit decreasing" heuristic

The first heuristic that is going to be presented is named *Bandwidth-Best-Fit decreasing* (BBFd). The name of this heuristic has been given by analogy with the terminology used in the bin-packing problems with off-line resolution (*Best-Fit decreasing, First-Fit decreasing*, see [79]). In the *bin-packing* problems, the objective is to minimize the number of boxes given a set of objects of different sizes that must be placed inside those boxes. In our context, the goal is to minimize the network bandwidth consumption without taking account of the number of frames. The underlying idea of the heuristic is to place a signal in the frame that minimizes the most the supplementary bandwidth consumption induced by the signal.

The heuristic builds only one solution whose frames have characteristics that help to guarantee the respect of the signals freshness constraint (according to inequality 7.2). The feasibility of the frames is assessed with the Audsley algorithm. If the solution is feasible then a local optimization procedure is applied. If the Audsley algorithm fails to find a feasible priority assignment, the heuristic applies some transformations on the frames by isolating the most demanding signals (those with the smallest freshness constraint), in order to shift some load to lower priority levels - this procedure is termed *decomposition* of the frames. The general algorithm of the BBFd heuristic is given in figure 7.5.

Sections 7.2.1 and 7.2.2 detail the steps of the heuristic. Section 7.2.3 presents the local optimization procedure applied in the solution.

```
FrameList BBFD(NodeList N) {
    FrameList F;      /* list of frames from all the ECUs */
    for each ECU N_i ∈ N do {        /* i ∈{1,...,m} where m is the number of ECUs */
        SignalList S_i, S'_i;
        S_i = list of signals produced in N_i;
        S'_i = sort(S_i);                /*S'_i is the list of signals of S_i sorted in decreasing
                                            order of bandwidth consumption */
        FrameList F_i = {};              /* list of frames sent by N_i */
        for each signal s_{i,j} ∈ S'_i do
            if( insert(s_{i,j}, F_i) == ERROR ) then
                return {};     /* FAIL - no solution */
        F = F + F_i;
    }
    FrameList F_f̄ = {};
    do {
        F_f̄ = Audsley(F);   /* F_f̄ is the list of frames for which no priority was found */
        if( F_f̄ ≠ {} ) then
            F = decompose(F, F_f̄);
    } while( F ≠ {} and F_f̄ ≠ {} );
    if( F_f̄ == {} ) then
        return F;   /* SUCCESS */
    return {};      /* FAIL - no solution */
}
```

Figure 7.5: Description of the BBFd heuristic.

7.2.1 Insertion: function *insert()*

This step is executed when the heuristic tries to insert a signal into a frame (see figure 7.5). It begins by constructing a set with the frames that by accepting the signal will not violate the maximum data size imposed by the network protocol (including an empty frame). Then, it sorts the frames in increasing order of augmentation of bandwidth consumption caused by accepting the signal. The signal is placed in the first frame for which a valid relative deadline is calculated (see section 7.1.2).

The algorithmic description of the *insert* function of the heuristic is given below. This function receives signal $s_{i,j}$ and tries to insert it in one of the frames composing set F_i.

1. $insert(s_{i,j}, F_i)$:
   (a) set X is composed of the frames of F_i that by accepting $s_{i,j}$ will not violate the maximum data size imposed by the network protocol: $X = canAccept(F_i, s_{i,j})$
   (b) an empty frame is added to set X: $X = X + \{f = \emptyset\}$
   (c) set $\overrightarrow{X}$ is constructed by sorting the frames of set X in increasing order of augmentation of bandwidth consumption caused by accepting $s_{i,j}$: $\overrightarrow{X} = bandwConsAugm(X)$
   (d) $\alpha = false$
   (e) for each $f_{i,k} \in \overrightarrow{X}$ do:
      i. $d = deadline(f_{i,k} + \{s_{i,j}\})$ /* see section 7.1.2 */
      ii. if $d \neq ERROR$ then
         A. $f_{i,k} = f_{i,k} + \{s_{i,j}\}$
         B. $\bar{D}_{f_{i,k}} = d$
         C. $T_{f_{i,k}} = \min(T_{f_{i,k}}, T_{\delta_{i,j}})$ /* $T_{\delta_{i,j}}$ is the activation period of the task producing $s_{i,j}$ */
         D. $\alpha = true$
         E. break
   (f) if $\alpha == false$ then return $ERROR$
   (g) return OK

The function stops its execution either when the signal is successfully inserted in a frame, or when there is not one single frame capable of accepting the signal. In particular, function *deadline*() returns $ERROR$ if, with the current frame signals composition, the calculated relative deadline is less than the time necessary to transmit the frame (see section 7.1.2).

7.2.2 Decomposition: function *decompose()*

The decomposition step is executed when the Audsley algorithm fails to find a feasible priority assignment for the set of frames (see figure 7.5). The idea is to reduce the deadline constraints of the frames by isolating the most demanding signals. The frame initially chosen for the decomposition is the one that exceeds the least its relative deadline at the lowest priority level that has not been successfully assigned. It is thus the frame that is the more likely to respect its relative deadline if it should contain less data. The algorithmic description of function *decompose* is given below. It receives set F that is the list of frames from all ECUs, and set $F_{\bar{f}}$ containing the frames for which no priority was found.

1. $decompose(F, F_{\bar{f}})$:
   (a) if all frames in $F_{\bar{f}}$ contain one signal then return $\{\}$
   (b) find the frame in $F_{\bar{f}}$ containing at least two signals, and which has the least difference between its worst case response time and its relative deadline at the lowest priority that has not been assigned. Let this frame be $f_{i,k}$
   (c) choose s from $f_{i,k}$ such that s is the signal with the smallest freshness constraint: $s = s' \in f_{i,k} \vdash \mathcal{F}_x^{s'} = \min_{\forall s_{i,j} \in f_{i,k}}(\mathcal{F}_x^{s_{i,j}})$ where x is the index of an ECU that contains at least an applicative task consuming signal $s_{i,j}$
   (d) remove s from $f_{i,k}$ and update its characteristics (transmission period, relative deadline and size, as shown in section 7.2.1): $f_{i,k} = f_{i,k} - \{s\}$

(e) create new frame $f_{i,n+1}$, insert s, and set the characteristics of $f_{i,n+1}$: $f_{i,n+1} = f_{i,n+1} + \{s\}$ where n is the number of frames transmitted by ECU i

(f) add $f_{i,n+1}$ to F: $F = F + \{f_{i,n+1}\}$

(g) return F

Function *decompose* terminates when there is no frame capable of being split, or when a signal has been successfully moved from an old frame to a new one.

7.2.3 Local optimization algorithm

When the BBFd heuristic returns a feasible solution ($F_{\bar{f}} = \{\}$ in figure 7.5), one can additionally execute a local optimization (LO) algorithm as it is classically done in optimization problems (see for instance [80]). We propose a LO strategy that basically tries to reduce the bandwidth usage by permuting pairs of signals or by moving some signals. This procedure is executed on each ECU, and the solution of the LO becomes the current best solution if it improves the bandwidth consumption and it is feasible. The following algorithm describes the LO strategy executed in each ECU i:

1. $\{f_{i,a}, f_{i,b}\}$ are two frames randomly chosen from the set of frames with more than one signal, and bc_{before} is the bandwidth consumption of $\{f_{i,a}, f_{i,b}\}$. Choose randomly one signal from each frame: $s_{i,a}$ from $f_{i,a}$ and $s_{i,b}$ from $f_{i,b}$ and execute the following operations:

   (a) remove $s_{i,a}$ from $f_{i,a}$ and insert it in $f_{i,b}$. If the size of $f_{i,b}$ is smaller or equal than the protocol maximum and if the deadline of both frames is valid (see section 7.1.2), then bc_{after1} = bandwidth consumption of $\{f_{i,a}, f_{i,b}\}$, otherwise $bc_{after1} = \infty$

   (b) same procedure as step 1a with $s_{i,b}$ instead of $s_{i,a}$. The corresponding bandwidth consumption is bc_{after2}

   (c) remove $s_{i,a}$ from $f_{i,a}$ and insert it in $f_{i,b}$, remove $s_{i,b}$ from $f_{i,b}$ and insert it in $f_{i,a}$; if the size of both frames is smaller or equal than the protocol maximum and if their deadline is valid, then bc_{after3} = bandwidth consumption of $\{f_{i,a}, f_{i,b}\}$, otherwise $bc_{after3} = \infty$

2. if one of the bandwidth consumptions $\{bc_{after1}, bc_{after2}, bc_{after3}\}$ is lower than bc_{before}, then update the set of frames of the ECU with the best solution, otherwise go to step 1

3. feasibility test of the new set of frames with the Audsley algorithm:

   (a) the new set of frames is not feasible, undo the changes operated at step 2 and go to step 1

   (b) the new set of frames is feasible, go to step 1

The possibility of improvement comes from the fact that, by swapping or moving signals, one can test a configuration with a signal located in a frame that did not exist, or not with the same content, at the time when the signal insertion was decided (see section 7.2.1).

7.3 The "Semi-Exhaustive" heuristic

The second heuristic is called *Semi-Exhaustive*. This term has been chosen because this strategy is an exhaustive search through the solutions space where one would cut the branches that are judged not promising. This heuristic begins by the construction on each ECU of the set of all possible frames. In practice, the complexity of this first step, determined in section 7.1.3, does not allow us to deal with cases where there are more than 12 signals in one ECU ($\mathcal{S}_i = 4\,213\,597$ for $n = 12$ see section 7.1.3). With less than 12 signals per ECU, the generation of the set of possible frames can be done using the technique detailed in [81]. Once the construction of all possible frames is achieved, the heuristic sorts on each ECU the set of frames in increasing order of bandwidth consumption. Then, it sorts the ECUs in increasing order of their set of frames that minimizes the most the bandwidth consumption. Finally, the heuristic builds possible solutions with a set of frames from each ECU, tests their feasibility, and stops as soon as one solution is feasible. The algorithmic description of the Semi-Exhaustive heuristic is given below:

1. for each ECU i:

   (a) construction of F_i^* that is the set of all possible partitions of the set of signals produced in ECU i. For example, if 3 signals $\{a, b, c\}$ are produced in an ECU, then 5 possible partitions exist: $[\{a, b, c\}]$, $[\{a, b\}, \{c\}]$, $[\{a\}, \{b, c\}]$, $[\{a, c\}, \{b\}]$, and $[\{a\}, \{b\}, \{c\}]$. A partition is possible if all of its frames respect the maximum data size imposed by the communication protocol and have a valid relative deadline (see section 7.1.2). The transmission period of a frame is the smallest activation period of the tasks producing the signals it contains, while the deadline of the frame is set according section 7.1.2

   (b) the set F_i^* is sorted in increasing order according to the partition bandwidth consumption (i.e. the bandwidth consumed by all the frames of a partition)

2. $F_{i,1}^*$ is the partition that minimizes the most the bandwidth usage on ECU i. The ECUs are sorted in increasing order according to $F_{i,1}^*$ (e_k is the index of the k-th ECU in this new order). For each F_i^*, one defines a depth p_i from which no further search will be performed: the partitions $F_{i,j}^*$ with $j > p_i$ won't be considered since they are a priori the least satisfactory in terms of bandwidth consumption. The values chosen for p_i enable us to control the complexity of the heuristic. Let a_{e_k} be a temporary index variable initialized to 1 on each ECU e_k

3. the current set of frames is composed of the frames contained in the partitions $F_{e_1,a_{e_1}}^* \bigcup F_{e_2,a_{e_2}}^* \bigcup \ldots \bigcup F_{e_m,a_{e_m}}^*$ where m is the number of ECUs in the network

4. the feasibility of the set of messages is tested with the Audsley algorithm:

   (a) the configuration is not feasible: the next solution to test is built as illustrated in figure 7.6, according to the algorithm:

   for u:=e_m downto e_1 do

   if (a_u<p_u) then {

   a_u:=a_u+1;

   break;

   } else

   a_u:=1;

   return to step 3

   (b) the configuration is feasible: return SUCCESS

As soon as the first feasible solution is found, the algorithm stops. Nevertheless it is possible to pursue the search for a given duration or until a fixed number of feasible solutions are found. The algorithm does, at most, $\prod_{i=1..m} p_i$ calls to the Audsley algorithm. The choice of the values of p_i should be done according to the available computation time. It is worth to note that the algorithm finds the optimal solution in terms of bandwidth consumption in the particular case where the set of partitions $F_{e_1,1}^* \bigcup F_{e_2,1}^* \bigcup \ldots \bigcup F_{e_m,1}^*$ is feasible.

7.4 Conclusion

This chapter presented the frame packing activity, which is in charge of constructing and configuring the frames that are sent by each ECU. The goal was to define the set of signals composing each frame, as well as the characteristics that would allow each frame to meet its relative deadline, and to respect the freshness constraints imposed on the signals.

We have seen that with the purpose of ensuring the respect of the freshness constraints, the frame packing heuristics use the notion of maximum age of a signal. To assign a relative deadline to each

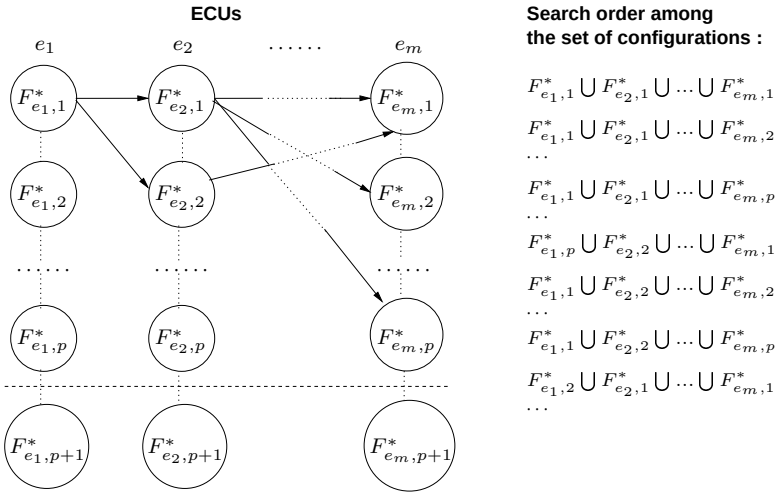


Figure 7.6: Search through the solutions space in the Semi-Exhaustive heuristic.

frame such that the maximum age of the signals does not violate their freshness constraints, one needs the characteristics of the middleware tasks, which creates a dependency cycle. This cycle is broken by considering worst-case behaviours for the middleware tasks. If under these behaviours the relative deadlines assigned to frames are feasible and the freshness constraints are met, then they will also be with the actual executions. We have then proposed worst-case behaviours for the middleware tasks executing in the producer and receiver sides according to a given signal.

Two frame packing heuristics were proposed that use these worst-case behaviours with the purpose of assigning a relative deadline to each frame. The first heuristic is based on known bin-packing heuristics, and can be used to solve problems of any size. The second heuristic is an exhaustive approach, and hence, can only be applied in limited size problems (less than 12 signals per ECU). Both heuristics represent an improvement in the field of frame packing since they raised and handled the issue of the feasibility of frames (compared with the proposals presented in chapter 2). Different versions of these heuristics were presented in two conferences [82, 78]. The version that consider the notion of freshness constraint presented in this document was published in [83].

At the end of this chapter one should:

- understand the basic principles of the execution of each frame packing heuristic proposed,
- know what are the characteristics of each frame that the heuristics determine, and specially,
- be aware on how the relative deadline of each frame is calculated, such that the maximum age of each signal does not violate its freshness constraints.

The activity presented in this chapter, is the first of the part of the methodology whose purpose is to try to configure the frames and the tasks with feasible parameters. In the next chapter, we are going to detail the activities that from the frames configuration, try to configure the applicative and middleware tasks.

Chapter 8

Configuration of the applicative and middleware tasks

8.1 Introduction

If the frame packing step of the methodology is successful, it returns a feasible set of frames. Moreover, the set of middleware tasks as well as the code executed by each task have also been defined in chapters 5 and 6. With all this information, the remaining steps of the methodology are:

- the complete characterization of the middleware tasks on each ECU, and
- the calculation of a feasible priority for the applicative tasks on each ECU.

The goal of this chapter is twofold. The first part is to address the configuration issues of the middleware, and specifically to define the parameters of the two middleware tasks that have been identified in chapter 5:

- the vector of execution times of the multiframe (or generalized multiframe) task, its activation period (or set of activation periods), and its relative deadline,
- the execution time, the activation period, and the relative deadline of the sporadic task, and
- a feasible priority for both tasks.

The second part handles the assignment of feasible priorities to the applicative tasks on each ECU. These two parts, composing the second and third steps of the timing constraints dependent part of the methodology (see chapter 4), are illustrated in figure 8.1.

The accomplishment of the first part of this chapter (characterization of the middleware tasks) depends on the parameters assigned to the frames. The second part of the chapter is dependent of the specification of the applicative tasks, and of the middleware tasks characteristics previously determined. To try to obtain a feasible assignment of priorities, the Audsley algorithm [58] (introduced in chapter 2) will be used.

8.2 Configuration of the middleware tasks

The characteristics of the middleware tasks are based on the parameters that constitute the frame packing configuration. In particular, these parameters are going to determine the execution time, the activation period, and the relative deadline. In the following, it is explained how these characteristics are calculated from the frame packing configuration. Then, it is detailed the assignment of priorities to the middleware tasks.

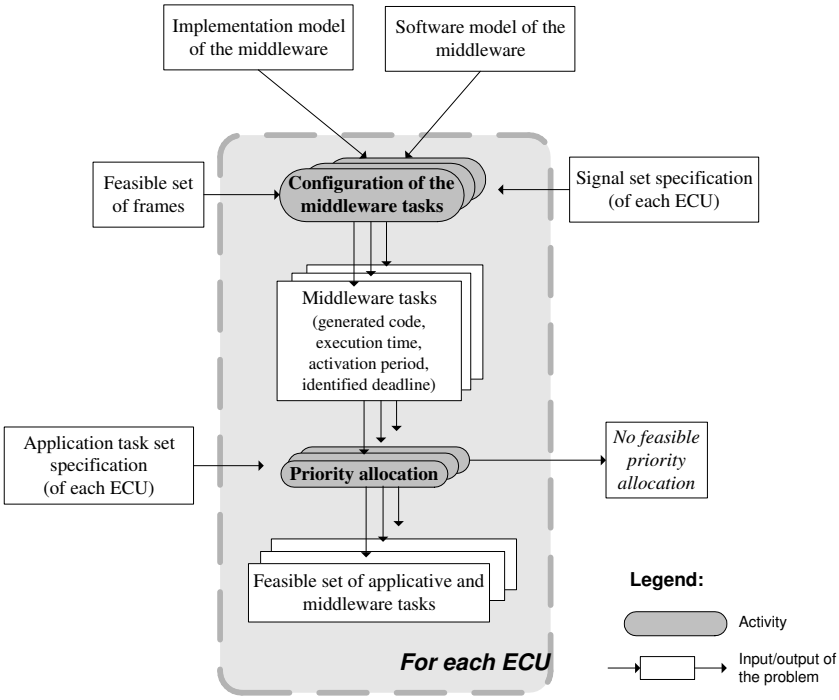


Figure 8.1: Input and output data of the activities responsible of the configuration of the middleware and applicative tasks on each ECU.

Specifically, these activities determine the complete configuration of the middleware tasks, and try to assign a feasible priority to each applicative task.

8.2.1 Task constructing and requesting transmission of frames

As pointed out in chapter 5, this task can be implemented either as multiframe [62], or as generalized multiframe [63]. We present in the following how to characterize a multiframe and a generalized multiframe task.

8.2.1.1 Implementation as a multiframe task

This model of task is characterized by a unique activation period and a set of execution times corresponding to successive instances of the task. The configuration of this task on each ECU i, denoted ϕ_i, is characterized by a tuple $(C_{\phi_i}, T_{\phi_i}, \bar{D}_{\phi_i})$ where:

- $C_{\phi_i} = \{c_{\phi_i}^{(0)}, c_{\phi_i}^{(1)}, ..., c_{\phi_i}^{(N-1)}\}$ is a set of N execution times; the execution time of instance a of task ϕ_i is given by $c^{(a \bmod N)}$ where $a \geq 0$, and the execution time of subsequent instances is obtained by cycling through the set C_{ϕ_i} in order,

- T_{ϕ_i} is the activation period defining the minimum time interval between two consecutive activations, and

- $\bar{D}_{\phi_i}$ is the relative deadline identifying the maximum time interval tolerated between the activation and the end of execution of task ϕ_i.

From the frame packing configuration, one derives the execution times and the activation period of the multiframe task ϕ_i as follows. It is assumed that the first emission request of all frames is issued by the first instance of the task (all frames are requested for transmission at instant 0). Let the set $Q_i = \{(Q_{f_{i,1}}, T_{f_{i,1}}), ..., (Q_{f_{i,k}}, T_{f_{i,k}})\}$ where $Q_{f_{i,k}}$ is the time needed to construct and request transmission of frame $f_{i,k}$, and $T_{f_{i,k}}$ is the transmission period of the frame. The activation period of ϕ_i, T_{ϕ_i}, is simply $\gcd(T_{f_{i,1}}, ..., T_{f_{i,k}})$. For each activation instant during the first hyperperiod, $\mathrm{lcm}(T_{f_{i,1}}, ..., T_{f_{i,k}})$, one determines the frames that are to be sent and, thus, the set of execution times for ϕ_i:

$$\forall\, 0 \leq a \leq \mathrm{lcm}(T_{f_{i,1}}, ..., T_{f_{i,k}})/T_{\phi_i} - 1,$$

$$c_{\phi_i}^{(a)} = \sum_{\{k \,|\, a \cdot T_\phi \bmod T_{f_{i,k}} = 0\}} Q_{f_{i,k}}$$

The relative deadline $\bar{D}_{\phi_i}$ is equal to the activation period T_{ϕ_i}, otherwise a set of frames may suffer a delay caused by the sending of a previous set. This situation is not further considered since it corresponds to a scenario where applicative tasks with a lower priority than the middleware cannot execute, and thus, cannot produce signals to be transmitted. Moreover, avoiding the multiple activation of a same task, allows car-makers to use a OSEK/VDX OS belonging to a conformance class that requires few resources[3].

Example of characterization: Let $Q_i = \{(3, 20), (6, 60), (4, 20)\}$ be the collection of pairs of:

- the time needed to construct and request transmission, and
- the transmission period

of frames $\{f_{i,1}, f_{i,2}, f_{i,3}\}$. The activation period (and relative deadline) of ϕ_i is:

$$T_{\phi_i} = \gcd(20, 60, 20) = 20.$$

Since $a \in [0, \mathrm{lcm}(20, 60, 20)/20 - 1] \Rightarrow [0, 2]$, the vector of execution times of ϕ_i is thus composed of 3 elements. These are determined as follows:

[3] According to [1], the BCC1 OSEK/VDX OS conformance class (which is the one that uses less resources) does not allow multiple activations of a task.

$$\begin{aligned} c_{\phi_i}^{(0)} &= 3+6+4=13 \\ c_{\phi_i}^{(1)} &= 3+4=7 \\ c_{\phi_i}^{(2)} &= 3+4=7 \end{aligned}$$

The execution times vector is then composed of the following elements:

$$C_{\phi_i} = \{13,\ 7,\ 7\}.$$

Figure 8.2 examplifies the execution of the task, and, in particular, states which frames are constructed by each instance of the task.

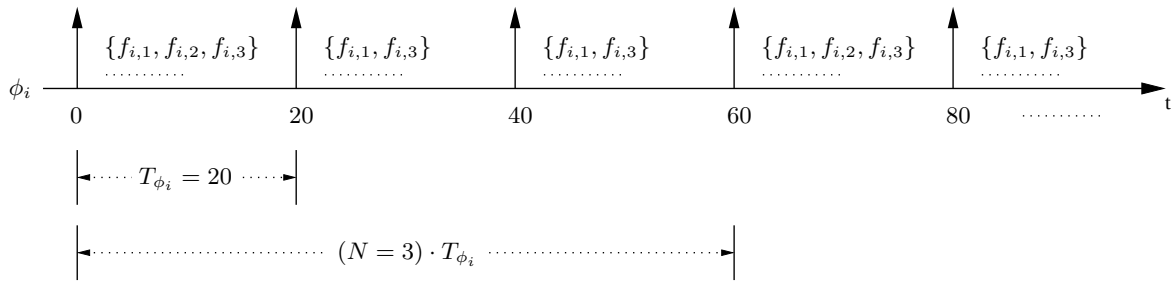


Figure 8.2: Execution of the multiframe task characterized by set $Q_i = \{(3,20),\ (6,60),\ (4,20)\}$.

Set Q_i expresses the settings of frames $\{f_{i,1},\ f_{i,2},\ f_{i,3}\}$ that were used to determine the parameters of the task.

8.2.1.2 Implementation as a generalized multiframe task

It may happen that one of the several execution times of a multiframe task is greater than the activation period. For instance, imagine that $Q_i = \{(3,15),\ (6,60),\ (4,20)\}$. With these settings, the multiframe task ϕ_i would have one execution time greater than its activation period (or relative deadline): $c_{\phi_i}^{(0)} > T_{\phi_i} \Leftrightarrow 13 > 5$. One can try to overcome this problem by implementing this task as generalized multiframe (GMF), which allows to model a more complex activation pattern than with a multiframe task. A GMF task ϕ_i is characterized by a tuple $(C_{\phi_i},\ T_{\phi_i},\ \bar{D}_{\phi_i})$ where:

- $C_{\phi_i} = \{c_{\phi_i}^{(0)},\ c_{\phi_i}^{(1)},\ ...,\ c_{\phi_i}^{(N-1)}\}$ is a set of N execution times,
- $T_{\phi_i} = \{t_{\phi_i}^{(0)},\ t_{\phi_i}^{(1)},\ ...,\ t_{\phi_i}^{(N-1)}\}$ is a set of N activation periods, and
- $\bar{D}_{\phi_i} = \{\bar{d}_{\phi_i}^{(0)},\ \bar{d}_{\phi_i}^{(1)},\ ...,\ \bar{d}_{\phi_i}^{(N-1)}\}$ a set of relative deadlines composed of N elements.

Again, one assumes that the first emission request of all frames is issued by the first instance of the task. Let the set $Q_i = \{(Q_{f_{i,1}},\ T_{f_{i,1}}),\ ...,\ (Q_{f_{i,k}},\ T_{f_{i,k}})\}$ where $Q_{f_{i,k}}$ is the time needed to construct and request transmission of frame $f_{i,k}$, and $T_{f_{i,k}}$ is the transmission period of the frame. One constructs the vector of activation periods (and relative deadlines) of a GMF task ϕ_i as follows:

$$t_{\phi_i}^{(0)} = \min(T_{f_{i,1}},\ ...,\ T_{f_{i,k}}),$$

$$t_{\phi_i}^{(j+1)} = \min_{\forall k}(a \cdot T_{f_{i,k}} - \sum_{l=0}^{j} t_{\phi_i}^{(l)}),$$

$$a = \min\{i \in \mathbb{N}^+ \mid i \cdot T_{f_{i,k}} > \sum_{l=0}^{j} t_{\phi_i}^{(l)}\}$$

This vector is built while $\min_{\forall k}(a \cdot T_{f_{i,k}}) \leq \text{lcm}(T_{f_{i,1}}, ..., T_{f_{i,k}})$. At the end, the following expression should be true:

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

The procedure used to construct the vector of execution times is the following:

$$c_{\phi_i}^{(0)} = \sum_{k} Q_{f_{i,k}},$$

$$c_{\phi_i}^{(j+1)} = \sum_{\{k \mid \sum_{l=0}^{j} t_{\phi_i}^{(l)} \bmod T_{f_{i,k}} = 0\}} Q_{f_{i,k}}$$

Example of characterization: Let set $Q_i = \{(3,15), (6,60), (4,20)\}$ be the collection of pairs of:

- the time needed to construct and request transmission, and
- the transmission period

of frames $\{f_{i,1}, f_{i,2}, f_{i,3}\}$. One constructs the vector of activation periods of ϕ_i in an iteratively way until condition $\sum_{j=0} t_{\phi_i}^{(j)} = \text{lcm}(15, 60, 20)$ is true:

$$\begin{array}{lclcl}
t_{\phi_i}^{(0)} & = & \min(15, 60, 20) & = & 15 \\
t_{\phi_i}^{(1)} & = & \min(2 \cdot 15 - 15, 1 \cdot 60 - 15, 1 \cdot 20 - 15) & = & 5 \\
t_{\phi_i}^{(2)} & = & \min(2 \cdot 15 - 20, 1 \cdot 60 - 20, 2 \cdot 20 - 20) & = & 10 \\
t_{\phi_i}^{(3)} & = & \min(3 \cdot 15 - 30, 1 \cdot 60 - 30, 2 \cdot 20 - 30) & = & 10 \\
t_{\phi_i}^{(4)} & = & \min(3 \cdot 15 - 40, 1 \cdot 60 - 40, 3 \cdot 20 - 40) & = & 5 \\
t_{\phi_i}^{(5)} & = & \min(4 \cdot 15 - 45, 1 \cdot 60 - 45, 3 \cdot 20 - 45) & = & 15
\end{array}$$

The activation periods (and relative deadlines) vector is then composed of the following elements:

$$T_{\phi_i} = \{15, 5, 10, 10, 5, 15\}.$$

The set of execution times can then be constructed as follows:

$$\begin{array}{lclcl}
c_{\phi_i}^{(0)} & = & 3+6+4 & = & 13 \\
c_{\phi_i}^{(1)} & = & 3 & = & 3 \\
c_{\phi_i}^{(2)} & = & 4 & = & 4 \\
c_{\phi_i}^{(3)} & = & 3 & = & 3 \\
c_{\phi_i}^{(4)} & = & 4 & = & 4 \\
c_{\phi_i}^{(5)} & = & 3 & = & 3
\end{array}$$

Task ϕ_i has thus this list of execution times:

$$C_{\phi_i} = \{13, 3, 4, 3, 4, 3\}.$$

Figure 8.3 illustrates the execution of the task, specifying which frames are constructed by each instance of the task.

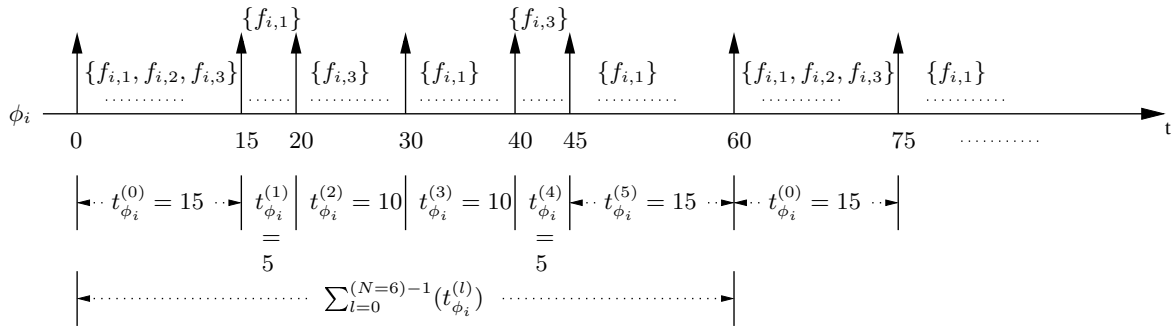


Figure 8.3: Execution of the GMF task characterized by set $Q_i = \{(3, 15), (6, 60), (4, 20)\}$.

Set Q_i expresses the settings of frames $\{f_{i,1}, f_{i,2}, f_{i,3}\}$ that were used to calculate the parameters of the task.

8.2.2 Task receiving and handling frames

This task, implemented as an interrupt service routine on OSEK/VDX OS, is sporadic [64]. Each sporadic task is characterized by a unique execution time, activation period and relative deadline. Since a sporadic task does not follow a strict releasing rate, its activation period is set to the minimum inter-activation time interval. The configuration of this task, denoted ω_i on ECU i, is given by a tuple $(C_{\omega_i}, T_{\omega_i}, \bar{D}_{\omega_i})$. Its characterization is achieved as follows:

- C_{ω_i} is the time necessary to handle the largest frame that ω_i can receive (information obtained from the frame packing configuration),
- T_{ω_i} is the time to transmit the smallest frame that ECU i can receive, and
- $\bar{D}_{\omega_i}$ is equal to T_{ω_i}, otherwise frames can be lost.

8.2.3 Priority assignment

The assignment of priorities to the middleware is done independently of the applicative tasks. The reason justifying such decision is the fact that being a software layer providing services to the application level, the middleware can be considered as a system service, and thus, with a priority that exceeds the one from any application level component. As a result of this decision, it is assumed that the two middleware tasks on each ECU have the two highest priorities. In the first place, if communication services are the most priority on each ECU, one avoids losing frames due for instance to buffer overflow. Secondly, middleware tasks can prevent a faulty task from jeopardizing the system behaviour, for instance, by consuming all the CPU time due to a software bug. The middleware can thus respond to this situation by applying the right mechanisms (activate a new task, re-initialize signal values, reboot the ECU, etc).

Moreover, in OSEK/VDX OS, interrupt service routines have necessarily a higher level of priority than tasks. Hence, task ω_i has a higher priority than ϕ_i. This is justified by the fact that the frame reception task ω_i is implemented as an OSEK/VDX OS interrupt service routine triggered by communication controller interrupts. Task ϕ_i, which is activated periodically by OSEK/VDX OS timing alarms, is implemented as a classical OS task.

8.3 A feasible priority for the applicative tasks

With fully characterized middleware tasks, it is possible to try to determine a feasible priority assignment for the set of applicative tasks on each ECU. This objective can be achieved with the Audsley

algorithm [58] (presented in chapter 2) that is optimal in the sense that if a solution exists then it will necessarily be found.

Recall that the strategy of the algorithm is to start from the lowest priority (m), and to look for a task that is feasible at this level. In case of failure, one can conclude that the set of tasks is not schedulable. The first feasible task at level m is assigned to that priority, then the algorithm tries to find a feasible task at level $m-1$, and so on until the highest priority of the system.

When verifying if a task is feasible at a certain level, the algorithm performs a feasibility test (i.e. worst-case response time computation). This test must however take into account the fact that middleware tasks have the two highest priorities, and that there will be a blocking time interval caused by the Priority Ceiling Protocol used by OSEK/VDX OS. Particularly, when considering the interference delay caused by the middleware tasks, one must keep in mind that the task constructing frames is either multiframe or generalized multiframe. The interference induced by this task can be computed as shown in [71], although it is less trivial than for periodic and sporadic tasks.

If the algorithm is successful, a feasible priority allocation for the set of applicative tasks on an ECU is obtained. However, if no such allocation exists, then two choices are possible: either the applicative tasks set and their constraints have to be re-worked (*e.g.* code optimization, restructuring, etc), or the allocation of tasks on the ECUs has to be re-examined. In both scenarios a new configuration of frames will be constructed, which will change the characterization of the middleware and thus, the interference on the applicative tasks.

8.4 Conclusion

At the end of the previous chapter the reader was aware about some implementation details of the middleware. The set of tasks as well as the code implementing the communication services provided by the middleware were presented. Moreover, the activity that tries to construct a set of frames whose characteristics meet feasibility and freshness constraints was detailed.

The current chapter presented the activities that, on each ECU, build the complete configuration of the middleware tasks, and try to determine a feasible priority assignment for the applicative tasks. The configuration parameters of the middleware tasks were calculated with the help of the characteristics of the frames. It also was decided to set the middleware tasks with higher priorities than the applicative tasks, which prevents faulty applicative tasks from taking control of the system.

The use of the Audsley algorithm was proposed to accomplish the activity that tries to determine a feasible priority assignment for the applicative tasks. The issue concerning the inexistence of a feasible priority allocation for the applicative tasks was raised. The solutions proposed to solve this problem were to modify the set of applicative tasks (re-design the set of automotive functions) and their constraints, and/or to change the ECU where these tasks are placed. The work presented in this chapter was published in [66, 74].

Chapter 9

Quality of service monitoring through notification services

The methodology presented in the previous chapters aims to develop a middleware providing communication services. The goal of this chapter is to study the possibility of evolving the middleware by adding notification services. They would give to applicative tasks the state of the transmission and reception of the signals produced and consumed. The semantics of transmission and reception states will be given later.

Specifically, we wish to evaluate the consequences of integrating notification services in terms of the implementation and the software model of the middleware. This evaluation will try to validate our methodology, by testing the effectiveness of the activities in charge of the creation of these models. We are interested in verifying that the implementation and the software models can still be constructed using the same context (OSEK/VDX OS, CAN network), the same input data, and the same methods. Let us then begin by presenting the new services considered for the middleware.

9.1 Notification services

We are interested in adding notification services to the middleware. These services must provide the applicative tasks with the information on the transmission of their produced signals, and on the reception of the signals they consume. Concretely, two new services are added to the list of communication services discussed in chapter 4:

- a service informing applicative tasks whether the transmission of previously produced signal values is respecting the freshness constraint or not, and
- a service notifying applicative tasks whether a signal value available for consumption respects the freshness constraint or not.

These services will be called transmission and reception state respectively. Moreover, the two existing communication services (see chapter 4) are improved in the following way:

- the service allowing applicative tasks to provide a produced signal value, will return the current transmission state of the signal, and
- the service allowing applicative tasks to obtain the last value received of a signal, will also return the current reception state associated to the value.

All these services are executed when they are called by the applicative tasks. We have decided to avoid the utilization of notification mechanisms that trigger the execution of applicative tasks, in order to keep their activation model simple and not overly pessimistic for the scheduling analysis. In the following, the detailed description of the semantics of the transmission and reception states is given.

9.1.1 Transmission state

The transmission state returned by the middleware to the applicative tasks can be one of the following values:

- -1 = service not available
- 1 = waiting for the construction of the frame, or for the event establishing that the frame was transmitted
- 0 = the last signal transmitted did not respect its freshness constraint
- 2 = the last signal transmistted respected its freshness constraint

Recall that the middleware is asynchronous from the application, and hence, the set of values foresees all possible situations of a produced signal value: before, during, and after the transmission. Moreover, as we will see hereafter, some mechanisms (like task activation through alarms) must exist in order to implement this service. When these mechanisms are not provided, the application receives an information of service not available.

9.1.2 Reception state

The values for the reception state of a signal are the following:

- -1 = service not available
- 0 = the current value is not respecting the freshness constraint imposed on the signal
- 1 = the current value is respecting the freshness constraint imposed on the signal, but has already been consumed or its reception state already been consulted
- 2 = the current value is respecting the freshness constraint imposed on the signal

As for the transmission state, these values anticipate all possible scenarios of a received signal value. Moreover, when a reception state of value 2 is read, the middleware will change it to 1. Future readings of the reception state (or consumptions of the signal value) before the arrival of a new value, will alert the application for the fact that no new value has arrived since the last reading, which enriches even more the semantics of the reception state. More details on the implementation of the middleware in order to accomplish these services are given hereafter.

9.2 Consequences on the implementation model of the middleware

The implementation model of the middleware, presented in chapter 5, was achieved by an activity whose goal was to identify a set of tasks representing the middleware on each ECU. This activity took as input data the functionalities and the activation events of the middleware, the OSEK/VDX OS specification, and some task identification strategies. We are now going to study the consequences on this activity caused by the integration of the notification services.

9.2.1 Service returning the transmission state

This service is strongly related to the service in charge of the construction and transmission request of frames. It is thus logical to implement both services in the same task.

9.2.1.1 Implementation based on the frames sender task

To be able to construct and transmit frames and verify their transmission state, the frames sender task would execute in the following way. After requesting the transmission of a frame, this task sets a timeout equivalent to the frame relative deadline. If the timeout expires (triggering the middleware frames sender task) before the completion of the frame transmission, then the freshness constraint of the signals composing the frame will not be respected (service value is set to 0). Otherwise, if the frame transmission completion event arises (triggering the middleware frames sender task) before the expiration of the timeout, then the freshness constraint of the signals composing the frame will be respected (service value is set to 2). Every time the applicative task produces a signal value, the service value is set to 1 (allowing the task to know during future productions if something has changed since the last one).

Consequences of using the frames sender task

The advantage of this implementation scheme is in the fact that one task is used to provide both services. However, there is a drawback concerning the methodology because the frames sender task is now also activated by other types of events (timeout expirations and transmission completion of frames), instead of just transmission periods expirations (as stated by the used task identification strategy in chapter 5). Another disadvantage of this execution is that the middleware frames sender task loses its well defined activation scheme. Besides being activated by periodic timing alarms for the construction and transmission of frames, it is now also triggered either by frames transmission completions or by timeout expirations. In order to let the frames sender task be activated only by transmission periods expirations, and hence keep its well defined activation scheme, the solution is to use a separate task.

9.2.1.2 Implementation based on a separate task

The idea is to use another task that would be activated by the frames sender task through the OSEK/VDX OS *ChainTask* function. Its purpose is to wait for the raising of events, either frames transmission completions or timeouts. According to the event received, the service value is set to 0 or 2.

Consequences of using a separate task

This solution partially validates our methodology because the frames sender task is only activated by one type of event (transmission periods expirations), but the same does not happen with the separate task. Indeed, this task, named transmission handler in the following, is still activated by timeout expirations and frames transmission completion events, and follows an aperiodic activation scheme.

Usually, in-vehicle communication platforms do not provide frame transmission completion events to the task level[4]. In this case, the transmission handler task will be just activated by the timeouts expiration events, and, at that time, will use the network controller API to verify whether the frame was sent or not. Each different timeout set by the frames sender task is assigned to a different event name inside OSEK/VDX OS, which helps the transmission handler task to determine the frame that must be verified. Consequently, this task must know which frame corresponds to each event name, and which signals compose each frame transmitted by the frames sender task.

This implementation scheme, illustrated in figure 9.1, is compliant with our methodology, and allows the transmission handler task to follow the activation scheme of the frames sender task, which is periodic with either a constant value in a multiframe scenario, or different values in a GMF situation. Nevertheless, the schedulability analysis must consider this relation of precedence between both tasks, and must also take into account the fact that the transmission handler task suspends itself while waiting for the timeouts expiration. Moreover, in OSEK/VDX OS only extended tasks are allowed to wait for events, such as timeouts. Thus, this implementation schemes implies the utilization of an OSEK/VDX OS belonging to a higher conformance class.

[4] CAN controllers, such as Philips SJA1000 [84], Intel 87C196CB [85], and Infineon C50x [86], provide frame transmission completion events under the form of interrupts. However, these interrupts are not transmitted to the applicative level software but used by the drivers of the controller to know when to initiate the transmission of the next frame waiting in the buffer.

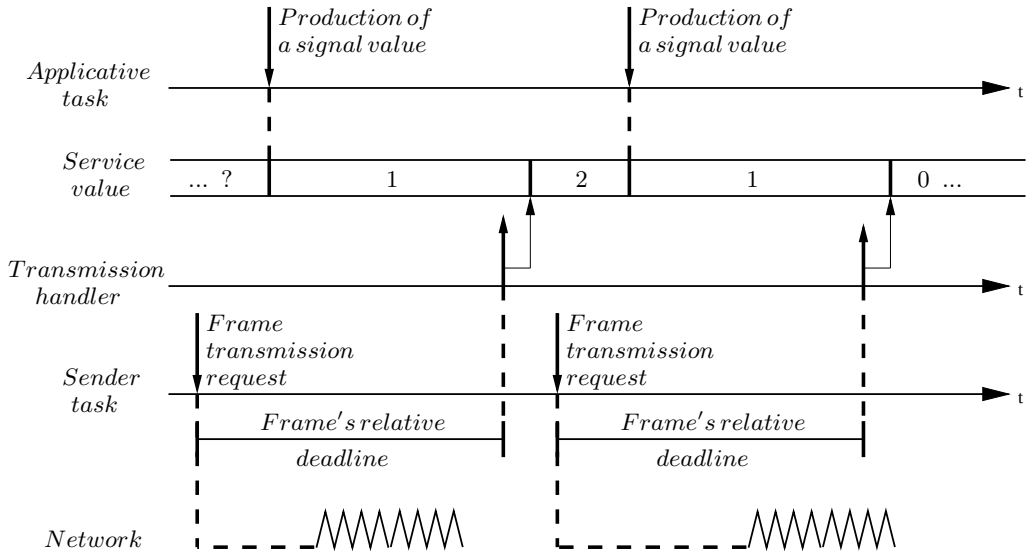


Figure 9.1: Execution of the transmission handler task proposed to implement a notification service associated to the transmission of produced signals.

9.2.2 Conclusion on the transmission state

We have proposed two implementation schemes for the transmission state service. The scheme based on a separate task, under the condition that frame transmission completion interrupts raised by the network controller are disabled, is compliant with our methodology, while the scheme that uses the frames sender task is not. However, both schemes suffer from some drawbacks that make them inaccurate.

The first inconvenience is that if the relative deadlines of the frames (used to set the timeouts) are greater than, or equal to, the activation period of the frames sender task, this situation can lead to multiple requests for the activation of the frames sender or the transmission handler task (depending on the used implementation scheme). In case the separate task implementation scheme is used for example, while the transmission handler task is waiting for a timeout expiration, the frames sender task can activate a new instance of the transmission handler. Multiple requests for the activation of a task introduce delays on the older instances, and requires the utilization of more resources. Moreover, it forces the system designer to use an OSEK/VDX OS belonging to a higher conformance class.

The second drawback concerns the preemption of the frames sender or the transmission handler task (depending on the used implementation scheme). If one of these tasks cannot execute when activated with the purpose of verifying the transmission state (because a higher priority task is running), then the verification of the frames relative deadline respect is not accurate. Finally, the last inconvenience is related with the methodology. While determining a frame relative deadline, the frame packing algorithm must consider a worst-case behaviour for the middleware tasks, where is included the transmission handler if it is used. Since the execution parameters of these tasks depend on the frames relative deadline, which is being determined by the frame packing algorithm, it is difficult to consider worst-case behaviours.

All these problems demonstrate how difficult it is to implement a solution capable of providing an accurate notification service to producer applicative tasks based on the freshness constraints. To overcome this problem, we might have to decrease our desired quality of service. Instead of notifying applicative tasks about the transmission of signal values being respecting or not their freshness constraints, we can just inform if the value has been used to construct a frame or not. When the frame carrying a signal value is constructed, the service value of the signal is changed in order to express this fact. When the applicative task produces a new value for this signal, the service value is also changed. This third implementation scheme of the service mainly gives to the applicative tasks the on-line guarantee that the middleware is executing (and vice-versa) in terms of construction of transmission request of frames. It is thus a service

with a different purpose, but when compared with the other solutions (see table 9.1), it presents the important advantage of not implying any modifications on the initial implementation model, and being perfectly adapted to the methodology.

	Verification of the signals freshness constraint		**Verification of the signals transmission**
	with the frames sender task	**with a separate task**	**with the frames sender task**
Adapted to the tasks identification strategy	no	yes, if frame transmission completion interrupts disabled	yes
Calculation of the worst-case behaviour	difficult	difficult	easy
Activation model of the task(s)	aperiodic	periodic (multiframe or GMF)	periodic (multiframe or GMF)
Multiple activation of tasks	yes*	yes*	no
Accuracy of the result	fair (because of preemption)	fair (because of preemption)	high

*yes, if the frames relative deadline is greater than, or equal to, the activation period of the frames sender task, no otherwise

Table 9.1: Comparison between the proposed implementation schemes for the transmission state.

9.2.3 Service returning the reception state

Recall that the goal of this service is to notify applicative tasks whether a signal value available for consumption respects its freshness constraint or not. This service is dependent on the one in charge of receiving and handling frames. If both services are implemented in the interrupt service routine (ISR), one can reduce the overheads in terms of resources utilization.

9.2.3.1 Implementation based on the interrupt service routine

By implementing both services in the ISR, one obtains the following execution. Let us define D_f^+ as being the longest time interval between two consecutive arrivals of frame f (without considering the lost of frames caused by electro magnetic interferences). This interval, illustrated in figure 9.2, is expressed by:

$$D_f^+ = T_f + R_{\phi_i} - C_{\phi_i} + R_f - \bar{C}_f$$

where:

- T_f is the transmission period of f (or the activation period of the sender middleware task when constructing and requesting the transmission of f),

- R_{ϕ_i} is the worst-case response time of the sender middleware task when constructing and requesting the transmission of f,
- C_{ϕ_i} is the execution time of the middleware sender task when constructing and requesting the transmission of f,
- R_f is the worst-case response time of f, and
- $\bar{C}_f$ is the time necessary to transmit f over the CAN network.

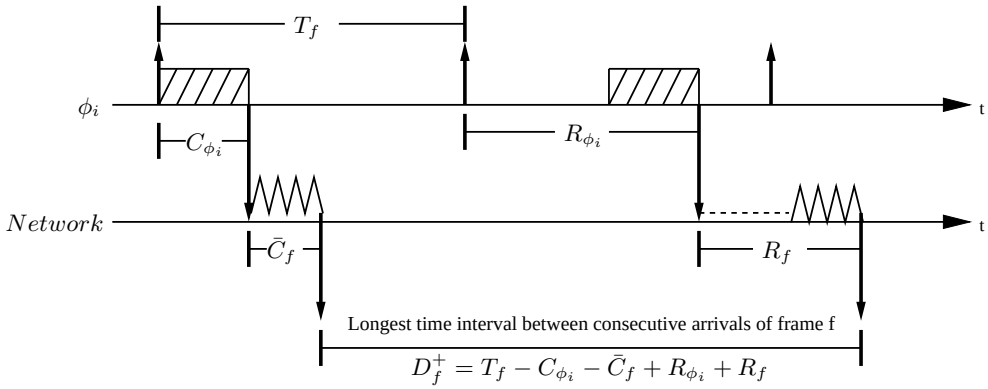


Figure 9.2: Representation of D_f^+, the longest time interval between two consecutive arrivals of frame f on a CAN network.

After receiving one instance of f, the ISR sets a timeout equivalent to D_f^+. If the timeout expires (triggering the ISR) then the next instance of f has not yet arrived and thus, the signal values available (that are usually transmitted in f) are no longer respecting their freshness constraints (service value of those signals is set to 0 and the timeout is not reset). Moreover, if meanwhile the next instance of f arrives (triggering the ISR) after the timeout expiration, then the signal values contained in the frame are not respecting their freshness constraints. In this case, they are made available or not, depending on the configuration made by the system designer. Anyway, the service value continues to be 0 but the timeout is reset. The ISR can determine if the timeout has already expired by using function *GetAlarm* provided by OSEK/VDX OS.

Otherwise, if the next instance of f arrives (triggering the ISR) before the timeout expiration, then the signal values contained in the instance of f are made available and respect their freshness constraints (service value of those signals is set to 2 and timeout is reset). When an applicative task consumes a signal value, the service value is set to 1 if it is currently equal to 2, but, it is not changed if it is currently equal to 0 (allows the task to know if new "fresh" values have arrived since the last consumption).

The main problem of this execution scheme lies in the impossibility for OSEK/VDX OS ISRs to be activated by timeouts (timing alarms). Indeed, according to this OS specification, alarms are exclusively assigned to tasks. Thus, reception states must be handled by a separate task.

9.2.3.2 Implementation based on a separate task

This implementation scheme is represented in figure 9.3. The idea is to have a task activated only when a frame does not arrive before the timeout expiration.

After receiving frame f, the ISR sets the timeout equivalent to D_f^+. If a new instance of f arrives before the timeout expiration, then the ISR resets the timeout. Otherwise, if a new instance of f does not arrive before the timeout expires, then the task is activated (by an OSEK/VDX OS alarm) and it knows that the currently available values of the signals transmitted in f are not respecting their freshness constraints. In this case, the task does not reset the timeout. If meanwhile the next instance of f arrives, the ISR determines that the alarm has expired (using function *GetAlarm*) and resets it.

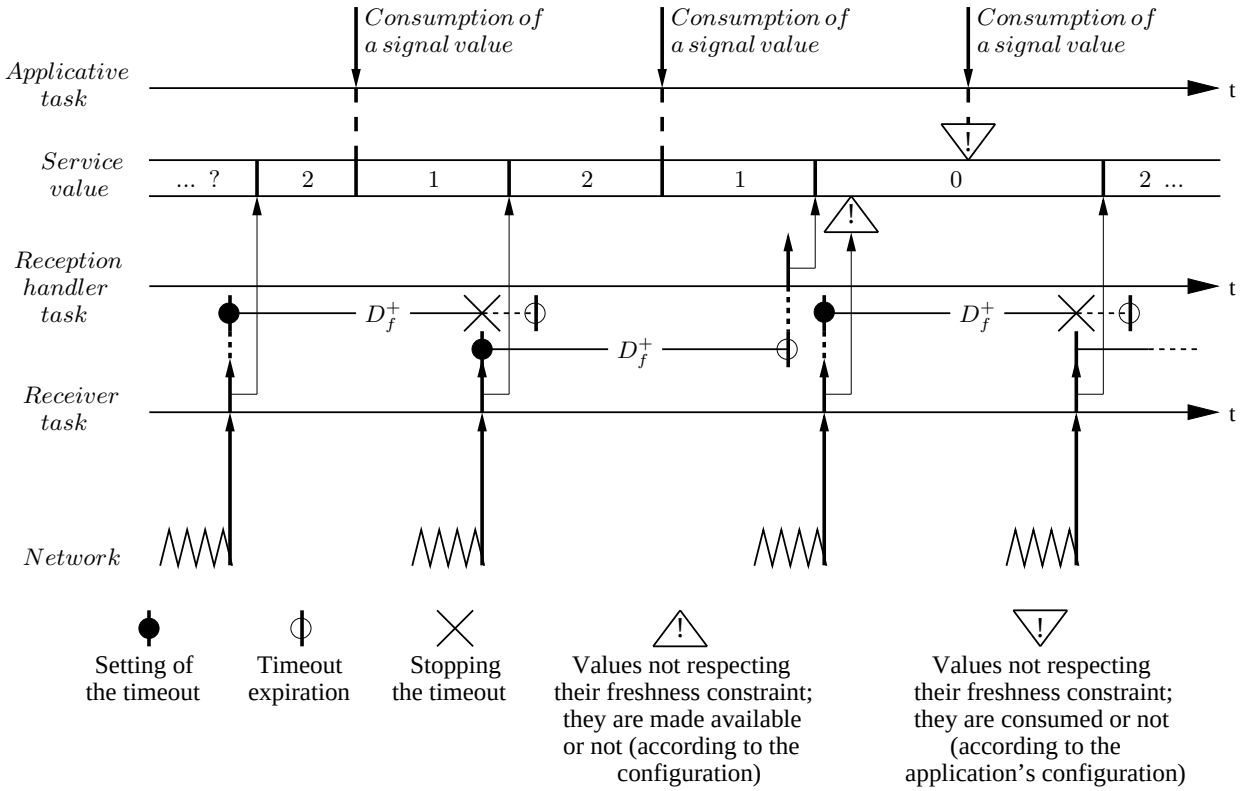


Figure 9.3: Example of cooperation between the middleware tasks implementing a service that returns the reception state of a signal.

The reception state of an available signal value indicates whether it respects the freshness constraint or not.

9.2.4 Conclusion on the reception state

The implementation scheme based on a separate task validates our methodology because there is a new task, called reception handler, which is activated by a single type of events. These events are the OSEK/VDX OS alarms equivalent to the longest time interval between two consecutive arrivals of the frames (D^+).

When activated, the reception handler task must know which frame has not arrived. To do so, the task checks if there is an expired alarm. One inconvenience of this scheme is the need to define one alarm for each different frame received on an ECU. Moreover, the fact that frames arrive sporadically in CAN, and the values of D^+ are independent from each other, may lead to a situation where multiple instances of the reception handler task are activated simultaneously. This scenario requires the utilization of an OSEK/VDX OS belonging to a higher conformance class, as previously mentioned in section 9.2.2.

The sporadic arrival of frames, the independence between the values of D^+, and the fact that the task only executes if the timeouts expire, make this task aperiodic (as one can verify in figure 9.3). Nevertheless, it would be possible to determine the maximum number of activations during a time interval t (one instance of the task for each alarm, and all instances activated simultaneously). With this information, one can derive the worst-case activation pattern of the task, and thus perform a schedulability analysis. However, while determining a frame relative deadline, the frame packing algorithm must consider a worst-case behaviour for the reception handler task. The activation of this task depends on the values of D^+, which on their turn are dependent on the characteristics of the frames being determined by the frame packing algorithm. It is thus difficult to consider a worst-case behaviour for the reception handler task.

Another drawback of this scheme, also present in the implementation of the transmission state, is the preemption of the reception handler task. If this task cannot execute when activated because a higher priority task is running, it may happen that meanwhile the ISR (that has a higher priority than any task) receives the late instance of the frame and resets the timeout. When the reception handler task will be able to execute, the alarm associated with the timeout will not be seen as expired. This scenario can jeopardize the accuracy of this service.

Once again, it is difficult to implement a solution capable of providing a good quality of service. We must then use the same solution as for the transmission state and decrease our desired quality of service. For this purpose, instead of informing if an available signal value respects its freshness constraints, we can notify the applicative tasks of the presence or not of a new value since the last consumption. When a new signal value is made available, then the service value is set in order to demonstrate this fact. When applicative tasks consume a signal value, then the service value is also modified. This way, applicative tasks are aware that new frames are being received, and the middleware is executing and handling them. Moreover, the middleware can also verify if an applicative task is executing. This verification can become the basic mechanism used by the middleware to detect an hazardous behaviour from the applicative tasks.

9.2.5 Conclusion

We have considered the implementation of notification services in the middleware either using the set of tasks identified for the communication services, or using a set of separate tasks. For each of the services and the possible implementation scenarios, we have listed the reasons that in some cases prevent their concretization, and in other cases allow their implementation but negatively affect their accuracy. To preserve the set of tasks identified for the communication services, and propose services returning accurate results, the solution was to decrease the desired quality of service. The set of notification services that at the beginning of the chapter had the goal of verifying the respect of the freshness constraints, was transformed in such a way that became to be composed of:

- a service that informs applicative tasks if the previously value of a produced signal has been used to construct a frame or not, and
- a service notifying applicative tasks if the current value of a signal to be consumed has been updated since the last consumption or not.

Notice that this solution also validates our methodology because the initial set of middleware tasks for the communication has not been modified, and hence, the implementation model of the middleware determined at chapter 5 remains valid. In the following, we are going to verify if the same situation occurs with the software model.

9.3 Consequences on the software model of the middleware

The software model of the middleware does not require any profound modifications. The set of classes will remain the same, except that it is now necessary to change some methods in order to reflect the existence of the notification services. These methods are easy to identify because of the utilization of design patterns for the construction of the software architecture of the middleware. The methods to modify belong to classes *Proxy_Scheduler*, *Signals*, and *Core*.

The class on the left of figure 9.4 illustrates the new class *Proxy_Scheduler*. One can verify that the two notification services have been added: *checkTransmState* and *checkRecepState*. Moreover, the declaration of methods *sendSig* and *consumeSig* was changed so that these methods might return the transmission and reception states respectively. The code implementing these methods has also changed. Inside method *sendSig*, after the newly produced value has been saved in class *Signals*, it is necessary to change the service value. The same happens inside method *consumeSig*, before returning the value to be consumed.

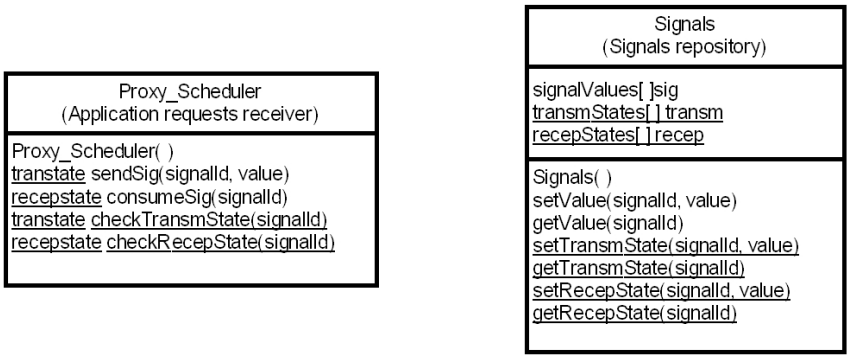


Figure 9.4: Modifications introduced in classes *Proxy_Scheduler* and *Signals* in order to reflect the existence of notification services.

On the right of figure 9.4 one can see the new class *Signals*, to which have been added the attributes and the methods necessary to work with the services values. Nevertheless, recall that this class will be transformed into an OSEK/VDX OS shared resource when transposing the class diagram to the set of middleware tasks. Finally, class *Core* must be modified inside methods *notify* and *receiveFrame*. In the first, while obtaining signal values from class *Signals* in order to construct frames, it is necessary to change the service values of the signals. In method *receiveFrame*, for each newly arrived signal value that is saved in class *Signals*, the associated service value is updated.

9.4 Conclusion

We have seen in this chapter the difficulty associated with the implementation of notification services whose goal would be to verify the respect of the freshness constraints. Implementation schemes using the initial set of middleware communication tasks, or requiring the utilization of a separate set of tasks were considered. Schemes based on new tasks could be useful in order to preserve the activation scheme of the middleware tasks for communication, and to have the middleware adapted to the specification of OSEK/VDX OS (ISRs cannot be activated through OSEK/VDX OS alarms).

However, all the implementation schemes considered suffer from a certain inaccuracy, mainly caused by the fact that the new tasks can be preempted during their execution, and the delays induced can affect the accuracy of their results. Making these tasks non-preemptive would, on the one hand, cause delays on the frames sender task, and, on the other hand, not avoid their preemption by the frames receiver ISR. Notice that this set of notification services are those proposed by the standard OSEK/VDX COM (see chapter 2) for performing a service of deadline monitoring. This chapter has thus stated the difficulties that one can encounter when trying to implement the proposals made by this standard.

To overcome these problems, we decided to propose a different set of notification services. This new set is less demanding in terms of quality of service, but does not require the creation of new tasks. Thus, the implementation model defined in chapter 5 remains valid. The software model did not suffer any deep modifications. For instance, no new classes were added to the middleware software architecture. Nevertheless, some new methods were integrated and some of the previous ones were changed. In consequence to the utilization of design patterns to construct the software architecture, these modifications were easy to identify and perform.

Part III

Conclusion

Chapter 10

Conclusions

Dans cette thèse nous avons étudié la conception d'un intergiciel embarqué dans l'automobile. Plus précisément, nous avons proposé une méthodologie qui envisage son développement sous deux volets. Le premier traite les aspects génériques de l'intergiciel, comme les modèles de composants logiciels (séquences de code, objets, méthodes) et d'implémentation (tâches en charge de l'exécution de l'intergiciel, principes génériques de déploiement des objets et du code). Ce volet est générique car les modèles résultants sont spécifiés une seule fois et seront instanciés pour chaque système. Le deuxième volet cible la configuration correcte et optimisée des activités du système embarqué (tâches et trames réseau). Le terme "correcte" signifie que cette configuration doit garantir le respect de toutes les contraintes temporelles exigées par le niveau applicatif tandis que le terme "optimisée" porte sur la minimisation de la bande passante et des objets à gérer par le système d'exploitation. Nous détaillons ci-dessous les conclusions auxquelles nous sommes arrivés concernant chacune des ces deux parties de cette méthodologie.

10.1 Contributions

La première composante de la méthodologie visait la spécification, d'une part, d'un modèle d'implémentation, et, d'autre part, d'une architecture de composants logiciels. Concrètement, le modèle d'implémentation devait identifier un ensemble de tâches capables de représenter l'intergiciel sur chaque ECU. Pour cela, nous avions la spécification du système d'exploitation cible OSEK/VDX OS, les fonctionnalités que l'intergiciel devrait implémenter ainsi que les événements qui les activaient. Une étude comparative de stratégies pour identifier quelles tâches doivent exécuter l'intergiciel nous a amené à proposer une solution reposant sur deux tâches uniquement (une tâche OSEK/VDX OS et une tâche d'interruption). Cette solution présente l'avantage de permettre un plus grand nombre possible de tâches applicatives (OSEK/VDX impose une limite, différente selon la classe de conformité, du nombre global de tâches gérées). Cette propriété est importante car elle favorise l'implémentation de nouvelles fonctions automobiles, selon le modèle de conception incrémentale largement utilisé dans l'industrie automobile.

Un autre problème lié à ces tâches est de définir à quel modèle chacune d'elles peut se référer, de façon à pouvoir analyser leur comportement temporel. Nous avons montré que les tâches OSEK/VDX OS appartiennent aux modèles *multiframe* ou GMF (*generalized multiframe*) tandis que les tâches d'interruption relèvent du modèle de tâches sporadiques. Ces modèles permettent une analyse d'ordonnançabilité, et peuvent être implémentés sur OSEK/VDX OS. Finalement, nous avons remarqué que la stratégie utilisée pour l'identification de l'ensemble de tâches, reste encore valable si les paramètres de configuration évalués par le deuxième volet de la méthodologie (en particulier les caractéristiques des trames) changent. Cette propriété accroît la possibilité d'une ré-utilisation du modèle d'implémentation.

Le modèle de composants logiciels cible la construction d'une architecture logicielle orientée objets selon une approche *design patterns*. A partir d'un catalogue de design patterns, et de la liste de services de communication fournis par l'intergiciel, nous avons identifié les design patterns pertinents pour notre problème et spécifié leur composition. Le résultat a été représenté par un diagramme de classes UML, qui en conjonction avec le modèle d'implémentation a permis ensuite l'identification des séquences de code à

exécuter par chacune des tâches intergicielles.

Un des avantages de ce diagramme de classes réside dans le fait d'exprimer l'ensemble des composants logiciels de l'intergiciel d'une manière générale et indépendante du modèle d'implémentation. De plus, l'utilisation des design patterns pour la conception du diagramme permet de facilement documenter, modifier, et ré-utiliser l'architecture logicielle générique définie. Par ailleurs, l'identification des séquences de code exécutées par chaque tâche a permis de déterminer les objets à instancier dans chacune des tâches. Même si en pratique le développement ne devait pas faire appel à un langage orienté objets, les séquences de code identifiées dans notre travail restent valides et peuvent être groupées de façon à constituer une seule fonction. Ce processus de groupement peut être facilement automatisé.

Le deuxième volet de la méthodologie a pour but d'instancier les modèles génériques pour effectuer la configuration des tâches et des trames pour une application, des propriétés temporelles requises et une distribution données. Le but est, d'une part, de déterminer les caractéristiques de chaque tâche intergiciel sur chaque ECU et, notamment, d'attribuer des priorités faisables aux tâches applicatives. Ce processus de configuration doit également construire et caractériser l'ensemble de trames à envoyer à partir de chaque ECU. Tout ce processus de configuration doit aboutir à une solution correcte (respect de propriétés applicatives) et optimisée (minimisation de bande passante réseau et du nombre de tâches).

Le processus de configuration a été divisé en trois étapes : la première s'est occupée de la construction des trames, la deuxième visait la caractérisation des tâches de l'intergiciel déterminées par le modèle d'implémentation, et la dernière étape a abordé l'attribution des priorités faisables aux tâches applicatives. La configuration de trames prend en entrée la spécification des tâches applicatives et des signaux. Nous avons proposé deux heuristiques et avons évalué leurs performances. Ces heuristiques définissent une solution qui respecte des échéances sur les trames. Nous avons montré qu'elles assurent les contraintes de fraîcheur sur les signaux. Le résultat obtenu est ensuite utilisé pour configurer les tâches de l'intergiciel. Ceci a permis de s'affranchir de la boucle d'interdépendance entre la configuration des trames et celle des tâches présentes sur chaque ECU telle qu'elle apparaît dans la spécification initiale du problème de configuration globale.

La caractérisation des tâches de l'intergiciel prend en entrée un ensemble de trames configurées et détermine les règles ou profils d'activation de chaque tâche ainsi que les temps d'exécution et les échéances de chaque job (instance de tâche). Le résultat obtenu permet de quantifier l'interférence causée par les tâches de l'intergiciel sur les tâches du niveau applicatif. Le processus d'attribution de priorités faisables aux tâches applicatives peut alors se faire à l'aide de l'algorithme d'Audsley.

Dans une dernière partie de la thèse, nous avons essayé de valider notre méthodologie dans une perspective d'intégration de nouveaux services à l'intergiciel. En particulier, nous avons étudié les conséquences sur les modèles d'implémentation et de composants logiciels, de l'intégration de services de notification aux tâches applicatives. Nous avons remarqué le fait que les services proposés étaient un raffinement de ceux qui étaient considérés par le standard OSEK/VDX COM. La principale conclusion de cette étude est la difficulté d'implémenter de tels services de telle manière qu'on puisse conserver la possibilité d'effectuer des analyses d'ordonnançabilité sur l'ensemble de tâches sur chaque ECU. Une solution à laquelle nous sommes parvenus est de proposer des services moins exigeants en termes de qualité de service, le problème essentiel ne venant pas du modèle générique que nous proposons, mais de l'impossibilité d'obtenir un total déterminisme dans la caractérisation des participants (tâches et trames) du système embarqué.

10.2 Perspectives

La perspective immédiate du travail présenté dans ce document est une implémentation sur une plateforme réelle avec, comme objectif, de vérifier l'implémentabilité de l'approche proposée, de quantifier l'influence de l'architecture logicielle basée sur les 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. Quelques résultats liés à ces problèmes seront bientôt disponibles, car nous encadrons un stage d'ingénieur CNAM ayant comme sujet la génération automatique d'une image binaire de l'intergiciel à partir des

modèles d'implémentation et de composants logiciels. L'environnement d'exécution de l'intergiciel est le système d'exploitation temps-réel RTAI avec émulation des services CAN sur la base d'un réseau Ethernet.

Une deuxième perspective serait d'inclure dans notre méthodologie un modèle d'erreurs de transmission de trames, en particulier, dans l'activité liée à la construction de la configuration de trames. En effet, nous avons considéré pour le calcul de l'âge maximal d'un signal que les valeurs produites ne sont jamais corrompues pendant leur transmission. Pourtant, la réalité est loin d'être ainsi ; en effet, les perturbations électromagnétiques engendrent souvent la perte ou la corruption des trames. Des travaux comme ceux de Tindell et Burns [87] et de Navet [88], qui étudient la relation entre la perte de trames et le respect des échéances relatives, serviraient alors de point de départ pour étendre notre méthodologie. Intégrer ces mécanismes de recouvrement de fautes de transmission conduit à augmenter l'âge maximal d'un signal, ce qui implique probablement une dégradation au niveau des résultats fournis par nos algorithmes de construction de la configuration de trames mais, en contre-partie, augmente la robustesse aux perturbations des configurations proposées.

Sur un plan plus technique, d'autres approches pour la configuration de trames pourraient être explorées comme, par exemple, la programmation par contraintes [89]. Cette méthode a prouvée être efficace pour le placement de tâches applicatives sur un réseau d'ECUs [90]. Cependant, l'existence d'un intergiciel n'a pas été considérée, et la minimisation de la consommation de bande n'a pas été prise en compte.

La plate-forme de communication dans cette thèse suppose une distribution autour du réseau CAN, un réseau basé sur les priorités. Le futur des réseaux embarqués dans l'automobile passera certainement aussi par ceux guidés par le temps, et, en premier lieu, FlexRay. Il nous semble donc intéressant comme perspective, d'étudier la pertinence de notre méthodologie dans un tel contexte et d'identifier les modifications à aboutir à la méthodologie, de manière à aboutir sur une implémentation faisable sur une plate-forme FlexRay.

Enfin, une perspective ambitieuse serait d'appliquer les principes que nous avons développés dans le contexte des services de communication, à tous les services d'un intergiciel, par exemple celui défini dans le consortium AUTOSAR et, ainsi, fournir une contribution solide et formelle à un générateur automatique d'empreintes d'intergiciel AUTOSAR.

Appendix A

The *Mediator* design pattern

This appendix presents an example of design pattern: the *Mediator* pattern [41]. It uses an object to coordinate state changes between other objects. Putting the logic in one object to manage state changes of other objects, instead of distributing the logic over the other objects, results in a more cohesive implementation of the logic and decreased coupling between the other objects. This pattern should be used when:

- a set of objects communicate in well-defined but complex ways,
- the reutilization of an object is difficult because it uses and communicates with many other objects, and
- a behaviour distributed among several classes should be customizable without much sub-classing.

For example, consider a dialog box that uses a window to show a set of widgets (buttons, menus, entry fields, etc). Often there are dependencies between the widgets, like when text appears in an entry field, other buttons may become enabled. Moreover, different dialog boxes may have different dependencies between widgets. To avoid these problems, a separate mediator object can encapsulate the collective behaviour. It serves as an intermediary, and is responsible for controlling and coordinating the interactions of a group of objects. Objects do not refer to each other explicitly, and are only aware of the existence of the mediator.

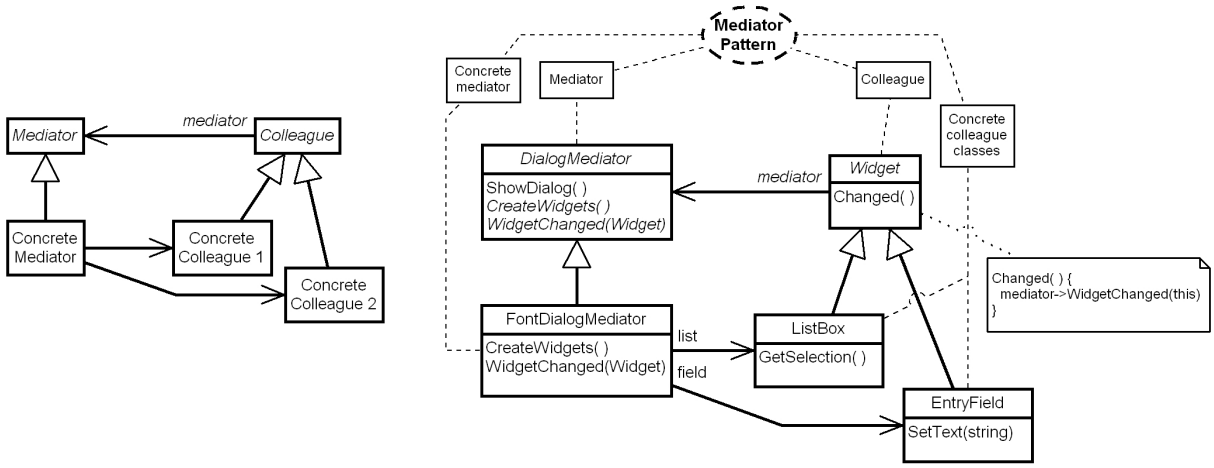


Figure A.1: Class diagrams representing the structure of the *Mediator* design pattern.

The diagram on the left illustrates the generic structure of the pattern. The diagram on the right shows the structure of the pattern applied in an example composed of a dialog box and a set of widgets. Moreover, this last diagram illustrates the UML notation for the representation of a design pattern. An ellipse displays the name of the pattern, while the rectangles announce the role played in the pattern by each class.

Figure A.1 presents two class diagrams. The diagram on the left illustrates the generic structure of the pattern. The diagram on the right shows the structure of the pattern applied in the dialog box example. The behaviour of the pattern concerning its utilization in the dialog box context is detailed in figure A.2.

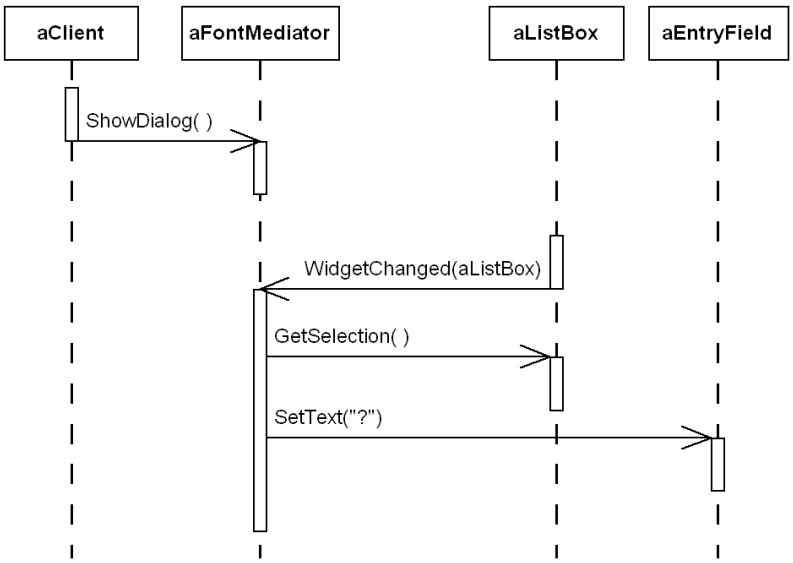


Figure A.2: Sequence diagram depicting the behaviour of objects participating in the *Mediator* design pattern.

Appendix B

Performances evaluation

This appendix presents the results of a performances evaluation made on our frame packing algorithms during 2003 and published in [73]. The performance metrics were the network bandwidth consumption and the capability to find schedulable solutions. Our algorithms were evaluated by comparison with two effective off-line bin-packing heuristics (First-Fit decreasing and Best-Fit decreasing), and with a naive strategy that consists of inserting one signal per frame.

However, the context assumed for this performances evaluation was different from the one considered in this thesis. The work developed in 2003 assumed that:

- the maximum age of a signal was defined as being the longest time interval between the production of the signal on the sender side, and its reception at the receivers side,
- the execution of the applicative and middleware tasks was not considered when determining a frame's relative deadline, and
- only one freshness constraint was associated to each signal, which was equal to the signal's production period.

Moreover, the method used to determine a frame's relative deadline was also different. The method considered in 2003 took into account the possible offsets between the production dates of the signals and the actual transmission dates of the frame. Let us for instance consider the example shown on figure B.1 with two signals s_1 and s_2 having respectively a period $T_1 = 10$ and $T_2 = 14$, and a freshness constraint $\mathcal{F}_1 = 10$ and $\mathcal{F}_2 = 14$. The signal s_2 produced at time 14 is actually sent at time 20, and the relative deadline of the frame must thus be equal to 8 in order to respect the timing constraint of s_2.

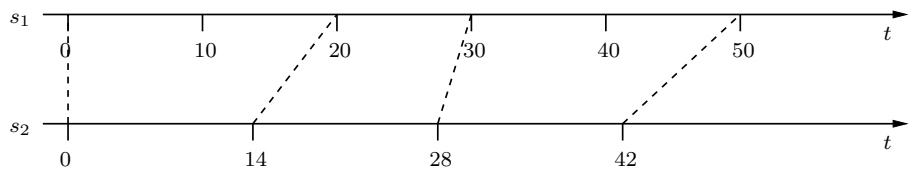


Figure B.1: Two signals with production periods equal to 10 and 14.

The signals are transmitted in a frame synchronized with the signal having the smallest period. The dotted line indicates when the signal with period 14 is actually transmitted.

To determine the deadline of the frame, one must find the largest offset between a production date and the transmission of the next frame. One wants to include signal s_i in frame f_k already containing the signals $s_1, s_2, ..s_n$. One denotes s_{min} the signal with the smallest period of the set $\{s_1, s_2, ..s_n\} \cup s_i$, the period of f_k becomes $T_k^* = T_{min}$. The relative deadline of f_k is $D_k^* = \min\{\mathcal{F}_j - \text{Offset}(T_{min}, T_j) \mid s_j \in \{s_1, s_2, ..s_n\} \cup s_i\}$ where $\text{Offset}(a, b)$ returns the largest possible duration between the production date of a signal with period $b \geq a$ and the transmission of the frame of period a that contains the signal. It has been shown in [91] that $\forall k_1, k_2 \in \mathbb{N}\ k_1 \cdot a - k_2 \cdot b = q \cdot \gcd(a, b)$ with $q \in \mathbb{Z}$. In our context, one imposes $a > k_1 \cdot a - k_2 \cdot b \geq 0$ and thus $\text{Offset}(a, b) = (\frac{a}{gcd(a,b)} - 1) \cdot gcd(a, b) = a - gcd(a, b)$. In our example, the deadline must be set to 6.

Under these conditions, experiments were conducted, first, to evaluate the Bandwidth-Best-Fit-decreasing (BBFd) heuristic (with and without the local optimization algorithm), and then, to compare BBFd (with the local optimization algorithm) with the Semi-Exhaustive (SE) algorithm.

B.1 Bandwidth-Best-Fit-decreasing performances evaluation

The performances of the BBFd heuristic were evaluated with regard to two metrics that are crucial in our context: the feasibility of the system and the network bandwidth consumption. The competing strategies were:

- *One Signal per Frame* (1SpF): each frame contains only one signal,
- *First-Fit decreasing* (FFd): the signals are sorted decreasingly according to their size and they are inserted in the first frame that can contain them,
- *Best-Fit decreasing* (BFd): the signals are sorted decreasingly according to their size and they are inserted in the frame that will have the smallest space left after the insertion.

The heuristics were implemented in C++ and the feasibility of each solution is tested by the *rts* software (written by Jörn Migge, see [92]) that implements the Audsley algorithm (see chapter 2) and the response time computations on CAN.

B.1.1 Bandwidth consumption

Only the results induced by configurations that are feasible with all strategies were taken into account, since the use of non-feasible set of frames is ruled out in the context of in-vehicle applications. Figure B.2 shows the average network load on 100 feasible configurations for an useful load varying from 15 to 35% (the useful or nominal load is the load brought by the data alone). Signals were randomly generated with a production period ranging from 5 to 100 ms (uniform distribution - step of 5ms), a freshness constraint equals to the production period, a size between 1 and 8 bytes (uniform distribution - step of 1 byte). For each test, the number of generated signals was the average number of signals necessary to reach the desired nominal load level. The signals were randomly located on a set of 10 ECUs. The network transmission rate was 500 Kbits/s and the overhead was 8 bytes for one data frame that can contain up to 8 data bytes (CAN network).

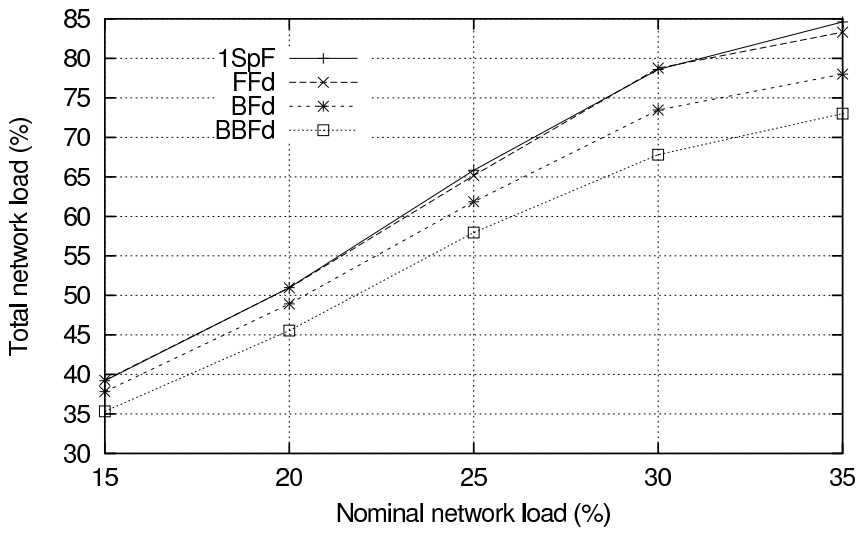


Figure B.2: Total network load for a nominal load ranging from 15 to 35%.

Average results on 100 feasible configurations for the 4 competing strategies.

As it can be seen on figure B.2, the relative performance of the heuristics remains identical whatever the network load. BBFd always produces the best results with a gain varying from 9.9 to 13.7% over the 1SpF approach, and from 6.3 to 7.6% over BFd which is the closest heuristic. The gains in terms of bandwidth consumption do exist for all network load without being very important.

B.1.2 Configurations feasibility

The conditions of the experiments were those described in the bandwidth consumption performances evaluation. The results were based on 100 randomly generated tests. Table B.1 shows the results obtained for the 4 competing strategies. Up to 25% of nominal load, all the heuristics find a feasible solution. Beyond that load level, the performances strongly differ, but the relative performances remain identical as for the bandwidth consumption. BBFd is always the best. In particular, it enables us to obtain 13 supplementary feasible configurations at the highest load level.

nominal load	15%	20%	25%	30%	35%
1SpF	100	100	100	97	36
FFd	100	100	100	97	54
BFd	100	100	100	97	76
BBFd	100	100	100	100	89

Table B.1: Number of feasible configurations over 100 tests for a nominal load varying from 15 to 35%.

B.1.3 Local optimization algorithm

The performance of the local optimization algorithm (LO) procedure was evaluated by the gain of network bandwidth consumption, and by the percentage of cases where it improved the solution of BBFd. The conditions of the experiments were those described in the bandwidth consumption performances evaluation. LO, which was executed *10 000* times in each ECU, was applied on 100 feasible solutions computed with BBFd. Table B.2 shows the resulting network bandwidth usage.

nominal load	15%	20%	25%	30%	35%
BBFd	35.32	45.54	57.96	67.81	73.01
BBFd - with LO	35.21	45.36	57.63	67.47	72.73

Table B.2: Average network bandwidth consumption over 100 feasible solutions for a nominal load varying from 15 to 35%.

Although the improvements are modest, BBFd with LO is clearly always better than BBFd, with a gain varying between 0.3% and 0.5%. Table B.3 shows the percentage of cases where LO brought a gain against BBFd alone.

nominal load	15%	20%	25%	30%	35%
%	56	64	87	94	95

Table B.3: Percentage of cases where BBFd with LO decreased the bandwidth consumption level starting from 100 feasible solutions for a nominal load varying from 15 to 35%.

The improvement with LO tends logically to be more important (Table B.2) and more frequent (Table B.3) when the load increases, since BBFd is likely to be more distant from the optimal solution.

B.2 Semi-Exhaustive performances evaluation

The experiments were conducted with the parameters described in the bandwidth consumption performances evaluation, except for the number of signals on each ECU and the number of ECUs, which are no longer randomly chosen. Indeed, to ensure that the SE heuristic can be applied (no more than 11

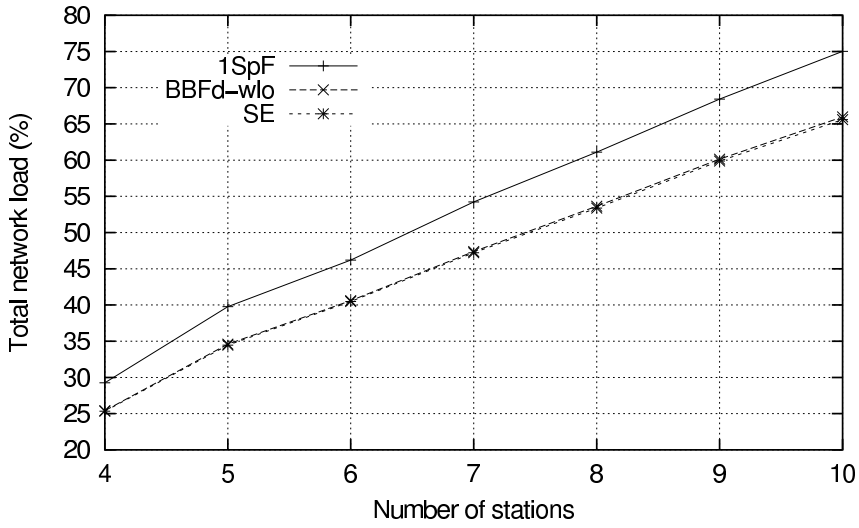


Figure B.3: Total network load for a number of stations varying from 4 to 10 with 10 signals by station.

The corresponding nominal load is 11% with 4 stations, 13.75% with 5 stations, 16.5% with 6 stations, 19.25% with 7 stations, 21.5% with 8 stations, 24% with 9 stations, and 27.5% with 10 stations. Average results over 100 feasible configurations with *One Signal per Frame* (1SpF), BBFd-wlo, SE heuristics.

signals on each ECU), the number of signals is fixed to 10 per station, while the number of stations varies according to the desired nominal load.

On figure B.3, one observes that SE and BBFd with the local optimization algorithm (BBFd-wlo) are always significantly better than 1SpF (up to 12.56% for SE and 12.06% for BBFd-wlo) which shows the effectiveness of the proposed heuristics. The performance of SE and BBFd-wlo are very close to each other (the curves are almost superposed on figure B.3), but SE remains better whatever the load. However, the gain is limited since it ranges from 0.3 to 0.56%. From a schedulability point of view, SE, as well as BBFd-wlo, always found a feasible solution in our experiments (made with freshness equal to production period) for nominal loads lower than 35%. Beyond this load level, the computation time for SE may become problematic due to the lack of feasible solutions.

Index

échéance relative, 4
âge maximal, signal, 38, 78
âge, signal, 38, 78

activation period, 4
age, signal, 38, 78
applicative task, 4
Audsley, 27, 78, 94

design patterns, 23, 60

ECU, 3

fraîcheur, 5, 76
frame packing, 75
freshness, 5, 76

generalized multiframe, 55, 92

intergiciel, 8, 13

maximum age, signal, 38, 78
methodology, 47
middleware, 8, 13
multiframe, 55, 91

networks, in-vehicule, 6

OSEK/VDX, 6, 18, 54

période d'activation, 4
préemption, 5
preemption, 5
priorité, 6
priority, 6

réseaux, 6
relative deadline, 4
response time, 7

services, communication, 45
signal, 5
sporadic, 57, 94

tâche applicative, 4
task identification strategies, 53
temps de réponse, 7

Bibliography

[1] OSEK Consortium. OSEK/VDX operating system, version 2.2.3, February 2005. Available at `http://www.osek-vdx.org`.

[2] OSEK Consortium. OSEK/VDX time-triggered operating system, version 1.0, July 2001. Available at `http://www.osek-vdx.org`.

[3] ISO. *ISO 11898 - Road vehicles - Interchange of digital information - Controller Area Network for high-speed Communication.* International Standard Organization, 1994. ISO 11898.

[4] ISO. *ISO 11519-2 - Road vehicles - Low Speed serial data communication - Part 2: Low Speed Controller Area Network.* International Standard Organization, 1994. ISO 11519-2.

[5] ISO. *ISO 11519-3 - Road vehicles - Low speed serial data communication. Part 3: Vehicle Area Network (VAN).* International Standard Organization, 1994. ISO 11519-3.

[6] SAE. SAE J1850 standard, class B data communications network interface, May 1994.

[7] LIN Consortium. LIN (Local Interconnect Network) specification package, version 2.0, September 2003. Available at `http://www.lin-subbus.org`.

[8] H. Kopetz et al. *Specification of the TTP/A Protocol.* University of Technology Vienna, September 2002.

[9] TTTech Computertechnik GmbH. *Time-Triggered Protocol TTP/C, High-Level Specification Document, Protocol Version 1.1*, November 2003. Available at `http://www.tttech.com`.

[10] FlexRay Consortium. FlexRay Communication System, Protocol Specification, version 2.0, June 2004. Available at `http://www.flexray.com`.

[11] ISO. *ISO 11898-4 - Road vehicles - Controller Area Network (CAN) - Part 4: Time-Triggered Communication.* International Standard Organization, 2000. ISO 11898-4.

[12] MOST Cooperation. MOST Media Oriented Systems Transport specification, version 2.3, August 2004. Available at `http://www.mostcooperation.com`.

[13] R. Santos Marques. Proposition d'un middleware tolérant aux fautes dans le contexte des applications embarquées dans les automobiles. Stage de dea, Univ. Henri Poincare Nancy 1, July 2002. MSc. Thesis / Stage de DEA.

[14] W. Eckerson. Searching for the middle ground. *Business Communications Review*, pages 46–50, 1995.

[15] P. Bernstein. Middleware: a model for distributed system services. *Commun. ACM*, 39(2):86–98, 1996.

[16] C. Britton. *IT Architectures and Middleware: Strategies for Building Large, Integrated Systems.* Addison Wesley, 2004.

[17] The Microsoft Corporation. The Microsoft .NET platform, 2002. `http://www.microsoft.com/net/`.

[18] OMG Object Management Group. *The Common Object Request Broker: Core Specification*, version 3.0.2 edition, December 2002.

[19] OMG Object Management Group. *Real-Time CORBA (Static Scheduling)*, version 1.2 edition, January 2005.

[20] OMG Object Management Group. *Real-Time CORBA (Dynamic Scheduling)*, version 2.0 edition, November 2003.

[21] T. Quinot, F. Kordon, and L. Pautet. From functional to architectural analysis of a middleware supporting interoperability across heterogeneous distribution models. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA'01)*, Rome, Italy, September 2001.

[22] OMG Object Management Group. *OMG Unified Modelling Language: Superstructure*, version 2.0 edition, October 2004.

[23] J. Hugues, L. Pautet, and F. Kordon. Refining middleware functions for verification purpose. In *Proceedings of the Monterey Workshop 2003 (MONTEREY'03)*, Chicago, IL, USA, September 2003.

[24] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture - a system of patterns.* John-Wiley and Sons, 1996.

[25] D. Schmidt. The ADAPTATIVE communication environment: an object-oriented network programming toolkit for developing communication software. In *Proceedings of the 12th Annual Sun Users Group Conference, SUG*, San Francisco, June 1994.

[26] D. Schmidt. Applying design patterns and frameworks to develop object-oriented communication software. In Peter Salus, editor, *The Handbook of Programming Languages.* MacMillan Computer Publishing, 1997.

[27] D. Schmidt, D. Levine, and S. Mungee. The design of the TAO real-time Object Request Broker. *Computer Communications*, 21(4), April 1998.

[28] D. Schmidt and C. Cleeland. Applying patterns to develop extensible ORB middleware. *IEEE Communications Magazine*, 37(4):54–63, April 1999.

[29] A. Rajnak, K. Tindell, and L. Casparsson. Volcano communications concept. Technical report, Volcano Communications Technologies AB, 1998.

[30] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a revolution in on-board communications. Volvo Technology report, Volvo, 1999.

[31] K. Tindell and A. Burns. Guaranteeing message latencies on controller area network (CAN). In CiA, editor, *Proceedings of the 1st International CAN Conference (ICC'94)*, pages 2–11, Mainz, Germany, 1994.

[32] A. Rajnak. Volcano - enabling correctness by design. In Richard Zurawski, editor, *Embedded Systems Handbook*, pages 43.1–43.18. CRC Press Taylor and Francis Group, Boca Raton, FL, USA, 2005.

[33] OSEK Consortium. OSEK/VDX communication specification, version 3.0.3, July 2004. Available at `http://www.osek-vdx.org`.

[34] OSEK Consortium. OSEK/VDX fault-tolerant communication, version 1.0, July 2001. Available at `http://www.osek-vdx.org`.

[35] EAST-EAA ITEA project, 2004. `http://www.east-eea.net`.

[36] AUTOSAR Consortium, 2004. `http://www.autosar.org`.

[37] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In Springer-Verlag, editor, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, Jyväskylä, Finnland, 1997.

[38] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02), year = 2002, address = Seattle, USA, pages = 161–173.*

[39] U. Freund, T. Riegraf, M. Hemprich, and K. Werther. Interface based design of distributed embedded automotive software - the TITUS approach. Technical Report Bericht 1547, VDI - Verein Deutscher Ingenieure, 2000.

[40] TTTech Computertechnik AG. The time-triggered technology and the two-level design approach, 2005. `http://www.tttech.com/technology/docs/general/TTTech-Two_Level_Design_Approach.pdf`.

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley, 1995.

[42] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-oriented software architecture - patterns for concurrent and networked objects.* John-Wiley and Sons, 2000.

[43] B. Douglass. *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[44] F. Buschmann. Real-time constraints as strategies. In *Proceedings of the Third Annual European Pattern Languages of Programming Conference*, 1998.

[45] R. Jimenez-Peris, M. Patino-Martinez, and S. Arevalo. Multithreaded rendezvous: a design pattern for distributed rendezvous. In *Proceedings of the 1999 ACM symposium on Applied computing*, pages 571–579, San Antonio, Texas, United States, 1999. ACM Press.

[46] B. Selic. An architectural pattern for real-time control software. *In Vlissides, J.M., J.O. Coplien, and N.L. Kerth. (eds.) Pattern Languages of Program Design 2. Addison-Wesley*, 1996.

[47] R. McKegney. Application of patterns to real-time object-oriented software design. Master's thesis, MSc. Thesis, Department of Computing & Information Sciences, Queen's University, July 2000.

[48] J. Hugues, L. Pautet, and F. Kordon. Contributions to middleware architectures to prototype distribution infrastructures. In *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, San Diego, CA, USA, JUNE 2003.

[49] T. Quinot. *Conception et réalisation d'un intergiciel schizophrène pour la mise en oeuvre de systèmes répartis interopérables.* PhD thesis, Université Pierre et Marie Curie - Paris VI, 2003.

[50] OMG Object Management Group. *UML Profile for Schedulability, Performance, and Time*, version 1.1 edition, January 2005.

[51] P. Tessier, S. Gérard, C. Mraidha, and J. Geib. A component-based methodology for embedded system prototyping. In *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, page 9, Washington, DC, USA, 2003. IEEE Computer Society.

[52] T. Phan, S. Gérard, and F. Terrier. Real-time system modeling with ACCORD/UML methodology: illustration through an automotive case study. *Languages for system specification: Selected contributions on UML, systemC, system Verilog, mixed-signal systems, and property specification from FDL'03*, pages 51–70, 2004.

[53] J. Pasaje, M. Harbour, and J. Drake. Mast real-time view: A graphic UML tool for modeling object-oriented real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, page 245, Washington, DC, USA, 2001. IEEE Computer Society.

[54] B. Douglass. *Real-time UML second edition - developing efficient objects for embedded systems.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[55] M. Saksena, P. Karvelas, and Y. Wang. Automatic synthesis of multi-tasking implementations from real-time object-oriented models. In *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 360. IEEE Computer Society, 2000.

[56] B. Selic, G. Gullekson, and P. Ward. *Real-time object-oriented modeling.* John Wiley & Sons, Inc., 1994.

[57] A. Colin, I. Puaut, C. Rochange, and P. Sainrat. Calcul de majorants de pire temps d'exécution : état de l'art. *Technique et Science Informatiques*, 22(5):651–677, 2003.

[58] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS164, University of York, November 1991.

[59] L. Sha, T. Abdelzaher, K. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.

[60] C. Norström, K. Sandström, and M. Ahlmark. Frame packing in real-time communication. Technical report, Mälardalen Real-Time Research Center, Sweden, 2000.

[61] P. Pop, P. Eles, and Z. Peng. Schedulability-driven frame packing for multicluster distributed embedded systems. *Trans. on Embedded Computing Sys.*, 4(1):112–140, 2005.

[62] A. Mok and D. Chen. A multiframe model for real-time tasks. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96)*, page 22. IEEE Computer Society, 1996.

[63] S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.

[64] A. Mok. *Fundamental Design Problems for the Hard Real-Time Environments.* PhD thesis, Massachusetts Institute of Technology (MIT), 1983.

[65] R. Santos Marques and 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.

[66] 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.

[67] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[68] P. Feiler. Real-time application development with OSEK: A review of the OSEK standards. Technical Report CMU/SEI-2003-TN-004, Software Engineering Institute, Carnegie Mellon, 2003. Available at `http://www.sei.cmu.edu/pub/documents/03.reports/pdf/03tn004.pdf`.

[69] R. Yerraballi. Real-time operating systems: An ongoing review. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS'2000), WIP Section*, Orlando Fl, October 2000.

[70] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[71] H. Takada and K. Sakamura. Schedulability of generalized multiframe task sets under static priority assignment. In *Proceedings of the 4th International Workshop on Real-Time Computing Systems and Applications (RTCSA'97)*. IEEE Computer Society, 1997.

[72] K. Tindell. An extendible approach for analyzing fixed priority hard real-time tasks. Technical Report YCS189, University of York, 1992.

[73] R. Santos Marques and F. Simonot-Lion. Design-patterns based development of an automotive middleware. In *Proceedings of the 6th IFAC International Conference on Fieldbus Systems and their Applications (FeT'2005)*, Puebla, Mexico, November 2005.

[74] Holger Giese and Oliver Niggemann, editors. *Postworkshop Proceedings of the Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 3), Heinz Nixdorf MuseumsForum, Paderborn, Germany. October 13 and 14, 2005*, HNI-Verlagsschriftenreihe, Paderborn, Germany, 2006. (under preparation).

[75] S. Yacoub and H. Ammar. UML support for designing software systems as a composition of design patterns. In Martin Gogolla and Cris Kobryn, editors, *Proceedings of the 4th International Conference on the Unified Modeling Language - Modeling Languages, Concepts and Tools - (UML 2001)*, pages 149–165, Toronto, Canada, October 2001. Springer.

[76] R. Saket and N. Navet. Frame packing algorithms for automotive applications. Technical Report INRIA RR-4998, Institut National de Recherche en Informatique et en Automatique (INRIA), 2003. Available at `http://www.inria.fr/rrrt/rr-4998.html`.

[77] M. Abramowitz and I. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, USA, 1974.

[78] R. Santos Marques, N. Navet, and F. Simonot-Lion. Frame packing under real-time constraints. In *Proceedings of the 5th IFAC International Conference on Fieldbus Systems and their Applications (FeT'2003)*, pages 185–192, Aveiro, Portugal, July 2003.

[79] 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.

[80] T. Osogami and H. Okano. Local search algorithms for the bin packing problem and their relationships to various construction heuristics. *Journal of Heuristics*, 9(1):29–49, 2003.

[81] M. Orlov. Efficient generation of set partitions. [Online], 2002. Available at `http://www.cs.bgu.ac.il/~orlovm/papers/partitions.pdf`.

[82] R. Santos Marques, N. Navet, and F. Simonot-Lion. Construction de trames sous contraintes temps-réel. In *Proceedings of the 11th International Conférence on Real-Time Systems & Embedded Systems 2003 (RTS'2003)*, pages 235–255, Paris, France, April 2003.

[83] R. Santos Marques, N. Navet, and F. Simonot-Lion. Configuration of in-vehicle embedded systems under real-time constraints. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'2005)*, Catania, Italy, September 2005.

[84] Philips Electronics. *Application note - SJA1000 Stand-alone CAN controller - AN97076*, December 1997.

[85] Intel Corporation. *87C196CB supplement to 8XC196NT user's manual*, August 2004.

[86] Infineon. *AP16021 - C50x and C16x family - CAN interrupt structure with the example of the C167CR*, October 2005.

[87] K. Tindell and A. Burns. Guaranteed message latencies for distributed safety critical hard real-time networks. Technical Report YCS229, University of York, 1994.

[88] N. Navet. *Évaluation de performances temporelles et optimisation de l'ordonnancement de tâches et messages.* PhD thesis, Institut National Polytechnique de Lorraine, Nancy, France, November 1999.

[89] C. Elekin. *An optimization framework for scheduling of embedded real-time systems.* PhD thesis, Chalmers University of Technology, Sweden, 2004.

[90] P. Hladik, H. Cambazard, A. Déplanche, and N. Jussien. Guiding architectural design process of hard real-time systems with constraint programming. In *the 3rd Taiwanese-French Conference on Information Technology (TFIT'06)*, Nancy, France, 2006.

[91] G. Quan and X. Hu. Enhanced fixed-priority scheduling with (m,k)-firm guarantee. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000)*, pages 79–88, Orlando, FL, USA, November 2000.

[92] J. Migge. rts tool, 2002. Program and manual available at `http://www.loria.fr/~nnavet`.

Résumé

Notre objectif est de proposer une méthodologie pour le développement d'un intergiciel embarqué dans l'automobile offrant des services de communication aux applications. Le cadre d'utilisation de nos travaux est la conception de systèmes embarqués dans les véhicules. Ces applications requièrent un intergiciel capable de fournir des services standards de communication, qui cachent la localisation des participants aux échanges, qui masquent l'hétérogénéité des plates-formes de communication, et qui garantissent le respect des contraintes temporelles imposées sur l'échanges et sur l'exécution des participants.

La méthodologie proposée vise la conception d'un intergiciel optimisé et pour cela aborde deux aspects : la spécification d'une architecture d'implémentation, et la construction d'une configuration faisable. 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 minimise son utilisation des ressources disponi-bles. 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 plate-forme concrète (sous la forme d'un ensemble de tâches).

La procédure proposée pour la construction de la configuration de l'intergiciel calcule les caractéristiques temporelles faisables de l'intergiciel et des trames émises par les stations d'un réseau CAN. Elle prévoit aussi une étape pour le calcul d'une allocation de priorités faisable pour les tâches de l'application sur chaque station. L'optimalité de la configuration est atteinte en assurant le respect de toutes les contraintes temporelles imposées sur les échanges et sur l'exécution des tâches de l'application et de l'intergiciel.

Mots-clés: Systèmes Embarqués dans l'Automobile, Génie Logiciel, Intergiciel, Optimisation, Configuration.

Abstract

Our objective is to propose a methodology for the development of an automotive embedded middleware that provides communication services to the applicative level software. This work is focused on the design of automotive functions, where the nowadays context demands a middleware capable of offering standard communication services, hiding the localization of the participants in the exchanges, masking the heterogeneity of communication platforms, and ensuring that the timing constraints imposed on the exchanges and on the execution of the participants are met.

The proposed methodology is aimed for the design of an optimized middleware. For this purpose, it deals with two topics: the specification of an implementation architecture, and the construction of a feasible configuration. The implementation architecture is optimized because the middleware is well adapted to its execution environment (operating system OSEK/VDX OS), and minimizes the utilization of the available resources. It contributes, on the one hand, to a specification of a software architecture (built using design patterns), and, on the other hand, to mechanisms allowing to deploy this software architecture onto a concrete platform (under the form of a set of tasks).

The algorithm proposed for the construction of a configuration determines feasible timing characteristics for the middleware and for the frames exchanged over a CAN bus. It covers also the calculation of a feasible set of priorities for the applicative tasks executing on each station of the bus. The correctness of the configuration is achieved by ensuring that the timing constraints imposed on the exchanges and on the execution of the applicative and middleware tasks are met.

Keywords: Automotive Embedded Systems, Software Engineering, Middleware, Optimization, Configuration.