Sommaire

1

Introduction

Dans le contexte automobile actuel, les systèmes embarquent de plus en plus de logiciel. Cela induit une complexité croissante des calculateurs embarqués. Historiquement, les calculateurs étaient développés de manière *ad-hoc* afin d'optimiser au maximum les ressources utilisées. Cependant, cette méthode de développement ne permet pas nécessairement de réutiliser le logiciel embarqué et conduit à des temps de développement plus important. Pour cette raison, depuis quelques années, l'industrie automobile a mis en place un standard qui a pour objectif de permettre une meilleure réutilisation du code, une standardisation des interfaces pour une meilleure communication entre les différents protagonistes, et une abstraction des couches basses du calculateur [16].

Cependant, avec l'approche qui est dorénavant utilisée, il est possible que deux fonctionnalités, qui étaient précédemment totalement décorrélées, soient implémentées sur le même calculateur et puissent donc interférer l'une avec l'autre. C'est pour cette raison qu'une norme a été développée pour l'automobile afin de donner des méthodes pour faire un développement de systèmes électriques/électroniques (E/E) sûrs de fonctionnement [57].

L'émergence et la croissance des systèmes logiciels dans les architectures automobiles conduisent à un certain nombre de challenges développés par Broy dans [31] en 2006. Depuis lors, quelques uns d'entre eux ont été traités par différents protagonistes industriels, mais un certain nombre reste toujours d'actualité, comme la possibilité de faire des mises à jour partielles efficaces.

Nous traitons des mises à jour applicatives pour des calculateurs enfouis dans le véhicule. Cela signifie que les systèmes d'infotainement qui peuvent être présents ne sont pas concernés par ce processus.

D'autre part, bien qu'il soit à l'heure actuelle possible de faire des mises à jour dans les calculateurs embarqués des véhicules, les méthodes couramment employées manquent pour certaines de standardisation. En effet, il existe actuellement deux possibilités pour faire des mises à jour :

— Soit des patchs mémoires bas niveau. Cette technique n'est pas standardisée et nécessite une connaissance approfondie du contenu du calculateur.

— Soit un rechargement complet du calculateur. Lorsque les mises à jour visent

des éléments applicatifs de faible taille, cette technique n'est pas optimale. En effet, dans ce cas, l'intégralité du code contenu dans le calculateur est modifié, ce qui représente plusieurs méga-octets de code. Si la mise à jour qui doit être ajoutée ne représente que quelques kilo-octets, cette technique est clairement non appropriée.

Pour cette raison, nous proposons dans ce travail une nouvelle méthode qui permet de faire des mises à jour partielles du calculateur, de manière standardisée, tout en respectant le standard AUTOSAR et en prenant en considération son impact à plusieurs niveaux dans le calculateur. Tout d'abord, du point de vue architectural et processus de développement, puis du point de vue temps-réel, et enfin du point de vue de l'implémentation.

La possibilité de faire des mises à jour partielles présente des intérêts non seulement pour le constructeur automobile, mais également pour les possesseurs de véhicules.

Tout d'abord, pour les personnes ayant une voiture, faire des mises à jour partielles pourrait leur permettre de tenir leur véhicule à jour sans nécessairement devoir retourner au garage. En effet, étant donné qu'une mise à jour partielle présente une quantité réduite de données, il serait alors possible de faire des mises à jour OTA (Over-The-Air), et donc de réduire le temps d'indisponibilité du véhicule. De plus, de cette manière chaque client pourrait personnaliser de manière fine son véhicule, que ce soit au moment de l'achat ou bien pendant la durée de vie du véhicule. Enfin, cela permettrait d'ajouter des options qui n'existaient pas au moment de l'achat du véhicule et de lutter ainsi contre l'obsolescence.

En ce qui concerne le constructeur automobile, les avantages sont multiples :

— **Infrastructure** : dans l'état actuel des choses, l'infrastructure débarquée utilise un réseau 2G qui présente des débits assez faibles. Dans le cadre d'une mise à jour à distance, une mise à jour partielle permettrait donc d'utiliser cette infrastructure au plus juste sans la surcharger inutilement avec l'intégralité du code d'un calculateur.
— **Débit du bus CAN** : une fois la mise à jour parvenue jusqu'à la voiture en utilisant une passerelle dédiée, il est ensuite nécessaire de la faire transiter jusqu'au calculateur de destination. Pour cela le bus utilisé serait le CAN qui présente un débit assez faible. Ce d'autant plus que d'autres messages utilisent également ce canal pour transiter.
— **Reprise de parc** : une fois la voiture produite, il arrive qu'il soit nécessaire de faire des modifications à grande échelle sur l'intégralité des calculateurs. Diminuer le temps nécessaire pour ces mises à jour pourrait faire gagner significativement du temps (par exemple passer de 10 minutes/calculateur en rechargeant tout à 1 minute/calculateur avec une mise à jour partielle).
— **Ligne de production** : les temps de cycle sur les lignes de production sont très limités et déterminés de manière précise. Lors du chargement d'un calculateur, le véhicule est immobilisé pendant un temps précis pendant lequel rien d'autre ne peut être fait. Gagner quelques minutes ou même secondes sur une ligne de production en ne chargeant que le strict minimum pourrait être très intéressant.
— **Répondre à une norme européenne** : les constructeurs automobiles sont obli-

gés de fournir aux garagistes indépendants les moyens de réparer leurs véhicules. La mise à jour partielle permet de leur fournir le minimum de code possible afin de protéger au maximum la propriété intellectuelle de Renault.

Généralement la durée de vie des systèmes qui nous concernent dans le cadre de ce travail est de plusieurs années. Il est donc intéressant d'avoir un système qu'il est possible de maintenir facilement et dont n'importe quelle partie peut être remplacée.

Ce mémoire présente notre approche suivant 5 chapitres. Le Chapitre 2 présente le contexte dans lequel nous travaillons, ainsi que les différentes méthodes qui existent à l'heure actuelle pour les mises à jour dans des systèmes embarqués, distribués et à composants. Nous présentons également l'architecture logicielle AUTOSAR qui correspond au cadre dans lequel ce travail prend place et définissons ainsi la terminologie correspondante.

Le Chapitre 3 aborde ensuite le processus de développement pour les applications embarquées de type AUTOSAR. Nous décrivons ici le processus de développement complet ainsi que l'étape supplémentaire que nous avons ajoutée par rapport à la méthodologie du standard et des critères précis d'allocation. Nous définissons également des concepts liés à la mise à jour partielle, à savoir la notion d'espace d'adaptation et de *container*, qui permettent de réserver des espaces dans l'architecture pour des mises à jour ultérieures. Enfin, nous expliquons comment ces concepts supplémentaires peuvent être ajoutés dans le processus de développement, et les modifications qui sont alors nécessaires.

Le Chapitre 4 définit un certain nombre de concepts liés à l'ordonnancement temps-réel, et montre comment ces concepts peuvent être appliqués dans le cas d'un système répondant au standard AUTOSAR. Nous décrivons ici également les notions spécifiques à AUTOSAR et étudions comment les concepts issus de la théorie de l'ordonnancement temps-réel peuvent être appliqués spécifiquement dans le cadre de la mise à jour partielle.

Le Chapitre 5 décrit ensuite les aspects liés plus en détail à l'implémentation des mises à jour. En particulier, nous expliquons comment les différentes versions d'une application et de ses mises à jour peuvent être gérées grâce à la notion d'arbre de mise à jour, et le processus de création de nouvelles fonctionnalités. Nous détaillons également comment la mémoire doit être gérée pour permettre, non seulement de stocker les mises à jour, mais également de les exécuter. Enfin, les aspects liés à la sûreté de fonctionnement sont traités dans ce Chapitre.

Le Chapitre 6 présente la validation expérimentale des concepts définis précédemment en utilisant une application fournie par Renault. Cette application gère les clignotants et les warnings. Deux mises à jour sont possibles pour cette application. Nous décrivons donc dans ce Chapitre les aspects architecturaux et temps-réel, non seulement pour l'application initiale, mais également pour chacune des mises à jour possibles. Ainsi, nous montrons la faisabilité de notre approche sur un exemple de taille suffisamment raisonnable pour être significatif d'une application embarquée automobile.

En conclusion, nous tirerons les leçons de cette étude en soulignant les contraintes auxquelles nous avons du obéir pour effectuer l'adaptation dynamique dans le contexte AUTOSAR.

2

Contexte, problématique et état de l'art

Sommaire

2.1 Introduction

Cette section présente tout d'abord la notion de système embarqué ainsi que les contraintes relatives à ce type de système. Dans ce contexte, nous discutons ensuite de la possibilité de faire évoluer le système grâce à des mises à jour et décrivons les différents travaux effectués dans le domaine de la mise à jour en général.

2.2 Systèmes embarqués

A l'heure actuelle, les systèmes embarqués jouent un rôle important dans notre vie de tous les jours. Même s'ils sont généralement invisibles pour l'utilisateur final, leur bon fonctionnement de manière continue et sans erreur revêt une importance majeure. D'autre part les architectures logicielles des systèmes embarqués sont conçues spécifiquement pour répondre à des besoins particuliers.

Un système embarqué est un système autonome qui possède des ressources temporelles et matérielles limitées. Généralement le comportement attendu est temps-réel, c'est-à-dire que le système doit respecter des contraintes temporelles précises. Il doit alors être capable de traiter des informations et de répondre dans un temps adapté à l'évolution de son environnement. Un système embarqué fait partie d'un système plus grand.

2.2.1 Temps-réel

Une distinction qualitative peut être faite entre deux types de systèmes temps-réel : les systèmes temps-réel stricts (*hard real-time*) et les systèmes temps-réel mous (*soft real-time*). Dans le premier cas, la non-satisfaction de contraintes temporelles peut conduire à des situations catastrophiques, par exemple dans le cas des fonctions critiques de l'aéronautique. Dans le second cas, des dépassements peuvent être tolérés dans une certaine mesure, par exemple dans le cas du streaming vidéo. Il existe un continuum entre les systèmes temps-réel mous et stricts. En effet, certains systèmes peuvent adresser simultanément des contraintes des deux types, par exemple un système qui perdrait certains messages mais informerait l'utilisateur [93].

Pour garantir le respect des propriétés temps-réel, des vérifications préalables peuvent être faite sur un système. Par exemple, dans le cas de systèmes utilisant un système d'exploitation temps-réel, il est possible de faire une analyse d'ordonnancement qui vérifie que les contraintes temps-réel sont respectées [64].

2.2.2 Sûreté

Les contraintes de sécurité-innocuité peuvent être critiques pour certains systèmes (par exemple, pour les systèmes de surveillance du coeur des centrales nucléaires), et dans ce cas des tests spécifiques doivent être mis en place. Pour d'autres systèmes, comme les smartphones, ces contraintes peuvent être relâchées dans la mesure où une défaillance

peut facilement être corrigée et n'entraîne pas de conséquences potentiellement dramatiques.

On parle de *sûreté de fonctionnement* pour un système lorsque celui-ci est capable de "délivrer un service de confiance justifiée" [2]. L'une des facettes de la sûreté de fonctionnement est la *disponibilité*, c'est-à-dire que le système doit être prêt à être utilisé à tout instant. Un système embarqué doit avoir la meilleure disponibilité possible. En effet, l'indisponibilité peut mener à des conséquences catastrophiques si le système n'est pas en mesure de répondre à une sollicitation extérieure et de réagir de manière appropriée.

2.2.3 Ressources limitées

Par nature, les systèmes embarqués ont des ressources limitées. Généralement il s'agit de ressources temporelles et de mémoire disponible. En effet, avoir plus de temps disponible signifie généralement avoir un calculateur plus puissant, donc plus onéreux. Une quantité de mémoire plus importante est aussi plus chère. Les applications destinées aux systèmes embarquées doivent donc être conçues de manière à optimiser les ressources disponible et par conséquent limiter les coûts.

2.2.4 Modèle de développement

De plus en plus, le logiciel pour les système embarqué est conçu à partir de modèles [24, 88, 71, 56, 29, 4]. Il s'agit alors de 'Model-Based Design". L'idée est de proposer une approche structurée pour le développement de systèmes complexes.

L'avantage proposé par ces approches est un développement plus aisé et rapide grâce à des outils spécialisés. Par exemple, dans [88], les auteurs expliquent comment l'utilisation de cette approche permet d'accélérer le développement de systèmes de contrôle pour les locomotives.

L'objectif est également de faciliter les communications entre les différents intervenants sur le logiciel, de mieux gérer la complexité et d'améliorer la qualité du logiciel [29]. Cette approche permet de répondre aux besoins de développement pour les systèmes embarqués, en terme de tests, de prédictibilité et de système distribués [4]. Les problèmes liés à la conception de système embarqué et certaines réponses sont détaillées dans [4].

Un exemple de conception de système embarqué dans le domaine de l'aéronautique grâce à une approche "Model-Based" est donnée dans [56]. Les auteurs de ce travail expliquent comment les coûts et les risques au cours du développement peuvent être réduits grâce à cette approche. L'industrie de l'automobile se tourne également vers cette approche [71].

Ces approches peuvent également permettre de s'insérer au mieux dans une chaine outillée existante afin de permettre des reconfigurations en ligne du système [24].

2.3 Mise à jour

Étant donné le contexte particulier qui est lié aux systèmes embarqués, de nombreux problèmes émergent pour leur mise à jour [58]. En effet, les systèmes embarqués

doivent répondre à des contraintes de disponibilité importantes. Il peut être inacceptable que certains systèmes embarqués deviennent inutilisables et/ou doivent être redémarrés lors d'une mise à jour [46], alors que cette contrainte est parfaitement acceptable dans d'autres cas.

A l'heure actuelle, il est relativement fastidieux de faire des mises à jour ou des évolutions dans les systèmes embarqués car, historiquement, leur conception est faite de manière *ad hoc* afin d'optimiser au mieux les ressources utilisées. Toutefois, la flexibilité est une caractéristique souhaitable pour ces systèmes. En effet, ils doivent de plus en plus se plier à l'évolution constante de l'environnement et/ou aux désirs de changement des utilisateurs. Ces derniers ont eu l'habitude de pouvoir faire des modifications facilement sur leurs smartphones ou leurs ordinateurs.

Enfin, l'un des problèmes est lié aux ressources limitées dont disposent les systèmes embarqués : comment dans ce cas optimiser l'utilisation des ressources alors que nous souhaitons pouvoir faire des modifications sur le système ? En effet, ces modifications sont potentiellement plus consommatrices de ressources. En outre, il est nécessaire d'ajouter des mécanismes supplémentaires afin de permettre la flexibilité du système, ce qui est en contradiction avec l'optimisation des ressources disponibles.

Afin de réaliser la mise à jour, d'autres aspects doivent être étudiés, par exemple l'acheminement de la mise à jour au système, la gestion du côté distribué de cette mise à jour, ou encore la granularité des mises à jour. Ces aspects plus spécifiques sont abordés dans les chapitres suivants, et l'état de l'art correspondant est détaillé à ce moment là.

La mise à jour dans les systèmes embarqués automobiles peut être vue sous trois angles :

— la mise à jour de systèmes embarqués,
— la mise à jour de systèmes distribués
— les architectures pour mise à jour.

La reconfiguration dynamique, *i.e.* la mise à jour du système soit sous forme d'ajout de fonctionnalité, soit sous forme de modification interne, pendant son exécution, est un problème qui intéresse les chercheurs depuis de nombreuses années [28, 48, 92, 86]. Cependant mettre à jour des systèmes tels que ceux utilisés dans l'industrie automobile n'est pas aisé [58, 4] et de nombreux problèmes se posent :

— Comment optimiser les ressources tout en faisant évoluer le système ?
— Comment assurer la cohérence globale du changement si l'évolution n'est pas monolithique ? En effet, dans le cas où une mise à jour est composée de plusieurs éléments, il faut que toutes les parties soient intégrées, ou qu'aucune ne le soit.
— Comment minimiser l'indisponibilité du système lors des mises à jour ? Comme expliqué précédemment (section 2.2.2), la disponibilité du système est un enjeu pour la sûreté de fonctionnement. Il faut donc s'assurer que les changements ne rendent pas le système indisponible pendant une durée inacceptable par rapport à la fonction à réaliser.
— Comment garantir la sécurité-innocuité du système ? En effet, les systèmes embarqués peuvent réaliser des fonctions critiques et il faut s'assurer que toute modification apportée ne perturbe pas le bon fonctionnement de celui-ci.

2.3.1 Mise à jour pour les systèmes embarqués

Faire évoluer un système embarqué ayant des ressources limitées peut être beaucoup plus problématique que dans le cas de systèmes plus traditionnels.

Pour ces raisons, il faut prévoir à l'avance et insérer des mécanismes spécifiques lors de la conception du système pour permettre des évolutions *a posteriori*. Les questions majeures sont les suivantes :

— Comment minimiser l'utilisation de ressources tout en permettant les évolutions ? Par exemple, dans [42], les mises à jour sont gérées par un élément ayant des ressources plus importante et distribuées aux éléments ayant des ressources limitées.
— Comment limiter au maximum le temps d'indisponibilité du système pour faire ces évolutions ? Certaines approches permettent de faire des mises à jour "à chaud", c'est-à-dire sans éteindre le système et en continuant à l'utiliser [5, 55].
— Comment conserver les propriétés de sécurité-innocuité du système ?

Les caractéristiques les plus intéressantes et recherchées lorsque l'on souhaite rendre possible les mises à jour dynamiques dans un système embarqué sont les suivantes : simplicité, transparence pour les applications et utilisation du moins de ressources possible [46, 58]. Dans [104], les auteurs proposent un classement des différentes méthodes permettant de faire des mises à jour dans les systèmes embarqués. Ces systèmes sont classés en fonction des critères principalement liés à la facilité pour un utilisateur à effectivement faire des mises à jour dynamiques dans le système.

Lorsque l'on parle d'évolution pour les systèmes embarqués, il existe plusieurs types selon l'approche qui est choisie. Il existe quatre types possibles d'évolution ou de mise à jour pour ces systèmes :

— les évolutions reposant sur les modèles,
— les évolutions reposant sur le binaire,
— la reconfigurations du système lui-même,
— les mises à jour pour les systèmes d'exploitation.

Les trois premiers types nous intéressent tout particulièrement et sont détaillées dans la suite de cette section. La mise à jour pour les systèmes d'exploitation nous intéresse dans une moindre mesure car notre objectif est surtout focalisé sur les évolutions des parties applicatives du système.

2.3.1.1 Modèles et évolutions

Les mises à jour utilisant des modèles présentent un intérêt particulier dans le cadre du Model-Based Design (MBD) qui se développe de nos jours de plus en plus dans l'industrie. Les avantages de ce type de procédé sont nombreux, tout particulièrement lorsque les systèmes en question sont complexes. En effet, bien que cette approche soit différente des méthodes traditionnelles de développement, elle permet de faciliter grandement la conception, la simulation, le test et la vérification. De plus, le MBD facilite les communications grâce à un environnement de développement homogène et une réutilisation beaucoup plus facile des éléments.

Becker *et al.* [24] proposent une approche intéressante pour ajouter plus de flexibilité dans une architecture embarquée de type AUTOSAR en travaillant directement au niveau du modèle AUTOSAR. Ce travail présente un outil appelé dTool développé afin de définir les reconfigurations possibles du système en fonction de l'évolution de l'environnement. Ainsi, pour chacune des situations possibles, un nouveau modèle doit être créé. Un automate de reconfiguration est également créé dans dTool pour permettre de prendre une décision de reconfiguration en fonction de la situation. L'objectif des auteurs ici est de proposer une approche qui puisse s'interfacer facilement avec une chaine d'outil AUTOSAR existante. L'inconvénient majeur de l'approche proposée est qu'elle est fortement attachée à un outil spécifique qui permet de faire des développement d'application AUTOSAR, à savoir SystemDesk distribué par dSpace. Elle manque donc de transversalité par rapport au standard lui même.

Une autre approche plus générale est proposée dans [32]. Dans ce travail les auteurs présentent une suite outillée qui repose sur UML. Elle n'a pas pour vocation d'être spécifique au domaine automobile, mais s'adresse aux systèmes temps-réel critiques en général. L'approche utilise une partie restreinte des possibilités de modélisation d'UML afin d'utiliser des outils de model-checking et de générer du code. De plus, cette approche permet de vérifier un certain nombre de propriétés temporelles du système. Toutefois, le modèle UML ne fait pas partie intégrante du processus de développement tel qu'il est défini dans le standard AUTOSAR.

Enfin, dans [94], les auteurs proposent un modèle formel pour supporter les mécanismes de mise à jour dynamique des composants. Dans les systèmes embarqués, les mécanismes permettant les mises à jour sont majoritairement réalisés de manière *ad hoc*. Ces mécanismes peuvent donc être non seulement très complexes, mais également source d'erreur. Ainsi, il est très difficile d'ajouter de manière automatique dans tous les nouveaux systèmes des mécanismes permettant les mises à jour. En analysant les différentes techniques proposées pour faire des mises à jour dans différents domaines, les auteurs ont défini le concept de Dynamic Update Connector (DUC) qui correspond à un élément du système qui gère toutes les connections entre les composants. De cette manière lorsqu'un nouveau composant vient remplacer un ancien, le DUC gère toutes les connections de manière transparente pour le reste du système. Ce composant gère également le transfert d'état. Les différentes interactions entre les composants, ainsi que les mécanismes de mises à jour, peuvent ensuite être décrites en utilisant CSP [52] (Communicating Sequential Processes) afin de vérifier de manière formelle trois propriétés : la transparence pour les clients, l'obtention de réponse aux requêtes des clients de la part du serveur (propriété d'équité), et l'absence de deadlock (propriété de vivacité).

2.3.1.2 Évolutions au niveau binaire

Les mises à jour des systèmes embarqués au niveau du binaire se concentrent plus sur l'intégration de la mise à jour dans le système d'un point de vue technique sans vraiment se concentrer sur la partie fonctionnelle qui y correspond. Dans ce contexte, il s'agit de faire une mise à jour différentielle du binaire, c'est-à-dire, déterminer quelles parties ont été changées, sans pour autant reprogrammer toute la mémoire de l'ECU

(Electronic Control Unit). La mise à jour repose alors sur un système qui permet de construire un arbre de différence entre le binaire actuel et le binaire de mise à jour. Des exemples d'approche qui permettent d'identifier les différences entre deux fichiers binaires sont `bdiff` [91] ou encore `Vdiff/Vdelta` [91]. Ces deux approches cherchent les plus longues chaines identiques et représentent les différences entre les deux fichiers par une séquence d'instructions de changements. Ceci peut se révéler techniquement difficile dans un contexte embarqué car les applications sont stockées en mémoire flash et un secteur de flash ne peut pas être partiellement ré-écrit : il faut tout d'abord l'effacer intégralement avant de pouvoir écrire à nouveau [45].

L'approche proposée par Nakanishi *et al.* [73] permet de faire des mises à jour de bas niveau de manière différentielle. Dans cette approche, des patchs logiciels sont construits à partir du binaire présent dans le système et du nouveau binaire qui doit être chargé. Ainsi un fichier binaire est reconstruit à partir des différences trouvées entre l'ancien et le nouveau binaire. Cela permet de limiter la taille des patchs qui doivent être envoyés. Des mécanismes spécifiques doivent cependant être ajoutés dans le système embarqué lui-même pour pouvoir reconstruire le binaire complet après que le patch a été réceptionné. Cette approche est très intéressante, tout particulièrement dans le cadre d'une mise à jour OTA (*Over-The-Air*) car elle permet d'avoir une taille de binaire à envoyer très réduite (comparée à l'envoi complet du nouveau binaire).

2.3.1.3 Reconfigurations du système

Habituellement, lorsque l'on parle de reconfiguration du système, il s'agit de modifications faites au sein du système lui-même, sans nécessairement ajouter des éléments extérieurs. Il s'agit donc de modifier le comportement du système sans nécessairement introduire de nouvelles fonctionnalités. Par exemple, il est possible de n'exécuter qu'une partie de l'ensemble des fonctionnalités ou encore de n'exécuter certaines fonctionnalités que dans un contexte donné. Cette reconfiguration peut se faire de manière adaptative en fonction du contexte ou bien à la demande de l'utilisateur. Ce concept peut être appliqué dans le cas où le système rencontre un problème et qu'il se met dans un "mode dégradé" où seuls les éléments essentiels sont exécutés : dans ce cas une reconfiguration peut être nécessaire pour atteindre ce mode dégradé.

Ces évolutions permettent de préserver les propriétés de sécurité-innocuité du système et de fournir une continuité de service même si certaines fonctionnalités non prioritaires peuvent être désactivées en cas d'absolue nécessité. Cela permet de répondre à deux contraintes majeures concernant le temps d'indisponibilité qui se voit réduit ainsi que les propriétés de sécurité qui sont conservées. Cependant, les évolutions qui sont permises dans le cas de reconfiguration du système lui-même sont assez limitées car elles ne prennent pas en compte l'ajout de nouvelles fonctionnalités : seuls des états pré-définis et connus du système peuvent être atteints.

Les mises à jour reposant sur la reconfiguration peuvent également permettre de répondre à des problématiques de sécurité-innocuité du système. Dans [51] par exemple, les auteurs proposent de mettre en place un système de monitoring qui détecte les problèmes qui peuvent se produire dans le système surveillé et de placer ce dernier dans un état

dégradé : seules les fonctionnalités essentielles sont alors conservées. Dans le domaine de l'adaptation pour la sécurité-innocuité, Schneider et Trapp, dans [100] développent le concept de "survivabilite" pour un système embarqué automobile, c'est-à-dire décider dynamiquement de ne garder que les fonctions critiques, en cas de problème. Les auteurs proposent une méthode pour modéliser, analyser et vérifier au moment de la conception les mécanismes permettant de faire une adaptation dynamique du système. Pour cela, un nouveau concept appelé MARS est défini afin d'étendre les possibilités de modélisation du système pour spécifier de manière formelle les mécanismes d'adaptation.

Dans [43], les auteurs proposent une approche différente, fondée sur une reconfiguration au niveau des tâches. En fonction du contexte, un algorithme détermine si une tâche doit s'exécuter ou non. Cette approche est principalement destinée à être mise en place sur des systèmes d'*infotainement*. Dans ce contexte, un *middleware* spécifique au projet DySCAS [1] (Dynamically Self-Configuring Automotive System) est mis en place. Cette approche tranche avec la manière plus classique de voir les systèmes automobiles embarqués, car généralement dans ce domaine les tâches sont fixées statiquement au moment de la conception et aucune modification ne peut être faite *a posteriori*. Leur approche a un certain nombre de limites, en particulier elle se confine à un seul ECU (Electronic Control Unit). Chaque tâche est représentée par un nouveau modèle dans lequel elle peut être chargée, activée (dans ce cas la tâche peut s'exécuter) ou désactivée (la tâche ne peut plus s'exécuter). Le choix de l'exécution ou non des tâches repose sur la disponibilité des entrées de la tâche ainsi que sur la charge CPU.

Becker *et al.* [24] insistent sur le besoin pour les systèmes automobiles actuels d'avoir une plus grande flexibilité, par exemple dans le cas où un capteur est défaillant. Étant donné l'importance de la chaîne outillée utilisée dans le contexte d'AUTOSAR, leur objectif est d'intégrer au mieux aux modèles de l'application AUTOSAR, réalisés dans un outil donné, des modules spécifiques qui permettent la reconfiguration du système.

Ainsi, ces approches pour les mises à jour peuvent être vues comme complémentaires. Elles répondent à diverses questions. Les mises à jour reposant sur les modèles définissent des évolutions qui peuvent être intégrées au système en utilisant des mises à jour différentielles de bas niveau (binaire). Enfin, il est également possible d'ajouter à cela des mécanismes qui permettent au système de se reconfigurer en cas de problème afin d'assurer au mieux les propriétés de sécurité-innocuité [51].

2.3.2 Mise à jour pour les systèmes distribués : cohérence globale

Certaines contraintes particulières sont à prendre en compte lors de la mise à jour de systèmes distribués. La première d'entre elles est liée à la cohérence globale. En effet, dans les systèmes distribués, il est possible qu'une mise à jour impacte plusieurs nœuds du système. Il est important dans ce cas de s'assurer que tous les éléments de la mise à jour sur les différents nœuds sont bien ajoutés au système ou bien qu'aucun d'entre eux ne l'est.

Une autre contrainte concerne l'état du système : comment gérer l'état courant du

1. `ftp://ftp.cordis.europa.eu/pub/ist/docs/dir_c/ems/dyscas-v1_en.pdf`

système ? L'état représente non seulement la valeur des variables (doit-on garder leur valeur actuelle ou bien les réinitialiser après la mise à jour ?), mais également les flots de contrôle (doit-on reprendre à zéro ou bien continuer avec les nouveaux éléments ?).

Enfin, un concept important qui peut être développé pour les mises à jour est celui de la *separation of concerns*, c'est-à-dire de séparer le logiciel permettant de faire des mises à jour de celui qui réalise effectivement les fonctionnalités.

2.3.2.1 Cohérence de la mise à jour

Les systèmes distribués peuvent comporter un grand nombre d'éléments, qui exécutent des programmes de manière concurrentielle. Dans un tel système, d'une part les modes de défaillance de chacun des éléments sont indépendants. D'autre part, la transmission de communications entre eux prend un temps non nul et il n'existe pas nécessairement d'horloge globale. Toutefois, l'exécution de certains programmes peut dépendre d'exécutions situées sur des nœuds distants (c'est-à-dire sur un élément différent et relié par un bus de communication). Un exemple de système distribué et temps-réel est le réseau de capteurs sur lequel de nombreuses études portent : par exemple, Han *et al.* proposent une étude des différentes techniques qui existent pour mettre à jour les réseaux de capteur [49]. Leur approche utilise un modèle d'architecture distribuée qui permet de créer et d'évaluer des outils de gestion logicielle pour les mises à jours de réseau de capteurs. En particulier l'évaluation des mécanismes pour la mise à jour repose sur trois critères principaux : l'environnement d'exécution des nœuds, le protocole de distribution des mises à jour et la compression des données transportées.

Un autre problème important qui peut se poser lorsque la mise à jour porte sur plusieurs éléments du système concerne la cohérence globale du système, qui doit être assurée. Il est donc important de garantir que la mise à jour est faite soit en totalité (tous les éléments concernés sont modifiés), soit pas du tout (aucun élément n'est modifié) afin de ne pas se trouver dans un état intermédiaire qui serait dommageable pour le bon fonctionnement du système. Ce concept a été développé par Gray qui définit qu'une transaction doit posséder les propriétés suivantes : être cohérente, atomique et durable [47].

Du point de vue du flot de contrôle, il faut également garantir une continuité dans l'exécution. En effet, il faut s'assurer par exemple que lorsqu'un service est demandé par un autre composant, ce service est bien fourni et une seule fois. Pour cela, Kramer et Magee [60] ont défini le concept d'état de *quiescence*, c'est-à-dire un état dans lequel le composant ne traite et n'émet pas de requêtes, et n'est sollicité par aucun autre élément du système. Lorsqu'un composant est dans cet état, il est alors possible de le mettre à jour sans porter préjudice au système puisque ce composant n'est pas sollicité et ne devrait pas l'être pendant la mise à jour. Toutefois, d'autres recherches ont montré que cet état de *quiescence* est trop restrictif et que les mises à jour peuvent avoir lieu à condition de s'assurer de la cohérence de l'exécution [20, 66, 103].

Par exemple, dans [66], les auteurs considèrent que la cohérence de version pour les transactions est un critère de sécurité-innocuité dans le cas de reconfiguration dynamique. Ce travail utilise les dépendances dynamiques à l'échelle globale, *i.e.* les contraintes

actuelles sur les composants, qui peuvent évoluer au cours du temps, afin de permettre plus de reconfigurations pour le système. Pour cela le concept de "cohérence de version" est définit pour les transactions qui implique qu'il n'existe pas un couple de sous-transactions (T1, T2) d'une même transaction, où T1 est exécuté sur un composant non mis à jour et où T2 sur un composant mis à jour. De cette manière ils définissent un état intermédiaire moins contraignant que l'état de *quiescence* mais plus contraignant que l'état de tranquilité [103].

André *et al.* proposent un framework pour prévoir le plan d'action qui permet de garder la cohérence du système [3]. Cette étape est extrêmement importante dans le cas des systèmes distribués, et ce tout particulièrement dans le cas de systèmes distribués de grande taille car un grand nombre d'éléments peuvent être impactés par la mise à jour. Il n'est toutefois pas forcément aisé de trouver un plan de reconfiguration ou de mise à jour qui soit optimal pour le système étant donné que de nombreux facteurs doivent être pris en compte pour y parvenir. Ainsi, Arshad *et al.* proposent un outil qui permet, à partir d'un modèle du système, de déterminer un plan de reconfiguration qui soit le meilleur possible (moindre nombre d'étapes ou temps de reconfiguration) [6].Pour cela ils ont développé un outil appelé *Planit* qui utilise la planification temporelle pour déterminer les déploiements possibles du système à partir d'un modèle.

Pour réaliser une mise à jour qui ne perturbe pas la cohérence du système, il faut non seulement déterminer le moment auquel la mise à jour peut être faite, mais également assurer la cohérence des flots de données et de contrôle, et enfin prévoir les actions qui permettent d'obtenir ce résultat. Par exemple, Banno *et al.* [20] proposent une méthode pour gérer la cohérence des mise à jour dans un système distribué en utilisant un outil appelé FREJA (Framework for Runtime Evolution of Java Applications). Le niveau de granularité pour les mises à jour est la classe et repose sur un *Update Controller* centralisé qui gère les mises à jour. Ce contrôleur délègue ensuite la gestion bas niveau des mises à jour à des *Update Clients* situés sur chacun des hôtes du système distribué. Enfin, un protocole spécifique est mis en place au sein de ce système afin d'assurer la cohérence globale. Ce dernier repose sur 4 étapes : 1/ vérifier qu'aucun élément pouvant invoquer le composant à mettre à jour n'est couramment exécuté, 2/ vérifier que le composant à mettre à jour lui-même n'est pas couramment exécuté et si c'est le cas bloquer l'accès au composant, 3/ vérifier si le composant à mettre à jour est inutilisé (étapes 1 et 2), procéder à la mise à jour, 4/ débloquer l'accès au composant.

2.3.2.2 État de la mise à jour

Ces systèmes présentent des caractéristiques spécifiques, qui rendent la mise à jour complexe : leur complexité architecturale, et pour certains, l'impossibilité de s'arrêter complètement pour la mise à jour [59]. En effet, éteindre le système conduit à un temps d'indisponibilité qui peut être considéré dans certains cas comme inacceptable. D'autre part, redémarrer complètement un système a pour conséquence la perte de son état actuel (par exemple si le système avait un processus non terminé, il faut reprendre là où il s'était arrêté lorsque cela est possible). Reconstruire cet état peut alors être long et complexe [42].

Pour gérer ce problème, il est donc nécessaire de s'assurer de la cohérence des flots de données et de contrôle. Concernant la cohérence des données : lorsqu'un composant est remplacé par un autre, il faut s'assurer que le nouveau composant a des données correctes [60]. Généralement, la solution la plus simple pour assurer cette cohérence est de copier les données vers le nouveau composant afin que ses données soient correctement initialisées [77].

Afin de réduire les contraintes importantes apportées par la notion de *quiescence*, Vandewoude *et al.* proposent un nouveau concept, à savoir celui de *tranquilité*, comme alternative [103]. Si une séquence d'actions initiée par un nœud du réseau est appelée "Transaction", il n'est pas nécessaire d'attendre la fin de cette dernière si le nœud qui doit être mis à jour ne lui fourni plus aucun service (même si ce nœud a été utilisé lors de la transaction). De plus, il est également possible de mettre à jour un nœud utilisé par une transaction si cette dernière a commencé mais n'a utilisé aucun service fourni par le nœud en question. Le nœud est alors dans un état de *Tranquilité* si : 1) il n'est engagé dans aucune transaction commencée, 2) il n'initie aucune nouvelle transaction, 3) il ne traite aucune demande ou 4) aucun nœud connexe n'est engagé dans une transaction dans laquelle le nœud qui doit être mis à jour a déjà été utilisé et doit à nouveau l'être. Cela nécessite d'avoir une hypothèse forte concernant la cascade de nœuds utilisés dans une transaction. Considérons trois nœuds N1, N2 et N3 tels que N1 débute une transaction. Lors de cette transaction une action est initiée par N1 sur N2 (A1) puis par N1 sur N3 (A2). N3 initie ensuite une action sur N2 (A3) puis la transaction se termine. Lorsque A1 est terminé, N2 peut alors être considéré comme étant tranquille ; cependant ce dernier est ensuite utilisé par N3 lors de la même transaction. Il faut donc être certain de l'indépendance entre A1 et A2 pour que la mise à jour sur N2 soit faite entre A1 et A2.

2.3.2.3 Séparation des éléments pour la mise à jour du reste du système : "separation of concern"

L'une des clefs de l'évolution des systèmes distribués réside dans la "separation of concerns", *i.e.* la séparation de la gestion des mises à jour et de la partie applicative qui réalise une fonction spécifique.

Un exemple de technique pour permettre les mises à jour graduelles d'un système distribué tout en limitant son indisponibilité est proposé dans [1]. Leur approche repose sur une infrastructure spécifique qui permet de mettre à jour les nœuds du système. Cette mise à jour est faite grâce à la "couche de mise à jour" intégrée dans chaque nœud et doit répondre aux exigences suivantes : modularité, généralité, conservation de l'état persistant, déploiement automatique et contrôlé, et continuité de service. L'inconvénient majeur de cette technique de mise à jour est qu'elle repose sur une architecture dédiée, incompatible avec l'architecture AUTOSAR actuelle et non prévue pour des systèmes embarqués.

Un autre travail sur les systèmes distribués est plus orienté sur la reconfiguration dynamique de systèmes distribués en utilisant Argus, un langage qui permet de construire des systèmes distribués plus fiables [28]. Le problème qui se pose également dans ce cas

est de pouvoir remplacer des parties de l'application tout en gardant un état cohérent et en minimisant les perturbations.

Dans [42], les auteurs proposent également une approche pour permettre la mise à jour dynamique de systèmes distribués présentant des ressources contraintes. L'objectif de ce travail est de présenter un outil qui permet une mise à jour des nœuds du réseau fondé sur le code binaire de l'application. Leur approche se soucie également de préserver l'état de l'application. Elle repose sur un nœud qu'ils nomment "manager" et qui dispose de ressources plus importantes que les autres nœuds du réseau. Il peut traiter les mises à jour avant de les envoyer au(x) nœud(s) concerné(s). Ce "manager" peut donc construire à partir du nouveau binaire et de l'ancien un ensemble de changements. Chaque élément de l'ensemble peut ensuite être intégré, au niveau du code binaire, dans chacun des nœuds du système distribué concerné par cette mise à jour.

2.3.3 Mise à jour pour les systèmes à composants

Il existe plusieurs types d'architectures qui ont pour vocation de permettre une adaptation plus facile des systèmes, telles que les architectures orientées services (SOA : Service-Oriented Architecture) [81]. Le principe de ce type d'architecture est de mettre en relation des éléments fournissant un ou plusieurs services et d'autres consommant des services. Ils reposent ensuite sur un *middleware* qui met en relation les différents éléments. Il conserve également l'indépendance de ces éléments en masquant l'identité des fournisseurs et des consommateurs. L'intérêt de ce type d'architecture est tout d'abord une meilleure réutilisabilité, mais également une meilleure interopérabilité des éléments car il est possible d'avoir des technologies différentes fournissant et utilisant les services.

Les systèmes à composants sont également une option architecturale qui repose sur l'indépendance entre les différents éléments et une meilleure réutilisabilité. Il existe de nombreuses définitions de composants, la plus communément reprise est celle de Szyperski [99] : *"A software component is a binary unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties"*.

Ainsi un composant doit pouvoir être développé par une personne différente de celle qui l'intègre. L'intégration correspond ensuite à un assemblage de composants afin de former une application. Un exemple de structure à composants logiciels est CORBA [76]. Ce dernier permet d'assembler des composants logiciels en utilisant des services d'un *middleware* (logiciel tiers qui gère les échanges entre les composants).

2.3.3.1 Architecture à composants et systèmes embarqués

Les systèmes à composants peuvent être utilisés pour des applications embarquées devant avoir un comportement temps-réel. Ces approches permettent un développement plus facile des applications en réutilisant des composants existants et en simplifiant l'architecture globale. C'est pour cette raison que l'architecture proposée par le standard AUTOSAR est elle aussi fondée sur une approche à composants. D'autres exemple de systèmes à composants existent également pour les systèmes embarqués temps-réel : par

exemple ROBOCOP [67] existe pour les produits électroniques grand public connectés, et offre une grande robustesse, Koala [102] pour les produits électroniques grand public (par exemple les télévisions) propose une meilleure flexibilité sans perdre en efficacité ou encore SaveCCM [50] pour les systèmes critiques de contrôle dans les véhicules.

Une étude récente par Hosek *et al.* [54] propose une comparaison des différentes structures à composants orientées systèmes embarqués temps-réel. Cette étude utilise les critères suivants :

— support de fonctionnalités avancées,
— existence d'outils,
— méthode,
— support de la reconfiguration dynamique et d'applications distribuées,
— documentations,
— statut de développement.

L'avantage apporté par les composants pour les systèmes embarqués réside principalement dans leur réutilisabilité qui permet de réduire les coûts de développement, les délais de mise sur le marché et permettent d'avoir des systèmes distribués à grande échelle ainsi que des productions de masse. De plus, cela permet d'améliorer la qualité du logiciel en utilisant des composants déjà testés et utilisés dans des systèmes, et qui ont donc fait leurs preuves. Enfin la taille des composants étant réduite (comparée à une application dans son intégralité), cela facilite les tests unitaires [104].

De nombreux problèmes se posent lorsque les approches à composants sont utilisées [4]. L'un de ces problèmes est le développement d'une architecture logicielle qui prend en compte la nature des composants et leur composition. Cette architecture doit permettre de

— spécifier la structure du système,
— spécifier son comportement,
— vérifier l'ordonnancement,
— vérifier l'exécution des composants,
— spécifier les composants logiciels selon des critères de décomposition systématique et qui ne dépendent pas d'une application particulière, pour améliorer la réutilisabilité.

2.3.3.2 Architectures et mises à jour

OSGi (Open Service Gateway initiative) est une plateforme utilisant le langage JAVA, qui permet de gérer de manière modulaire les éléments qui constituent une application afin de permettre une reconfiguration sans arrêt de service [95]. Chen *et al.* ont proposé une approche pour permettre des mises à jour dynamiques de services [35]. Elle est fondée sur OSGi et utilise une méthode de spécification des mises à jour des services reposant sur l'utilisation des automates finis ainsi qu'un outil pour faire des vérifications formelles de certaines propriétés.

Li *et al.*, dans [63] ont proposé une adaptation de cette plateforme dans un contexte automobile. Son objectif est de permettre une meilleure flexibilité des services télématiques embarqués dans les voitures, mais ne concerne donc pas les calculateurs enfouis

dans le véhicule.

Enfin, Concierge [89] propose une version de OSGi conçue pour les systèmes embarqués. Son objectif est de prendre en compte au mieux les contraintes sur les ressources de ce type de système en réduisant l'empreinte mémoire et en améliorant les performances de la plateforme.

L'inconvénient majeur de ces approche fondées sur OSGi dans notre contexte est leur incompatibilité avec AUTOSAR car elle repose sur une architecture à composants et des interfaces spécifiques qui sont différentes.

D'autres approches pour la mise à jour dans les systèmes à composants existent. Par exemple *Gravity* [34], proposé par Cervantes *et al.*, présente une architecture à composants qui interagissent via des services. Cela permet alors d'introduire le concept des architectures orientées service dans un modèle à composants, qui doivent alors être capable de s'adapter dynamiquement, pendant leur exécution, à l'apparition de nouveaux services lors de l'ajout de composants. L'objectif ici est de pouvoir s'adapter facilement à l'ajout ou la suppression de composants pendant l'exécution de l'application. L'approche proposée ici repose sur une architecture spécifique qui n'est pas nécessairement adaptée pour les systèmes embarqués.

Plasil *et al.* ont mis au point une architecture à composants appelée SOFA (SOFtware Appliance) et une extension à cette architecture, à savoir DCUP (Dynamic Component UPdating), qui permet de faire des mises à jour ou des échanges de composants de manière transparente pour le système [86]. Leur approche offre des réponses intéressantes aux questions plus générales de gestion de version dans l'application (comment savoir dans quel contexte la mise à jour est intégrée, comment lier clairement une mise à jour à une version). Ils offrent également des réponses concernant la gestion de l'état d'un composant au moment de la mise à jour et de la transparence de cette dernière pour l'utilisateur final.

Il est également important de prendre en compte le fait que ces systèmes qui sont soumis à évolution peuvent être critiques. Dans ce cas, il peut être intéressant de faire évoluer les mécanismes de sécurité en même temps que le système lui même. Dans le cas des systèmes à composants, Stoicescu *et al.* proposent une approche pour rendre les mécanismes de tolérance aux fautes évolutifs afin de permettre des évolutions à grains fins dans l'architecture [97].

Des travaux précédents ont traité la question de la mise à jour applicative dans AUTOSAR. Par exemple, Axelsson et Kobetski [18] proposent d'encapsuler une machine virtuelle JAVA dans un composant logiciel. Cette machine virtuelle pourrait ensuite recevoir les différents modules et les exécuter. Ainsi, les mises à jour seraient séparées du système de base. Cependant cette approche demande d'avoir une puissance de calcul bien supérieure à celle disponible sur les calculateurs embarqués traditionnellement utilisés dans les voitures. Elle semble donc compliquée à adapter à un vrai système embarqué automobile. D'autre part, en utilisant cette approche, l'architecture AUTOSAR elle-même n'est plus respectée puisque les mises à jour sont séparées du système principal et mise dans une architecture à part.

2.4 Classification des résultats

Nous avons présenté ici un certain nombre de résultats concernant les approches pour la mise à jour dans différents systèmes. Le positionnement de ces approches est résumé dans la table 2.1. Bien que certains résultats portent également sur la mise à jour off-line ([24] [32] [6]), ce tableau présente uniquement les différents résultats concernant la mise à jour en ligne. Nous pouvons voir ici que la majorité des résultats existants ne se positionnent pas nécessairement sur le domaine de l'automobile. D'autre part, beaucoup d'entre eux sont pour des systèmes distribués qui ne sont pas forcément adaptés dans le contexte embarqué qui est celui de l'automobile.

	Automobile		Autre	
	Embarqué	Distribué	Embarqué	Distribué
Architecture ou Modèle	[63] [18]	[94] [63]	[89]	[20] [103] [1] [35] [34] [86]
Bas Niveau	[73] [43]		[49]	[42] [49]
Reconfiguration	[100] [51] [43]			[28]

TABLE 2.1 – Positionnement des travaux présentés dans l'état de l'art

2.5 Architecture logicielle embarquée automobile

Les voitures se modernisent de plus en plus, si bien qu'il est maintenant possible de trouver plus de 60 calculateurs embarqués dans certains modèles de voitures. Il y a donc eu une augmentation importante de la quantité de logiciel embarqué dans les voitures au cours de ces 15 dernières années. Une quantité croissante de fonctionnalités est à présent réalisée en utilisant du logiciel et de nombreuses innovations actuelles (jusqu'à 90% [30]) sont possibles grâce à celui-ci.

De plus, les véhicules sont maintenant de plus en plus connectés (les communications *car-to-car* [33] ou *car-to-infrastructure* [90] sont des sujets auxquels les industriels s'intéressent de près). Cette ouverture des véhicules vers leur environnement peut à la fois permettre des évolutions plus faciles pour ces systèmes mais également poser des problèmes liés à la sécurité-immunité [74, 75, 53].

La complexité des systèmes embarqués évoluant de manière croissante, de nombreuses architectures logicielles sont apparues dans les différents domaines (par exemple AUTO-

SAR [16] pour l'automobile ou ARINC 653 [87] dans le domaine aéronautique) afin de rendre cette complexité plus simple à gérer et de permettre une meilleure compatibilité entre des parties de logiciel développées par différentes entreprises. De plus, cela permet d'assurer une cohérence globale. Ainsi, lors de la conception d'un système embarqué, il faut se plier à un certain nombre de standards pour respecter l'architecture logicielle correspondante.

Le logiciel embarqué automobile se divise en deux domaines. Le premier est lié à *l'infotainment*, c'est-à-dire les systèmes multimédias et l'autre corresponds aux calculateurs enfouis qui permettent de réaliser des fonctions qui ne sont pas nécessairement visibles pour l'utilisateur, comme les contrôles moteurs. Dans chacun de ces domaines, les procédés, architectures et méthodes de développement sont distinctes. Dans le cadre de cette thèse, nous nous intéressons principalement aux calculateurs enfouis. Ces derniers utilisent majoritairement une architecture de type AUTOSAR.

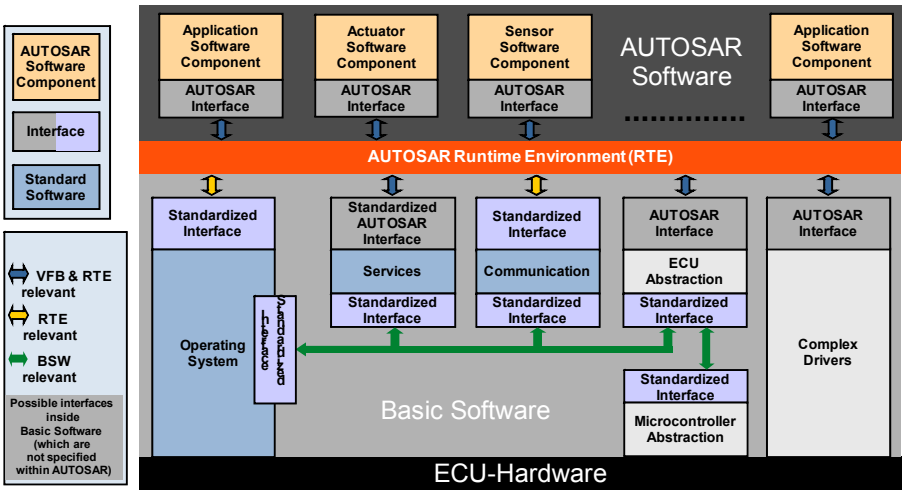
Cette augmentation massive du logiciel crée néanmoins des problèmes qui n'existaient pas avant. Par exemple, des fonctionnalités totalement décorrélées peuvent désormais se retrouver sur le même calculateur et interférer les unes avec les autres [31]. Dans ce contexte, la vérification et la validation du logiciel, sa maintenance ainsi que sa réutilisabilité posent donc des problèmes majeurs.

Historiquement, les calculateurs embarqués étaient développés de manière *ad hoc*, c'est-à-dire que le logiciel associé à chaque calculateur était développé spécifiquement à chaque fois. Ceci était possible car peu de logiciel était embarqué dans les véhicules. Cependant ce procédé rendait très difficile la réutilisation du logiciel, ainsi que sa modification, sa maintenance ou encore l'intégration de parties de logiciel issues de différents fournisseurs. D'autre part, dans le cas d'un logiciel *ad hoc*, il est difficile de changer d'architecture matérielle. Ainsi, pour des raisons de simplification et afin de résoudre ces problèmes, les différents acteurs de l'industrie automobile ont créé le standard AUTOSAR (AUtomotive Open System ARchitecture).

AUTOSAR présente une architecture en couche qui permet d'abstraire le hardware. Il s'agit d'une architecture modulaire facilitant la construction du logiciel à partir de briques élémentaires. L'objectif du standard est de réduire les coûts de développement logiciel. En effet, les composants applicatifs peuvent être réutilisés facilement et être portés d'une plateforme à une autre. Il est également possible d'utiliser des COTS (Commercial Off-The-Shelf) [69] et de simplifier l'architecture matérielle en mixant des fonctions sur des calculateurs génériques.

Dans le standard AUTOSAR, un ECU (Electronic Control Unit) présente une architecture logicielle divisée en quatre couches, comme montré par la figure 2.1. La couche la plus basse correspond au matériel. Au dessus du matériel se trouve le logiciel de base, également appelé couches basses (*BSW ou Basic Software* [9]). La couche la plus haute, quant à elle, est la couche applicative qui contient les différents composants logiciels (*SWC*) réalisant les fonctions (*AUTOSAR* Software [10]). Enfin, entre la couche applicative et le logiciel de base se trouve le middleware spécifique à AUTOSAR appelé *Run-Time Environment* (*RTE*) [12].

Une présentation plus détaillée des couches logicielles propres à AUTOSAR ainsi que



Note: This figure is incomplete with respect to the possible interactions between the layers.

FIGURE 2.1 – Détails de l'architecture en couche AUTOSAR (EXtraite des spécifications AUTOSAR [13]).

de la couche applicative plus particulièrement peut être trouvée dans [107].

2.5.1 Software component et *runnable*

Il existe différents types de composants logiciels (*SWC* ou Software Component). Dans la version 4.0 du standard AUTOSAR ils sont au nombre de 8 : Application, Sensor/Actuator, Parameter, ServiceProxy, Service, ECU-abstraction, Complex Device Driver, NV Block. Nous nous intéressons ici uniquement aux SWC de type Application.

Les Software Components correspondent au logiciel applicatif, c'est-à-dire à la partie de l'architecture qui se trouve au dessus du RTE. Un SWC peut être vu comme un ensemble de *runnables* (fragment de code "élémentaire"), de ports d'entrée/sortie qui permettent la communication avec les autres SWC via le RTE et d'IRV (Inter-Runnable Variables) qui permettent aux runnables d'un même SWC de communiquer entre eux. Il définit également les évènements qui permettent aux runnables d'être déclenchés (*RTEEvents*) ainsi que les *Exclusive Areas* (qui permettent de gérer les accès concurrents de données). Le SWC peut être vu comme un regroupement structurel qui n'a pas d'existence propre dans l'implémentation finale.

Un SWC peut soit être atomique, c'est-à-dire qu'il ne peut pas être subdivisé en Software Components plus petits et doit par conséquent être assigné sur un seul ECU, soit être une composition, c'est-à-dire un regroupement de SWC connectés entre eux. Notons que dans le cas d'une composition, celle-ci ne peut contenir que des SWC et pas de *runnables* propres.

2.5.1.1 Runnable

Le SWC est divisé en plusieurs fragments de code élémentaire. Chaque fragment de code est appelé runnable. Un runnable peut être défini comme une séquence d'instruction qui peut être déclenché par le RTE. Ils doivent ensuite être alloués dans des tâches du systèmes d'exploitation afin d'être exécutés. En effet, les SWCs correspondent à une décomposition hiérarchique de runnables qui n'ont pas d'existence propre dans l'implémentation finale. Toutefois ce sont des éléments nécessaires pour l'architecture du logiciel. La répartition des runnables dans les tâches du système d'exploitation est totalement indépendante de leur répartition dans les SWC. Ainsi, un runnable peut être soit périodique, soit non périodique. Un runnable peut soit s'exécuter lors de la réception d'un évènement soit attendre un évènement pour poursuivre son exécution (Wait Point).

Les runnables sont classés en 3 catégories : 1A, 1B et 2 en fonction de la façon dont les données sont traitées et de la présence ou de l'absence de "Wait Point", c'est-à-dire de point d'attente d'un évènement. Lorsque les runnables sont alloués sur les tâches qui sont le support de leur exécution, le type de tâche dans lequel le runnable peut être exécuté dépend de sa catégorie. En effet, les runnables de catégorie 1 ne peuvent pas attendre d'évènement, alors que les runnables de catégorie 2 le peuvent. La sous division de la catégorie 1 est liée au mode de communication choisi pour les données : *implicite* ou *explicite*. Dans le mode de communication *implicite*, toutes les données sont lues au début de l'exécution et écrites à la fin. Les runnables utilisant ce mode de communication sont de type 1A. Dans le mode de communication *explicite* les données sont lues et écrites au fur et à mesure de l'exécution. Les runnables qui utilisent ce mode de communication sont de type 1B. Les runnables de catégorie 2 utilisent exclusivement un mode de communication explicite.

2.5.1.2 Ports

Les ports d'un SWC sont ce qui lui permet de communiquer avec le reste de l'application ou avec le BSW via le RTE. Un port n'appartient qu'à un seul SWC à la fois et ne peut pas recevoir et envoyer en même temps. Chaque port doit être associé à une interface qui définit la nature des informations véhiculées entre deux ports. Une interface peut être considérée comme un "type" pour le port. Un port ne peut être associé qu'à une seule interface. Cette dernière décrit également les opérations ou données associées.

2.5.1.3 Inter-Runnable Variable (IRV)

Les runnables d'un même SWC n'utilisent pas nécessairement le RTE (dans la version 4 d'AUTOSAR), et aucunement les ports pour communiquer. En effet, des données peuvent être réservées pour permettre la communication, indépendamment des tâches sur lesquelles ces runnables sont alloués. Ces données (IRV) ne peuvent êtres lues ou écrites que par les runnables appartenant à un SWC donné. Les données sont écrites par un runnable et lues par un autre.

L'utilisation d'IRV permet une encapsulation des données dans la mesure où les autres SWC ne peuvent pas y accéder.

2.5.2 Run-Time Environment (RTE) et Virtual Functional Bus (VFB)

Dans AUTOSAR, une application est modélisée par une composition de SWCs interconnectés [7]. Le RTE est l'interface concrète qui permet aux différents SWCs de communiquer non seulement entre eux mais également avec le Basic Software.

Le VFB(Virtual Functional Bus) permet la conception des fonctions indépendamment du matériel ou de l'allocation fonctionnelle entre les calculateurs : il réalise l'abstraction des interconnexions entre les différents composants logiciels. L'existence du VFB est purement conceptuelle. Le RTE est l'implémentation concrète du VFB au niveau de l'ECU. La figure 2.2 illustre, sur un exemple extrait des spécifications AUTOSAR 4.0, la différence entre le niveau VFB et le niveau RTE. Ainsi, la partie haute de cette figure montre différents composants qui communiquent via le VFB il s'agit d'une vue architecturale qui montre les différents SWCs présents au final dans le système. Ensuite ces SWCs sont alloués sur les différents ECUs disponibles au cours du processus de développement. Le RTE de chaque ECU doit ensuite être généré. Quel que soit l'emplacement physique des SWCs, la communication doit se faire de manière transparente pour ces derniers : le SWC ne sait pas où se situe physiquement la source (ou la destination) des messages qu'il reçoit (ou envoie).

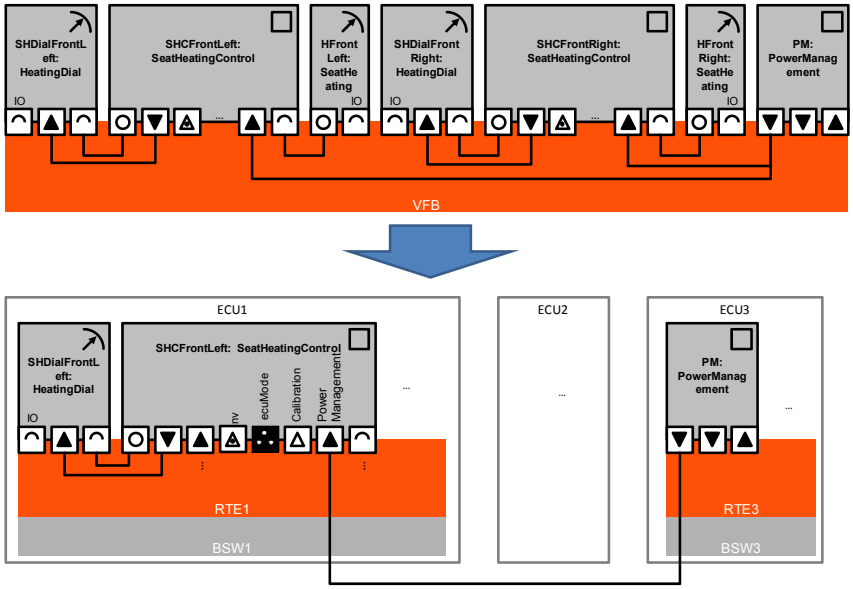


FIGURE 2.2 – Exemple d'allocation de software components sur plusieurs ECU. Visualisation des niveaux VFB et RTE (extrait des spécifications AUTOSAR 4.0 [13]

Le RTE correspond à un code de "glue" qui est souvent généré automatiquement en fonction des spécifications de conception de l'application. Il implémente les besoins en communication des composants, les évènements qui déclenchent l'exécution des runnables et définit le contenu des tâches applicatives. La cohérence globale du RTE doit

être assurée par les outils de génération. La fonction du RTE est de permettre la communication entre les différents SWCs ainsi qu'avec le Basic Software.

Le RTE permet également de déclencher l'exécution des runnables grâce à des évènements ; il dépend de l'ECU sur lequel il est implémenté et n'est donc pas générique que ce soit du point de vue fonctionnel ou du point de vue de la portabilité.

2.5.3 Basic Software

La figure 2.1 montre que le Basic Software contient de nombreuses briques logicielles. Cependant, dans cette section nous ne détaillons que celles qui sont les plus importantes du point de vue de la mise à jour dynamique dans les systèmes embarqués automobiles. Dans le cas particulier de ce travail, nous nous intéressons plus en détail au système d'exploitation, communément abrégé "OS" (pour Operating System) ainsi qu'à la pile de communication. Le premier des deux éléments, l'OS, gère l'exécution des runnables. En effet, c'est lui qui exécute, entre autre, des tâches qui contiennent elles-mêmes les runnables. La pile de communication, quant à elle, gère les flux entrants et sortants sur les différents bus logiciels disponibles, et par conséquent la communication entre l'ECU et son environnement.

2.5.3.1 Système d'exploitation (OS)

Afin de garantir les propriétés temps-réel du système embarqué automobile dans le cadre d'une architecture AUTOSAR, le standard définit un système d'exploitation associé appelé AUTOSAR OS [11]. Il est fondé sur un standard précédent appelé OSEK [78].

Un OS de type AUTOSAR doit posséder un ordonnancement à priorité fixe (chaque tâche possède une priorité qui ne varie pas au cours de l'exécution), offrir une protection contre un mauvais usage des services de l'OS et être capable de gérer les interruptions. Ces dernières doivent avoir des priorités supérieures à celles des tâches. Ces caractéristiques principales de l'OS associé au standard sont largement héritées de OSEK.

Dans le modèle architectural proposé par AUTOSAR, l'implémentation d'un composant logiciel se fait par l'intermédiaire de l'allocation de chacun des runnables qui le composent sur une ou plusieurs tâches de l'OS.

Le système d'exploitation temps-réel qui nous intéresse ici dispose de deux types de tâches, à savoir les tâches basiques et les tâches étendues que nous détaillons ici.

Modèles de tâches La différence principale entre les deux types de tâches peut se résumer par le fait que les tâches **étendues** peuvent être dans un état d'attente d'un évènement extérieur alors que les tâche **basiques** ne le peuvent pas. Ainsi, la synchronisation entre une tâche basique et son environnement ne se fait qu'au début et à la fin de son exécution. Alors que les tâches basiques consomment peu de mémoire, les tâches étendues apportent une meilleure cohérence (en ayant plus de points de synchronisation).

Ainsi, il n'existe que trois états possibles pour les tâches basiques : *running*, lorsque la tâche est en train de s'exécuter, *ready*, lorsque la tâche est prête à être exécutée et *suspended*, lorsque la tâche est passive. Les tâches étendues quant à elles peuvent

atteindre un quatrième état : *waiting*, lorsque la tâche attend un évènement pour pouvoir poursuivre son exécution. Notons que lorsqu'une tâche étendue est dans l'état d'attente d'un évènement elle n'empêche pas l'exécution des autres tâches. La figure 2.3 montre les différents états possibles pour chacun des types de tâches, ainsi que les transitions possibles entre ces états.

— Lorsqu'une tâche est dans l'état *suspended* et qu'elle est activée (`activate`) parce qu'elle doit être exécutée, elle passe alors dans l'état *ready*.
— Lorsque la tâche est dans l'état *ready* et que les ressources nécessaires sont libres pour l'exécuter, elle peut alors être exécutée (`start`) et passer dans l'état *running*.
— Lorsque la tâche est en cours d'exécution, elle peut soit terminer son exécution (`terminate`), soit être préemptée (`preempt`) si une interruption est activée ou bien si un tâche de priorité supérieure est prête à être exécutée, soit, s'il s'agit d'une tâche étendue attendre un évènement (`wait`).
— Lorsque la tâche reçoit l'évènement qu'elle attendait, elle passe alors à nouveau dans l'état *ready* pour poursuivre son exécution (`release`).

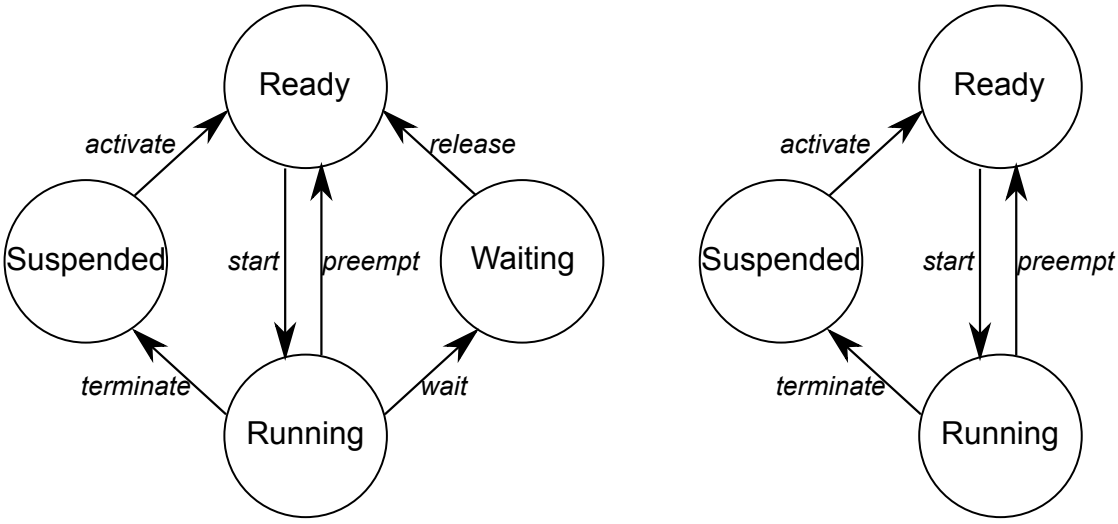


FIGURE 2.3 – Modèles de tâches a) Étendue (gauche) b) Basique (droite)

OS-Application L'une des différences principales dans AUTOSAR OS par rapport à OSEK réside dans le fait que les objets de l'OS (tâches, ISR, alarmes, évènements, *schedule table*, ressources) n'ont pas de propriétaire. Ainsi tout objet A peut manipuler tout autre objet B, d'où il peut résulter des propagations de fautes importantes. C'est pour cette raison que le concept d'OS-Application a été défini dans AUTOSAR.

Une OS-Application correspond à un groupe d'éléments de l'OS (tâches, ressources, ...) qui ne peuvent accéder qu'aux éléments appartenant au même groupe. Elle permet d'avoir des mécanismes de protection ciblés pour certains groupes d'éléments. Ceci est particulièrement intéressant dans le cadre de la protection pour des éléments ayant

des niveaux de criticités différentes : une solution est donc de considérer que seuls des éléments ayant un niveau de criticité identique peuvent être mis dans un même groupe.

2.5.3.2 Pile de communication CAN

La pile de communication dans le Basic Software AUTOSAR gère les messages en émission ou bien en réception des différents ECUs via différents protocoles. La figure 2.4 montre le positionnement d'une pile de communication dans le Basic Software de l'architecture AUTOSAR. Cette dernière s'étend entre le RTE et le matériel, et est sous-divisée en différentes couches en fonction de son niveau d'abstraction. Chaque niveau est réalisé par un certain nombre de modules.

Il est à noter que cette architecture doit être répétée pour chacun des protocoles de communication existant (CAN, FlexRay, Ethernet,...). Chaque protocole implémente ensuite de manière spécifique chacun des éléments. Nous ne détaillons ici que la pile de communication CAN car c'est celle qui nous intéresse dans le cadre de ce travail.

Cette pile est sous divisée en 3 éléments (voir figure 2.4). Tout en haut (au plus près du RTE) se trouvent les services de communication qui gèrent le réseau de communication. En dessous se situe la couche d'abstraction de l'ECU qui permet de faire l'interface avec les pilotes. Elle fournit également l'API pour accéder aux périphériques indépendamment de leur emplacement. Enfin, le dernier élément est la MCAL (Microcontrolleur Abstraction Layer) qui contient les pilotes internes, c'est-à-dire les modules logiciels qui permettent d'accéder au matériel (en l'occurrence le bus de communication).

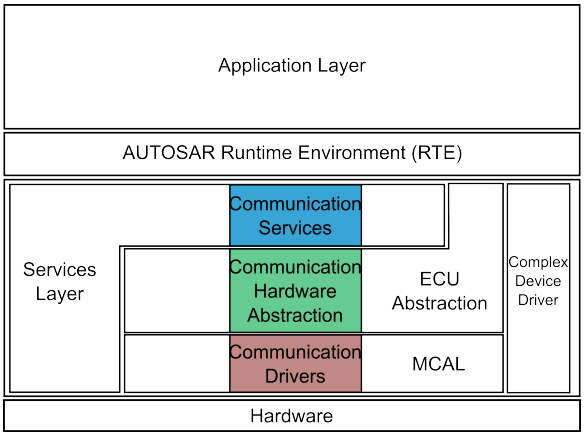


FIGURE 2.4 – Schéma d'une pile de communication AUTOSAR

L'impact de l'adaptation des runnables dans le réseau n'est pas étudié dans cette thèse.

2.6 Synthèse

Nous nous intéressons ici à des systèmes embarqués automobiles. Les systèmes sur lesquels nous travaillons sont de type temps-réel strict. En effet, ils doivent répondre rapidement à des simulations extérieures, comme par exemple celle du conducteur. D'autre part les messages envoyés ne doivent pas être perdus. Par exemple, lorsque le frein de la voiture est actionné, il n'est pas acceptable que la demande ne soit pas traitée parce que le paquet a été perdu, ou bien qu'elle soit traitée trop tard. Dans ce cas les conséquences peuvent être catastrophiques.

Les systèmes automobiles enfouis, qui sont des systèmes embarqués et distribués, doivent répondre à des problèmes de disponibilité et ont des ressources limitées. Pour ces raisons, mettre à jour ces systèmes est un problème non trivial. De plus il faut se conformer au standard architectural AUTOSAR. Il s'agit d'une architecture à composant qui a été développée pour faciliter la réutilisation du logiciel.

Dans ce contexte, nous souhaitons faire des mises à jour applicatisve du logiciel automobile. Bien que la version 4 du standard AUTOSAR spécifie les caractéristiques nécessaires pour utiliser des calculateurs multicœurs, nous ne nous intéressons dans le cadre de ce travail que au cas des calculateurs monocœurs.

De plus, nous considérons que les modifications faites sur le système sont uniquement de type logiciel. Ainsi, même s'il pourrait être intéressant d'ajouter des capteurs d'une part, puis le logiciel permettant d'utiliser ces capteurs d'autre part, cela représente une quantité trop importante de modifications pour utiliser des mises à jour partielles.

Il est important de noter que les mises à jour du système doivent avoir une taille bien inférieure à celle du code total embarqué dans le calculateur. Typiquement le code d'un calculateur représente quelques Mo, et une mise à jour ne doit pas excéder quelques dizaines de Ko. Dans le cas où la mise à jour est très volumineuse, et par conséquent que les modifications du système sont très importantes, il est plus efficace de recharger complètement le calculateur.

3

Architecture du système pour l'adaptation

Sommaire

3.1 Introduction

Le développement d'applications embarquées dans un contexte automobile AUTOSAR repose grandement sur une chaine outillée [106]. De nombreuses étapes sont nécessaires pour parvenir au binaire final tout en respectant l'architecture en couches présentée dans la section 2.5 du chapitre 2.

Le problème qui se pose ici dans le cadre de la mise à jour de systèmes embarqués de type AUTOSAR est principalement lié au manque de flexibilité. En effet, dans un processus de développement standard, après la génération du code, il est presque impossible de faire des modifications ultérieures, à moins de recharger intégralement le logiciel dans le calculateur. Ainsi, notre objectif est de proposer une méthode pour permettre des mises à jour partielles dans un calculateur AUTOSAR. Il est donc souhaitable de faire des modifications de manière différentielle, en ajoutant uniquement les nouveaux éléments ou en modifiant seulement des parties de logiciel. Sachant que nous souhaitons conserver et respecter l'architecture AUTOSAR, la solution proposée consiste à ajouter à l'avance dans l'architecture logicielle des espaces libres qui permettent de loger de nouvelles fonctionnalités. Ce chapitre détaille comment sont définis ces "espaces" et comment les intégrer dans l'architecture logicielle, ainsi que dans le processus de développement.

En premier lieu les concepts qui améliorent la flexibilité dans une architecture de type AUTOSAR sont présentés. Les deux concepts clefs pour la modification et l'ajout de fonctionnalités dans les applications embarqués AUTOSAR sont les "*Espaces d'Adaptation*" et les "*Containers*". Ces notions définissent des espaces dans lesquelles les mises à jour peuvent être placées.

Il faut ensuite intégrer ces concepts au processus de développement. Dans un premier temps, nous décrivons le processus de développement d'une application automobile embarquée dans le contexte d'AUTOSAR. Ce processus est celui utilisé au sein de l'entreprise Renault, et pour des raisons de confidentialité certaines parties ne peuvent pas être détaillées de manière spécifique. Cependant, l'essentiel nécessaire à la compréhension du processus de développement est développé et présenté ici. Bien que cette partie se repose sur la méthodologie AUTOSAR [15], certains points ne sont pas clairement explicités. Nous développons donc dans cette section quelles sont les ajouts et contributions spécifiques qui ont été apportées au sein de l'entreprise.

Enfin la troisième partie de ce chapitre présente les modifications qui doivent être faites sur le processus initial de développement d'application embarquée afin d'intégrer les espaces d'adaptation nécessaires ainsi que les modules de gestion des mises à jour.

3.2 *Espace d'adaptation* et *container*

L'une des hypothèses prise pour définir ces notions est que la granularité des mises à jour est le runnable. En effet, comme expliqué dans la section 2.5.1.1, un SWC est une décomposition hiérarchique qui contient un certain nombre de runnables. Ce sont ces runnables qui réalisent effectivement une partie de nos chaînes de calcul. Les SWCs n'ont aucune existence propre en tant que tel dans l'implémentation finale.

Étant donné que nous nous plaçons au niveau implémentation final pour ajouter la mise à jour, nous partons du principe que s'il est possible d'ajouter dans un système tous les runnables d'un SWC, cela revient à ajouter un SWC. Il est à noter toutefois, que bien que l'intégration se fasse au niveau du code, le travail architectural préparatoire est nécessaire pour déterminer le contexte dans lequel la mise à jour s'intègre. Nous nous concentrons donc dans ce travail sur les mécanismes permettant d'ajouter un runnable. Notons que ces mécanismes peuvent être actionnés plusieurs fois de suite afin d'ajouter un SWC complet.

3.2.1 Différentes vues d'une application

Les mises à jour doivent être intégrées dans une application existante de manière la plus transparente possible. Pour cette raison, nous présentons ici les différentes vues d'une application, dans le contexte d'une modélisation AUTOSAR.

Les différentes caractéristiques qui sont présentées dans cette section sont la base de la définition de nos concepts d'espace d'adaptation et de *container*.

3.2.1.1 Vue haut niveau

Cette vue correspond au niveau applicatif d'une application sur un ECU. Le tableau 3.1 montre les différentes caractéristiques que nous considérons pour le logiciel applicatif dans le cadre d'un ECU. Les éléments qui nous intéressent sont :

- les runnables (puisque ce sont eux qui réalisent la fonctionnalité), qui possèdent eux-mêmes un certain nombre de caractéristiques que nous détaillons ultérieurement,
- les tâches qui exécutent les runnables,
- les connexions internes (*i.e.* celles qui passent uniquement par le RTE) et
- les données d'entrées sorties (qui transitent par un bus matériel). Ces deux types de données sont différenciés car ils mettent en œuvre des mécanismes différents, et ne sont pas traités de la même manière lors des mises à jour.

Il est important de noter que les tâches qui sont considérées dans ce cas précis sont uniquement les tâches qui hébergent le logiciel applicatif (les runnables). Il existe également dans le système d'autres tâches qui permettent aux éléments du Basic Software d'être exécutés, mais elles ne sont pas prises en compte ici. Elles doivent toutefois être prises en compte pour l'analyse d'ordonnancement.

	Détails	Commentaires
Tâches	Entités exécutables et ordonnançables	Tâches et leurs caractéristiques
Runnables	Unités fonctionnelles d'exécution	Runnables et leurs caractéristiques
Connexions Internes	Communications entre les différents SWCs	Communications utilisant le RTE.
Données [IN/OUT]	Données utilisant la pile de communication	Communications avec des SWCs situés sur d'autres ECUs

Table 3.1 – Caractéristiques d'une application AUTOSAR

3.2.1.2 Vue niveau runnable

Le niveau Runnable correspond aux caractéristiques spécifiques de chacun de ces éléments. Ce qui nous intéresse plus particulièrement ici sont les caractéristiques présentées dans le tableau 3.2, à savoir, le mode d'activation, les données et la catégorie. Le mode d'activation ainsi que la catégorie du runnable permettent de déterminer en partie les caractéristiques requises pour les tâches du système. En effet, les tâches applicatives qui sont présentes dans le système ont uniquement pour objectif d'exécuter les runnables, et par conséquent, les caractéristiques de ces dernières doivent être déterminées de manière à répondre aux besoins.

	Détails	Commentaires
Mode d'Activation	Périodique / Sporadique	Un runnable peut être périodique ou déclenché par évènement. Il faut préciser la période/l'évènement associé.
Données	Données reçues et émises par le runnable et mode correspondant (implicite/explicite)	Permet de gérer les communications et de déterminer la catégorie du runnable.
Catégorie	1A/1B : Pas de point d'attente - Communication Implicite/Explicite 2 : Présence de point d'attente	Un point d'attente correspond à l'attente d'un évènement pour poursuivre l'exécution

Table 3.2 – Caractéristiques d'un runnable

3.2.1.3 Vue niveau implémentation

Lorsque l'application est définie, il faut la charger dans une cible embarquée. Pour cela, il est nécessaire de prendre en compte le support d'exécution. Ce support d'exécution revêt deux aspects : d'une part les tâches qui exécutent le logiciel applicatif, et d'autre part les caractéristiques intrinsèques liées à la cible, à savoir l'espace mémoire

utilisé ainsi que la charge CPU totale.

Le tableau 3.3 présente les caractéristiques pour une tâche du système d'exploitation.

	Détails	**Commentaires**
Type	Basique / Étendu	Seules les tâches étendues peuvent attendre des évènements (point d'attente)
Mode d'activation	Périodique ou évènementielle	Les tâches périodiques sont déclenchées par des alarmes et les évènementielles par l'arrivée d'un évènement
Préemptive	Full-préemptive Non-préemptive	Suspendue par l'arrivée d'une tâche ayant une priorité supérieure ou d'une interruption Suspendue uniquement par une interruption
Priorité		Plus le nombre est grand plus la priorité est grande
Trigger	Alarme Évènements	Tâches périodiques Tâches sporadiques

TABLE 3.3 – Caractéristiques d'une tâche AUTOSAR OS

3.2.2 *Espace d'adaptation*

Un espace d'adaptation correspond aux définitions de tous les paramètres possibles pour qu'un runnable puisse être ajouté dans le système. Cette définition correspond à une approche dite *pré-câblée* pour laquelle le budget temporel (temps disponible pour l'exécution) alloué à une mise à jour doit être défini par avance. Cette contrainte peut être relâchée si l'approche choisie est une approche dite *opportuniste* dans laquelle les mises à jour utilisent le temps libre restant dans le système. Une vérification de l'ordonnançabilité du système peut alors être faite de manière simplifiée (nous détaillons les concepts liés à l'ordonnancement dans le chapitre 4).

La figure 3.1 montre un exemple d'espace d'adaptation pour un runnable. Les caractéristiques qui sont considérées ici sont de natures différentes. D'une part, les axes "activation mode" et "category" (en vert) représentent des caractéristiques de niveau architectural. Ces caractéristiques sont extraites de la "vue runnable" (Tableau 3.2). D'autre part, les axes "trigger" et "priority" (en rouge) correspondent à des caractéristiques d'implémentation extraites de la "vue implémentation" (Tableau 3.3). Le budget temps alloué est rajouté ici, mais correspond néanmoins à une caractéristique d'implémentation bien qu'elle n'apparaisse pas directement dans les caractéristiques des tâches (en effet, le temps d'exécution d'une tâche n'est pas une caractéristique *a priori*).

Prendre des valeurs sur chacun des axes définit le contour d'un espace d'adaptation ayant des caractéristiques données. Cet espace d'adaptation est ensuite matérialisé dans l'application sous la forme d'un *container*.

Les communications n'apparaissent pas sur cette définition d'espace d'adaptation. Il s'agit toutefois d'une notion qu'il est important de prendre en compte par ailleurs

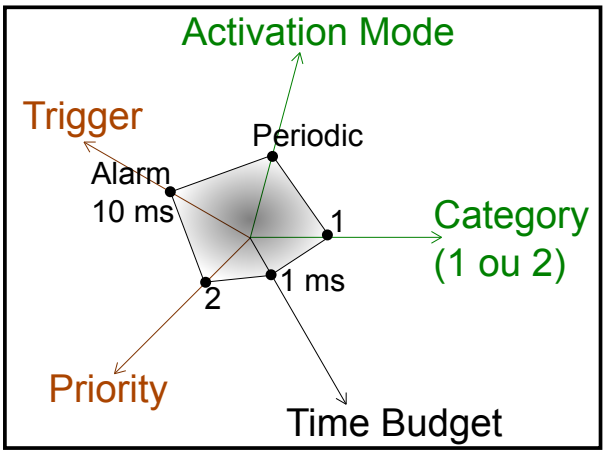


FIGURE 3.1 – Exemple d'espace d'adaptation

pour la mise à jour. Cela s'explique par la grande diversité des communications dans un calculateur automobile (plusieurs centaines, voire plusieurs milliers de données échangées dans un seul calculateur). Chaque mise à jour a des besoins spécifiques de communication qui dépendent de ses spécifications fonctionnelles, des canaux de communications **ad hoc** doivent être ajoutés dans le système au moment de la mise à jour.

3.2.3 *Container*

Le *container* est l'implémentation dans le système d'un espace d'adaptation. Il est fait pour accueillir un nouveau runnable. Il peut donc être défini comme la réalisation physique au sein de l'application de l'espace d'adaptation qui implémente toutes les caractéristiques de ce dernier et qui agit comme un espace réservé dans l'application destiné à accueillir les futures mises à jour. Pour qu'un nouveau runnable puisse être placé dans un *container* il faut que les caractéristiques de ce runnable correspondent à celles du *container*.

Lorsque le *container* est créé et placé dans le système, un budget temps maximum lui est alloué. Ce budget doit être pris en compte lors de l'analyse d'ordonnancement. Cette hypothèse est liée à l'approche *pré-câblée* qui est prise en compte ici. Elle peut être relâchée si une approche *opportuniste* est considérée. Dans ce cas, il est nécessaire de mettre en œuvre certains résultats issus de la théorie de l'ordonnancement temps-réel (voir chapitre 4).

Ainsi, nous considérons ici que les *containers* doivent avoir des caractéristiques temporelles et structurelles définies lorsque celui-ci est placé dans l'application. Bien que le budget temporel doive être pris en compte dans l'analyse d'ordonnancement, lorsque le *container* est vide il ne consomme pas son budget temps. Il n'utilise du temps d'exécution que lorsqu'une mise à jour est effectivement placée à l'intérieur. Cette mise à jour doit avoir un WCET (Worst-Case Execution Time) impérativement inférieur au budget temps alloué au *container*.

Le *container* associé à l'espace d'adaptation doit être placé dans une tâche ayant des caractéristiques compatibles, en particulier du point de vue du mode d'activation et du type. Le budget temporel alloué au *container* doit donc être ajouté au temps total d'exécution de la tâche. Cette approche est donc relativement pessimiste et si un grand nombre de *containers* est ajouté dans l'application, la charge CPU restant libre dans l'application initiale peut être assez faible. Un exemple de *container* placé dans une tâche est donné par la figure 3.2. Sur cette figure, les flèches représentent les communications. Étant donné que chaque mise à jour a des besoins propres en terme de communication, le *container* ne possède aucune flèche. Les détails de l'implémentation pratique des *containers* sont donnés ultérieurement.

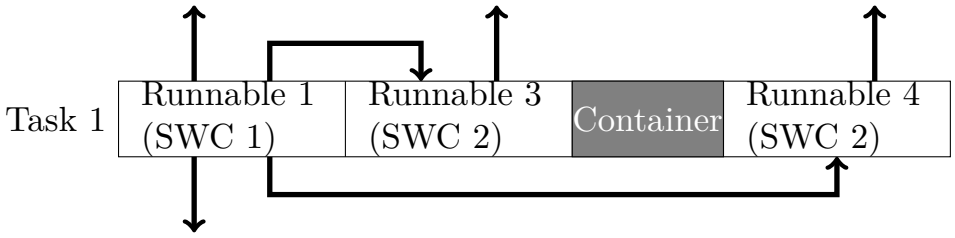


FIGURE 3.2 – Exemple de tâche avec un *container*

3.3 Développement d'une application logicielle embarquée : processus standard

Lors du développement d'une application embarquée de type AUTOSAR, un processus précis doit être suivi. Ce processus repose sur un certain nombre d'outils nécessaires pour créer une application embarquée de type AUTOSAR [106].

La figure 3.3 montre les différentes étapes nécessaires à la création d'une application embarquée de type AUTOSAR telle qu'elle a été définie chez Renault. Il s'agit d'une approche dite *top down*, c'est à dire que le point de départ sont les spécifications fonctionnelles pour arriver au final au code des calculateurs. Ce processus respecte la méthodologie AUTOSAR [15], mais apporte un certain nombre de contributions supplémentaires. Tout d'abord une étape supplémentaire est rajoutée : elle consiste a faire une modélisation complète du point de vue de l'architecture logicielle fonctionnelle. Cette étape n'est pas forcément spécifique à une conception AUTOSAR. Il s'agit d'une architecture globale et haut niveau du logiciel. Ensuite des critères spécifiques de regroupement des runnables dans les composants logiciels sont explicités. Enfin les critères d'allocation pour les SWCs sur les ECUs et pour les runnables sur les tâches sont détaillés. Ces éléments n'apparaissent pas explicitement dans le standard AUTOSAR.

La première étape du processus de développement est d'identifier les besoins fonctionnels, c'est-à-dire quelles sont les spécifications qui doivent être remplies par notre système et les caractéristiques qu'il doit avoir. Lorsque les besoins ont été identifiés, le processus de développement peut débuter. Cette section décrit les différentes étapes de

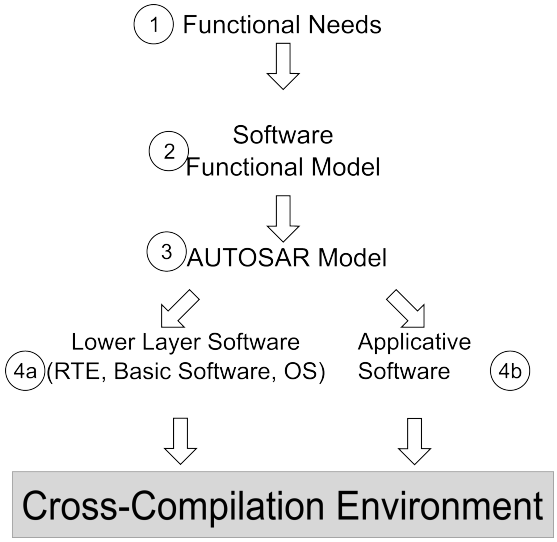


FIGURE 3.3 – Processus simplifié de développement d'une application AUTOSAR.

ce processus. Tout d'abord comment le modèle logiciel global est créé (étape 2 de la figure 3.3). Cette étape définit les éléments logiciels applicatifs dont nous disposons et qui doivent ensuite être agencés de manière à respecter l'architecture AUTOSAR (étape 3 de la figure 3.3). Enfin, pour chacun des calculateurs, des outils permettent de générer et/ou de configurer d'une part les couches basses et d'autre part le logiciel applicatif lui-même. Chacune de ces étapes est détaillée dans les sections suivantes.

En se plaçant dans un contexte avec une architecture de type AUTOSAR, nous devons non seulement respecter l'architecture elle-même avec sa division en couche comme détaillée dans la section 2.5, mais également la démarche de développement requise par le standard.

3.3.1 Architecture logicielle fonctionnelle globale

Pour répondre à un besoin fonctionnel dans le domaine de l'automobile, il est généralement nécessaire d'avoir des organes mécaniques, électroniques et logiciels. Le processus qui est développé ici se concentre uniquement sur la partie logicielle.

L'étape présentée dans cette section, à savoir l'*architecture logicielle fonctionnelle globale* ne fait pas partie des étapes décrites par la méthodologie AUTOSAR. C'est une étape qui a été rajoutée dans le développement d'application chez Renault. Nous avons, en effet, considéré qu'il est nécessaire d'avoir une vision globale des différents éléments présents dans un système et qui réalisent un ensemble de fonctionnalité. Cela permet d'avoir une meilleure cohérence dans un système. De plus, c'est une manière de regrouper de manière formelle les informations concernant une application et d'éviter ainsi les redondances. Les éléments disponibles et nécessaires de l'application sont donc clairement identifiés. Ainsi, cette première étape du processus de développement a été

pensée pour améliorer le dossier de conception d'une application et pour permettre une meilleure analyse haut niveau sur le plan structurel et comportemental. Cette étape est indispensable lorsque par la suite nous souhaitons faire des mises à jour.

Lors de la conception du logiciel, la première étape consiste en une description des différents cas d'utilisation correspondant à chacune des fonctions. Si nous prenons l'exemple d'une fonction A, il faut en déterminer tous les cas d'utilisation. La figure 3.4 montre un exemple de diagramme sur lequel les différents cas d'utilisation pour une fonction donnée sont décrits. Ce diagramme donne uniquement des informations basiques.En effet, il ne montre que les cas d'utilisation et les acteurs affectés par ou agissants sur ces cas d'utilisation.

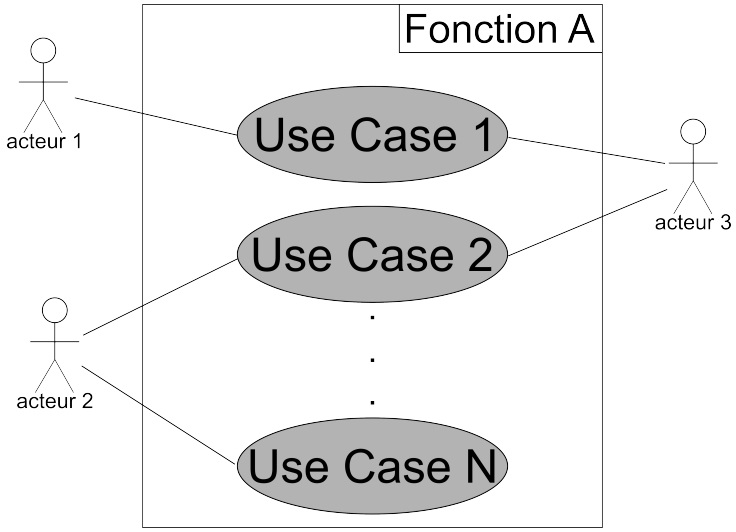


FIGURE 3.4 – Exemple de Use Cases

Chacun de ces cas d'utilisation doit ensuite être détaillé en utilisant, par exemple, un ou plusieurs diagramme(s) UML qui définit quels éléments doivent interagir et comment ces éléments sont inter-connectés. Chacun de ces "éléments" est appelé une *action*. Une chaine de calcul est représentée par un ensemble d'actions, connectées par des flots de données et/ou des flots de contrôles. Cet ensemble réalise tout ou seulement une partie d'un cas d'utilisation. Une action a une taille limitée dans la mesure où elle doit être élémentaire.

Chacun des cas d'utilisation est défini et implémenté indépendamment par une ou plusieurs chaines de calcul. Ainsi, il est possible qu'il existe des éléments communs entre les chaines de calcul des différents cas d'utilisation. Une seconde étape de conception consiste à regrouper les actions communes à différentes chaines de calcul. Cela peut aboutir à la création de chaines de calcul plus complexes. La figure 3.5 montre l'exemple de deux chaines de calculs spécifiques dans le cas de deux cas d'utilisation. Ces deux chaines de calcul ont ici des éléments communs (*Action 3*, *Action 4* et *Action 5*). Ces chaines peuvent donc être regroupées afin de former une chaine plus complexe. La fi-

gure 3.6 montre comment ces chaines sont regroupées. Le regroupement des actions pour former les chaines de calcul est un travail d'architecture logiciel.

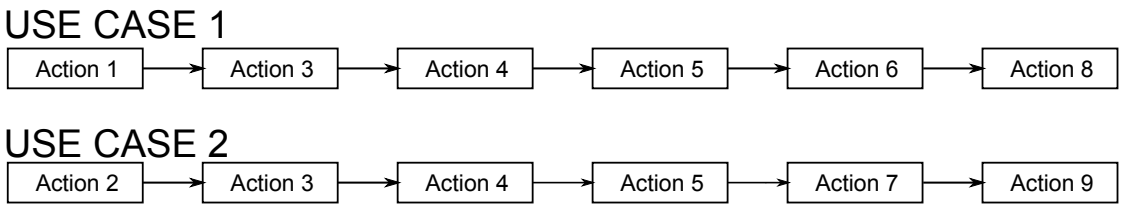


FIGURE 3.5 – Exemple de chaines de calcul spécifiques pour deux cas d'utilisation

La figure 3.6 montre un exemple de chaine de calcul. Le premier niveau de division dans la chaine de calcul est la décomposition en actions qui doivent avoir une taille limitée dans la mesure où chaque action doit être élémentaire. La granularité des actions n'est pas nécessairement celle des runnables. En fonction de la taille et de la nature de ces actions, il est ensuite possible de regrouper ces actions pour former des runnables. Il est également possible d'avoir des actions qui sont directement prises pour devenir des runnables. Il est généralement nécessaire de regrouper plusieurs actions afin de créer un runnable. Il est important de noter que, l'un des objectifs étant la réutilisabilité des composants logiciels, le groupement des actions doit être fait de telle sorte que les actions qui dépendent spécifiquement du matériel (par exemple les capteurs et les actionneurs) doivent être séparées des actions calculatoires.

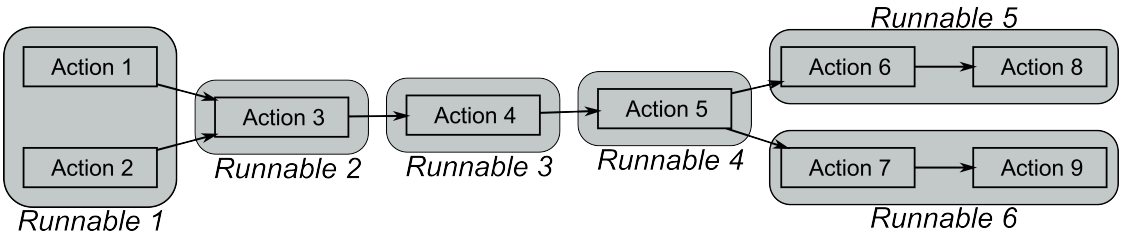


FIGURE 3.6 – Exemple de chaine de calcul

On peut noter que le modèle fonctionnel global ne se concentre que sur la partie applicative. Seuls sont exprimés ici certains besoins que peut avoir le logiciel applicatif vis-à-vis des couches basses, mais la modélisation des éléments permettant de répondre à ces besoins n'apparait pas ici. En effet, dans le cas du modèle AUTOSAR, les éléments du Basic Software sont des modules standards configurés en fonction des besoins de l'applicatif. Il n'est pas nécessaire de les modéliser ici. Le travail de configuration est fait plus loin dans le processus de développement.

Le modèle fonctionnel global doit également contenir un certain nombre d'informations concernant les contraintes haut niveaux. Ces dernières permettent de déterminer des critères d'implémentation. Parmi ces informations, il est possible de retrouver les contraintes de temps de bout-en-bout sur les chaines de calcul. Le mode d'activation des différentes actions ou runnables (en fonction de la granularité) doit également être

présent : il détermine les flots de contrôle et la manière dont les runnables sont activés (périodiquement, sporadiquement, sur évènement). D'autre part ce modèle architectural permet d'avoir les flots de données entre runnables.

Cependant des informations supplémentaires concernant les contraintes temporelles sur les runnables, et en particulier concernant les flots de données et leur chronologie sont également nécessaires. Un autre type de diagramme UML, à savoir le diagramme de séquence, permet d'avoir ces informations. La figure 3.7 présente un exemple de diagramme de séquence. L'échange dans le temps des messages entre les différents runnables présents dans l'application est représenté ici (c'est un choix architectural de présenter ici le diagramme de séquence pour les runnables et non pour les actions). Ce diagramme indique la nature, ainsi que la chronologie des messages échangés.

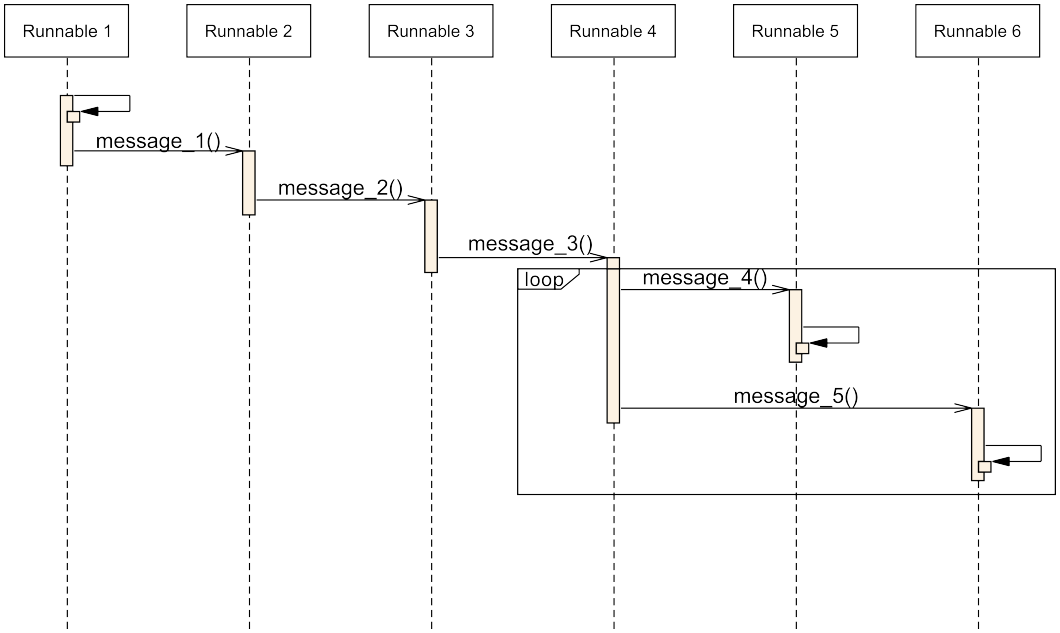


FIGURE 3.7 – Exemple de diagramme de séquence

3.3.2 Architecture logicielle AUTOSAR

Lorsque tous les runnables et leur caractéristiques propres (caractéristiques temporelles, flots de données, flots de contrôle), ainsi que les caractéristiques temporelles de l'application (contraintes de temps de bout-en-bout), ASIL (Automotive Safety Integrity Level [57]), contraintes de sécurité particulières ont été définies dans le modèle fonctionnel global, les différents éléments applicatifs peuvent être organisés de manière à correspondre au modèle architectural du standard AUTOSAR.

Une des premières étapes pour l'architecture au format AUTOSAR est de regrouper les runnables qui ont été précédemment définis au sein de composants logiciels. Les critères qui permettent de procéder à ce groupement ne sont pas clairement définis dans

le standard. Ceux que nous présentons ici ont été définis par Renault dans le processus de développement. Le regroupement doit obéir à une logique permettant la meilleure réutilisation possible des composants ainsi créés. Par exemple, tous les runnables d'un même SWC doivent être placés dans la même OS-Application (groupement d'objets OS qui permet de mettre en place des mécanismes de protection) : pour cette raison il est préférable que tous les runnables d'un même SWC aient le même niveau ASIL.

L'objectif de ce modèle AUTOSAR est :

— de définir les différentes communications entre les composants applicatifs,
— de répartir les composants sur les ECU disponibles,
— de configurer les couches du Basic Software de chaque ECU pour répondre aux besoins des composants qui lui sont alloués.

3.3.2.1 Conception niveau Virtual Functionnal Bus (VFB)

La première étape du modèle AUTOSAR consiste à définir les SWCs ainsi que leur communications. Il s'agit ici d'une vue haut niveau qui ne dépend pas de l'allocation future sur les différents ECUs disponibles. Cette conception est la vue VFB (Virtual Functional Bus), c'est-à-dire les communications entre les différents SWCs d'un point de vue global. Cependant, afin de garder la meilleure indépendance possible entre les composants logiciels (c'est un des buts de AUTOSAR), chaque composant ignore l'identité de celui avec qui il communique. D'autre part, l'allocation des composants sur les ECUs ne doit pas avoir d'impact, du point de vue des SWCs, sur les communications : même si les mécanismes bas niveau sont différents dans le cas de communication entre ECUs, cela doit être totalement transparent pour les SWCs.

Les critères qui permettent de regrouper des runnables au sein d'un même composant sont principalement des critères de cohérence et de sécurité-innocuité. En effet, puisque 1) l'objectif est d'assurer la meilleure réutilisabilité possible des SWCs et 2) les runnables d'un même SWC doivent être placés dans une même OS-Application, alors il faut faire en sorte que les runnables d'un même SWC aient une cohérence fonctionnelle et que le niveau d'ASIL soit cohérent. Pour une meilleure réutilisabilité, il faut également éviter de réunir au sein d'un même composant des runnables directement dépendants du matériel et des runnables calculatoires. En effet, les unités qui peuvent être réutilisées d'une application à une autre sont les SWCs et non les runnables.

La figure 3.8 montre ici la vue VFB pour l'exemple de la section 3.3.1. Les 6 runnables, qui ont été répartis dans différents composants logiciels, sont bien présents ici. Les runnables 1, 5 et 6 qui sont aux extrémités de la chaine de calcul, peuvent correspondre à des capteurs ou des actionneurs. Ils ont été placés dans des composants spécifiques. Les runnables 2, 3 et 4, quant à eux sont au milieu de la chaine et peuvent correspondre à des runnables de traitement, ils sont ici groupés ensemble au sein d'un même composant. Il s'agit ici d'une répartition possible, mais d'autres auraient également été valables. Une fois de plus il s'agit d'un choix architectural. Les liens qui apparaissent sur la figure 3.8 représentent les flots de données entre les runnables de SWCs différents.

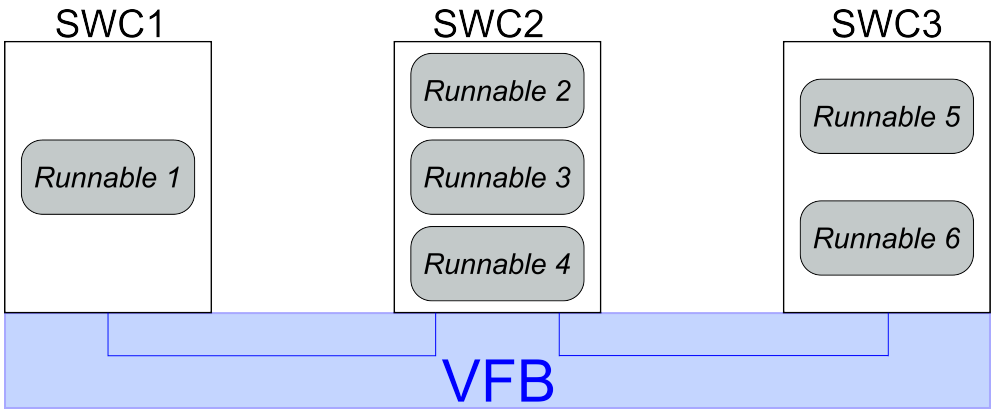


FIGURE 3.8 – Répartition des runnables en SWCs - vue VFB

3.3.2.2 Allocation sur les calculateurs

Une fois que la vue VFB est réalisée, il est possible de déterminer l'allocation des composants sur les différents calculateurs disponibles. Chaque SWC atomique est donc alloué sur un calculateur. Un SWC atomique est un composant qui ne peut pas être sous divisé en composants logiciels plus petits. Les composants logiciels non atomiques sont dits "compositions" et ne peuvent contenir que des SWCs. Ils n'ont pas de comportement interne propre (pas de runnables appartenant exclusivement à la composition, tous les runnables doivent appartenir à un sous-SWC).

Pour la répartition, il faut prendre en compte les critères de couplage entre les composants : si deux SWC échangent un grand nombre d'informations, il est préférable de les placer dans le même calculateur afin de limiter le nombre de messages qui transite sur les bus de communication ainsi que de réduire les latences. D'autre part, la position de certains composants (en particulier les capteurs et les actionneurs) sont généralement définis à l'avance en fonction de la topologie du réseau.

De plus, des contraintes de sécurité-innocuité peuvent se rajouter pour déterminer l'allocation des SWC sur les ECUs telles que l'indépendance entre certains SWCs ou la charge des calculateurs. Enfin des questions de coût peuvent aussi intervenir.

Les critères de répartition des composants logiciels sur les calculateurs ne sont pas définis par le standard. Ceux qui sont présentés dans les paragraphes précédents sont ceux définis par le processus de développement Renault.

En fonction de la répartition des composants sur les différents calculateurs, il est nécessaire d'utiliser des canaux de communications bas niveaux, comme le CAN pour envoyer des messages. Le fait que les messages transitent par un bus physique est totalement transparent pour les messages, mais doit être pris en compte lors de la vérification des contraintes temporelles de bout-en-bout.

La figure 3.9 montre ici une répartition possible des composants logiciels dans différents ECUs. Il est nécessaire ici d'utiliser des canaux de communication bas niveau afin de permettre les communications entre le SWC1 et le SWC2.

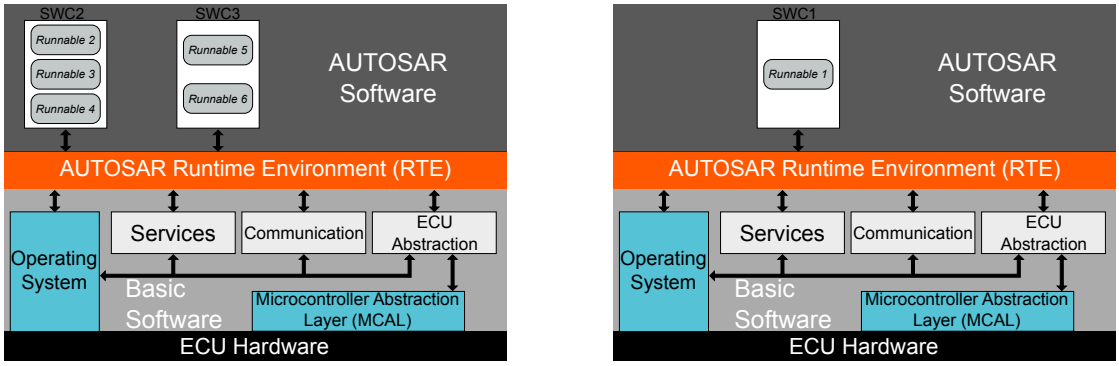


FIGURE 3.9 – Allocation des SWCs sur les ECUs - vue RTE

3.3.3 Création du code des ECUs

Pour chaque calculateur, il est ensuite nécessaire de générer le code correspondant, d'une part aux couches basses et d'autre part aux couches applicatives. Ces couches sont générées pour chacun des ECU. Le périmètre de cette section se limite à un seul ECU.

3.3.3.1 Création du logiciel applicatif

Lorsque les différentes parties du logiciel ont été définies sous forme de runnables appartenant à des SWCs et que leur allocation sur chacun des ECUs est connue, le logiciel applicatif correspondant doit être créé. Ce dernier correspond au code qui doit effectivement être exécuté pour remplir les fonctions définies précédemment.

Ce code doit non seulement répondre aux spécifications définies dans le modèle logiciel fonctionnel global, mais également au découpage qui a été défini dans le modèle AUTOSAR. Ainsi, il faut non seulement répondre aux contraintes liées à ses interactions et ses modes de vie (différentes périodes de la vie véhicule correspondant à l'exécution de code différent, par exemple à l'initialisation) mais également être en mesure d'isoler chacun des runnables puisque ce sont eux qui sont ensuite exécutés par les tâches.

L'approche qui est utilisée ici est dite "model-based", c'est-à-dire que le code n'est pas écrit manuellement pour chaque runnable, mais généré à partir d'un modèle comportemental plus haut niveau. Cela nécessite donc d'avoir des outils permettant la génération de code conforme aux spécifications AUTOSAR. Pour des raisons de confidentialités, il n'est pas possible de détailler les outils utilisés par Renault au cours de ce processus de développement.

Nous pouvons également noter que le code des runnables doit être compatible avec les interfaces qui ont été définies dans le modèle AUTOSAR. Il existe donc des dépendances fortes entre les différentes vues de ce processus et l'ordre de réalisation des étapes est importante.

3.3.3.2 Allocation des runnables dans les tâches

Afin de pouvoir être exécutés, les différents runnables doivent être placés dans des tâches de l'OS. Les caractéristiques de chacune des tâches sont définies à partir de celles des runnables. Les critères d'allocation présentés ici ne sont pas présents dans le standard. Il s'agit de critères développés pour le processus de développement AUTOSAR.

Chaque runnable évènementiel est placé seul dans une tâche évènementielle. Pour les tâches périodiques, il est possible d'avoir plusieurs runnables dans une même tâche. AUTOSAR autorise des runnables ayant une période plus grande à être alloués dans des tâches de période plus faible. Par exemple un runnable qui a une période à 20 ms peut être placé dans une tâche à 10 ms et exécuté une fois toutes les deux activations de la tâche.

Il existe plusieurs critères qui permettent de déterminer l'allocation de runnables dans une même tâche. Il est impératif de séparer les éléments du Basic Software des éléments applicatifs (runnables). En effet, les différents modules du Basic Software possèdent également des éléments qui doivent être exécutés pour le bon fonctionnement de l'ECU. Cependant ces éléments ne doivent pas être mélangés avec des runnables au sein d'une même tâche. Cela permet une meilleure séparation des éléments : une faute sur un élément du Basic Software ne doit pas perturber le logiciel applicatif et vice versa. De plus, les équipes qui travaillent sur le Basic Software ne sont pas nécessairement les mêmes que celles qui travaillent sur le logiciel applicatif. Ainsi, la répartition des runnables dans les tâches répond à la logique suivante :

- **Architectural** : Les runnables ne sont pas placés dans des tâches contenant des éléments du Basic Software.
- **Criticité** : Les niveaux ASIL (tels que définis dans la norme ISO 26262 [57]) sont également à prendre en compte lors de la répartition : des runnables ASIL A ne doivent pas être dans une même tâche que des runnables ASIL D afin d'éviter qu'un problème dans un runnable ASIL A ne puisse perturber la bonne exécution de celui ASIL D.
- **Type** : Les types de calcul effectués par les runnables sont également à prendre en considération : les machines à état, les calculs bouclés et les calculs linéaires ne doivent pas être mélangés.
- **Temps-réel** : Les contraintes temps-réel, telles que les périodes ou les contraintes de temps de bout-en-bout.
- **Propriété intellectuelle** : Des considérations de propriété intellectuelle (par exemple si plusieurs équipementiers fournissent du code pour un même ECU) ainsi que de test et de maintenance peuvent intervenir dans la répartition des runnables dans les tâches.

3.3.3.3 Génération du RTE et configuration des couches bas niveaux

Les couches bas niveau présentent 3 points d'intérêt différents pour notre étude : le RTE, l'OS et la pile de communication. Les autres éléments ne sont pas détaillés ici car nous nous concentrons uniquement sur les éléments qui peuvent avoir une influence

concernant la mise à jour.

Contrairement au logiciel applicatif, le Basic Software n'est pas directement généré à partir de modèles. En effet, il est constitué d'un ensemble de modules pré-existants (qui peuvent par exemple être fournis par un équipementier automobile), et qui doivent être configurés afin de répondre aux besoins du logiciel applicatif. La pile de communication ainsi que l'OS sont configurés et non générés.

L'OS doit être configuré en fonction des différentes tâches qui doivent être exécutées. Le type de tâche (applicative ou Basic Software) n'est pas important dans ce cas, seules les caractéristiques ont un impact sur la configuration de l'OS. C'est également à ce moment là que les différentes OS-Applications sont définies, ainsi que la répartition des différentes objets de l'OS au sein de ces OS-Applications. Le fichier de configuration de l'OS encapsule toutes les informations temporelles définies dans le modèle fonctionnel global. En particulier, il faut définir toutes les tâches qui sont exécutées ainsi que leurs caractéristiques. Il faut également définir les différents évènements (pour les tâches non périodiques), les sections critiques, les alarmes (pour les tâches périodiques) et les compteurs (qui déclenchent les alarmes). Ces différentes informations peuvent être extraites des différents diagrammes du modèle UML.

La configuration de la pile de communication dépend des messages échangés entre l'ECU concerné et les autres ECUs présents dans le véhicule. En effet, il faut définir quels messages transitent sur les bus matériels disponibles. Chaque communication émise par un composant logiciel sur le RTE et à destination d'un ECU distant doit être alloué sur un signal. De même pour les communications en provenance d'ECUs distants et à destination de composants logiciels. Pour chacun de ces signaux il faut spécifier l'émetteur, le destinataire, sa taille et la fréquence à laquelle il doit être envoyé. Dans le cas du bus CAN, une fois que tous les signaux émis et reçus par les calculateurs sont connus, l'étape suivante consiste à optimiser la messagerie. Cette étape est assez complexe et est réalisée par un expert du domaine (et à l'aide d'un logiciel dédié). Les différents signaux sont regroupés dans des trames CAN, un identifiant est alloué à chacune de ces trames. Ce sont les identifiants de ces messages qui permettent à chaque calculateur de savoir s'il est destinataire ou non du message.

Le RTE, quant à lui, est généralement généré de manière *ad hoc* pour chacun des ECUs. Étant donné le grand nombre de communications qui existe au sein d'un ECU, il est conseillé d'utiliser un *RTE generator* pour obtenir le code du RTE. En effet, le RTE doit non seulement gérer les communications entre les SWCs de l'ECU, mais également les communications avec des ECUs distants (qui elles passent par le Basic Software via la pile de communication). D'autre part, il gère aussi le déclenchement de l'exécution des runnables. Par exemple, le code des tâches applicatives se trouve dans le code généré par le *RTE generator*.

3.4 Modifications du processus de développement

Le processus standard tel qu'il a été décrit dans la section 3.3 ne permet pas d'avoir un degré de flexibilité suffisant pour pouvoir faire des mises à jour dans le système. Les

mises à jour sont possibles grâce à l'introduction des concepts définis dans la section 3.2, à savoir les *espaces d'adaptation* et les *containers* qui leurs sont associés.

L'objectif de cette section est de décrire les modifications nécessaires pour intégrer ces concepts dans le processus de développement. La figure 3.10 présente les étapes qui doivent être ajoutées au processus afin d'introduire de la flexibilité dans l'ECU final. L'application est préparée pour recevoir les mises à jour. Cela consiste à modifier les étapes 2, 3 et 4a du processus de développement afin d'ajouter tous les éléments nécessaires.

On peut voir ici que plusieurs éléments sont nécessaires. D'une part, l'identification de la version courante de l'application est une étape importante. En effet, il faut être en mesure de déterminer dans quel contexte spécifique une mise à jour doit s'intégrer. D'autre part, il faut non seulement pouvoir loger la mise à jour dans l'architecture initiale (c'est le rôle de nos *containers*), mais il faut également être en mesure de déployer tous les mécanismes bas niveau qui sont nécessaires au bon acheminement, au stockage et à l'exécution de cette mise à jour.

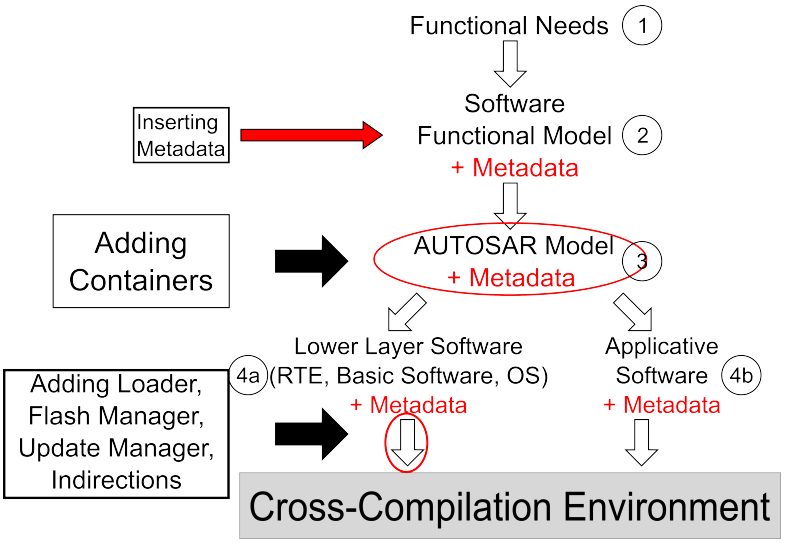


FIGURE 3.10 – Modifications nécessaires dans le processus de développement pour permettre les mises à jour.

3.4.1 Détermination de la version de l'application

Assez tôt dans le processus de développement, il est nécessaire d'ajouter des méta-données afin de pouvoir identifier les différents éléments présents dans l'application. Ces méta-données permettent d'avoir une traçabilité (une signature) des éléments présents et de leurs versions. L'agrégation de l'ensemble de ces méta-données permet ensuite de déterminer la version de l'application elle-même. Elles portent sur les différents runnables et *containers* présents dans l'application. Elles permettent l'identification des caracté-

ristiques de chacun des éléments de l'application, comme la version, l'emplacement dans la mémoire ou encore les caractéristiques temps-réel.

L'objectif premier de ces méta-données est de pouvoir déterminer à tout instant quels sont les éléments précisément présents dans le système afin de savoir quel est le contexte dans lequel une nouvelle mise à jour s'insèrerait. Cela permet également de retrouver tous les modèles correspondant à cette version du système ainsi que toutes les données nécessaires à l'implémentation de mise à jour (par exemple l'emplacement mémoire de chacun des éléments ou encore les *containers* disponibles ou le temps d'exécution restant).

3.4.2 Ajout des *containers*

Les *containers* ainsi que les méta-données qui leurs sont propres sont insérées dans le modèle AUTOSAR (étape 3 de la figure 3.10). Ces derniers ne peuvent pas être intégrés plus tôt dans le processus car le concept lié est défini à partir de concepts propres à AUTOSAR. Le modèle fonctionnel global a pour vocation de rester général. D'autre part les *containers* ne remplissent pas de rôle fonctionnel propre.

De plus, insérer les *containers* à ce niveau permet de les garder dans les étapes suivantes et de les prendre en compte lors de la configuration des couches bas niveau. Notons qu'il est souhaitable d'ajouter des *containers* dans tous les ECUs présents afin d'avoir le maximum de flexibilité possible.

Le nombre de *containers* ajouté dans l'application dépend du type d'approche choisie. Dans le cas d'une approche pré-câblée, il faut choisir avec soin les *containers* qui sont ajoutés car un WCET propre leur est alloué, alors que dans le cas d'une approche opportuniste, il est possible d'ajouter autant de *containers* qu'on le souhaite car ces *containers* n'ont pas d'existence du point de vue temporel. Ils ne consomment pas de temps et ne viennent pas perturber l'ordonnancement tant qu'ils restent vides. Cette seconde approche présente toutefois l'inconvénient d'exiger une analyse temporelle supplémentaire lors de la modification du système.

3.4.3 Ajout des mécanismes bas niveau

Des mécanismes bas niveau sont nécessaires pour une gestion à un niveau de granularité fin des mises à jour. Au sein du calculateur, il faut être en mesure de gérer les mises à jour, à savoir, communiquer avec l'environnement pour les demander et les réceptionner, vérifier leur intégrité, les stocker en mémoire et les placer dans des *containers* pour leur exécution.

Une modification importante concernant le code généré pour le contenu des tâches consiste dans l'utilisation d'indirections afin de pouvoir ajouter de la flexibilité au système. En effet, il est souhaitable que le contenu des runnables et des *containers* puisse être modifié en cas de nécessité. C'est ce que ce mécanisme d'indirection nous permet de faire. Cette étape sera détaillée dans le chapitre 5.4.

Ces services ne sont pas présents par défaut dans le standard, et sont donc placés dans un "Complex Device Driver". La figure 3.11 donne une vue de l'organisation des

différents éléments ajoutés dans l'application. Les méta-données n'apparaissent pas sur cette vue car elles sont attachées à chacun des éléments du système.

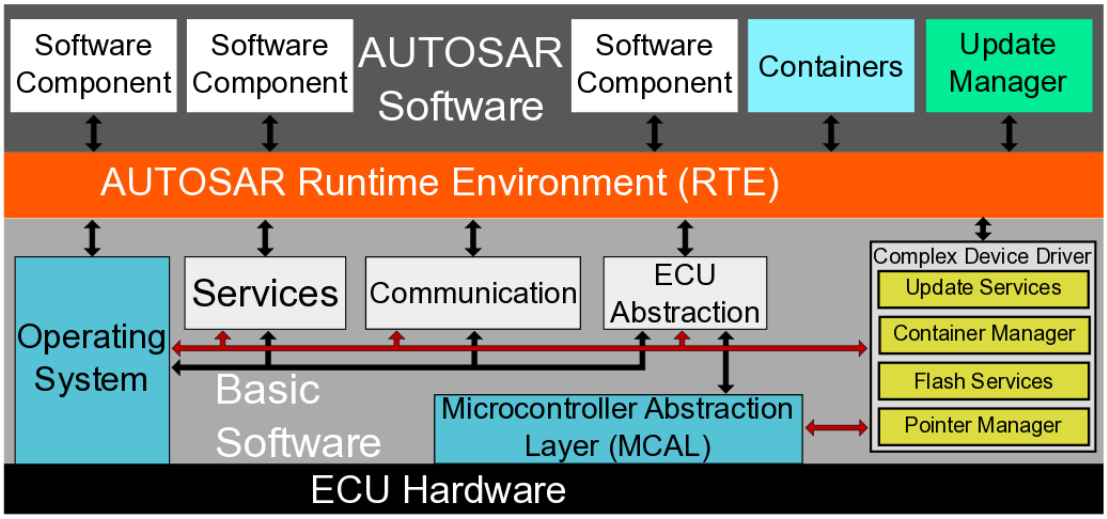


FIGURE 3.11 – Architecture AUTOSAR pour la mise à jour

3.4.3.1 Réception des mises à jour - *Update Services*

Un protocole de communication doit être mis en place entre le serveur contenant les mises à jour et le véhicule qui est destiné à recevoir ces mises à jour. Nous ne définissons pas ici de protocole particulier, dans la mesure où il en existe déjà dans le contexte automobile. D'autre part, la définition d'un protocole de communication à l'échelle industrielle n'est pas l'objectif de ce travail.

Dans un système complexe comme un véhicule complet avec l'ensemble de ses calculateurs, les mises à jour doivent d'abord passer par une passerelle qui est le lien entre le véhicule et l'extérieur. C'est cette dernière qui achemine ensuite la mise à jour au calculateur concerné. La mise à jour transite ensuite par un bus matériel interne (comme le bus CAN). Cela signifie que la messagerie doit être configurée pour prendre en compte les trames contenant les mises à jour dans une phase préalable à l'utilisation du véhicule.

Au sein du calculateur, il faut donc être en mesure de réceptionner la mise à jour transmise par paquets, et de la reconstruire. Le module qui permet la réception des mises à jour doit donc s'assurer que les mises à jour sont complètes et non corrompues, qu'elles lui sont bien destinées, et les stocker en mémoire volatile de manière temporaire.

Un module de gestion de la réception de la mise à jour doit donc être ajouté en supplément des autres modules déjà présents dans le Basic Software. Ce module est placé dans un "Complex Device Driver" du Basic Software et fait partie des "update services".

Il doit être en mesure de récupérer des informations issues de la pile de communication : en effet, les mises à jour transitent par le bus matériel, puis par la pile de communication présente dans le calculateur.

3.4.3.2 Stockage des mises à jour - *Container Manager* et *Flash Services*

Lorsque la mise à jour est arrivée dans son intégralité, et qu'elle a été vérifiée (signatures cryptographiques, etc), il faut la stocker en mémoire non volatile pour qu'elle puisse être exécutée et que cette mise à jour soit pérenne (*i.e.* qu'elle ne disparaisse pas si le calculateur doit être redémarré).

En outre, l'emplacement mémoire dans lequel la mise à jour doit être stockée est déterminé à l'avance et fait partie des données supplémentaires qui sont envoyées avec la mise à jour. En effet, les méta-données qui ont été ajoutées en amont lors de la préparation de l'application permettent de connaître à l'avance, non seulement la version de l'application, mais également de connaître les détails bas niveau (comme l'emplacement mémoire de chacun des éléments). Ainsi, il est possible de savoir par avance où les mises à jour doivent être stockées. Cela a également l'avantage de limiter les opérations qui doivent être faites directement dans l'ECU embarqué.

Ainsi, il faut ajouter dans le système un module capable de gérer ces aspects de stockage de la mise à jour. Notons que dans le cas d'une architecture de type AUTOSAR, il existe des services de la MCAL qui permettent d'écrire ou d'effacer la mémoire. Cependant, même si ces services sont utilisés, il faut mettre en place des mécanismes de plus haut niveau qui gèrent les données écrites ou effacées, l'ordre dans lequel les opérations doivent avoir lieu, etc. Ces mécanismes plus haut niveau sont placés dans un "Complex Device Driver" dans la mesure où il n'existe pas, à l'heure actuelle, dans le standard AUTOSAR de module du Basic Software permettant de réaliser ces opérations.

3.4.3.3 Exécution des mises à jour - *Pointer Manager*

Nous avons besoin d'ajouter en amont dans l'application un niveau supplémentaire d'indirection afin de pouvoir modifier les appels au code des runnables et pour utiliser les *containers*. Ainsi, un post-traitement sur le fichier contenant l'appel aux runnables et aux *containers* (habituellement créé par le RTE generator) est nécessaire afin d'ajouter ce niveau d'indirection.

Un module, le *Pointer Manager*, permet de modifier les runnables qui sont exécutés. Son rôle est de gérer les valeurs de la table d'indirection. Son rôle est aussi d'assurer la validité des valeurs contenues dans la table, mais également de garantir qu'il n'y a pas de corruption dans cette table (par exemple qu'un pointeur n'indique pas une case mémoire inaccessible).

D'autre part, un autre module doit gérer l'état des *containers*, le *Container Manager* : ceux qui sont utilisés, ceux qui sont disponibles et les caractéristiques associées. En effet, chaque fois qu'une mise à jour est ajoutée dans le système, il faut modifier en conséquence les méta-données associées à l'emplacement dans lequel cette mise à jour est placée.

3.4.4 Préparation off-line et intégration on-line

La préparation de l'application afin de permettre d'intégrer des mises à jour est intégralement faite avant le chargement et l'exécution. Ainsi l'ajout de *containers*, de signatures et de modules de gestion pour la mise à jour doivent se faire off-line (c'est-à-dire avant l'exécution, au cours du processus de développement). Le protocole de communication entre le véhicule qui doit recevoir les mises à jour et le serveur qui permet l'envoi de ces dernières doit également être défini au préalable ainsi que la messagerie correspondant aux mises à jour. La diffusion des trames depuis la passerelle aux calculateurs concernés, ainsi que leur réception, sont configurées de manière statique dans la messagerie.

Des vérifications en ligne doivent cependant être faites pour pouvoir intégrer les mises à jour dans le système. Tout d'abord, la validité et l'intégrité des données de mise à jour doivent être vérifiées en ligne, d'une part par la passerelle, et d'autre part par le calculateur final. Les indirections et les *containers* sont aussi gérés en ligne par le calculateur. En effet, même si l'emplacement où une mise à jour est placée est déterminé au moment de la création de l'application, le chargement effectif dans le *container* et la modification des indirections sont faits dans le calculateur.

La figure 3.12 montre les différentes étapes de création et d'intégration d'une mise à jour dans un calculateur de type AUTOSAR. A gauche de la figure 3.12, les étapes faites off-line pour la conception d'une mise à jour : tout d'abord la mise à jour doit être créée, puis testée, tout d'abord unitairement, puis intégrée dans son futur contexte d'exécution. Après que la mise à jour est correctement validée, cette dernière peut être intégrée dans le calculateur de destination. Pour cela, le gestionnaire de mise à jour la réceptionne, puis la place dans un *container* pour qu'elle puisse être exécutée.

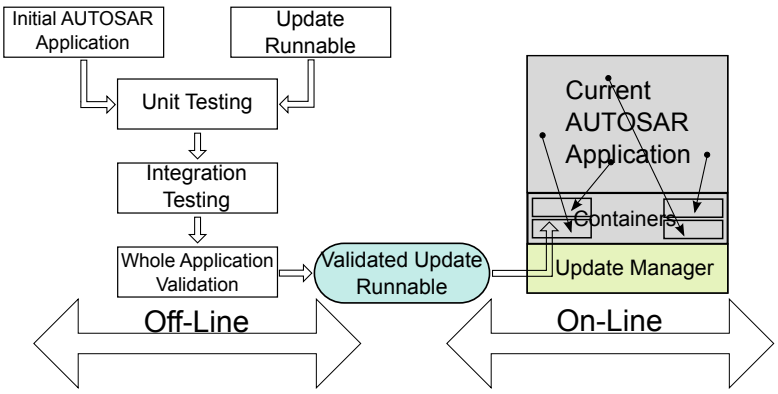


FIGURE 3.12 – Création et intégration d'une mise à jour

3.5 Types de mise à jour

Dans cette section, nous décrivons les différentes mises à jour possibles ainsi que les conséquences de l'ajout d'une ou plusieurs de ces mises à jour sur le système. Pour cela, dans un premier temps, nous prenons le cas d'un système simple composé d'une chaine de calcul à trois runnables. Cette chaine est illustrée par la figure 3.13.

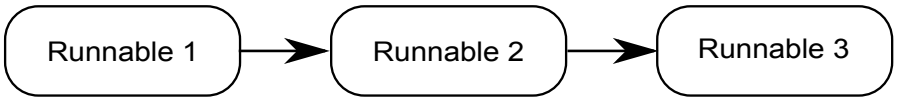


FIGURE 3.13 – Exemple simple d'une chaine de calcul.

Une mise à jour dans notre contexte peut correspondre à deux besoins différents. D'une part, le remplacement d'une fonctionnalité existante par une nouvelle version plus performante : l'*upgrade*. D'autre part, l'ajout de fonctionnalité initialement absente au sein de l'application : dans ce cas, nous parlons d'un *update*. Ainsi, le terme *mise à jour* regroupe les deux types possibles.

Mettre à jour un système a une influence sur trois paramètres :

— la gestion de la mémoire,
— les communications (ces points seront traités plus en détail au Chapitre 5),
— le temps d'exécution des tâches.

Bien que les aspects liés au temps-réel soient détaillés au Chapitre 4, nous décrivons ici les problèmes qui se posent lors de l'ajout de mise à jour.

Les mises à jour sont faites au sein de chaines de calcul existantes et en impactent un ou plusieurs éléments. Le niveau de granularité ici est le **runnable**.

3.5.1 Upgrade

Un *upgrade* peut être composée d'un ou plusieurs runnables, mais la structure de la chaine de calcul n'est pas modifiée. Dans ce cas les flots de données et de contrôles doivent rester inchangés. La figure 3.14 donne un exemple d'*upgrade*. Dans ce cas seul le runnable 2 est remplacé par une autre version. Ainsi, les modifications faites dans le systèmes sont relativement minimes :

— Concernant les communications, les canaux ne sont pas modifiées,
— La nouvelle version du runnable doit être stockée en mémoire, bien que la version précédente ne soit pas nécessairement supprimée
— La majorité des propriétés temps-réel reste identique. Toutefois, le pire temps d'exécution (WCET) du nouveau runnable peut être différent de celui initialement présent dans l'application, ce qui peut poser des problèmes pour l'ordonnancement.

Le WCET du runnable initial est noté C et celui du nouveau runnable C'. Si C' est supérieur à C, cela peut avoir un impact, non seulement sur la tâche τ_i qui contient le runnable 2, mais également sur toutes les tâches de priorité inférieure. Le WCET

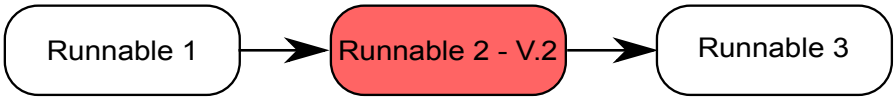


FIGURE 3.14 – Exemple d'upgrade pour une chaine de calcul.

de τ_i augmente donc, et il faut vérifier que le système est toujours ordonnançable (voir chapitre 4).

D'autre part si C' est inférieur à C, intuitivement, il est possible de penser que cela est une bonne chose car cela augmente le *slack time* dans le système. Le *slack time* correspond au temps qui reste libre dans le système après que toutes les tâches ont terminé leur exécution. Diminuer le temps d'exécution d'une tâche pourrait a priori sembler être une bonne chose, mais dans les fait cela peut sérieusement perturber l'ordre d'exécution des autres tâches et donc par conséquent le bon fonctionnement du système.

3.5.2 Updates

Les *updates* sont quelque peu différents car il s'agit alors d'ajouter dans le système un élément qui n'existait pas. L'impact sur le système va alors être plus important.

— Il n'est pas intéressant d'ajouter un runnable qui ne peut pas communiquer avec son environnement. Il est donc nécessaire d'ajouter, en plus du nouveau runnable, un ou plusieurs canaux de communication. Cependant, il faut que les éléments du système, avec lesquels ce nouveau runnable communique, soient en mesure de traiter ces données.

— Concernant la mémoire, il va falloir non seulement stocker cette nouvelle fonctionnalité, mais également les canaux de communication qui vont être ajoutés. De plus, il va falloir ajouter dans la mémoire la nouvelle version des éléments existants qui vont être modifiés pour communiquer avec les éléments qui ont été ajoutés. La taille d'un *update* est donc plus importante que celle d'un *upgrade*.

— Enfin, du point de vue temporel, l'impact pour les runnables existants qui sont modifiés est le même que celui décrit précédemment pour les *upgrades*. Cependant, pour le(s) runnable(s) qui sont nouveau(x), le WCET des tâches dans lesquelles ils vont être ajoutés va nécessairement augmenter. L'étude de l'impact sur l'ordonnancement doit donc être analysé tout particulièrement.

La figure 3.15 montre un exemple pour lequel un nouveau runnable "UPDATE" est ajouté à la chaine de calcul initiale. Il faut alors non seulement ajouter le nouveau runnable, mais également ajouter un canal de communication entre ce runnable et "runnable 2", ainsi que modifier "runnable 2" afin qu'il puisse utiliser la donnée envoyée par "UPDATE".

Du point de vue de l'ordonnancement, il y a alors plusieurs modifications possibles. En fonction de l'emplacement des runnables, plusieurs cas de figure peuvent se présenter : soit "UPDATE" est placé dans la même tâche que celle qui contient "runnable 2", soit il est placé dans une tâche différente. Si les deux runnables sont dans la même tâche, alors seul le WCET de cette tâche est modifié. Il n'y a pas d'impact sur la matrice

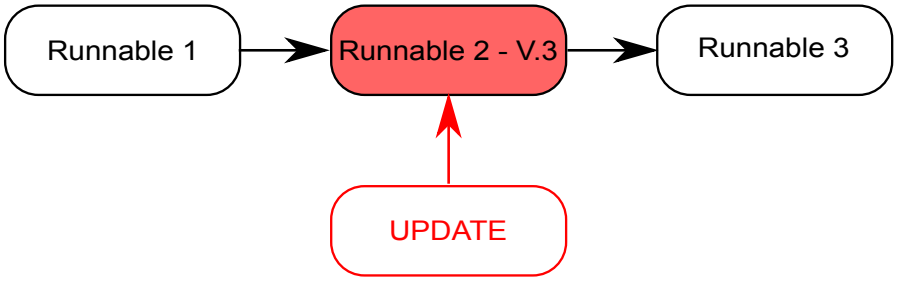


FIGURE 3.15 – Exemple d'update pour une chaine de calcul.

de précédence. Si "runnable 2" et "UPDATE" sont dans des tâches différentes, alors le WCET des deux tâches est modifié et la matrice de précédence peut être modifiée. Dans ce cas une analyse plus poussée concernant l'ordonnancement est nécessaire afin de vérifier également les contraintes de précédence et l'impact de la modification du WCET de deux tâches sur le système global.

4

Modélisation temps-réel et adaptation dans AUTOSAR

Sommaire

4.1 Introduction

Lorsque le système a été modifié afin de pouvoir accueillir les mises à jour d'un point de vue architectural, il faut étudier l'impact de ces mises à jour sur le comportement temps-réel du système, et en particulier s'assurer que les modifications ne perturbent pas l'ordonnançabilité du système. En effet, si le système devenait non ordonnançable suite à l'ajout d'une mise à jour, cela poserait des problèmes majeurs pour la sécurité-innocuité.

Nous supposons ici que les nouveaux éléments sont placés dans des tâches existantes du système d'exploitation. Le processus de mise à jour ne crée donc pas de nouvelles tâches. D'autre part, les systèmes que nous étudions dans le contexte AUTOSAR sont des systèmes monocœurs et l'étude de faisabilité dans le cas de systèmes multicœurs ne rentre pas dans le cadre de ce travail.

D'après l'étude que nous avons pu faire de quelques systèmes embarqués automobiles, nous avons pu constater que la majorité des runnables et des tâches (plus de 70%) sont périodiques (source ingénierie Renault). Pour cette raison, et puisque les tâches évènementielles ont besoin d'éléments de l'OS spécifiques, les mises à jour se concentrent uniquement sur les runnables périodiques du système. Les nouveaux éléments qui peuvent être ajoutés doivent également être périodiques.

L'impact majeur que représente les mises à jour sur le système consiste en une modification du WCET (Worst Case Execution Time) des tâches dans lesquelles les mises à jour sont allouées. Dans le cas d'un ajout de nouvelle fonctionnalité, le temps d'exécution de la tâche augmente. Cependant, dans le cas où une fonctionnalité existante est modifiée, le temps d'exécution de la tâche peut soit être augmenté soit être diminué, en fonction du WCET de la nouvelle fonctionnalité. Une analyse d'ordonnancement est donc nécessaire pour prendre en compte ces variations de temps d'exécution.

4.2 Définitions et modèle de tâche classique

Cette section présente deux définitions clefs, ainsi que les résultats liés à ces concepts. Tout d'abord, la notion de pire temps d'exécution, ou *Worst Case Execution Time*, qui est un élément clef de l'analyse d'ordonnancement temps-réel. Ensuite le concept d'offset pour une tâche, c'est à dire le décalage à l'origine. Ces deux éléments font partie des briques de base de la modélisation temps-réel.

4.2.1 Worst Case Execution Time (WCET)

Le WCET est un élément très important de l'analyse d'ordonnancement. En effet, pour savoir si un système de tâches est ordonnançable, il faut connaitre le pire temps d'exécution pour chacune des tâches. Cependant, déterminer cette valeur n'est pas trivial, et de nombreuses techniques ainsi que des outils spécifiques existent. Le WCET peut être obtenu de deux manières différentes : soit en effectuant une mesure du temps d'exécution du code sur une cible embarquée, soit en effectuant un calcul statique qui détermine le temps nécessaire pour chaque instruction et qui combine ces temps. Cependant, dans ces deux cas, il s'agit d'une estimation du temps d'exécution et non une

valeur exacte. Ce d'autant plus qu'entre deux exécutions d'un même fragment de code, le temps d'exécution peut être différent : par exemple dans le cas d'une instruction "if" si les deux branches nécessitent des temps d'exécution différents.

L'estimation faite pour le WCET est généralement pessimiste. En effet, avoir une estimation optimiste conduirait à sous-estimer les temps d'exécution des éléments et, par conséquent, l'analyse d'ordonnancement qui en résulterait ne serait plus sûre. Toutefois, il est également important de réduire au maximum le pessimisme de l'estimation. Un pessimiste trop important conduirait à une surcharge artificielle du système. C'est pour cette raison que des outils existent et qu'ils visent à réduire le pessimisme.

Du point de vue industriel, l'outil le plus utilisé est AbsInt [2] qui détermine le WCET à partir du code source et pour un couple compilateur/cible matérielle donné. L'outil repose sur une analyse statique du code binaire et prend en compte les caches et pipelines de la cible matérielle. L'objectif est de calculer au plus juste le WCET de chaque élément. Pour cela, la première étape consiste à rechercher le "critical path", c'est-à-dire le chemin d'exécution d'une fonction qui va prendre le plus de temps. Ensuite, pour chaque instruction de ce chemin, et en fonction des caractéristiques du matériel, le temps nécessaire à l'exécution est évalué. L'addition du temps d'exécution de chacune des instruction permet d'avoir une estimation du WCET.

Dans le milieu de la recherche, OTAWA [19] est un outil complet qui permet de déterminer le WCET d'un élément en utilisant différentes techniques. Cela signifie qu'il est possible de comparer les résultats obtenus avec différentes méthodes de détermination de WCET afin d'obtenir un résultat au plus juste.

Enfin, une étude européenne de 2008 [108] propose une vue d'ensemble des différentes techniques et outils qui permettent d'évaluer le WCET.

4.2.2 Offset

Un *offset* correspond au déphasage à l'origine pour une tâche, c'est-à-dire le temps qui s'écoule entre le début de l'exécution (t=0) et la première exécution de la tâche. Il s'agit d'un élément volontairement ajouté lors de la création du système.

Ajouter des offsets dans un système de tâches est très intéressant puisque cela permet de faire du *load balancing*. En effet, si différentes tâches commencent leur exécution à des instants différés dans le temps, le *load balancing* permet de lisser la charge du système au cours du temps. Ainsi, certains systèmes peuvent être non ordonnançables si toutes les tâches commencent à t=0, et devenir ordonnançables si des offsets sont introduits. La figure 4.1 donne un exemple de deux tâches périodiques. La tâche τ_1 a un WCET de 2 et une période de 5 et la tâche τ_2 a un WCET de 4 et une période de 7.5. Dans le cas où il n'y a pas d'offset ajoutés dans le système (à gauche de cette figure), le système de tâches n'est pas ordonnançable. Cependant, si un offset de 2 est ajouté avant la première exécution de la tâche τ_2, alors le système devient ordonnançable.

Toutefois, donner une preuve mathématique que l'ensemble de tâches est ordonnançable en présence d'offset est un problème connu pour être co-NP-complet [84]. Pour cette

2. `http://www.absint.com/`

FIGURE 4.1 – Système de deux tâches non ordonnançable sans offsets (à gauche), et ordonnançable avec offsets (à droite)

raison, une hypothèse très répandue est de considérer que toutes les tâches commencent à t=0.Cette hypothèse repose sur la preuve qui a été faite par Liu et Layland [64] que le pire cas possible pour un ensemble de tâches survient lorsque toutes les tâches démarrent en même temps. Cela signifie que s'il est possible de démontrer que le système est ordonnançable selon cette hypothèse, alors il sera ordonnançable en présence d'offset. Si le système n'est pas ordonnançable si toutes les tâches commencent en même temps, cela ne permet pas de conclure directement pour le cas où des offsets sont présents dans le système.

Malgré la complexité de prise en compte des offset dans les systèmes temps-réels, des travaux existent pour calculer le temps de réponse de ces systèmes [72] [80].

4.2.3 Contraintes de précédence

Cette section traite des contraintes de précédence, c'est-à-dire des relations qui peuvent exister entre les différentes tâches d'un système. Il peut, par exemple, s'agir d'une relation de type flot de donnée pour laquelle une tâche ne doit pas commencer son exécution tant que des données produites par une autre tâche ne sont pas disponibles. Nous traitons ici le cas général des contraintes de précédence et nous appliquerons plus spécifiquement ces concepts à des systèmes de type AUTOSAR ultérieurement.

Dans un premier temps, nous donnons les définitions liées aux contraintes de précédence, ainsi qu'à la modélisation de ces contraintes en utilisant des concepts issus de la théorie des graphes. Nous décrivons ensuite différents moyens de prendre en compte ces relations en balayant un certain nombre de travaux existant.

4.2.3.1 Définitions et modélisation

Dans un système temps réel, il est assez rare d'avoir des tâches totalement indépendantes les unes des autres. Il arrive qu'une tâche τ_i ne puisse pas commencer son exécution tant que la tâche τ_j n'a pas terminé son exécution. Le concept de précédence a été défini par Mangeruca *et al.* [68] par "any job τ_{ik} precedes a job τ_{jh} if the latter cannot start until τ_{ik} has finished". Cela doit permettre d'avoir des communications déterministes entre les différentes tâches du système [96] et correspond dans notre cas à des dépendances de données entre les tâches : la donnée produite par τ_i doit être disponible avant que τ_j ne puisse commencer son exécution [44]. Dans cette approche, cela signifie que les données sont de type “last is best”, à savoir que seule la dernière valeur produite

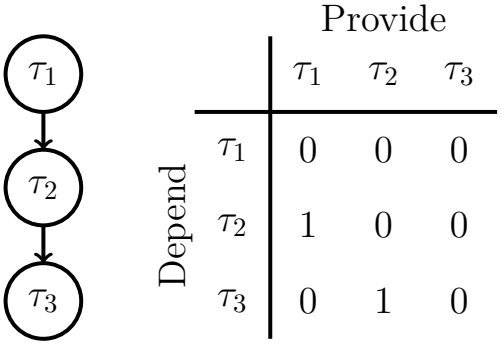


FIGURE 4.2 – Exemple de graphe de précédence pour trois tâches et matrice d'adjacence associée

par la tâche nous intéresse. Le cas où les données produites par plusieurs exécutions successives d'une même tâche sont nécessaires à l'exécution de la tâche suivante n'est pas considéré. Cela ne correspond pas aux cas pratiques que nous avons pu observer dans le domaine de l'automobile.

Les contraintes de précédence entre les tâches peuvent être représentées en utilisant un graphe orienté noté G(E,V). La théorie des graphes permet également d'associer à chaque graphe une matrice d'adjacence. Dans le cas d'un graphe ayant N sommets, la matrice d'adjacence associée est une matrice carrée de taille NxN. Cette matrice est composée de 1 qui correspondent à un arc entre deux sommets et de 0 qui indiquent que les sommets sont indépendants. La figure 4.2 présente un exemple de graphe de précédence et de matrice de précédence associée pour trois tâches. Ici τ_1 doit s'exécuter avant τ_2 qui doit elle-même s'exécuter avant τ_3.

Il est important de noter que le graphe de précédence pour un système donné doit absolument être acyclique pour que le système puisse être ordonnancé en respectant les contraintes de précédence. Si le graphe n'est pas acyclique, alors le système n'est pas causal et il n'est alors pas possible de trouver un ordre d'exécution qui respecte toutes les contraintes de précédence [44].

4.2.3.2 État de l'art

Le problème de précédence est difficile. Il a été prouvé par Baruah *et al.* [21] que ce type de problème ne peut pas être résolu de manière efficace sur un processeur simple (à moins de prouver que P=NP) et est au coeur de nombreuses recherches [36, 40, 44, 68].

Chetto *et al.* [36] proposent un test qui permet de décider si un groupe de tâches sporadiques arrivant à un moment quelconque dans le temps peut, ou non, être exécuté. Le système est décomposé en tâches périodiques indépendantes et en tâches sporadiques ayant des contraintes temporelles et de précédence. Ce travail utilise un algorithme de type EDF (Earliest Deadline First) qui est un algorithme d'ordonnancement dynamique.

Dans [40], Cucu *et al.* présentent un algorithme permettant de déterminer si un système ayant des contraintes de précédence, de latence et de périodicité peut être or-

donnancé. Dans ce cas, la latence entre deux tâches τ_i et τ_j est définie par le temps maximum qui peut s'écouler entre le début de l'exécution de τ_i et le début de l'exécution de τ_j, sachant que τ_i doit avoir terminé son exécution avant le début de τ_j. Cette notion n'a d'intérêt que s'il existe une relation de précédence entre les tâches.

Afin d'éviter les deadlocks ou les problèmes d'ordonnancement, Forget *et al.* [44] proposent une méthode d'ordonnancement de tâches dépendantes sans mécanisme de synchronisation. L'idée repose sur une politique d'ordonnancement de type Deadline Monotonic (la tâche ayant la deadline la plus petite a la priorité la plus grande) et ajuste la deadline de la tâche en fonction des contraintes de précédence.

Une condition nécessaire et suffisante pour ordonnancer un système ayant des contraintes de précédence est fournie par Mangeruca *et al.* dans [68]. Cette contrainte mathématique est ensuite traduite pour les algorithmes de type FP (Fixed Priority) ou EDF.

Dans le cas où les dépendances entre les taches sont de type flot de données, il a été défini qu'une dépendance de données entre deux tâches τ_i et τ_j signifie que τ_j a besoin d'une donnée produite par τ_i pour pouvoir s'exécuter correctement [44].

Une méthode largement utilisée pour résoudre les problèmes liés aux contraintes de précédence consiste à utiliser des mécanismes de synchronisation comme les sémaphores ou les mutex. Cependant, l'inconvénient de cette technique est l'ajout de délais supplémentaires dans l'exécution. Dans le cas d'ordonnancement à priorité fixe, il est également possible de résoudre le problème en utilisant les priorités des tâches. La tâche produisant la donnée doit avoir une priorité plus grande que celle qui la consomme. De cette manière la tâche productrice s'exécute toujours avant la tâche consommatrice (sous réserve qu'il n'y a pas de problème de période).

4.2.4 Modèle de tâches classique

Un système embarqué temps-réel de type AUTOSAR peut être représenté de la même manière que n'importe quel autre système embarqué temps réel. Il existe deux types de tâches : les tâches périodiques qui sont exécutées à intervalle de temps fixé, et les tâches sporadiques qui sont exécutées lorsqu'un évènement particulier se produit. Les *tâches sporadiques* sont assez compliquées à prendre en compte, car, par essence, il n'est pas aisé de prévoir à quels moments l'évènement qui les déclenche arrive. Pour faciliter l'analyse d'ordonnancement, une méthode couramment employée pour gérer les tâches sporadiques est de considérer que ce sont des tâches périodiques ayant une période égale au temps minimum entre deux arrivées de l'évènement déclenchant la tâche sporadique. Cette approche est, certes, pessimiste, mais elle permet d'assurer l'ordonnançabilité du système.

Les *tâches périodiques* peuvent être représentées par quatre paramètres :

— leur offset Φ, *i.e.* le temps qui s'écoule avant la première activation de la tâche,
— leur pire temps d'exécution (WCET, Worst Case Execution Time) noté C, *i.e.* le temps maximal possible pour une exécution de la tâche,
— leur période T et
— leur deadline D, *i.e.* le temps dont la tâche dispose pour terminer son exécution à chaque fois qu'elle est activée.

Une tâche peut donc être représentée par le quadruplet (ϕ, C, T, D). L'exécution de chaque tâche périodique est composée d'une succession de *job*s. Un *job* correspond à une exécution de la tâche. Le premier job de la tâche τ_i est noté τ_{i1} et est exécuté à t=Φ_i. Les jobs suivants τ_{ik} de la tâche sont exécutés à t=$\Phi_i + kT_i$.

4.3 Analyse d'ordonnancement et de sensibilité

Il existe un certain nombre de travaux existant sur les aspects temps-réel, et en particulier sur l'analyse d'ordonnancement dans les systèmes à priorité fixe. Dans une première partie, nous présentons tout d'abord des résultats permettant de faire une analyse d'ordonnancement, et nous nous concentrons tout particulièrement sur l'analyse proposée par Bini et Buttazzo [26]. Cela nous permet ensuite de présenter l'analyse de sensibilité, c'est-à-dire comment le système se comporte face à la modifications de certains de ses paramètres.

4.3.1 Analyse d'ordonnancement

Un facteur qu'il est intéressant de prendre en compte lorsqu'un système temps-réel est considéré est le "facteur d'utilisation". Le facteur d'utilisation est défini par $U_i = C_i/T_i$ pour chaque tâche et le facteur d'utilisation total pour les N tâches du système par $U_p = \sum_{i=1}^{N} U_i$. Il a été montré que dans le cas d'une politique d'ordonnancement de type "Rate Monotonic", une condition suffisante (mais non nécessaire) pour que un ensemble de n tâches soit ordonnançable est d'avoir $U_p \leq n(2^{1/n} - 1)$ [64], si et seulement si les hypothèses suivantes sont vérifiées

1. toutes les tâches ont un offset de zéro,
2. les deadlines sont égales aux périodes,
3. les tâches sont indépendantes

La limite du facteur d'utilisation lorsqu'il y a un grand nombre de tâches dans le système est de 69%. Il est néanmoins possible d'avoir un système ordonnançable si le facteur d'utilisation est plus important. En effet, il est tout à fait possible d'avoir un système chargé à 90% et utilisant un algorithme "Rate Monotonic" qui soit ordonnançable [62].

Il existe un grand nombre de méthodes qui permettent de faire une analyse d'ordonnancement pour un système de tâches temps-réel et de prendre en compte plus ou moins de contraintes. Nous allons nous concentrer ici uniquement sur une méthode proposée par Bini et Buttazzo [26], car c'est sur cette méthode que repose l'analyse de sensibilité (voir section 4.3.2) que nous utilisons pour vérifier qu'une mise à jour est possible. Cette dernière est particulièrement intéressante lorsqu'il s'agit d'ajouter de nouvelles fonctionnalités dans les tâches ou bien d'évaluer la flexibilité temps-réel dont nous disposons dans le système. Dans le cas où la charge est importante la méthode proposée dans [26] permet de déterminer si le système est ordonnançable ou non en un temps raisonnable.

Définition 1 *Soit τ_i une tâche quelconque de l'ensemble de tâches Γ_n. T_i est la période de τ_i, D_i sa deadline et C_i son WCET.*

La région $\mathbb{M}_n$ qui contient toutes les valeurs possibles pour les WCET des tâches de l'ensemble de tâches Γ_n est définie telle que :
$\mathbb{M}_n(T_1, ..., T_n, D_1, ..., D_n) = \{C_1, ..., C_n \in \mathbb{R}_+^n : \Gamma_n$ *est ordonnançable avec un algorithme à priorité fixe}*

Cette région de l'espace délimite toutes les valeurs possibles pour les temps d'exécution de chacune des tâches telles que le système reste ordonnançable. Les auteurs ont alors proposé un théorème afin de déterminer de manière plus précise cette région :

Théorème 1 *[26] La région des systèmes ordonnançables $\mathbb{M}_n$ est définie par* $\mathbb{M}_n(T_1, ..., T_n, D_1, ..., D_n) = \{C_1, ..., C_n \in \mathbb{R}_+^n :$

$$\bigwedge_{i=1..n} \bigvee_{t \in \mathcal{P}_{i-1}(D_i)} C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j \leq t \ \}$$

où $\mathcal{P}_i(t)$ est l'ensemble des points d'ordonnancement tels que :

$$\begin{cases} \mathcal{P}_0(t) = t \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right) \bigcup \mathcal{P}_{i-1}(t) \end{cases}$$

Le *C-space* correspond à l'espace dans lequel chaque coordonnée est le WCET des tâches. Un point dans cet espace correspond à un ensemble de valeurs de WCET pour chacune des tâches de l'ensemble considéré. Le Théorème 1 permet de déterminer une région dans cet espace contenant tous les points pour lesquels le système est ordonnançable. D'un point de vue des mises à jour dynamiques du système, déterminer la distance entre le point correspondant au système actuel et les limites de cet espace permet de connaitre la flexibilité disponible dans le système pour les mises à jour. La figure 4.3 montre un exemple de *C-space* pour un système de 3 tâches (cet exemple est tiré de [26]). Chacune des inéquations obtenues avec le Théorème 1 correspond à un demi espace et l'assemblage de l'ensemble des inéquations nous permet d'obtenir une représentation visuelle de l'espace des WCET possibles pour le système de tâches.

Bien que cette méthode d'évaluation pour l'ordonnançabilité d'un système est initialement conçue pour des ordonnancement de type RMS (Rate Monotonic Scheduling), il est possible de l'utiliser pour n'importe quel système à priorité fixe. L'inconvénient majeur de cette approche dans le contexte qui nous intéresse, est lié à l'hypothèse que toutes les tâches sont indépendantes. Il n'est donc pas possible de prendre en compte les contraintes de précédence en appliquant directement cette méthode. Toutefois, il est nécessaire de prendre en compte ces contraintes lors de l'analyse d'ordonnancement. Il faut donc soit traiter ces contraintes séparément, soit utiliser une autre approche qui permette de les prendre en compte.

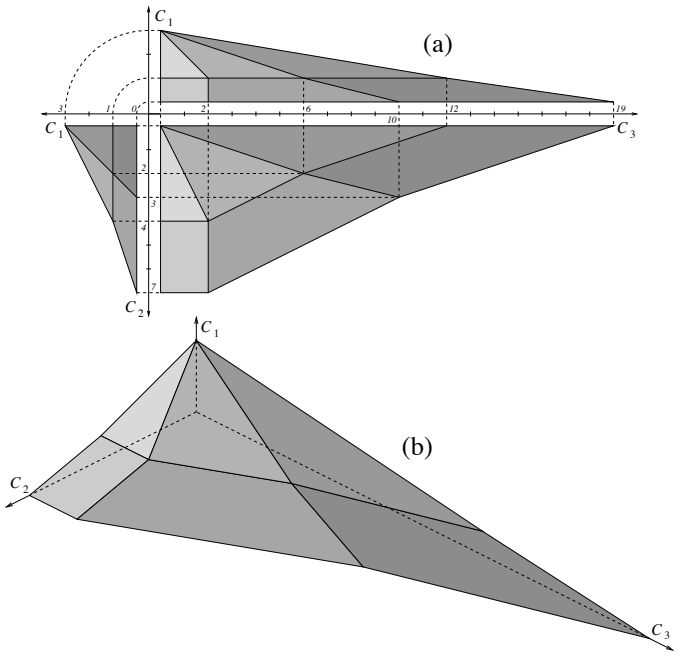


Fig. 6. The schedulability region when $D_i < T_i$: (a) the projection view and (b) the isometric view.

FIGURE 4.3 – Exemple de *C-Space* extrait de[26]

4.3.2 Analyse de sensibilité

En utilisant les résultats de leur analyse d'ordonnancement présentée dans la section 4.3.1, Bini *et al.* ont été plus loin en définissant une manière de faire une analyse de sensibilité [27]. Les auteurs définissent ici une méthode qui permet de déterminer si un système reste ordonnançable dans le cas où les paramètres d'une ou plusieurs tâches varient. Ces paramètres sont le WCET de la tâche ou sa période (on ne peut faire varier qu'un des deux paramètres). Cela peut s'avérer utile, par exemple, lorsque le système est porté sur une cible matérielle différente. L'intérêt tout particulier de cette technique est qu'elle évite de refaire une analyse d'ordonnancement complète.

L'élément de base utilisée par l'analyse de sensibilité est présentée par le Théorème 1 qui constitue une condition nécessaire et suffisante pour déterminer l'ordonnançabilité du système. À partir de ce test, Bini et Buttazzo ont défini le facteur d'élasticité pour les tâches [27]. Ce facteur d'élasticité peut être défini soit pour la période, soit pour le WCET des tâches. Nous nous intéressons ici uniquement au facteur d'élasticité pour le WCET. Chacun des temps d'exécution de chacune des tâches d'un ensemble de tâches Γ_n peut être écrit sous la forme d'un vecteur noté $\mathbf{C}$. La variation des temps d'exécution entre le système initial et le système modifié peut être notée sous la forme [27] :

$$\mathbf{C}' = \mathbf{C} + \lambda \mathbf{d} \tag{4.1}$$

avec $\mathbf{d}$ un vecteur non négatif qui correspond à la direction dans laquelle le système est modifié et λ une mesure de la variation imposée sur le système initial. Dans le cas où seul le WCET de la $i^{ème}$ tâche du système est modifiée, alors $\mathbf{d} = (0, ...0, 1, 0, ..., 0)$: le seul "1" correspond au $i^{ème}$ élément du vecteur. Les i premiers éléments du vecteur $\mathbf{d}$ sont notés $\mathbf{d}_i$ [27]. L'altération tolérable entre le système initial et le système modifié, selon la direction $\mathbf{d}$ peut être quantifiée par :

$$\lambda^{max} = \min_{i=1,...,N} \quad \max_{t \in \mathcal{P}_{i-1}(D_i)} \frac{t - \mathbf{n}_i \cdot \mathbf{C}_i}{\mathbf{n}_i \cdot \mathbf{d}_i} \tag{4.2}$$

avec $\mathbf{n_i} = \left(\left\lceil \frac{t}{T_1} \right\rceil, \left\lceil \frac{t}{T_2} \right\rceil, ..., \left\lceil \frac{t}{T_{i-1}} \right\rceil, 1 \right)$.

Le coefficient λ^{max} permet également de déterminer si le système est ordonnançable ou non : si λ^{max} est positif, alors le système est ordonnançable, sinon, il ne l'est pas. Dans le cas où le coefficient est négatif, cela permet de quantifier les modifications à faire dans la direction $\mathbf{d}$ pour que le système devienne ordonnançable.

Dans le cas où la direction de modification ne contient qu'un seul 1 à la $k^{ième}$ position, le WCET d'une seule tâche est changé. Il est alors possible de noter ΔC_k^{max} la variation maximale possible du temps d'exécution de la $k^{ième}$ tâche. A partir de l'équation 4.2, nous obtenons [27] :

$$\Delta C_k^{max} = \min_{i=k,...,N} \quad \max_{t \in \mathcal{P}_{i-1}(D_i)} \frac{t - \mathbf{n}_i \cdot \mathbf{C}_i}{\lceil t/T_k \rceil} \tag{4.3}$$

Intuitivement, les facteurs ΔC^{max} pour chacune des tâches représentent la distance entre le point d'ordonnancement actuel dans le *C-space* et ses frontières. Cependant pour pouvoir appliquer ces formules, il faut respecter un certain nombre de conditions. Tout d'abord, toutes les tâches doivent être des tâches périodiques et indépendantes. L'indépendance est un critère qui n'est quasiment jamais respecté, en particulier dans le domaine automobile. En effet, une tâche qui ne communique pas est *a priori* inutile. D'autre part, les tâches de l'ensemble Γ_n doivent être triées dans l'ordre de priorité, et toutes les tâches doivent avoir une priorité différente. Cette hypothèse n'est pas si restrictive, car éviter d'avoir des tâches de priorité identiques permet d'assurer un meilleur déterminisme à l'exécution. Toutefois, si l'algorithme d'ordonnancement utilisé est RMS strict et que deux tâches ont la même période, il est alors possible d'avoir des tâches de même priorité. Enfin, cette méthode ne prend pas en compte les éventuels offsets qui peuvent être présents dans l'application.

Malgré ces restrictions importantes, l'analyse de sensibilité apporte une réponse très pertinente au problème de variation de WCET que peut poser la mise à jour d'un système. Pour cette raison, nous étudions ici les possibilités d'application de cette méthode tout en utilisant d'autres moyens pour prendre en considération les restrictions qu'elle impose.

4.4 Modèle de tâches et AUTOSAR

La modélisation du système se fonde principalement sur le modèle de tâches associé au système (c'est-à-dire comment sont représentées les tâches), ainsi que sur les contraintes de précédence qui sont indépendantes du modèle de tâche lui-même. Les contraintes de précédence ont un rôle concernant l'ordre d'exécution de certaines tâches dans le système. Cela peut, par exemple, être lié à des dépendances de données ou de flots de contrôle. Ces informations se retrouvent dans le modèle fonctionnel global présenté dans la section 3.3.1.

Ces différents aspects sont très importants pour pouvoir par la suite effectuer une analyse d'ordonnançabilité sur le système.

4.4.1 Contraintes de précédence, tâches et runnables

Dans cette section nous nous intéressons tout particulièrement aux contraintes de précédence qui, initialement, sont exprimées pour les runnables et non pour les tâches. Les contraintes de précédence pour les tâches vont donc grandement dépendre de l'allocation qui a été faite des runnables dans les tâches applicatives.

4.4.1.1 Priorités et précédences

L'une des contraintes que nous avons dans le cadre d'AUTOSAR est que l'ordonnancement doit être à priorités fixes. Un ordonnancement de type Rate Monotonic est considéré comme optimal (puisque si un ensemble de tâches ne peut pas être ordonnancé avec RMS, il ne peut pas non plus être ordonnancé avec un autre algorithme à priorité fixe) [64], nous considérons donc que c'est cet algorithme qui est utilisé. Au sein de Renault, il n'est pas forcément aisé d'avoir des retours d'expérience concernant les politiques d'ordonnancement puisque ces étapes de conceptions sont majoritairement sous-traitées aux équipementiers.

Dans le cadre d'un ordonnancement de type RMS strict, il est possible d'avoir des tâches avec la même priorité. Cette dernière est déterminée uniquement en fonction de la période des tâches : plus la période de la tâche est faible, plus la priorité est grande. Cependant afin d'avoir un meilleur déterminisme lors de l'exécution, il est préférable d'éviter d'avoir deux tâches avec la même priorité. Pour cette raison, nous supposons que toutes les tâches ont des priorités différentes. Dans le cas où plusieurs tâches sont à la même période, chaque période est associée à une plage de priorité. Le principe de RMS est tout de même conservé : plus la période est faible, plus la plage de priorité correspondante est élevée. En outre, cette hypothèse permet également de respecter une des hypothèses de l'analyse de sensibilité.

Les priorités qui vont être allouées aux tâches qui ont la même période dépendent également des contraintes de précédence. Pour garantir le respect de ces contraintes, seules les tâches avec une priorité plus élevée peuvent envoyer des données aux tâches de priorité inférieure. Ainsi, les contraintes de précédence sont uniquement gérées en utilisant le graphe de précédence et la matrice associée. Pour que le système respecte

les contraintes de précédence, grâce aux priorités, il faut que la matrice d'adjacence soit triangulaire inférieure (à condition que les tâches de l'ensemble soient ordonnées par ordre de priorité avec la plus prioritaire en premier).

4.4.1.2 Des contraintes de précédence sur les runnables aux contraintes sur les tâches

Dans la théorie de l'ordonnancement temps-réel, les contraintes de précédence sont généralement directement liées aux tâches et définissent un ordre partiel d'exécution entre elles. Bien que ces contraintes de précédence puissent être liées à des contraintes diverses, nous ne considérons ici que des contraintes liées aux échanges de données entre les éléments du système.

Or, nous avons défini au Chapitre 3 que le modèle fonctionnel logiciel global définit des communications entre les runnables et non entre les tâches. Ces runnables sont cependant ensuite alloués dans des tâches de l'OS pour être exécutés. Les dépendances de données entre tâches sont donc directement héritées des dépendances de données entre les runnables et de l'allocation qui a été faite.

Il est tout à fait possible d'appliquer les notions de graphe de précédence et de matrice d'adjacence, non pas directement à des tâches mais aux runnables. Ensuite, à partir de l'allocation qui a été faite des runnables sur les tâches, le graphe de précédence pour les tâches peut être déduit. Cela peut être fait en utilisant un algorithme très simple décrit par l'algorithme 1 et se fonde sur la matrice d'adjacence pour les runnables, et sur les tâches du système et les runnables qu'elles contiennent. Chacune des tâches τ_i de l'ensemble Γ_N est représentée un 5-uplet $(\Phi_i, C_i, T_i, D_i, Run_i)$ où Run_i est le tableau de tous les runnables alloués dans la tâche, dans l'ordre de leur exécution. Dans cette représentation, le WCET de τ_i est considéré comme étant la somme de tous les WCET de chacun des runnables qu'elle contient. Cela signifie que la tâche est uniquement un support d'exécution pour les runnables et qu'elle n'a pas d'autre contenu qui lui serait propre.

Sachant que le nombre de runnables d'une application est nécessairement plus grand ou égal au nombre de tâches applicatives du système, si la matrice de précédence pour les runnables est de taille PxP, alors P$\geq$N. Le pire cas possible est obtenu quand il n'y a qu'un seul runnable par tâche. Cependant, ce n'est généralement pas le cas puisqu'une application peut contenir plus de 200 runnables, alors que le nombre de tâches excède rarement 100 pour des raisons d'efficacité [25].

4.4.1.3 Allocation des runnables sur les tâches

AUTOSAR permettant d'avoir des runnables à des périodes différentes au sein d'une même tâche, cela veut dire que le WCET d'une telle tâche peut varier significativement entre deux exécutions. Dans ce cas, prendre le pire cas possible pour C peut s'avérer extrêmement pessimiste : un système qui serait ordonnançable dans la pratique en prenant en compte les variations de temps d'exécution au sein de la tâche peut être considéré par le calcul comme non ordonnançable dans le cas où l'hypothèse prise est celle du pire

Algorithm 1: Construction de la matrice de précédence pour l'ensemble de tâches Γ_N

```
Data: Runnables_Precedence_Matrix, Γ_N
Result: Tasks_Precedence_Matrix
∀ (α, β) Tasks_Precedence_Matrix(α, β) = 0 ;
for All elements (i,j) of Runnables_Precedence_Matrix do
    if Runnables_Precedence_Matrix(i,j) = 1 then
        for All Tasks τ_k ∈ Γ_N do
            if Runnable_i ∈ τ_k then
                precede = k
            end
            if Runnable_j ∈ τ_k then
                depend = k
            end
        end
        Tasks_Precedence_Matrix(precede,depend)=1;
    end
end
```

cas possible. Par exemple, pour un ensemble de deux tâches τ_1 et τ_2. La tâche τ_1 a un pire temps d'exécution qui varie : pour le premier job $C_1 = 2$ puis pour le second $C_1 = 3$ puis à nouveau $C_1 = 2$, etc. Si nous considérons uniquement le WCET, alors nous prenons la plus grande valeur possible pour C_1, à savoir $C_1 = 3$. Ainsi le système de tâches peut être décrit par : $\{\tau_1 : (0, 3, 5, 5), \tau_2 : (0, 5, 10, 10)\}$. Le facteur d'utilisation total pour ce sysème, tel qu'il a été défini par Liu et Layland [64], est $U_p = 110\%$. Ce facteur permet de déterminer le pourcentage d'utilisation du processeur. Un pourcentage supérieur à 100 signifie que le processeur est surchargé et ne pourra pas exécuter correctement toutes les tâches. Ainsi, si seul le pire cas possible est considéré, alors le système n'est pas ordonnançable.

Si nous considérons maintenant un modèle dans lequel la variation de temps d'exécution entre deux jobs de la tâche τ_1 peut être pris en compte, alors le système devient ordonnançable. La figure 4.4 présente les deux cas. A gauche, dans le cas où seul le WCET est considéré, il est clair que le système n'est pas ordonnançable puisque la tâche τ_2 n'a pas suffisamment de temps pour s'exécuter. A droite, lorsque la variation des temps d'exécution est prise en compte, le système devient alors ordonnançable.

Pour pallier ce problème, Mok et Chen ont introduit le concept de *Multiframe Task Model* [70]. Les différents jobs d'une tâche peuvent alors avoir des temps d'exécution différents en fonction de leur position dans un schéma d'exécution pré-défini. Cela permet une analyse d'ordonnançabilité plus précise. Chaque tâche est alors représentée par un couple (Γ, P), où P correspond au temps minimum entre deux jobs et Γ représente le tableau de tous les temps d'exécution possibles pour la tâche (le schéma d'exécution). Suivant cette représentation, la tâche τ_1 donnée en exemple ci-dessus pourrait être re-

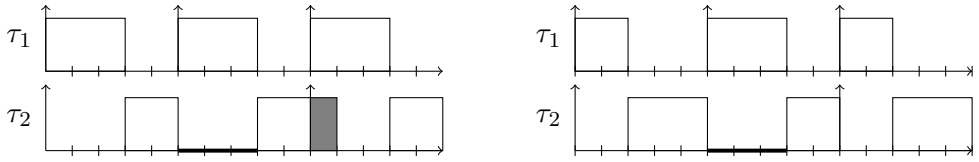


FIGURE 4.4 – Chronogramme des deux tâches. A gauche le cas où le WCET pessimiste est considéré. A droite le cas où la variation de temps d'exécution est considérée.

présentée par ({2,3},5). La définition donnée pour P permet également de prendre en compte les tâches sporadiques.

Cependant, bien que cette pratique soit permise par le standard, nous ne considérons pas que ce soit une bonne pratique pour plusieurs raisons. Tout d'abord, si l'algorithme d'ordonnancement choisi est RMS, alors la priorité doit uniquement dépendre de la période. Or si des runnables possèdent des périodes plus grandes que les tâches dans lesquelles ils sont placés, il peut y avoir des inversions de priorité. Par exemple, dans le cas de deux tâches τ_1 à 10 ms et τ_2 à 20 ms et deux runnables R_1 à 30 ms et R_2 à 20 ms, R_2 doit avoir une priorité plus grande que R_1 puisque la période de R_2 est la plus petite. Cependant si R_1 est placé dans τ_1 et ne s'exécute qu'une fois sur trois, alors il pourra pré-empter R_2 placé dans τ_2.

D'autre part, permettre à un runnable de ne pas s'exécuter à chaque activation de la tâche nécessite de rajouter du code "glue" spécifique dans le corps de la tâche. Cette opération est faite lors de la génération du RTE. Toutefois, pour permettre des mises à jour, ceci n'est pas possible, à moins d'effectuer des mises à jour au niveau des tâches et non plus au niveau des runnables. Cela forcerait donc à faire des mises à jour à un niveau de granularité plus grand.

Pour ces raisons, nous formulons l'hypothèse que, dans un système qui a vocation à être mis à jour, les runnables doivent nécessairement être placés dans des tâches ayant la même période qu'eux.

4.4.2 Modèle de tâche dans le cadre d'AUTOSAR

Afin de pouvoir appliquer l'analyse de sensibilité aux futures mises à jour dans le système, il est nécessaire de poser un certain nombre d'hypothèses concernant la conception du système. De plus, lorsque les runnables sont alloués sur les tâches, les dépendances de données doivent être prises en compte. Cette section présente donc les hypothèses concernant les différents paramètres qui interviennent dans l'analyse d'ordonnancement, ainsi que le modèle final prenant en compte ces hypothèses.

Étant donné la complexité de la prise en compte des offsets dans l'analyse d'ordonnancement, nous considérons que les offsets sont nuls pour faire notre analyse. Cette hypothèse est grandement répandue dans la théorie de l'ordonnancement temps-réel. Cela permet également de répondre à l'une des hypothèses de l'analyse de sensibilité.

Dans un système de type AUTOSAR, les deadlines sont liées à des contraintes de bout-en-bout sur les chaines de calcul : entre le début de l'exécution du premier run-

nable de la chaine et la fin de l'exécution du dernier, le temps écoulé doit être borné, avec une borne définie par l'architecture logicielle. Cette borne comprend non seulement les temps d'exécution des runnables de la chaine, mais également les éventuels temps de latence de transmission des données, ainsi que les délais d'attente pour l'exécution (si d'autres éléments de priorité supérieure doivent s'exécuter avant). Ainsi, il n'est pas possible d'allouer des deadlines sur les runnables d'une chaine de calcul en sous-divisant la contrainte de bout-en-bout. Il est en effet très compliqué de trouver un critère approprié pour lier directement les contraintes de bout-en-bout aux runnables. En effet, cela reviendrait à avoir des deadlines pour les runnables et non directement pour les tâches, ce qui poserait un nouveau problème : comment traduire les deadlines des runnables en deadlines pour les tâches. Pour toutes ces raisons, nous considérerons que les deadlines des tâches sont égales aux périodes et que des mécanismes de sécurité-innocuité séparés (comme par exemple des watchdogs) sont ajoutés dans le système afin de surveiller les contraintes de temps de bout-en-bout (vérification en ligne de propriété de sécurité [39]).

Le modèle de tâche qui nous intéresse et que nous utilisons dans le cadre d'AUTOSAR est considérablement simplifié par rapport au modèle général présenté dans la section 4.2.4. L'ensemble des N tâches qui composent le système est appelé Γ_N et les tâches doivent y être ordonnées par ordre décroissant de priorité. $\Gamma_N = \{\tau_1, \tau_2, ..., \tau_n\}$ avec τ_1 la tâche avec la priorité la plus élevée et τ_n la tâche avec la priorité la plus faible. De plus, les tâches de Γ_N ont toutes des priorités différentes afin d'avoir un meilleur déterminisme et de répondre aux hypothèses de l'analyse de sensibilité.

Le modèle initial représente les tâches avec quatre paramètres : (Φ, C,T,D). Nous considérons ici que l'offset de toutes les tâches est nul, *i.e.* $\Phi = 0$ et que la deadline d'une tâche est égale à sa période, *i.e.* D = T. Ces deux paramètres peuvent donc être exclus car ils sont soit déjà connus, soit redondants. Concernant le WCET des tâches, il est égal à la somme de tous les WCET des runnables alloués dans la tâche. Cela signifie donc que l'ensemble des runnables alloués dans une tâche doit être ajouté au modèle. Nous notons Run_i l'ensemble des runnables alloués dans la tâche τ_i, R_k le k^{ième} runnable de l'ensemble Run_i et C_{R_k} le WCET du runnable R_k. Le modèle final est donné par :

$$\begin{aligned}
&\Gamma_N = \{\tau_1, \tau_2, ..., \tau_n\} \\
&\tau_i = (T_i, Run_i) \\
&Run_i = (R_1(C_{R_1}), R_2(C_{R_2}), ..., R_j(C_{R_j})) \\
&C_i = \sum_{l=1}^{j} C_{R_l} \\
&M[\Gamma_N]
\end{aligned} \tag{4.4}$$

où $M[\Gamma_N]$ représente la matrice de précédence pour le système de tâches ordonnées par priorité.

4.5 Mise à jour, AUTOSAR et temps-réel

Après avoir donné tous les concepts qui permettent de donner un modèle du système ainsi que les différents résultats issus de travaux précédents qui sont nécessaires pour déterminer si ce système est ordonnançable, nous décrivons dans cette section les différentes étapes pour faire une mise à jour.

Tout d'abord il faut définir la mise à jour, à la fois du point de vue structurel et temporel. Lorsque les éléments qui doivent être ajoutés sont clairement identifiés, il faut vérifier que les contraintes de précédences associés à ces éléments peuvent être vérifiées. Ce processus peut être itératif dans le cas où la mise à jour peut être placée à plusieurs endroits de l'application. Enfin, il faut procéder à une analyse de sensibilité afin de vérifier que le système avec la mise à jour est toujours ordonnançable.

4.5.1 Caractéristiques de la mise à jour

Les mises à jour doivent être atomiques, c'est-à-dire qu'une mise à jour ne doit correspondre qu'à une seule fonctionnalité. Lorsque la mise à jour est complètement définie, grâce au processus de développement, le chargement peut avoir lieu de manière différentielle dans les containers qui ont été ajoutés lors de la préparation de l'application initiale. L'allocation qui est faite des nouveaux runnables suit les mêmes règles que celles pour l'allocation initiale. Étant donné que les mises à jour sur lesquelles nous travaillons dans le cadre de cette thèse sont des mises à jour applicatives, les nouveaux runnables ne peuvent être ajoutés que dans des tâches applicatives. Les tâches permettant d'exécuter des modules du Basic Software ne seront donc jamais modifiées au cours du processus de mise à jour.

Une considération importante est que les runnables placés dans une tâche donnée doivent obligatoirement avoir la même période que cette dernière. Cette hypothèse doit être respectée lors de l'allocation des mises à jour dans les tâches applicatives du système.

4.5.2 Contraintes de précédence

Lorsqu'une mise à jour est intégrée au système, les contraintes de précédence associées à cette dernière doivent être prises en compte (à savoir les communications). L'ajout d'un runnable qui ne communique pas avec son environnement est considéré comme inutile. Toutefois si la mise à jour consiste uniquement à modifier un (ou plusieurs) runnable(s) existant dans l'application, seule l'analyse de sensibilité est nécessaire car les WCET peuvent changer. En effet, dans ce cas, si aucun canal de communication n'est ajouté, alors les contraintes de précédence restent identiques. Ainsi, vérifier le respect des contraintes de précédence n'a d'intérêt que si de nouvelles communications sont ajoutées dans le système.

Il est alors nécessaire de déterminer la matrice d'adjacence pour les runnables à partir du nouveau modèle fonctionnel logiciel global. À ce stade, la matrice d'adjacence pour les tâches peut être partiellement remplie avec les runnables qui ne sont pas modifiés et qui sont déjà placés dans des tâches du système. Ensuite plusieurs cas peuvent se présenter.

Soit chacun des nouveaux runnables ne peut être placé que dans une seule tâche ; dans ce cas, la nouvelle matrice d'adjacence pour les tâches peut être directement déterminée. Soit un ou plusieurs runnables peuvent être placés à différents endroits ; dans ce cas, il peut être nécessaire de déterminer l'emplacement optimal pour ces runnables. L'objectif est alors 1) de garder une matrice de précédence triangulaire inférieure (critère le plus important) 2) d'essayer de réduire cette matrice (critère secondaire).

Afin de minimiser la matrice de précédence, il est possible de placer un runnable qui envoie une donnée dans la même tâche que celui qui la reçoit (dans le cas où les autres critères d'allocation sont respectés), si l'ordre d'exécution est correct (celui qui émet s'exécute avant celui qui reçoit). Une seconde possibilité est de placer le runnable de sorte à ne pas modifier la matrice. Par exemple considérons le cas où un nouveau runnable R_k doit être ajouté dans un système et deux tâches τ_a et τ_b peuvent recevoir cette mise à jour. Si R_k envoie une donnée à un runnable placé dans τ_c et que c'est également le cas d'un runnable déjà placé dans τ_a, il peut être plus intéressant de placer R_k dans τ_a.

Une fois que tous les emplacements possibles pour un nouveau runnable ont été déterminés en utilisant les critères d'allocation, il est possible de tester itérativement toutes les configurations possibles afin de trouver celle qui est optimale. La matrice d'adjacence peut être considérée comme réduite si le nombre de "1" qu'elle contient est le plus faible possible. Pour chaque allocation possible pour un nouveau runnable, il va alors être nécessaire de déterminer la matrice d'adjacence $M[\Gamma_N]$ pour les tâches. Si la matrice est triangulaire inférieure, alors cette allocation est considérée comme acceptable. Il est ensuite possible de calculer la somme S des coefficients de la matrice :

$$S = \sum_{i=1}^{N} \sum_{j=1}^{i} M[\Gamma_N](i,j)$$

Ici la somme qui concerne les colonnes de la matrice s'arrête à i car il a déjà été vérifié que la matrice est triangulaire inférieure. Le second critère d'optimisation va donc consister à choisir le positionnement pour lequel la valeur de S est la plus petite possible.

Si aucune de ces options n'est acceptable pour le placement d'un nouveau runnable, il faut alors uniquement vérifier que la matrice reste triangulaire inférieure.

4.5.3 Analyse de sensibilité

Cette analyse, décrite à la partie 4.3.2, permet de déterminer si, malgré les modifications de temps d'exécution des tâches, le système est toujours ordonnançable. Ainsi une analyse d'ordonnancement simplifiée peut être faite grâce à l'analyse de sensibilité.

Cela peut également permettre de déterminer quel est l'emplacement le plus pertinent pour un nouveau runnable si jamais deux solutions sont équivalentes pour la matrice de précédence. En effet, dans ce cas le runnable peut être placé dans la tâche qui a le plus de flexibilité disponible, *i.e.* celle pour laquelle le ΔC_k est le plus grand (voir équation 4.3). En effet,, ce facteur représente la flexibilité possible pour la tâche τ_k. Il va donc être plus intéressant de placer la mise à jour dans la tâche ayant initialement le plus de flexibilité

possible (c'est-à-dire le ΔC_k le plus grand). Cela permet de réduire l'impact global sur le système.

L'impact sur la flexibilité globale disponible dans le système doit également être étudié avec le facteur λ_{max} (voir équation 4.2). Ce facteur permet, non seulement de déterminer quelle est la flexibilité globale disponible dans le système, mais également de savoir si le système est toujours ordonnançable ou non. Il faut donc calculer ce facteur pour savoir si la mise à jour peut être acceptée ou non au sein du système.

L'emplacement d'une mise à jour a un impact sur la valeur du λ_{max} du système après mise à jour, et donc sur la flexibilité disponible après mise à jour. Il est donc très important de choisir l'emplacement qui peut garder la valeur la plus grande possible pour ce facteur (à condition que les contraintes concernant les précédences décrites au paragraphe précédent soient respectées).

4.6 Application à un exemple simple

Nous considérons un exemple simple ici qui est un sous ensemble de l'exemple des clignotants qui est développé plus en détails dans le Chapitre 6. Nous ne considérons ici qu'une seule chaine de calcul, à savoir celle qui gère les clignotants. Les valeurs qui sont données ici pour les WCET ne sont pas le reflet de la réalité. Notre objectif est de montrer que même dans un système assez chargé, il est possible de faire des mises à jour en utilisant cette méthode.

4.6.1 Présentation du système et de la mise à jour

La chaine de calcul est présentée de manière succincte dans ce chapitre, mais le détail de l'architecture du système est présenté ultérieurement. La figure 4.5 présente un exemple de chaine de calcul correspondant aux clignotants. Pour simplifier, nous avons ici choisi de présenter les runnables sous forme anonyme. Le runnable le plus à gauche, *R1* correspond au capteur de clignotants, et les runnables les plus à droite sont les actionneurs (dans ce cas, les ampoules des clignotants). Les runnables centraux sont, quant à eux des runnables de traitement.

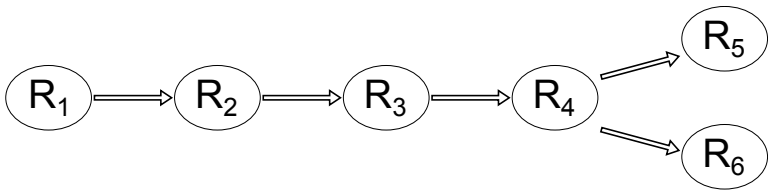


FIGURE 4.5 – Exemple de chaine de calcul initiale

Le système initial contient 6 runnables. Son objectif est de détecter la position du comodo de clignotant afin d'allumer le clignotant correspondant, tout en respectant les contraintes dictées par la règlementation en vigueur.

Nous proposons de faire une mise à jour simple qui va permettre d'utiliser des clignotants impulsionnels. Ainsi, au lieu de lire le niveau du capteur, il s'agirait de détecter une variation et de faire clignoter l'indicateur correspondant 3 fois. Cela permet d'indiquer, par exemple, un changement de voie. La figure 4.6 montre le comportement que doit avoir la mise à jour.

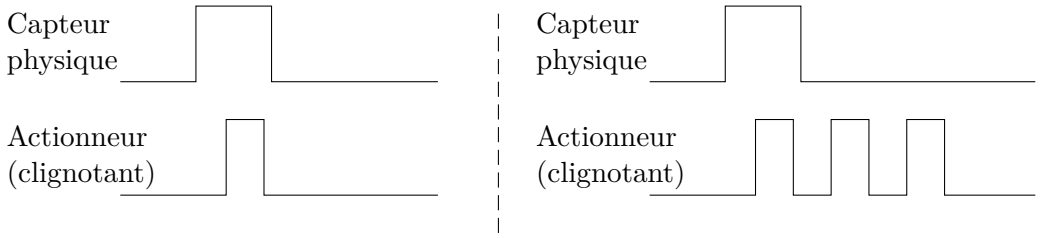


FIGURE 4.6 – Comportement nominal (à gauche) et comportement de la mise à jour (à droite)

Cette mise à jour va donc permettre d'ajouter une fonctionnalité qui n'était pas initialement présente dans l'application. Pour cela nous allons ajouter un runnable R' qui va être en mesure de détecter une impulsion sur le comodo. Dans le cas où une impulsion est détectée, le rôle de ce runnable est alors de maintenir le niveau afin que le clignotant correspondant clignote 3 fois.

R_2 va également être modifié pour avoir un rôle d'arbitrage entre R_1 et R'. La mise à jour va donc modifier R_2 et ajouter le runnable R' ainsi qu'une communication entre ces deux runnables. Il est également nécessaire que le runnable R' récupère les informations issues du runnable R_1 qui récupère les informations venues du capteur physique. Une communication supplémentaire entre R_1 et R' va donc devoir être ajoutée. La figure 4.7 présente les modifications qui doivent être faites sur la chaine de calcul initiale pour insérer la mise à jour.

Enfin, R_1 doit également être modifié pour envoyer une donnée à R'. Cette modification est cependant minime et a un impact très négligeable sur son WCET. Nous considérerons donc ici que l'ordonnancement n'est pas perturbé par le remplacement de R_1 et nous ne détaillerons pas ces modifications.

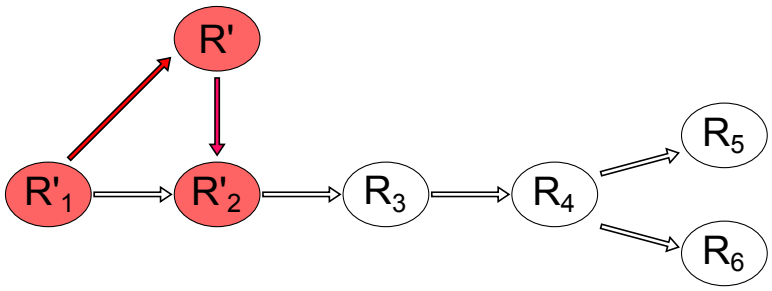


FIGURE 4.7 – Exemple de chaine de calcul mise à jour

4.6.2 Caractéristiques temps-réel de l'application initiale

Les contraintes de précédence pour le système initial sont présentées sur la figure 4.8, sous forme de matrice d'adjacence. Cette matrice est directement déduite de la figure 4.5 qui fait partie du modèle fonctionnel global.

	R_1	R_2	R_3	R_4	R_5	R_6
R_1	0	0	0	0	0	0
R_2	**1**	0	0	0	0	0
R_3	0	**1**	0	0	0	0
R_4	0	0	**1**	0	0	0
R_5	0	0	0	**1**	0	0
R_6	0	0	0	**1**	0	0

FIGURE 4.8 – Matrice d'adjacence pour l'application initiale

Chacun des runnables doit ensuite être placé dans des tâches, selon leur type et selon leurs caractéristiques temporelles. Il est important de noter que les WCET des runnables sont donnés ici uniquement à titre de démonstration. En outre, il y a un certain nombre de tâches du Basic Software qui doivent également être ordonnancées et sur lesquelles nous n'avons, du point de vue applicatif, aucune influence. Il faut donc en tenir compte dans l'analyse d'ordonnancement, mais elles ne pourront pas être modifiées, ni utilisées pour ajouter des mises à jour. La table 4.1 présente ici les tâches ainsi que leurs caractéristiques selon le modèle de tâches que nous avons défini dans la section 4.4.2. La tâche τ_2 présente dans cette table est grisée car il s'agit d'une tâche du Basic Software.

	Period	Priority	$R_k(C_{R_k})$
τ_1	5	4	$R_1(0.5)$
τ_2	5	3	WCET = 2.5
τ_3	10	2	$R_2(0.5)$, $R_3(0.25)$, $R_4(0.75)$
τ_4	10	1	$R_5(.25)$, $R_6(.25)$

TABLE 4.1 – Propriétés des tâches du système initial

A partir des informations données dans la table 4.1 ainsi que de la matrice d'adjacence de la figure 4.8, il est alors possible d'obtenir la matrice d'adjacence pour les tâches. Il est important de noter que cette matrice d'adjacence ne présente que les tâches applicatives. La figure 4.9 donne ici la matrice d'adjacence pour les tâches du système initial. Nous pouvons remarquer ici que cette matrice est bien triangulaire inférieure. Elle répond donc bien au critère d'ordonnançabilité que nous avons précédemment défini pour répondre aux contraintes de précédence.

Le facteur d'utilisation du système, tel qu'il a été défini dans la section 4.3.1 est de $U = 0,8$. Le système est donc relativement chargé. En effet, la limite théorique pour que ce système soit ordonnançable (selon la formule de Liu et Layland [64]) est $U_{théorique} \simeq 0,76$. Toutefois, même en ayant un système très chargé, la méthode de Bini

	τ_1	τ_3	τ_4
τ_1	0	0	0
τ_3	1	1	0
τ_4	0	1	0

FIGURE 4.9 – Matrice d'adjacence pour les tâches du système initial

et Buttazzo montre que ce système est tout de même ordonnançable.

De plus, l'utilisation de l'analyse de sensibilité nous permet de déterminer le ΔC_k^{max} pour chacune des tâches du système, ainsi que le λ^{max} qui donne la flexibilité globale du système. Nous obtenons pour ce système $\lambda^{max} = 0,25$ et $\Delta C_1^{max} = \Delta C_2^{max} = 1$, $\Delta C_3^{max} = \Delta C_4^{max} = 2$. Notons que le fait que λ^{max} soit positif montre bien que notre système est ordonnançable.

4.6.3 Caractéristiques temps-réel de l'application mise à jour

Nous ajoutons donc au système le runnable R'. Les caractéristiques fonctionnelles de ce runnable demande qu'il soit exécuté à une période de 5 ms. Ce runnable va donc être placé dans la tâche τ_1. Le WCET du runnable R' est sensiblement identique à celui de R_1 : $C_{R'} = 0,5$. Le runnable R'_2, quant à lui, remplace le runnable R_2 initialement présent. Ses caractéristiques temporelles sont donc sensiblement identiques. Toutefois, étant donné que l'arbitrage doit maintenant être pris en compte dans le runnable R'_2, le WCET de ce runnable est donc plus grand que celui du runnable initial R_2 : $C_{R'_2} = 0,6$.

4.6.3.1 Contraintes de précédence

Les runnables doivent être placés dans des tâches à la même périodes que la leur. Pour cette raison, le runnable R' doit être placé dans la tâche τ_1. R'_2 quant à lui reste dans la tâche τ_3 : seul le code du runnable R_2 est modifié.

Deux nouvelles dépendances de données sont ajoutées : une entre R_1 et R' et une entre R' et R'_2. Cela crée donc de nouvelles contraintes de précédence qui doivent être prises en compte : la matrice de précédence doit être modifiée en conséquence. Il y a donc une donnée produite par la tâche τ_1 et consommée également par τ_1 (entre R_1 et R'), et une donnée produite par τ_1 et consommée par τ_2. La matrice d'adjacence présentée par la figure 4.9 est donc légèrement modifiée. Cette modification est présentée par la figure 4.10.

	τ_1	τ_3	τ_4
τ_1	**1**	0	0
τ_3	1	1	0
τ_4	0	1	0

FIGURE 4.10 – Matrice d'adjacence pour les tâches du système mis à jour

Ainsi la matrice de précédence est toujours conforme, et les contraintes de précédence

peuvent donc être respectées. Il est important de noter que nous ne considérons ici des communications de type “last is best”, c'est-à-dire que seule la dernière valeur produite nous intéresse. Dans le cas où des données supplémentaires sont produites, et perdues, cela ne perturbe pas le bon fonctionnement du système.

4.6.3.2 Analyse de sensibilité

Nous appliquons maintenant les concepts de l'analyse de sensibilité à notre nouveau système. Nous avons maintenant $C_1 = C_{R_1} + C_{R'} = 1$ et $C_3 = C_{R_2} + C_{R_3} + C_{R_4} + (C_{R'_2} - C_{R_2}) = 1,6$. Notons que nous avons initialement $\Delta C_1^{max} = 1$ et $\Delta C_3^{max} = 2$. Intuitivement, nous pouvons penser que la mise à jour est possible, uniquement en regardant ces valeurs. Cependant elles ne sont valables que pour le cas où une seule tâche est modifiée et notre cas, deux tâches doivent être modifiées (τ_1 qui reçoit R' et τ_3 dans laquelle R_2 est mis à jour).

Pour ce nouveau système, le facteur d'utilisation est maintenant de $U = 0,91$, ce qui est largement au dessus de la valeur théoriquement admissible. Cependant, l'analyse de Bini et Buttazzo montre que malgré tout, ce système est toujours ordonnançable. De plus, nous pouvons déterminer les nouvelles valeurs pour λ^{max} et les ΔC_k^{max} de chacune des tâches. Ainsi, $\lambda^{max} = 0,99$ et $\Delta C_1^{max} = 0,45$, $\Delta C_3^{max} = \Delta C_4^{max} = 0,9$. Nous constatons ici que le facteur λ^{max} est toujours positif.

La figure 4.11 montre une coupe dans le *C-space*, selon les axes C_1 / C_3. Le point le plus à gauche représente le point d'ordonnancement correspondant au système initial, alors que le point le plus à droite représente le système mis à jour. Un temps d'exécution plus grand pour les tâches va donc entrainer un rapprochement de la frontière du *C-space*.

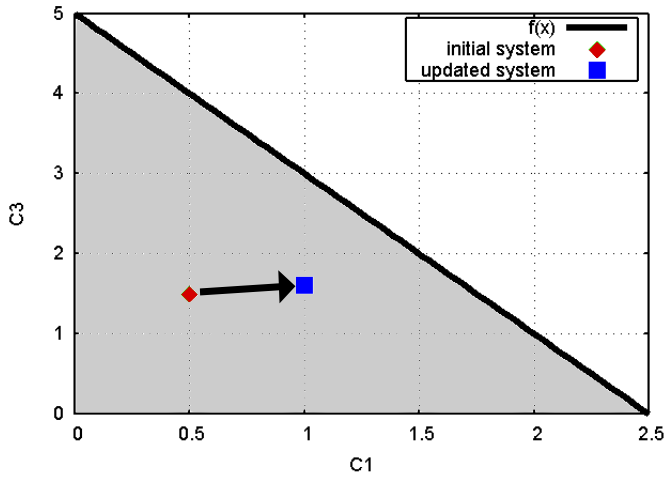


FIGURE 4.11 – Coupe du *C-space* dans le plan C_1 - C_3

4.7 Conclusion

Ce chapitre rappelle un certain nombre de concepts liés à la théorie de l'ordonnancement temps-réel et montre comment il est possible de lier ces éléments dans le cas d'un système de type AUTOSAR. Dans la théorie classique de l'ordonnancement, tous les critères temporels (deadline, WCET, contraintes de précédence) sont liés aux tâches.

Cependant, le formalisme AUTOSAR, qui fait porter les fonctionnalités sur les runnables et non directement sur les tâches, conduit à avoir des contraintes temporelles directement sur les runnables et non sur les tâches. Ces runnables sont ensuite alloués sur des tâches du système d'exploitation, qui vont alors être considérées uniquement comme support d'exécution. Les tâches héritent donc des caractéristiques des runnables.

Ce changement de paradigme conduit donc à adopter un modèle différent de la théorie classique de l'ordonnancement. Cependant, des algorithmes permettent ensuite de se replacer dans un contexte similaire et d'appliquer des résultats pertinents issus de la théorie classique de l'ordonnancement.

L'établissement de ce modèle pour le contexte AUTOSAR permet ensuite de caractériser d'un point de vue temps-réel l'ajout de nouvelles fonctionnalités, sous la forme de runnables, dans une application existante. Afin de déterminer si ces modifications sont possibles, d'une part les contraintes de précédence doivent être étudiées, et d'autre part il faut vérifier que le système est toujours ordonnançable. Pour cela, nous utilisons deux outils : tout d'abord, la matrice de précédence pour les runnables et pour les tâches, et ensuite l'analyse de sensibilité. La matrice de précédence permet, non seulement de vérifier que les contraintes de précédence sont vérifiées, mais également de déterminer une allocation optimale pour la mise à jour. L'analyse de sensibilité permet de vérifier que la mise à jour ne perturbe pas l'ordonnançabilité du système et, dans le cas où plusieurs emplacements seraient encore possibles, de déterminer quel emplacement est optimal du point de vue comportement temps-réel.

5

Mise en pratique des mises à jour

Sommaire

5.1 Introduction

Après avoir développé les aspects structurels (Chapitre 3) ainsi que les aspects comportementaux (Chapitre 4) de la mise à jour, ce chapitre présente un certain nombre d'aspects d'implémentations essentiels à la mise à jour. Nous détaillons ici en particulier la gestion des communications, la création des mises à jour, la gestion des versions et les aspects liés à la sûreté de fonctionnement.

Dans ce travail, nous ne traitons que la partie liée au calculateur embarqué. Bien qu'il existe d'autres aspects pour la mise à jour, en particulier un aspect "débarqué" (c'est-à-dire qui doit s'exécuter en parallèle de la partie embarquée), nous nous concentrons essentiellement sur les aspects qui nous paraissent essentiels pour les mises à jour dans le calculateur.

Les sujets traités ici couvrent un large spectre. Notre objectif n'est pas de traiter intégralement chacun d'entre eux, mais de poser les problèmes liés et d'identifier des techniques pour répondre à chacun de ces problèmes.

Nous commençons par décrire la gestion des communications pour les mises à jour, au niveau du RTE et du Basic Software. Nous décrivons également ici le processus de création des mises à jour, tout d'abord pour en récupérer le contexte d'exécution, puis pour insérer cette mise à jour dans le calculateur.

Le second point développé est la gestion de version. Cette partie se trouve, certes, du côté du serveur de mise à jour et non du côté de la carte embarquée, mais elle est primordiale. C'est cette gestion qui permet de savoir quelle est la version actuellement présente dans le calculateur et quelles sont les mises à jour possibles.

Nous discutons ensuite des problèmes de gestion de la mémoire. En effet, pour pouvoir ajouter de nouvelles fonctionnalités dans un calculateur, il est nécessaire d'avoir suffisamment d'espace libre. Cependant, même si une grande quantité d'espace mémoire est laissée libre au départ, ajouter des mises à jour peut finir par épuiser la mémoire disponible. Quelles sont alors les solutions qui s'offrent à nous pour libérer de la mémoire ? D'autre part, la gestion mémoire dans un calculateur embarqué ne se fait pas de la même manière que sur une cible débarquée. Cette section détaille les problèmes spécifiques au monde de l'embarqué de ce point de vue.

Enfin, nous traitons les aspects de sûreté de fonctionnement liés à la mise à jour. Faire une mise à jour partielle oblige à traiter séparément certains éléments. Cette séparation n'existe pas dans le cas d'un rechargement complet du calculateur. Pour cette raison, les aspects de sécurité-innocuité ne sont pas nécessairement identiques. Nous traitons ici les mécanismes de sécurité-innocuité propres à la structure déployée dans le cadre des mises à jour.

5.2 Gestion des communications et création des mises à jour

Pour définir une mise à jour, il faut procéder aux mêmes étapes que lors du développement standard d'une application (voir Chapitre 3). Sachant que grâce aux méta-données

qui ont été ajoutées lors de la préparation de l'application, il est possible de retrouver le modèle fonctionnel logiciel global correspondant à la version actuellement chargée dans l'application, la mise à jour peut donc directement être intégrée dans ce modèle. Cela permet de savoir quels sont les éléments précis (runnables et canaux de communication en particulier) qui sont modifiés ou ajoutés. Les étapes suivantes permettent de déterminer où sont placés ces éléments jusqu'à l'obtention du code binaire correspondant et des emplacements mémoires où ces derniers doivent être placés. Les mécanismes de sécurité-innocuité sont donc ajoutés à la mise à jour de la même manière que dans le cas d'un développement standard. Les analyses de *safety* sont effectuées hors ligne, lors de la préparation de l'application, sur toutes les versions et mises à jour, en se conformant à l'ISO 26262.

5.2.1 Gestion des nouvelles communications

De nouvelles communications doivent être ajoutées dans le cas où la mise à jour est de type *update*. Cela peut aussi être le cas pour une *upgrade*, par exemple, si des informations supplémentaires doivent être envoyées à un autre runnable existant. Sachant que les modifications doivent, en premier lieu, être faites au niveau "Architecture logicielle fonctionnelle globale" (voir section 3.3.1), il n'est pas possible de déterminer à l'avance si le nouveau runnable et les autres éléments de la chaine sont placés sur le même ECU ou bien sur des ECUs distants.

Dans les deux cas, il est nécessaire d'ajouter un ou plusieurs canaux dans le RTE. Non seulement pour l'ECU dans lequel le nouveau runnable est placé, mais également dans le RTE de l'ECU qui contient les éléments de la chaine de calcul avec lesquels le nouveau runnable communique. S'il s'agit du même ECU, cela simplifie la situation car une seule modification doit être faite et les considérations de cohérence globale distribuée disparaissent. Toutefois, dans le cas où il s'agit d'ECUs différents, il faut également modifier la pile de communication et la messagerie CAN afin de faire transiter de nouveaux messages sur le bus. Ces modifications ne sont pas triviales, et les systèmes actuels de messagerie ne permettent pas forcément de faire de telles modifications dynamiquement sur les bus matériels.

5.2.1.1 Canaux du RTE

Les nouvelles communications doivent nécessairement passer par le RTE afin de respecter l'architecture AUTOSAR. Ainsi il faut ajouter tous les éléments nécessaires afin de créer un canal. Trois éléments principaux sont nécessaires afin de rajouter un canal, à savoir une donnée RTE, une fonction qui permet au runnable destinataire de lire la donnée, une fonction qui permet au runnable émetteur d'écrire la donnée (accesseurs). Ces éléments sont des *Objets RTE*.

La donnée RTE peut être composée d'une ou plusieurs variables, chacune de type différent. Elle doit être stockée dans un espace mémoire spécifique. La méthode la plus simple pour gérer ces variables est d'utiliser une section spécifique du tas, dédié au

stockage de ces données. Il faut donc être en mesure de savoir quels emplacements mémoires sont déjà utilisés dans le tas afin de ne pas écraser des données existantes.

Lorsque les fonctions de lecture et d'écriture des données RTE ont été crées, il est nécessaire de les écrire dans la mémoire FLASH du calculateur. L'écriture de ces fonctions est faite de la même manière que l'écriture de runnables, à la suite du programme initial. Cependant, pour les fonctions RTE, il est nécessaire de connaître l'adresse mémoire à laquelle ces fonctions sont stockées afin de pouvoir les appeler et afin que l'exécution soit correcte. La figure 5.2 donne un exemple d'emplacements mémoires pour la mise à jour simple présentée sur la figure 5.1.

Cette mise à jour est donc constituée de quatre éléments : "New_runnable_1" qui envoie une donnée à "New_runnable_2" et les fonctions pour lire et écrire *data_1*, à savoir "read_data_1" et "write_data_1". "New_runnable_1" utilise la fonction "write_data_1" afin d'envoyer une donnée à "New_runnable_2". Dans le code source de la fonction "New_runnable_1", il y a donc un appel à la fonction 'write_data_1". Le code assembleur de cette fonction correspond donc à une instruction de branchement vers l'adresse de la fonction (voir figure 5.2). Le code source doit être compilé en code binaire de type PowerPC. La première passe du compilateur va uniquement transformer les instructions avec les bons codes correspondant à l'opération à effectuer (18 pour une instruction de type *BRANCH*). Cependant, les adresses mémoire des fonctions sont uniquement gérées pendant le link. Il faut donc que l'adresse mémoire de la fonction dans le calculateur soit connue pour que le link puisse se faire correctement et que les instructions binaires soient valides.

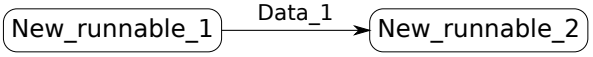


FIGURE 5.1 – Exemple d'éléments constitutifs d'une mise à jour

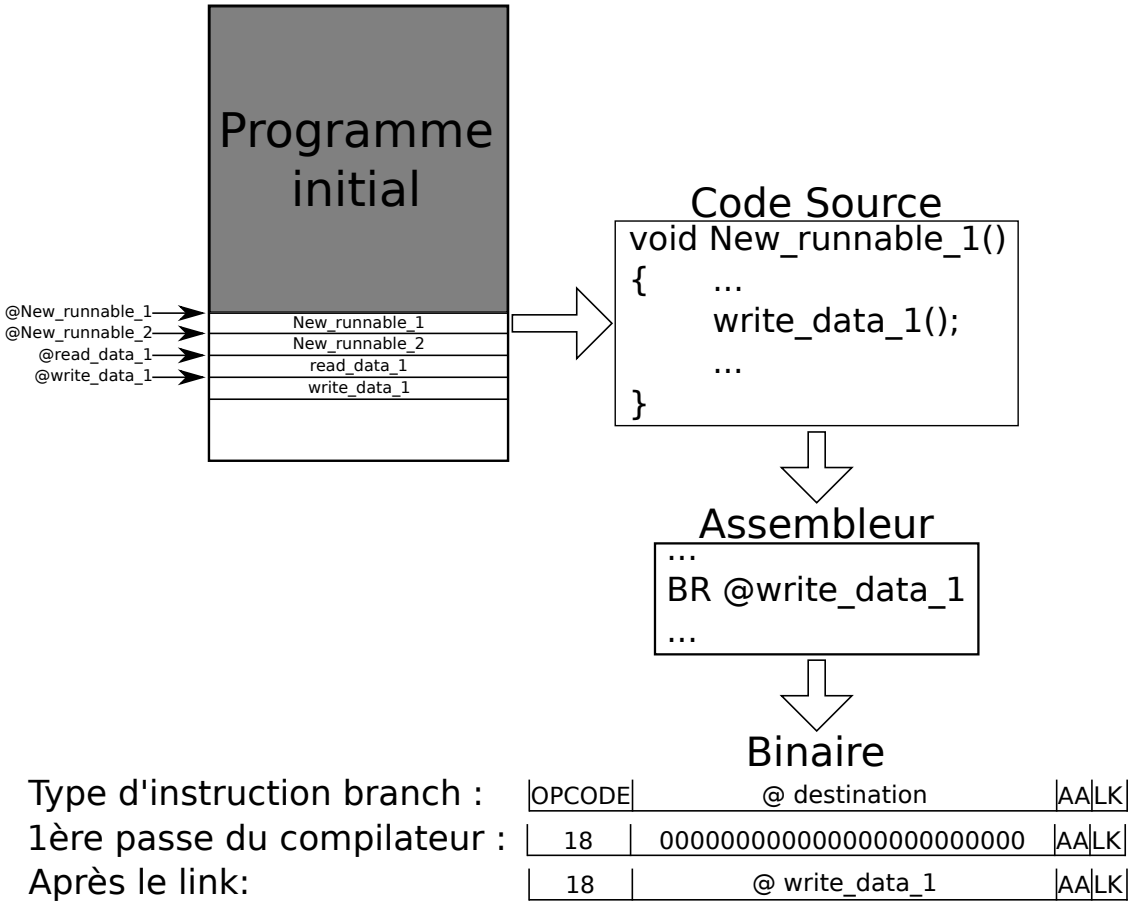


FIGURE 5.2 – Ajout de runnables et de fonctions RTE pour la mise à jour de la figure 5.1

5.2.1.2 Modifications du Basic Software

Lorsqu'une mise à jour de type *update* porte sur plusieurs calculateurs, il est nécessaire de modifier le Basic Software. La modification du Basic Software peut intervenir dans trois cas principaux pour les mises à jour :

1. ajout d'entrées/sorties matérielles,
2. modification ou ajout de librairies,
3. modification ou ajout de signaux venant de la pile de communication.

Le premier cas est improbable, car cela signifierait qu'il y ait des périphériques matériels inutilisés. Or, pour des raisons de réduction de coût, si un périphérique matériel est inutile, il n'est pas présent dans le calculateur. Il se peut que certaines fonctionnalités nécessitent l'ajout de matériels spécifiques (capteurs et/ou actionneurs), cependant ce cas n'est pas étudié dans le cadre de cette thèse.

Concernant les librairies, si une librairie existante devait être modifiée, alors tous les appels à cette librairie devraient alors l'être aussi. Cela impliquerait de modifier une

grande quantité de code non contigu, et par conséquent de faire des modifications lourdes sur la mémoire (à cause de la gestion par secteur de la mémoire flash). Dans ce cas, une mise à jour partielle ne serait pas appropriée et un rechargement complet du calculateur serait nécessaire.

L'ajout de librairie est quelque peu différent : il s'agit alors de rajouter de nouvelles fonctions, qui ne serait pas utilisées par le code existant, mais uniquement par les mises à jour. Cela peut être rapproché de l'ajout de fonctions de lecture et d'écriture des données RTE et les mécanismes seraient les mêmes. Ainsi, il est possible d'ajouter une librairie par une mise à jour partielle, mais pas de modifier une librairie existante.

Enfin, l'ajout ou la modification de signaux de la pile de communication sont également deux processus différents. Nous ne traitons ici que le cas de la pile de communication CAN, bien que d'autres types de bus matériels (FlexRay, LIN, Ethernet) existent dans l'automobile et soient également standardisés par AUTOSAR. En effet, la modification de signaux conduirait à une modification du RTE (qui remonte ces données au logiciel applicatif), ce qui pourrait être assez lourd à mettre en place, dans la mesure où il s'agirait de modifier du code existant et des appels à ces fonctions. L'ajout de signaux dans la pile de communication et, par conséquent dans le RTE, est possible à condition de gérer la configuration de la messagerie en PostBuild. La configuration des systèmes embarqués de type AUTOSAR peut intervenir à plusieurs moments du développement[105] :

— Soit avant la compilation, et il s'agit alors de configuration "PreCompile",
— Soit au moment du "Link", par exemple pour générer des librairies et les lier à des variables présentes en ROM,
— Soit au "Run-time", par exemple pour les routines de diagnostique,
— Soit en "Post-Build", lorsqu'il s'agit de charger des données de configuration, via un bootloader, dans un ECU contenant déjà du code.

Cela pourrait donc impliquer certaines modifications dans la gestion de la messagerie pour pouvoir faire une gestion des signaux en Post-Build, ce qui n'est pas nécessairement possible à l'heure actuelle. .

5.2.2 Création des mises à jour

Dans cette section nous supposons que les mises à jour sont au sein d'un seul ECU. Il n'y a pas de notion de mise à jour distribuée. Cela permet de simplifier deux problèmes : tout d'abord, il n'est pas nécessaire de s'assurer de la cohérence globale des mises à jour distribuées, ce qui est un problème non trivial. De plus, cela ne nécessite pas la modification de la configuration de la messagerie pour le bus matériel qui doit transmettre les messages puisque seules les communications internes à l'ECU peuvent être impactées. Notons que la mise à jour distribuée pourrait être faisable avec la méthode proposée ici à condition de ne pas modifier la messagerie sur le bus matériel et d'ajouter les mécanismes nécessaires pour assurer la cohérence globale du système.

Un point clef pour permettre une mise à jour est de connaître le contexte particulier dans lequel cette mise à jour sera exécutée. Pour cela, les méta-données qui ont été ajoutées aux différentes étapes du processus de développement (voir chapitre 3) ont un

rôle essentiel. En effet, à partir des données embarquées dans le calculateur il est possible de retrouver les modèles correspondants.

5.2.2.1 Mise à jour et méta-données

Les méta-données qui permettent de retrouver le contexte dans lequel une mise à jour s'exécute sont de deux types principaux : d'une part le numéro de version de l'application, et d'autre part l'ensemble des données uniques pour chacun des runnables. Ces données sont composées de l'identifiant unique associé au runnable et de l'ensemble des adresses mémoires correspondant à ce runnable.

L'identifiant unique du runnable correspond à un runnable donné dans une version donnée. Si un runnable est modifié, son identifiant unique change également. Pour pouvoir retrouver un runnable dans la mémoire, il faut non seulement connaître l'adresse de début du stockage, mais aussi le nombre d'octets stockés à cette adresse correspondant au runnable. En outre, un runnable n'est pas nécessairement stocké à une seule adresse mémoire. En effet, il peut arriver qu'il soit éclaté à plusieurs endroits de la mémoire. Ainsi, pour pouvoir l'identifier complètement, il faut avoir son identifiant, ainsi que les emplacements mémoires correspondants. L'ensemble de ces informations étant assez volumineux à stocker, il est nécessaire de les compresser en utilisant un algorithme de compression de données sans perte qui permette ensuite de retrouver les informations complètes. Notons que ceci est tout à fait possible car la décompression des données n'a pas vocation à être faite directement sur le calculateur, certains algorithmes de décompression pouvant être gourmand en ressources mémoires et calculatoires.

Par exemple, pour les runnables et les *containers*, les méta-données que nous avons utilisées sont :

— ID unique du runnable
— Statut (pour savoir si le pointeur de fonction pointe sur une fonction nulle ou non : lorsqu'il s'agit d'un *container*, cela permet de savoir si le *container* contient une mise à jour ou non)
— valeur du pointeur de fonction (c'est-à-dire l'emplacement de la mémoire correspondant à l'ID unique)

Il est important de noter que ces méta-données sont uniquement présentées ici à titre d'exemple et n'ont pas vocation à être exhaustives. Dans l'application globale, un certain nombre d'autres informations sont nécessaires, par exemple celles liées aux canaux de communication.

Après que les méta-données sont extraites de l'ECU, elles permettent de retrouver tous les identifiants uniques des runnables. Ces identifiants peuvent être sous la forme UUID (*Universally Unique IDentifier*), normalisée par l'ISO/IEC 9834-8 :2014, qui est pris en charge par les différents outils que nous utilisons au cours du processus de développement. Lorsque tous les identifiants des runnables contenus dans l'ECU sont connus, il est possible de trouver le modèle AUTOSAR correspondant à cette version spécifique. Puis à partir des identifiants placés dans le modèle AUTOSAR, il est possible de retrouver le modèle fonctionnel global correspondant. Le processus global qui permet, à partir

des informations embarquées dans le calculateur, de retrouver les différents niveaux de modèles du logiciel est présenté sur la figure 5.3.

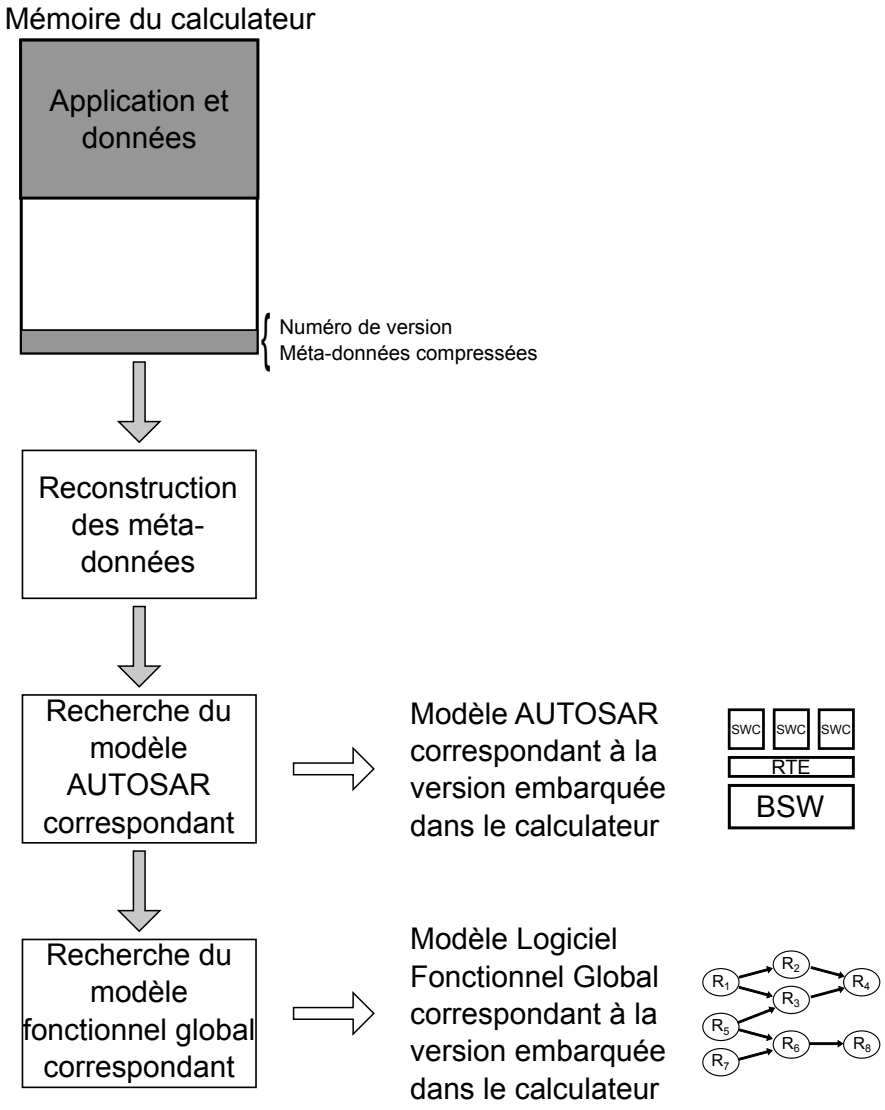


FIGURE 5.3 – Processus de récupération des modèles à partir des informations embarquées dans le calculateur

5.2.2.2 Insertion des mises à jour

Les méta-données portent sur les différents éléments présents dans le système (runnables, canaux de communication, ...). Cela signifie que lorsqu'une mise à jour est faite, il faut non seulement faire évoluer le numéro de version, mais également modifier les méta-données. La création de mises à jour suit un processus similaire au processus de

développement standard. Cependant, un grand nombre d'éléments restent inchangés ; le processus de développement s'en trouve donc simplifié.

La mise à jour est donc tout d'abord intégrée au modèle logiciel fonctionnel global. Les méta-données associées à chacun des éléments constitutifs de cette mise à jour doivent être intégrées au modèle. Cette étape permet de déterminer toutes les caractéristiques de la mise à jour. Cela signifie que toute mise à jour suit un processus de développement qui permet d'insérer tous les mécanismes de sécurité-innocuité nécessaires. Une *update* est constituée d'au moins deux runnables et d'un canal de communication : un nouveau runnable et un runnable existant modifié pour prendre en compte les communications avec le nouveau runnable. En effet, nous avons expliqué précédemment qu'un runnable qui n'interagit pas avec son environnement est considéré comme inutile.

Ensuite, les différents runnables constituant la mise à jour sont intégrés dans le modèle AUTOSAR. Notons que le modèle AUTOSAR ici peut être directement pris au niveau RTE, sans nécessairement passer par le niveau VFB car nous avons supposé que les mises à jour sont faites au sein d'un seul ECU (pas de mise à jour distribuée). Les méta-données correspondant au modèle AUTOSAR sont ajoutées à ce niveau. Les nouveaux canaux du RTE, qui doivent être créés en cas d'*update* sont générés à cette étape.

Le code correspondant aux mises à jour est généré en utilisant également une approche "Model-Based Design". En cas de modification de runnable existant, cela permet de repartir du modèle et de ne pas tout refaire à partir de zéro.

L'ensemble des éléments constitutifs de la mise à jour, à savoir les nouveaux runnables, les runnables modifiés et les objets RTE doivent ensuite être compilés et linkés. Cependant, pour des raisons de contraintes matérielles, les éléments d'une mise à jour sont placés dans la mémoire après l'application déjà existante. Cela signifie que les fonctions existantes ne sont pas supprimées de la mémoire. Cela signifie également que l'ordre dans lequel les mises à jour sont faites est important et que, même si les mêmes fonctionnalités sont présentes dans un ECU, la version du logiciel n'est pas nécessairement la même (nous détaillons ce point dans la section suivante). Cela demande donc une configuration spécifique du linker pour prendre en compte ces contraintes sur les emplacements mémoires. En effet, lorsque l'application initiale est compilée, les fonctions sont placées en mémoire par le compilateur. Cependant, lors de la compilation des mises à jour, certaines contraintes concernant les emplacements mémoires des fonctions déjà présentes doivent être imposées. Cela permet de faire un link correct entre la mise à jour et le reste de l'application.

La figure 5.4 résume ici les différentes étapes de la création d'une mise à jour. Sur cette figure les flèches grises correspondent aux différentes étapes du processus, qui permettent d'insérer la mise à jour dans le calculateur. Les flèches blanches présentent des étapes corollaires au processus de création de mise à jour : ici il s'agit de stocker les modèles correspondant à la nouvelle version du logiciel afin de pouvoir insérer de nouvelles mises à jour ultérieurement. Les notations utilisées ici font écho à la figure 5.3, mais les parties rouges présentent ce qui est ajouté pour la mise à jour.

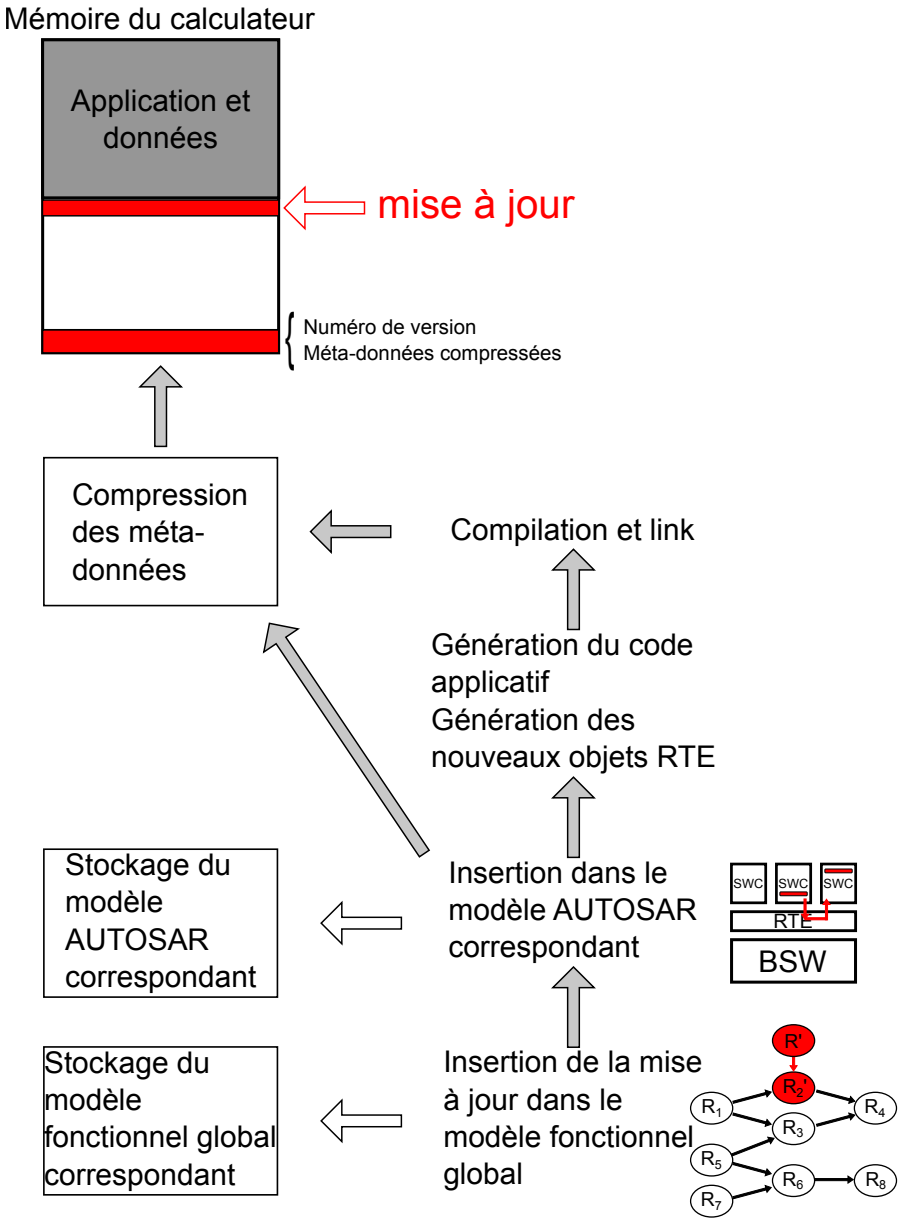


FIGURE 5.4 – Processus de création d'une mise à jour

5.3 Gestion de version

L'objectif de ce travail est de permettre à l'utilisateur final de faire des mises à jour de manière personnalisée, c'est-à-dire qu'il a la possibilité de décider quelles mises à jour faire et dans quel ordre. Les mises à jour successives peuvent alors être faites dans n'importe quel ordre et il est possible d'avoir un grand nombre de combinaison de fonctionnalités différentes pour chaque calculateur. Ainsi, cette méthode de mise à jour

pour les calculateurs va créer une combinatoire importante. La prise en compte et la maîtrise de toutes les versions possibles du système (selon les mises à jour choisies et leur ordre) sont donc des questions qu'il est impératif de traiter.

D'autre part, les systèmes embarqués automobiles sur lesquels nous travaillons peuvent être critiques : il est alors nécessaire de s'assurer que les contraintes de sécurité-innocuité liées à ces systèmes sont vérifiées. Pour cette raison, des tests préalables doivent être mis en place pour garantir le bon fonctionnement des systèmes, y compris lorsqu'une mise à jour est ajoutée. Si la combinatoire de toutes les versions du logiciel devient très importante, il peut alors être très difficile de tester toutes les combinaisons possibles. Cependant, les tests sont nécessaires, et pour cette raison, il faut être en mesure de mettre en place des tests qui vont cibler uniquement la mise à jour et son intégration dans son environnement plutôt que de tester à chaque fois l'intégralité de l'application.

Les méta-données placées dans l'application lors du processus initial de développement, et spécifiquement lors de sa préparation pour les mises à jour ultérieures (voir Chapitre 3) permettent de connaître, non seulement la version du système, mais également d'identifier précisément chacun des éléments présents dans le système. Cela permet donc à tout moment de connaître le contexte d'exécution et également de savoir quelles mises à jour sont présentes, ainsi que l'ordre dans lequel elles ont été ajoutées. De plus, à partir de ces informations, il est possible de déterminer quelles mises à jour peuvent alors être ajoutées dans le système. Le graphe présentant toutes les évolutions possibles d'un système donné est alors appelé **arbre de mise à jour**.

Cela permet également, dans une certaine mesure, d'éviter l'explosion combinatoire. En effet, certaines combinaisons de mises à jour peuvent ne pas être autorisées pour diverses raisons (compatibilité, sécurité-innocuité, etc). Ainsi l'arbre de mise à jour permet d'avoir toutes les combinaisons possibles pour un système donné.

Pour détailler l'intérêt des arbres de mises à jour prenons l'exemple simple d'une chaine de calcul contenant 5 runnables, pour laquelle 2 mises à jour sont possibles. La figure 5.5 présente cette chaine de calcul avec les deux mises à jour possibles : 1) ajouter le *runnable A* qui communique avec le *runnable 2* ("Mise à jour A"), 2) ajouter le *runnable B* qui communique avec le *runnable 2* ("Mise à jour B").

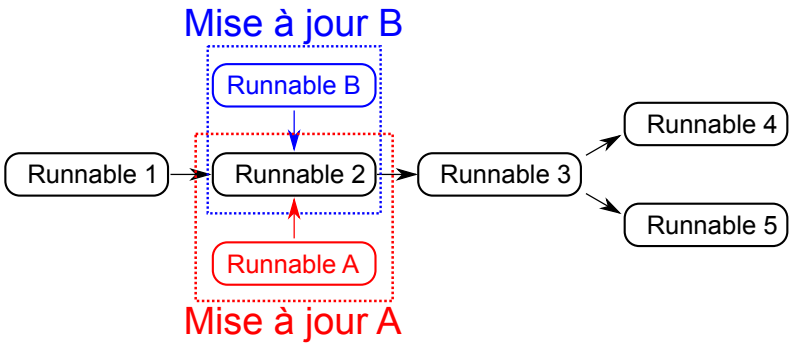


FIGURE 5.5 – Exemple de chaine de calcul et des mises à jour possibles

Ainsi, la première mise à jour peut être notée sous la forme MàJ_A={runnable

A, runnable 2 (VA), com A→2} : cette mise à jour requiert de changer la version du runnable 2, qui devient alors “VA”, pour signifier que cette version est compatible avec une communication avec le runnable A. La seconde mise à jour possible pour le système peut être notée sous la forme MàJ_B={runnable B, runnable 2(VB), com B→2} : le runnable 2 doit être dans une version “VB”, c’est-à-dire capable de prendre en compte les informations envoyées par le runnable B.

Le *runnable 2* possède donc au moins quatre versions : la version initiale (que l’on note “V1” pour tous les runnables), la version qui permet de communiquer en plus avec le runnable A (“VA”), la version qui permet de communiquer en plus avec le runnable B (“VB”) et enfin la version qui permet de communiquer à la fois avec A et B (“VAB”).

A partir des différentes mises à jour possibles du système et de leur combinaison, il est alors possible de déterminer l’arbre de mise à jour. Celui correspondant à notre exemple est donné par la figure 5.6. À partir de la version initiale du système il est donc possible de faire, soit la mise à jour A (branche A de l’arbre), soit la mise à jour B (branche B de l’arbre). Une fois que la première mise à jour a été faite, il est possible de faire la seconde. L’état du système qui possède les deux mises à jour, d’après l’arbre présenté ici est alors soit “MaJ_A_B”={runnable B, runnable 2, com B→2}, soit “MaJ_B_A”={runnable A, runnable 2, com A→2}.

Lorsque les deux mises à jour sont faites, et quel que soit l’ordre dans lequel elles ont été faites, les fonctionnalités présentes dans le calculateur sont les mêmes et le comportement final pour l’utilisateur est identique. Cependant, l’état du calculateur en lui même n’est pas identique : en effet, l’emplacement des mises à jour dans la mémoire du calculateur n’est pas forcément identique, les *containers* utilisés peuvent être différents, et les méta-données sont différentes. C’est pour cette raison que la version du logiciel dans le calculateur n’est pas la même, et que l’arbre de mise à jour conserve deux états distincts qui permettent de traduire l’ordre de chargement des mises à jour.

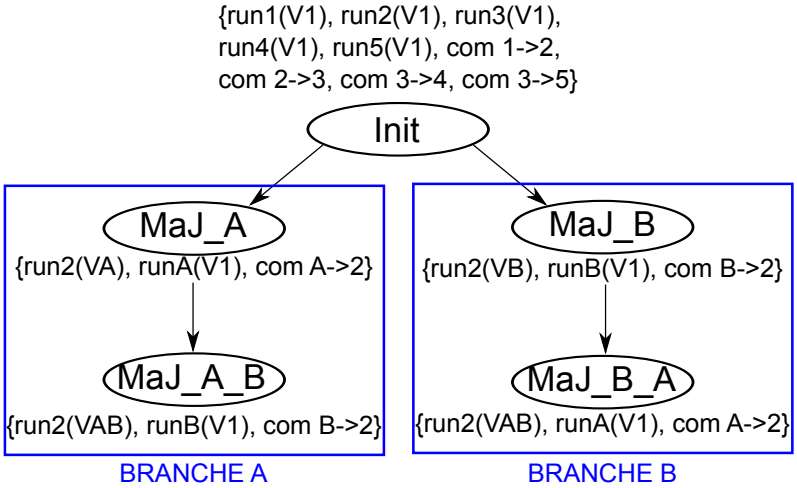


FIGURE 5.6 – Exemple d’arbre de mise à jour pour deux mises à jour possibles sur la chaine de calcul de la figure 5.5

Il est préférable que l'arbre de mise à jour soit construit au préalable et hors ligne, c'est-à-dire que, au fur et à mesure que de nouvelles mises à jour sont créées, l'arbre est également mis à jour pour déterminer quelles versions actuellement disponibles sont capables de gérer la nouvelle mise à jour. Il faut alors explorer l'arbre et, en fonction des éléments impactés par la mise à jour, et des dépendances aux autres éléments, savoir si cette mise à jour est compatible avec la version de chacun des nœuds de l'arbre. En plus de cette vérification structurelle, il faut également procéder à une vérification comportementale, à savoir une analyse d'ordonnancement grâce à l'analyse de sensibilité (voir Chapitre 4).

Lorsque plusieurs mises à jour successives ont lieu dans un système, il pourrait arriver que certaines parties du système deviennent incompatibles entre elles. En effet, certains éléments logiciels pourraient alors être plus récents que d'autre et ne plus pouvoir fonctionner ensemble. Une telle situation serait totalement inacceptable pour le bon fonctionnement du système. Pour cette raison, il est nécessaire de s'assurer de la cohérence du système. C'est l'un des rôles de l'arbre de mise à jour : s'assurer qu'à partir d'une version donnée du logiciel, une mise à jour est compatible avec l'ensemble des autres fonctionnalités du système. Il est important de noter que la cohérence doit, de toute façon, être assurée par des tests off-line.

5.4 Indirections et gestion de la mémoire

Afin d'avoir plus de flexibilité dans le calculateur pour pouvoir faire les mises à jour, nous ajoutons un niveau supplémentaire d'indirection pour l'appel des runnables dans les tâches. De cette manière il est possible de modifier le code appelé et de faire des mises à jour. Cette gestion des fonctions est en lien avec la façon dont nous avons choisi de gérer la mémoire embarquée dans le calculateur.

En effet, cela nous permet d'éviter de ré-écrire des parties importantes de la mémoire pour ajouter les mises à jour. Ces dernières sont ajoutées à la suite du code existant et le niveau d'indirection supplémentaire nous permet de savoir où est stocké la fonction qui doit être exécutée.

5.4.1 Indirections

Dans un processus standard de création d'application AUTOSAR, les runnables sont ajoutés dans les tâches et appelés dans un certain ordre lors de l'exécution. L'appel aux runnables se fait de manière directe. Un exemple de tâche créée par un processus standard est donné par la figure 5.7.

Afin d'avoir une meilleure gestion à grain fin (c'est-à-dire au niveau de chacun des runnables) du contenu des tâches, nous devons ajouter un niveau d'indirection pour l'appel de chacun des runnables. Au lieu d'appeler directement chaque runnable, nous allons placer un tableau avec les adresses des runnables dans la tâche, et appeler le contenu de chacune des cases dans l'ordre. Il y a une table d'indirection par tâche, et chacune contient un certain nombre de case "vides" : chacune d'entre-elles correspond

```
TASK(Task_100ms)
{
   /* start runnable */
   Runnable1();
   /* start runnable */
   Runnable2();
   /* start runnable */
   Runnable3();

   TerminateTask();
}
```

FIGURE 5.7 – Exemple de code d'une tâche avec appels directs aux runnables

à un *container* placé dans la tâche. Cependant le temps effectivement utilisé par le *container* est nul : il appelle alors une fonction vide. Ce sont ces slots vides qui permettent de rajouter des mises à jour *a posteriori*.

Un tableau possède deux types de caractéristiques : des caractéristiques générales, qui sont celles de la tâche (période, deadline, priorité) et des caractéristiques spécifiques à chacune des cases. En effet, chaque case doit avoir un élément descripteur qui permet de savoir à quel runnable et à quelle version de ce runnable elle correspond. Si la case correspond à un *container*, il est nécessaire d'avoir en plus de ces caractéristiques son état, à savoir s'il contient un runnable ou non. Le format de description pour chacune des cases du tableau est donc identique.

La figure 5.8 montre un exemple de tâche utilisant un appel indirect aux runnables. Notons qu'il s'agit ici uniquement d'un exemple qui n'a pas vocation à être complet, et qu'il faudrait une initialisation spécifique de la table d'indirection qui n'apparait pas dans cette figure.

La figure 5.9 présente ici un exemple de table d'indirection correspondant à ce qui est créé dans la figure 5.8. Chaque case de la table possède donc un descripteur correspondant à la fonction sur laquelle il pointe. Cette figure montre uniquement un exemple, et par conséquent une partie de table. Il peut y avoir un plus grand nombre de runnables et de *containers* dans chaque table d'indirection. Il est important de noter que chaque tâche possède une table d'indirection qui lui est associée.

Notons qu'un outil est nécessaire pour faire cette transformation. Il est tout à fait possible de faire ces modifications de manière automatique en utilisant un script de *post-processing* des fichiers de l'application. L'avantage d'un outil est qu'il permet de faire les modifications plus rapidement et de mieux s'insérer dans le processus de développement AUTOSAR qui consiste en une chaine outillée.

La figure 5.10 présente les interfaces de l'outil permettant de faire les modifications nécessaires afin d'ajouter un niveau d'indirection. Les informations concernant chacun des runnables sont extraites des fichiers de description des SWCs, qui sont au format `.arxml`. Ce format est celui défini par le consortium AUTOSAR comme format standard de description des différents éléments. Il s'agit d'une extension du format `xml` avec un

```
#define TABLE_SIZE 10

typedef void (*ptr_runnable)(void);
                        // define the function ptr for runnables
                        // runnables do not have return
                        // variables and do not have parameters

typedef struct{                 // structure for describing
        int ID_runnable;        // the content of a cell
        int empty;              // of the indirection table
        }descriptor;

typdef struct{                  // Content of each cell
        ptr_runnable ptr;       // of the indirection table
        descriptor desc;
        }tab_cell;

tab_cell tab_Task_100ms[TABLE_SIZE];

TASK(Task_100ms)
{
        for (i=0; i<TABLE_SIZE; i++)
        {
                /* start runnable */
                tab_Task_100ms[i].ptr();
        }
   TerminateTask();
}
```

FIGURE 5.8 – Exemple de code d'une tâche avec table d'indirection

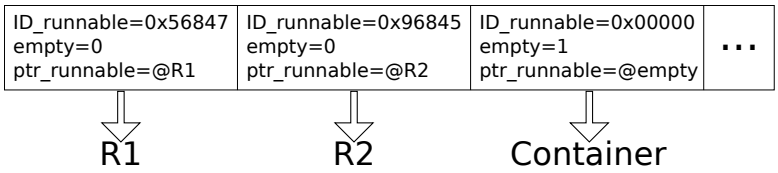


FIGURE 5.9 – Exemple de table d'indirection

certain nombre de balises spécifiques. Les méta-données associées aux runnables ainsi que leurs caractéristiques doivent pouvoir être extraites de cette collection de fichier. Une fois que ces informations concernant les runnables sont extraites, il est possible de modifier le fichier contenant le code des tâches (il s'agit, dans les applications que nous avons pu étudier, du fichier *Rte.c*) afin d'ajouter un niveau d'indirection, ainsi que toutes les informations nécessaires à identifier chaque runnable.

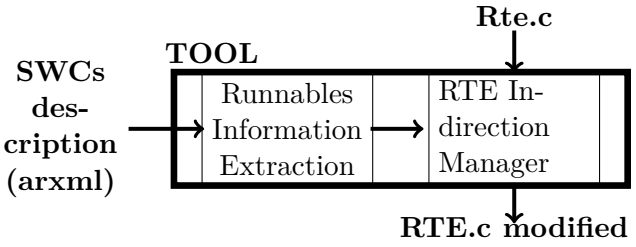


FIGURE 5.10 – Outil de gestion d'indirection

5.4.2 Gestion de la mémoire

Afin de pouvoir ajouter et exécuter des mises à jour dans un ECU embarqué, il est nécessaire de les stocker en mémoire. Il est important de stocker le code des mises à jour en mémoire permanente, qui n'est pas effacée lorsque l'ECU est éteint.

Nous décrivons dans un premier temps les différents types de mémoire qui existent dans un calculateur embarqué, puis nous expliquons comment les différents éléments d'une mise à jour sont gérés et stockés dans la mémoire.

5.4.2.1 Types de mémoire

Il existe deux types de mémoire dans un calculateur embarqué : la mémoire volatile (RAM, Random Access Memory) et la mémoire non-volatile (ROM, Read-Only Memory). La mémoire volatile est utilisée pour stocker des variables amenées à être modifiées et pour lesquelles un accès rapide est nécessaire. La mémoire non volatile (il peut par exemple s'agir de mémoire FLASH) est utilisée pour stocker les instructions du programme à exécuter.

Une différence majeure entre les deux types de mémoire est la persistance des données stockée. Une donnée stockée en ROM ne s'efface pas lorsque la source d'alimentation est coupée, alors que la RAM est effacée.

Dans le cas qui nous intéresse, à la fin de la compilation, le fichier binaire obtenu est de type .elf [37]. Dans ce fichier, l'application est divisée en trois sections principales : la section .bss qui contient toutes les variables non initialisées, la section .data qui contient les variables globales et statiques et la section .text qui contient le code exécutable. La section .text est composée de variables statiques (qui ne changent pas au cours de l'exécution), généralement stockées en ROM.

Dans la RAM, deux sections mémoire supplémentaires sont nécessaires au bon fonctionnement du programme : le tas (*heap*) et la pile (*stack*). Il s'agit ici de mémoire dynamique. La pile contient par exemple les variables locales à une routine (allouées au début de l'exécution, et desallouées à la fin) ou encore les adresses de retour (pour savoir où reprendre l'exécution, une fois que la routine a terminé son exécution). Le tas correspond à de la mémoire partagée à l'échelle du programme, qui peut, par exemple, être utilisée pour les allocations dynamiques de mémoire.

La mémoire ROM, dans les calculateurs que nous avons pu utiliser, est de type

FLASH. Pour le calculateur MPC5646C de chez *Freescale* que nous avons pu utiliser pour notre cas d'étude, la valeur par défaut contenue dans la mémoire est "1". La mémoire FLASH est constituée de secteurs, de tailles variables (entre 16 Ko et 128 Ko). Il n'est possible d'écrire en mémoire FLASH que si l'emplacement mémoire sur lequel nous souhaitons écrire a été au préalable effacé : il n'est pas possible d'écrire un "1" si un "0" est présent dans une case mémoire. De plus, il n'est possible d'effacer de la mémoire FLASH que par secteur complet.

5.4.2.2 Gestion et stockage des mises à jour dans la mémoire

La mise à jour est envoyée par paquets au calculateur, en utilisant un protocole qui permet d'identifier le début et la fin du code spécifique à la mise à jour, ainsi que de s'assurer de l'intégrité des données transmises. Un stockage temporaire en mémoire RAM est donc nécessaire afin de reconstituer intégralement la mise à jour (des paquets peuvent être perdus ou corrompus), et de vérifier son intégrité. En effet, il n'est pas souhaitable d'écrire dans la mémoire FLASH des données corrompues. Notons que ce processus nécessite d'avoir suffisamment d'espace en mémoire RAM pour pouvoir accueillir la mise à jour. La figure 5.11 présente ici les différentes étapes de la mise à jour concernant les modifications qui doivent être faites dans la mémoire. Ainsi, les étapes 1, 2 et 3 présentées sur cette figure correspondent à la reception et au stockage en RAM de la mise à jour dans son intégralité.

Lorsque la mise à jour est complète, il est nécessaire de vérifier son intégrité avant de l'écrire en mémoire non volatile. Pour vérifier son intégrité, il est possible, par exemple, d'utiliser un mécanisme de chiffrement à clef publique. Si jamais la mise à jour n'est pas correcte, elle est alors rejetée (étape 4 de la figure 5.11).

Pour pouvoir écrire en mémoire FLASH, il faut que la mémoire ait été effacée au préalable. Il y a alors deux options : soit il est considéré que la mémoire a été totalement effacée lorsque l'application a été chargée pour la première fois, soit aucune information n'est disponible concernant le contenu du secteur de FLASH. Dans le premier cas, il est possible d'écrire directement à la suite de l'application et aucune manipulation spécifique de la mémoire, autre que l'écriture des octets de la mise à jour, n'est nécessaire. Cependant, cette méthode est un peu plus fragile, si jamais un bit a été écrit par erreur dans la FLASH (par exemple à cause d'une interférence qui aurait perturbé la mémoire). La seconde méthode nécessite d'effacer le secteur de FLASH avant de pouvoir ré-écrire : il faut donc avoir un espace temporaire de stockage qui permette de déplacer le contenu utile du secteur de FLASH à effacer. Cette méthode présente l'avantage de permettre d'effacer une partie des données contenues dans ce secteur (par exemple, si nous souhaitons récupérer de la place dans la mémoire). Cependant, dans ce cas, le nombre de cycles de ré-écriture dans la mémoire peut être important, surtout si ce processus a lieu à chaque mise à jour (étape 5 de la figure 5.11).

Lorsque la mise à jour a été écrite dans la mémoire, il faut modifier les pointeurs de fonctions afin d'exécuter cette mise à jour. Il est donc nécessaire de faire des modifications dans la table d'indirection, telle qu'elle a été présentée dans la section précédente. Si la mise à jour est une nouvelle fonctionnalité, elle doit être placée dans un *container*, et les

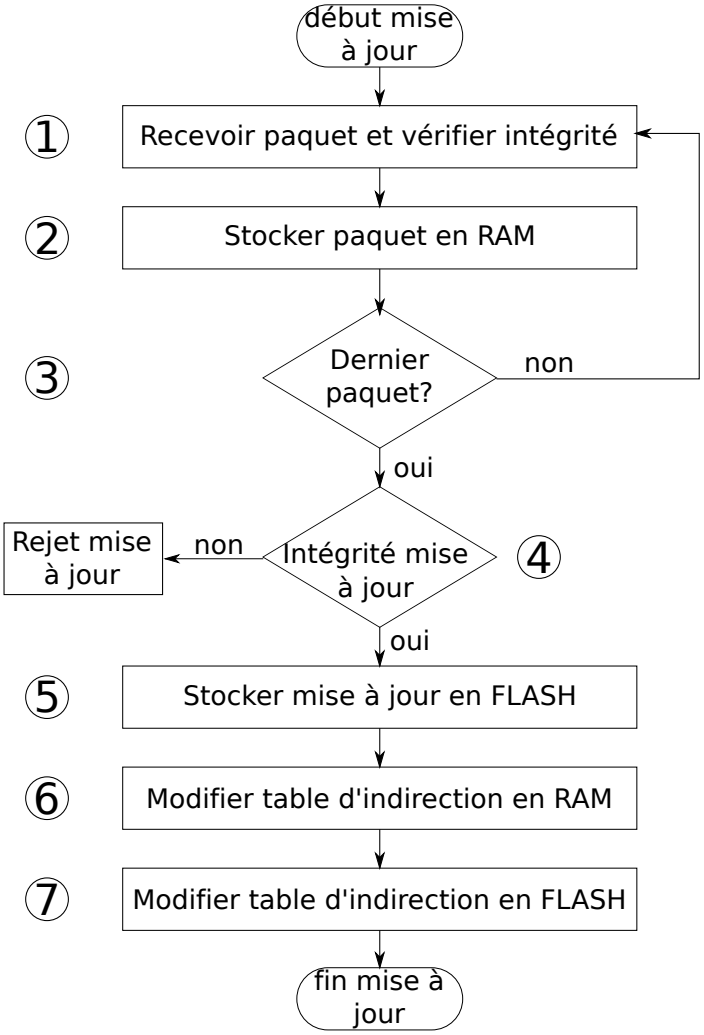


FIGURE 5.11 – Organigramme présentant le processus de réception et stockage d'une mise à jour

informations correspondantes doivent être modifiées. S'il s'agit d'un runnable existant qui est remplacé, la case de la table d'indirection correspondante doit être modifiée, ainsi que les méta-données telles que l'identifiant du runnable. Pendant l'exécution du programme, il est plus optimal de stocker la table d'indirection en mémoire RAM. Cela permet de la modifier plus simplement, et de diminuer le temps d'accès aux données. Toutefois, il est nécessaire d'avoir une copie de cette table en mémoire non volatile, à un emplacement spécifique, afin de ne pas perdre les valeurs de chacune des cases à chaque redémarrage du calculateur. Ainsi, à chaque mise à jour il faut modifier la table d'indirection en RAM ainsi que sa copie en FLASH (étapes 6 et 7 de la figure 5.11).

De plus les méta-données doivent être également mises à jour pour correspondre à la

version et aux éléments qui sont maintenant présents dans le calculateur. Ces données doivent être stockées à un endroit spécifique de la mémoire et modifiées à chaque mise à jour.

Il est important de noter que nous écrivons les mises à jour directement à la suite de l'application précédemment en mémoire. Cela signifie qu'il n'y a pas *a priori* de libération de l'espace mémoire. Ainsi, si une fonction n'est plus utile, elle n'est pas supprimée. Bien que cette méthode soit consommatrice de mémoire, elle a l'avantage de permettre de revenir à n'importe quelle version précédente. De plus, si une libération de l'espace mémoire était envisagée, la mémoire disponible pourrait rapidement être fragmentée. Étant donné le système d'écriture et d'effacement par secteur dans la FLASH, il pourrait être compliqué de réutiliser les espaces laissés libres.

5.5 Sûreté de fonctionnement

Lorsque des mises à jour partielles du logiciel sont faites, il est important de prendre également en compte les contraintes de sécurité-innocuité. Si certains mécanismes liés à la sécurité (notamment en conséquence de l'application de la norme ISO 26262 [57]) sont ajoutés au moment de la création des mises à jour, d'autres points de sécurité, liés à la gestion partielle des mises à jour, doivent être traités spécifiquement.

5.5.1 Généralités

A l'heure actuelle, avec l'augmentation massive de la quantité de logiciel dans les véhicules modernes, plusieurs fonctionnalités, totalement indépendantes en théorie, peuvent se retrouver en pratique sur un même calculateur. Des interactions et des dépendances qui n'existaient pas lorsque ces fonctionnalités étaient gérées avec des éléments mécaniques peuvent maintenant survenir. Cela peut poser problème si une fonction donnée interagit de façon non souhaitée avec une fonction de niveau de criticité supérieur. C'est pour cette raison que la sécurité-innocuité dans les systèmes embarqués est un élément qui doit impérativement être pris en compte. Ce constat est vrai dans de nombreux domaines, comme l'aéronautique, l'automobile, le nucléaire, le ferroviaire ou encore le spatial. Ainsi, dans [22], les auteurs présentent les différents standards qui existent dans ces domaines. Les processus de certification qui peuvent exister dans certains domaines sont également discutés, ainsi que l'interprétation et l'utilisation qui est faite de chacun des standards. La comparaison entre ces standards est également faite sur quelques points techniques.

Les concepts liés à la sûreté de fonctionnement (qui englobe les concepts de fiabilité, disponibilité et maintenabilité) sont le sujet de nombreuses recherches depuis les années 1980 [61]. En effet, certaines défaillances de systèmes ne sont pas acceptables et doivent être évitées par tous les moyens possibles. Les concepts permettant d'avoir des systèmes sûrs de fonctionnement et sécurisés sont développés par Avizienis *et al.* dans [17].

La sécurité-innocuité pour le logiciel embarqué dans les véhicules est un sujet important de préoccupation et c'est pour cette raison que des recherches ont été menées dans le

domaine afin de garantir que les systèmes respectent les propriétés de sécurité-innocuité. Par exemple, Heckemann *et al.* développent le concept de *Safety Cage* [51] qui permet une vérification formelle du comportement du système en fonction du contexte du véhicule. Il est possible de surveiller une ou plusieurs fonctions du véhicule ainsi que les flots de données correspondant. D'autres solutions existent également pour s'assurer de la sécurité-innocuité dans un véhicule. Par exemple, la possibilité pour le logiciel de s'auto-reconfigurer en fonction du contexte afin de garantir une continuité de service [100]. De cette manière, si un évènement indésirable se produit, seules les fonctionnalités essentielles sont conservées et les autres sont desactivées.

Une autre méthode couramment employée pour garantir la sécurité-innocuité est l'utilisation de méthodes de vérification en ligne [65, 41] [39] [83]. La première de ces méthodes [65, 41], qui repose sur le concept de *Safety Bag*, consiste à placer des assertions exécutables qui analysent les données des capteurs et, en fonction du résultat, peuvent corriger les effets sur les actionneurs. Le principe de cette approche est une instrumentation de l'application qui permet d'exécuter séparément le code non fonctionnel. Une autre possibilité est de synthétiser des moniteurs qui surveillent l'exécution du code. Pour cela, Cotard [38, 39] propose un outil appelé *Enforcer* qui permet de synthétiser des moniteurs en charge de vérifier en ligne un certain nombre de propriétés. Cette approche est également axée sur le milieu de l'automobile et a été testé dans le cadre du système d'exploitation temps-réel *Trampoline* [23]. Enfin, dans [83] les auteurs proposent une méthode pour générer des moniteurs à partir de modèles, dans le processus de développement AUTOSAR. Leur approche est fondée sur un framework qui s'intègre en parallèle du développement. Leur approche intervient, non seulement à partir des fichiers de description en `arxml`, mais également à partir des fichiers sources. Certaines étapes nécessitent des transformations manuelles, mais l'avantage de cette approche est qu'elle est totalement parallèle au processus standard et qu'il n'y a pas de boucle entre le framework et ce processus.

Il est également possible d'utiliser des wrappers pour instrumenter les applications AUTOSAR [85]. Les informations nécessaires sont tout d'abord extraites à partir du modèle AUTOSAR, puis utilisées afin de pouvoir injecter dans le code un certain nombre de mécanismes. Ainsi l'objectif final est de pouvoir accéder aux données produites par les composants logiciels et modifier ces données dans le but de faire de l'injection de fautes, et donc de tester la résilience du système à ces fautes.

5.5.2 Sécurité-innocuité : AUTOSAR et ISO 26262

Cette section décrit brièvement tout d'abord la norme ISO 26262 qui définit les notions liées à la sûreté de fonctionnement dans le milieu de l'automobile. Ensuite, nous détaillons les notions de sécurité-innocuité qui ont été introduites dans le standard AUTOSAR.

5.5.2.1 ISO 26262

L'ISO 26262 est la référence pour la sécurité-innocuité dans le domaine de l'automobile. Elle définit les différents niveaux de conception pour les systèmes électriques et électroniques (E/E) et à chaque étape du processus, elle décrit le cadre d'application, les activités et les méthodes à utiliser afin d'obtenir un système sûr de fonctionnement. En particulier, la partie 4 de la norme décrit le développement d'un produit au niveau "système", la partie 5 au niveau "matériel" et la partie 6 au niveau "logiciel".

Elle offre une approche reposant sur l'analyse de risque afin de déterminer pour chaque élément un niveau de criticité appelé ASIL (*Automotive Safety Integrity Level*). Les ASILs sont échelonnés entre A (niveau le moins critique) et D (niveau le plus critique). En fonction de l'ASIL de chacun des éléments, différentes approches peuvent être adoptées et un certain nombre de mécanismes ajoutés. Les questions de validation pour les systèmes E/E sont également mis en avant dans la norme. Enfin, les questions de relation entre les constructeurs et les fournisseurs sont détaillées.

Notre sujet de préoccupation particulier ici étant le logiciel, nous ne présentons rapidement que les mécanismes qui permettent de garantir la sûreté de fonctionnement pour le logiciel (partie 6 de l'ISO). Le tableau 5.1 présente les différents mécanismes que l'ISO recommande de mettre en place en fonction de l'ASIL des éléments. Il y a trois sujets principaux de préoccupation : les flots de données, les flots de contrôle et l'architecture.

Criticité	Flots de données	Flot de contrôle	Architecture
Faible (ASIL A/B)	vérification des intervalles pour les entrées et sorties, contrôle de plausibilité, détection d'erreur sur les données (signature ou redondance)		Mécanismes de récupération statiques
Moyenne (ASIL B/C)	vérification des intervalles pour les entrées et sorties, contrôle de plausibilité, détection d'erreur sur les données (signature ou redondance)	Watchdog, surveillance du flot de contrôle	Mécanismes de récupération statiques
Elevée (ASIL C/D)	vérification des intervalles pour les entrées et sorties, contrôle de plausibilité, détection d'erreur sur les données (signature ou redondance)	Watchdog, surveillance du flot de contrôle	Mécanismes de récupération statiques, diversification de la conception logicielle, redondance parallèle indépendante

TABLE 5.1 – Mécanismes de sécurité-innocuité logiciels pour la détection et la gestion d'erreur dans l'ISO 26262

5.5.2.2 AUTOSAR

La sûreté de fonctionnement revêt une importance capitale dans le domaine de l'automobile, ce qui a conduit en particulier à l'élaboration de la norme ISO 26262. C'est pour cette raison que dans la version 4 de AUTOSAR, un certain nombre de mécanismes devant être présents dans l'architecture embarquée ont été spécifiés [8]. Il s'agit donc de mécanismes qui vont permettre d'implémenter plus facilement au sein des architectures AUTOSAR les concepts définis par la norme ISO 26262 décrite brièvement précédemment.

Les mécanismes requis pour la sécurité-innocuité dans AUTOSAR couvrent 6 domaines :

— le flot de contrôle,
— la synchronisation,
— les contraintes temporelles,
— la pile de communication,
— la protection de bout en bout pour les données,
— le partitionnement de la mémoire.

Concernant le flot de contrôle, l'objectif est de pouvoir détecter les séquences d'instructions incorrectes et la violation de contraintes temporelles. Ainsi, les fautes logicielles doivent pouvoir être détectées (diversification), aussi bien que les fautes matérielles aléatoires et systématiques (redondance physique et temporelle).

Les questions liées à la synchronisation sont de plusieurs natures. Tout d'abord, il est souhaitable d'avoir des bases de temps synchronisées entre les différents ECU. Cela peut permettre de synchroniser l'exécution de différentes fonctionnalités logicielles réparties. Du point de vue des contraintes temporelles, il est important de s'assurer du déterminisme de l'application, en utilisant par exemple des mécanismes qui permettent de définir des sections critiques (mutex ou sémaphore). Pour cela, un ordonnancement statique des tâches est nécessaire, les tâches doivent avoir une priorité fixe, et les ISR doivent être remplacées par des routines de *polling* afin de garantir le déterminisme. Les seules préemptions qui peuvent avoir lieu doivent être par les tâches de priorité supérieure sur les tâches de priorité inférieures. Enfin une surveillance de l'exécution des tâches est mise en place pour vérifier le début de l'exécution, le non-manquement de deadlines et l'utilisation de ressources.

Les communications entre les ECUs doivent être contrôlées, et particulièrement les séquences de données. Il est important de s'assurer qu'aucun signal n'est manquant ou au contraire n'a été envoyé plusieurs fois. Cela permet de garantir une meilleure cohérence globale. De plus des mécanismes de protection de bout-en-bout sur les chaines de données doivent être mis en place, afin d'assurer l'intégrité des données envoyées. Cette protection peut être faite via une librairie AUTOSAR standard qui permet l'encapsulation des communications.

Enfin, un système de partition mémoire doit être présent afin de garantir l'absence d'interférences entre les composants de logiciels : le code qui s'exécute dans une partition mémoire ne doit pas modifier la mémoire d'une autre partition. Cela permet d'éviter la propagation de fautes logicielles systématiques et de fautes matérielles aléatoires. Un

autre aspect important pour prévenir la mauvaise utilisation de certains services est d'avoir différents modes pour l'exécution : le mode utilisateur et le mode superviseur. Chaque partition mémoire s'exécute dans un mode spécifique et certaines routines ne sont pas accessibles depuis le mode utilisateur.

5.5.3 Mécanismes de sécurité-innocuité

Bien que la version initiale de l'application, ainsi que chacune des mises à jour, suivent un processus standard de développement, au cours duquel les questions de sécurité-innocuité sont prises en compte, et les mécanismes nécessaires sont ajoutés, certaines spécificités dues à la mise à jour doivent être considérées. Par conséquent, il est nécessaire d'ajouter des mécanismes dédiés pour gérer deux points en particulier. Tout d'abord les mécanismes permanents qui doivent être ajoutés au cours de la préparation de l'application pour les mises à jour. Ensuite, pendant la création, le transfert et l'installation de la mise à jour des mécanismes supplémentaires doivent être mis en place pour assurer que les mises à jour sont correctes, non corrompues et qu'elles ne perturbent pas le bon fonctionnement du système.

5.5.3.1 Sûreté de fonctionnement et mécanismes pour la mise à jour

Tout d'abord, il doit être possible de revenir en arrière, soit à la version précédente, soit à la version initiale en cas de problème. Bien que cela puisse encombrer la mémoire du calculateur embarqué, car il faut alors laisser du code qui n'est pas exécuté, il est important en cas de problème de pouvoir revenir soit à une version précédente, soit à la version initiale.

D'autre part, l'ajout d'un niveau d'indirection, et par conséquent l'utilisation de pointeurs de fonction conduit à un changement de paradigme pour l'appel des runnables dans les tâches. Ainsi, il faut s'assurer que ces nouveaux mécanismes sont sûrs de fonctionnement.

Roll-Back L'état d'un ECU n'est pas géré de manière centralisé, c'est-à-dire qu'il n'existe pas de serveur centralisé capable de déterminer l'état de n'importe quel véhicule à tout instant. Il est donc nécessaire que l'ECU soit en mesure de gérer seul son état.

Cela signifie que l'ECU doit être capable à tout instant, et ce particulièrement dans le cas où une mise à jour est faite, de détecter une erreur, et en cas de problème de revenir soit à une version précédente du logiciel, soit à la version initiale. Ainsi, si un problème se produit au cours d'une mise à jour, ou bien s'il s'avère que, malgré toutes les vérifications faites, la mise à jour est corrompue, l'ECU peut revenir seul à la dernière version fonctionnelle du logiciel. Ainsi, les versions précédente du logiciel, qui sont toujours présentes dans l'ECU sont considérés comme des modes dégradés potentiel pour l'application. Cela peut être implémenté en utilisant une pile de version stockée en mémoire stable qui permet de retracer toutes les modifications qui ont été faites ainsi que l'ordre dans lequel ces modifications ont été effectuées. Cette pile de version fait partie

des méta-données relatives au calculateur. Dans le cas où un ECU spécifique permettrait de centraliser et de répartir les mises à jour, cette pile de version pour chaque ECU pourrait également être dupliquée sur cet ECU. Cependant, cela reviendrait à ajouter un ECU spécifique pour la mise à jour, ce qui n'est pas forcément possible dans le domaine automobile où la réduction des coûts est un critère de développement important.

En outre, cette méthode permet de retourner à la version initiale du calculateur si un problème important survient suite à une mise à jour. Par exemple si une mise à jour malicieuse est introduite dans le calculateur, et que plusieurs fonctionnalités sont alors corrompues, retourner à l'état initial pourrait permettre de mettre le calculateur dans un mode dégradé, mais sûr de fonctionnement.

Pointeurs et modifications L'utilisation de pointeurs de fonctions, stockés en RAM plutôt que d'appels statiques à des fonctions peut poser un certain nombre de problèmes du point de vue de la robustesse. En effet, si jamais un pointeur est corrompu, alors une mauvaise fonction pourrait être exécutée. Une conséquence plus grave de la corruption d'un pointeur serait de bloquer complètement le logiciel du calculateur. Cela pourrait alors conduire à des défaillances catastrophiques de l'ECU.

Ainsi, il est très important de vérifier que les pointeurs ont des valeurs correctes. Pour cela, la méthode qui est privilégiée est l'utilisation de redondance pour les données. Différentes méthodes de redondance peuvent être utilisées :

— duplication des adresses et comparaison avant le lancement du runnable,
— triplication des adresses et vote majoritaire avant le lancement du runnable,
— utilisation de codes correcteurs et détecteurs d'erreur, type code de Hamming ou code de Reed-Solomon,
— utilisation de signatures cryptographiques si seule la détection d'erreur est intéressante et non le recouvrement.

L'avantage de la duplication (ou de la triplication) réside dans la simplicité de comparaison entre la valeur du pointeur et sa copie. Cependant, le stockage de l'information prend plus de place. La signature de donnée, d'autre part, a l'avantage de prendre moins de place dans la mémoire, mais il faut vérifier que la signature correspond à la donnée, ce qui utilise des algorithmes, et donc du temps de calcul. Dans notre cas, étant donné que nous ne considérons pas les fautes malicieuses, une signature classique reposant sur des algorithmes cryptographiques symétriques (type AES) peut suffire.

Ainsi, chaque fois qu'un pointeur de fonction est utilisé plutôt qu'un appel direct à la fonction, il est nécessaire de faire une vérification, ce qui peut prendre du temps. Toutefois, le fait de stocker en RAM plutôt qu'en FLASH permet de diminuer les temps d'accès aux données, et donc de compenser l'ajout de mécanismes de vérification.

5.5.3.2 Sûreté de fonctionnement et création de mise à jour

Lorsque la mise à jour est créée, il est nécessaire d'ajouter un certain nombre de mécanismes de sûreté de fonctionnement en lien avec l'analyse de safety qui doit être faite. Après la création de la mise à jour, cette dernière doit être transférée dans le calculateur embarqué, et à ce moment là, un certain nombre de mécanismes de sécurité

doivent également être mis en place pour s'assurer, non seulement que la mise à jour est intègre, mais également que l'entité qui a émis cette mise à jour est fiable et que son exécution ne va pas avoir de conséquences néfastes.

Pendant la création de la mise à jour (offline) Lorsque la mise à jour est créée, il est nécessaire de suivre le processus de développement standard, et de passer par toutes les étapes de modélisation. De cette manière, à chaque étape, le développement doit suivre les règles liées à la sécurité-innocuité, et en particulier s'assurer d'être en conformité avec le standard ISO 26262.

Ainsi, au cours du développement, les analyses de *safety* doivent avoir lieu, ainsi que les tests unitaires qui doivent permettre de vérifier que la mise à jour est fiable et que le comportement global du système qui intègre la mise à jour correspond bien à ce qui est attendu.

Pour le transfert de la mise à jour dans le calculateur Il est nécessaire de mettre en place un système de signatures électroniques afin de s'assurer que la mise à jour qui a été envoyée au calculateur est non seulement intègre, mais également qu'il est possible d'avoir confiance dans l'émetteur de cette mise à jour (authentification).

Ce type de système doit permettre d'identifier de manière certaine l'identité de celui qui a signé et envoyé la mise à jour et qu'il ne soit pas possible de falsifier, de réutiliser la signature. Ainsi, lorsque la mise à jour est envoyée (par exemple depuis un serveur sécurisé du constructeur automobile), il est possible pour le calculateur de vérifier si cette mise à jour est conforme et si elle a bien été envoyée par le constructeur et non par une source frauduleuse. L'identification peut reposer sur un système de mécanisme à clefs publiques. Dans ce cas, la mise à jour est signée grâce a un système de chiffrement asymétrique : le constructeur possède une clef privée avec laquelle il signe les mises à jour, et dans chaque véhicule est embarqué la clef publique avec laquelle il est possible de vérifier l'authenticité de la mise à jour.

Ce système permet également de vérifier qu'il n'y a pas eu de problème pendant le transfert et que la mise à jour n'a pas été corrompue (par exemple par un bit qui aurait été mal transmis).

La sûreté de fonctionnement passe donc ici par une vérification de l'intégrité de ce qui est intégré dans le système initial.

Pendant l'installation de la mise à jour Une modification importante qui doit être faite, pour l'installation et surtout pour la bonne exécution ultérieure de la mise à jour, porte sur la table d'indirection qui contient tous les pointeurs de fonctions ainsi que la description de chacun d'entre eux. Nous avons décrit précédemment cette table, ainsi que les descripteurs associés.

Il est important que cette table soit protégée de toute modification frauduleuse ou accidentelle. Pour cette raison, en utilisant des mécanismes disponibles dans les calculateurs embarqués (tels que la MMU, "Memory Management Unit", qui permet de gérer les permissions de lecture et d'écriture dans la mémoire), il est nécessaire de protéger cette

table en ne la modifiant que dans un mode privilégié. Cela signifie que le segment de la mémoire qui contient la table d'indirection peut uniquement être lu en temps normal, et n'est modifié que lorsqu'une mise à jour est faite, qui se déroule en mode privilégiée. Il est également nécessaire lorsque cela se produit de dupliquer ces modifications en mémoire FLASH pour en assurer la persistance.

5.5.3.3 Conclusion

La figure 5.12 montre les différentes étapes de création et de chargement dans le calculateur d'une mise à jour. Ces étapes ont, pour la majorité, été décrites précédemment dans notre chapitre. Notre objectif ici est de pointer les différents mécanismes de sûreté de fonctionnement qui peuvent permettre d'assurer un niveau de confiance acceptable lors des mises à jour. Ces mécanismes peuvent permettre de répondre à des besoins de l'ISO 26262 allant jusqu'à l'ASIL D.

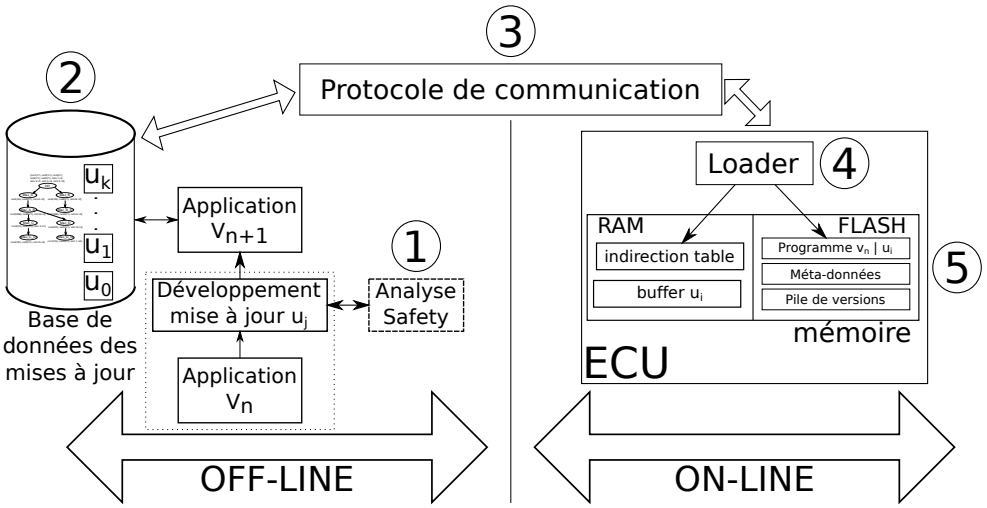


FIGURE 5.12 – Etapes de création et de chargement d'une mise à jour

Dans cette section, le modèle de faute que nous adressons correspond aux **fautes transitoires accidentelles**. En effet, bien qu'il puisse exister également des fautes malicieuses ou des attaques sur ce type de système, ces questions ne sont pas traitées dans le cadre de ce travail.

La première étape (étape 1 de la figure 5.12 consiste donc à créer la mise à jour selon le processus qui a été décrit section 5.2.2. Cette étape a lieu hors-ligne, c'est-à-dire en dehors du calculateur et indépendamment de son exécution. Le processus de développement des mises à jour doit se faire dans le contexte de l'application embarquée dans le calculateur et en respectant les règles, telles que celles recommandées par l'ISO 26262. Des tests unitaires, ainsi que des tests d'intégration des mises à jour doivent également être mis en place afin de s'assurer du bon fonctionnement de chaque mise à jour.

Une fois que la mise à jour est créée, elle doit être placée dans la base de donnée des mises à jour (étape 2 de la figure 5.12). Dans cette base de données, nous avons donc les éléments correspondant à chaque mise à jour élémentaire, ainsi que l'arbre de mise à jour qui permet de déterminer si chacune d'elle peut être (ou non) ajoutée à un système donné.

Lorsqu'une mise à jour est demandée par un utilisateur, il faut donc faire une requête sur la base de données, en fonction de la version déjà embarquée dans le calculateur, puis utiliser un protocole de communication pour envoyer la mise à jour au véhicule. Ce protocole doit bien sûr être sécurisé afin de permettre une identification, d'une part du véhicule demandant la mise à jour, et d'autre part du serveur qui va lui envoyer. Du côté utilisateur, l'identification peut être faite par une carte à puce, alors que du côté du serveur il est possible d'utiliser un mécanisme de signature cryptographique (KPi) (étape 3 de la figure 5.12).

Lors de l'étape 4, c'est-à-dire du chargement dans la mémoire du calculateur, il est nécessaire de faire des vérifications d'intégrité, par exemple en utilisant des codes de type CRC.

Il est ensuite nécessaire de modifier la table d'indirection, non seulement en mémoire RAM, mais également en mémoire FLASH. Cette modification est très importante et aucune erreur ne peut arriver lors de cette étape car cela aurait pour conséquence potentielle une défaillance catastrophique du système. Il est donc nécessaire de mettre en place une redondance des données, sous forme par exemple de signature électroniques ou de codes "détecteur-correcteur" d'erreurs.

La pile de version doit également être mise à jour de manière fiable, ainsi que les méta-données associées à cette mise à jour. Enfin, lors de l'exécution, la prévention de fautes transitoires peut être faite en utilisant de la redondance temporelle. Il s'agit alors d'exécuter plusieurs fois un même programme afin de vérifier que le résultat est le même. Si le résultat est identique, alors ce dernier est considéré comme acceptable. Dans le cas contraire, il ne peut pas être considéré comme fiable.

6

Application au cas des clignotants

Sommaire

6.1 Introduction

Ce chapitre présente la mise en pratique des différents concepts que nous avons développés jusqu'à présent sur une application réelle. Cette dernière gère les clignotants et les warnings, sur un seul ECU (pas de reconfiguration réseau). Elle est composée de 8 runnables (deux capteurs, runnables de traitement et deux actionneurs). Deux mises à jour sont considérées pour cette application : l'allumage des warnings en cas de freinage d'urgence et les clignotants impulsionnels (en cas d'appui bref sur le capteur le clignotant correspondant s'actionne 3 fois).

Une des raisons du choix de cette application est que nous avons accès à tous les éléments de l'application, ce qui facilite les manipulations. Nous avons accès directement au code source de l'application, ce qui est utile pour certains tests. En effet, pour des raisons de propriété intellectuelle, dans le cas où la création de l'ECU est déléguée à un équipementier, il est habituel que seul le code objet des composants soit disponible.

Cependant, il est important de noter que l'accès complet au code source des SWCs n'est pas nécessaire pour être en mesure de faire des mises à jour selon la méthode que nous avons présentée dans cette thèse, mais a permis d'en faciliter l'évaluation.

Nous présentons ici dans un premier temps l'application initiale que nous utilisons comme preuve de concept avec les différentes étapes de configuration. Puis nous présentons la préparation de l'application d'un point de vue temps-réel et structurel, et en particulier la préparation de l'application avec les *containers* afin de permettre une plus grande flexibilité.

Enfin, nous présentons les différentes mises à jour possibles pour cette application ainsi que les mécanismes qui permettent de gérer ces mises à jour.

6.2 Outils et matériel

Cette section présente rapidement la cible matérielle que nous avons utilisée pour exécuter l'application des clignotants et vérifier la faisabilité de notre approche sur une cible embarquée, ainsi que les outils utilisés. *Trampoline* est un OS temps-réel open source qui permet d'exécuter les tâches selon une politique d'ordonnancement à priorité fixe. Ensuite, nous avons utilisé l'outil *Otawa* pour estimer le WCET des runnables.

6.2.1 Cible matérielle

Nous avons effectué nos tests sur deux cibles matérielles de type PowerPC (Performance Optimization With Enhanced RISC - Performance Computing) par Freescale. Ce type de processeur est très utilisé pour les applications embarquées. Dans le domaine de l'automobile, ils peuvent être utilisés pour les fonctions liées aux essuie-glaces, à l'éclairage, ou aux lèves-vitres.

Les microprocesseurs de type RISC [82] sont construits de manière à utiliser un ensemble d'instructions réduit, ce qui, combiné à un processeur capable de traiter les instructions rapidement, permet de réduire le temps de traitement.

Le processeur est intégré dans une carte de démonstration également fournie par Freescale. Cette carte possède des bus de communication (SPI, CAN, I2C, RS232), une horloge, des LEDS, des boutons poussoir, etc.

6.2.2 Trampoline

Nous utilisons dans ce travail le système d'exploitation temps-réel *Trampoline* [23]. Il s'agit d'une implémentation open source des concepts définis dans le standard OSEK/VDX [79] compatible avec une cible PowerPC.

AUTOSAR OS repose également sur le standard OSEK/VDX, et des évolutions de *trampoline* ont permis de rendre le système d'exploitation compatible avec le standard AUTOSAR. *Trampoline* est non seulement compatible avec un certain nombre de cibles matérielles embarquées, mais également avec POSIX, ce qui permet de procéder à des tests préalables (bien que les aspects temporels et gestion de la mémoire ne soient pas nécessairement représentatifs).

6.2.3 Otawa

Otawa [19] est un outil qui permet, entre autre, d'évaluer le WCET d'éléments présents dans un fichier binaire.

Nous ne pouvons pas avoir accès au script de description précise de l'architecture matérielle du calculateur. Pour cette raison, notre objectif est uniquement d'avoir une estimation approximative des WCETs des runnables et non des valeurs précises et spécifiques au matériel cible utilisé. En effet, l'application que nous utilisons sera exécutée seule sur le calculateur. Pour cette raison, nous utilisons un script simple d'une architecture sans cache qui considère que chaque instruction demande 5 cycles pour être exécutée.

En utilisant cet outil, nous avons déterminé des valeurs pour les WCETs des runnables de notre application. La table 6.1 donne ici les valeurs obtenues. Ces dernières sont exprimées en temps de cycles et non en valeur absolue dans la mesure où nous utilisons un script de description architecturale générique et non spécifique à notre architecture matérielle.

Capteur 1	TssRunnable	560 cycles
Capteur 2	WlsRunnable	490 cycles
Traitement 1	TssPreprocessing	15 cycles
Traitement 2	WlsPreprocessing	30 cycles
Traitement 3	Logic	215 cycles
Traitement 4	Toggle	285 cycles
Actionneur 1	FlaRunnable	135 cycles
Actionneur 2	FraRunnable	135 cycles

TABLE 6.1 – WCET des runnables obtenus avec l'outil Otawa

A partir des WCETs exprimés en temps de cycles, il est possible d'obtenir les WCETs

en valeur absolue (t) en utilisant la formule suivante : $t = \frac{WCET}{Freq}$ où $Freq$ représente la fréquence du calculateur.

6.3 Application initiale et processus de développement

Cette section détaille l'application que nous utilisons pour mettre en pratique les concepts définis dans les chapitres précédents. Il s'agit d'une application gérant les clignotants en se conformant aux contraintes liées à la règlementation en vigueur. L'intérêt de cette application est qu'elle permet d'avoir un retour visuel, qu'elle est de taille intéressante et qu'elle comporte des runnables de différents types.

Nous donnons ensuite les différents modèles liés à cette application, à savoir le modèle UML haut niveau, et le modèle AUTOSAR.

Il est important de noter que les clignotants (et les warnings) ont un rôle sécuritaire dans la voiture et qu'il existe par conséquent un certain nombre de contraintes spécifiques liées à ces derniers. Les temps de réponse entre le moment d'activation du capteur et le moment où les actionneurs sont effectivement déclenchés, ainsi que le comportement global des actionneurs doivent être conformes à la règlementation européenne.

6.3.1 Règlementation liée aux clignotants

La conception et le positionnement des clignotants dans une voiture doivent répondre à un certain nombre de contraintes définies par la commission européenne [3]. Nous résumons ici les principales caractéristiques qui doivent être respectées par les clignotants, et qui peuvent être implémentées par du logiciel [101]. Il est important de noter que d'autres règles doivent être suivies, en particulier par rapport au positionnement des clignotants sur la voiture et l'angle de visibilité. Cependant, ces règles ne seront pas détaillées ici.

La présence de clignotants est obligatoire pour les véhicules. Leur implantation dépend cependant du type de la voiture (en particulier concernant la présence de répéteurs latéraux). Quelle que soit l'implantation des signalisations, tous les indicateurs de direction d'un même côté du véhicule doivent être allumés ou éteints par le même contrôleur et doivent clignoter de manière synchronisée. Le témoin de fonctionnement peut être visuel, auditif, ou bien les deux. En cas de problème de fonctionnement sur les clignotants, le témoin visuel doit soit rester éteint, soit rester allumé ou bien changer la fréquence de clignotement. Le témoin auditif, quant à lui, doit être clairement audible et doit changer de fréquence en cas de dysfonctionnement.

Les indicateurs de direction doivent clignoter 90 ± 30 fois par minute. Entre le moment où le contrôleur des clignotants est enclenché et le moment où l'indicateur s'allume pour la première fois, il ne doit pas s'écouler plus d'une seconde, puis après au maximum une demi seconde l'indicateur doit s'éteindre pour la première fois.

Les warnings, d'autre part, réutilisent une grande partie des contraintes précédemment définies pour les clignotants, mais ont également certaines contraintes propres. En

3. `http://www.unece.org/`

particulier, il doit y avoir un contrôleur manuel séparé spécifique pour les warnings qui permet de déclencher l'activation simultanée des indicateurs de direction et lors de cette activation, tous les indicateurs doivent clignoter en phase.

Il est important de noter que l'application présentée ici est une application de test. Les applications qui sont effectivement chargées dans les calculateurs des voitures peuvent différer.

Maintenant que nous avons expliqué les raisons de notre choix pour cette application ainsi que les contraintes qui doivent être respectées pour satisfaire la règlementation européenne, nous pouvons détailler l'application. Cette description repose sur le processus de développement qui a été décrit au Chapitre 3.

6.3.2 Besoins fonctionnels et évènements indésirables

La définition des besoins fonctionnels correspond à la première étape du processus de développement (étape 1 sur la figure 3.3).

L'application qui gère les clignotants doit répondre à deux besoins distincts :

— Lorsque le levier (comodo) est activé, il indique l'intention de tourner à droite ou à gauche du conducteur.

— En cas d'activation de la commande des warning, il faut faire clignoter de manière synchronisée les indicateurs de changement de direction.

Il existe des évènements indésirables pour les clignotants comme pour les warnings.

Un évènement indésirable pour les clignotants peut être l'actionnement du capteur et l'absence de réaction de l'actionneur sans que le conducteur n'en soit informé. Un autre problème qui pourrait se produire serait l'actionnement du mauvais indicateur.

Pour les warnings, le principal évènement indésirable serait l'absence d'action sur les indicateurs lumineux lorsque la commande est actionnée par le conducteur.

6.3.3 Modèle fonctionnel logiciel global

La première étape nécessaire est la conception du modèle fonctionnel logiciel global. Nous avons choisi ici d'utiliser le langage UML pour ce modèle car il s'agit d'un standard largement utilisé dans le monde industriel.

La première étape de ce modèle consiste donc à définir les cas d'utilisation de la fonction clignotant. Nous en avons défini ici deux qui correspondent à chacun des deux besoins fonctionnels exprimés à l'étape précédente. La figure 6.1 présente les deux cas d'utilisation identifiés. Dans les deux cas, les acteurs sont les mêmes : le conducteur agit sur le système et la conséquence est que les ampoules doivent s'allumer.

Nous traitons donc ensuite chacun des deux cas d'utilisation séparément dans un premier temps, puis nous groupons les éléments communs dans les chaines de calcul. Il est important de noter que les contraintes de type ASIL, contraintes de temps de bout en bout ou encore contraintes temps-réel sont également modélisées dans l'architecture fonctionnelle. Pour simplifier, ici les contraintes seront présentées directement dans la description.

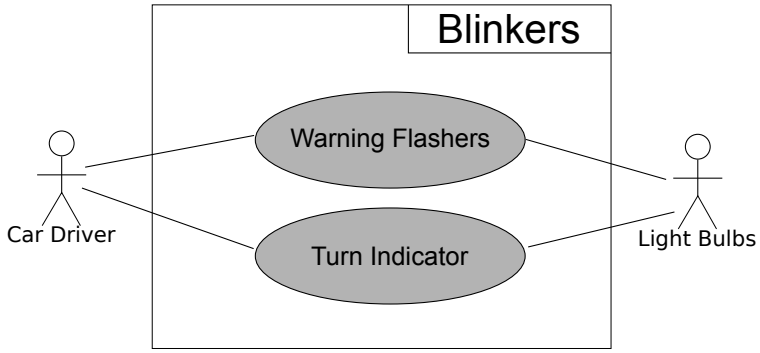


FIGURE 6.1 – Diagramme de cas d'utilisation pour les clignotants

En premier lieu, chacun des cas d'utilisation est traité séparément, à la fois du point de vue comportemental (avec un diagramme d'activité) et sur le plan temporel (avec un diagramme de séquence). D'autres diagrammes sont également nécessaires, en particulier pour faire figurer toutes les contraintes : les niveaux d'intégrité (ASIL [57]), les contraintes de temps de bout-en-bout ou les caractéristiques temps-réel de chacun des éléments (*minimum inter-arrival time*, *execution time*, *response time*).

6.3.3.1 Indicateur de changement de direction

La conception préliminaire des indicateurs de changement de direction (clignotants) est plutôt simple. La figure 6.2 montre les éléments de base («Part») nécessaires pour traiter les informations issues du capteur (comodo) et transmettre les commandes aux actionneurs.

L'élément "TurnSwitchSensor" représente le capteur permettant de savoir si l'indicateur de changement de direction doit être actionné. Il s'agit ici d'un élément du Basic Software : bien que les éléments du Basic Software n'apparaissent pas directement dans le modèle haut niveaux, les dépendances et les besoins des éléments applicatifs pour les services fournis par le Basic Software doivent être exprimés. A l'autre extrémité de la chaine de calcul, "FrontLeftActuator" et "FrontRightActuator" représentent les actionneurs. Les éléments "ToggleBulb" et "SelectBulb" sont, quant à eux, des éléments de traitement.

Il est important de noter que les éléments dont il est question ici ne sont pas nécessairement des runnables. Le groupement éventuel d'éléments en runnables est fait dans une étape ultérieure de la modélisation.

La figure 6.2 montre les différents éléments et les données qui sont échangées entre ces éléments. Cependant, des informations supplémentaires sont nécessaires, à savoir les questions temporelles concernant les échanges de messages. C'est pour cette raison que nous modélisons également le diagramme de séquence correspondant sur la figure 6.3, qui présente la chronologie entre les différents éléments dans le cas nominal. Il est également possible de modéliser les contraintes temporelles liées à la règlementation européenne. Par exemple le critère TLampOn doit permettre de répondre à la condition "entre le

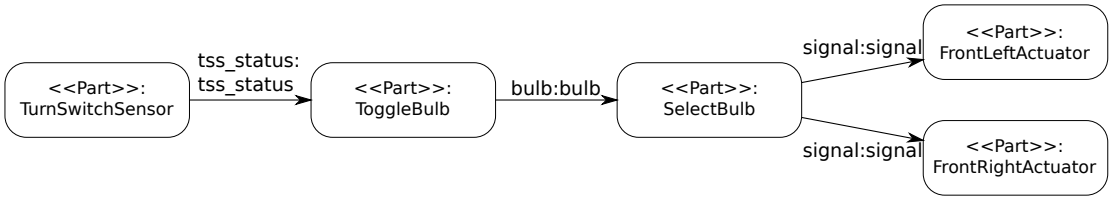


FIGURE 6.2 – Chaine de calcul pour les clignotants (conception préliminaire)

moment où l'indicateur s'allume pour la première fois et le moment où il s'éteint pour la première fois, il ne doit pas s'écouler plus d'une demi seconde".

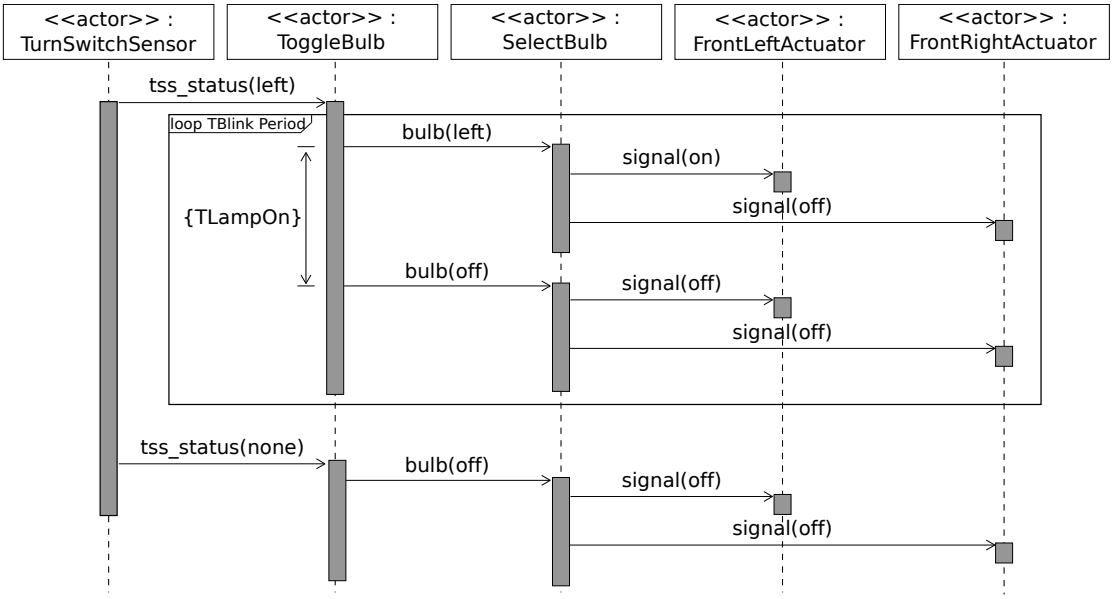


FIGURE 6.3 – Comportement pour les clignotants (conception préliminaire)

Les contraintes temporelles sont celles qui ont le plus grand impact sur les clignotants : il ne doit pas s'écouler plus de 100ms entre le début de l'exécution du premier élément de la chaine et la fin de l'exécution du dernier élément. Ces contraintes peuvent également être modélisées et ajoutées dans le modèle UML.

L'analyse de sécurité-innocuité permet de déterminer les ASILs associés à chaque évènement redouté. Il existe deux évènements redoutés pour les clignotants :

— La perte des indicateurs de direction,
— L'inversion des indicateurs de directions (par exemple, le conducteur actionne le capteur correspondant au clignotant gauche et c'est le clignotant droit qui clignote).

Le premier évènement indésirable est classé ASIL A, alors que le second est classé ASIL B. Pour cette raison, nous allouons à la chaine qui gère les clignotants un ASIL B.

6.3.3.2 Warnings

Les warnings présentent une chaine de calcul proche de celle des indicateurs de changement de direction, mais plus simple car les deux actionneurs doivent fonctionner simultanément. Il n'y a donc pas de module de choix entre l'actionneur droit et l'actionneur gauche. Ainsi, la figure 6.4 présente la chaine de calcul pour les warnings. L'élément le plus à gauche représente le capteur pour les warnings et les deux éléments à droite les actionneurs. Il n'y a ici qu'un seul élément de traitement : "Toggle".

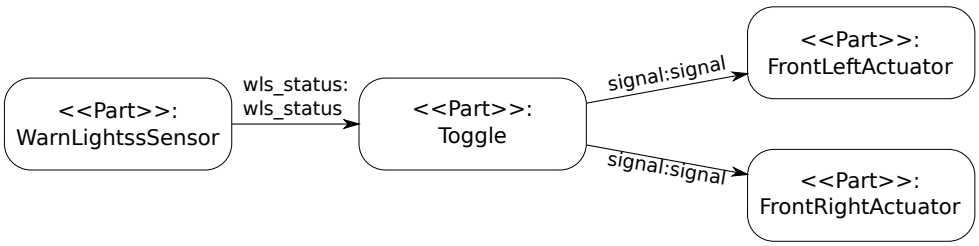


FIGURE 6.4 – Chaine de calcul pour les warnings (conception préliminaire)

La figure 6.5 présente le diagramme de séquence pour les warnings dans leur cas de fonctionnement nominal. Il est important de noter qu'un critère lié la règlementation européenne pour les warnings est la synchronisation pour l'allumage et l'extinction des actionneurs.

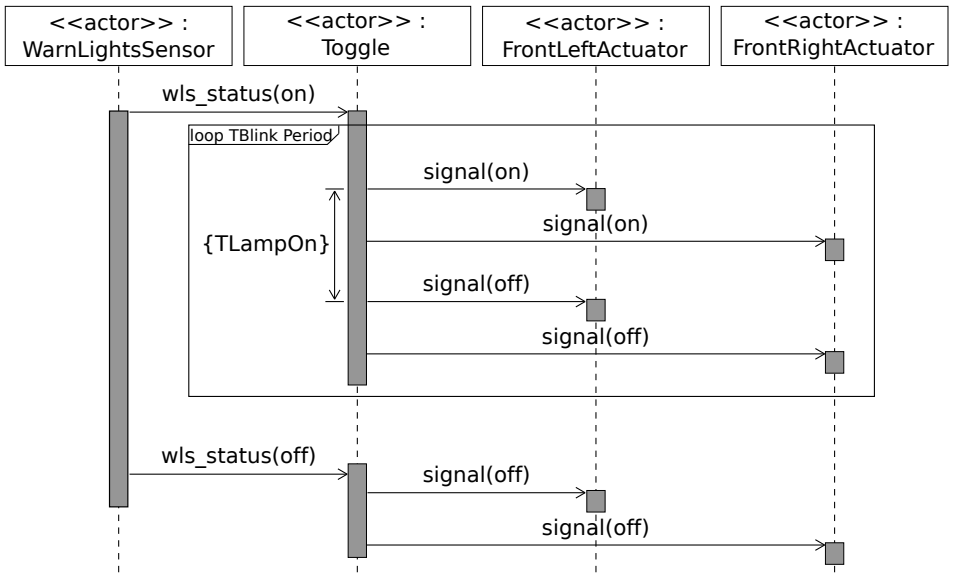


FIGURE 6.5 – Comportement pour les warnings (conception préliminaire)

Les contraintes temporelles pour les warnings sont directement liées à la règlementation européenne. Les contraintes de temps de bout en bout entre le premier élément

de la chaine (lecture du capteur) et le dernier élément (activation des actionneurs) ne doivent pas excéder 100ms.

Il existe également des contraintes de sécurité-innocuité liées aux warnings, tout particulièrement dans le cadre d'un développement ISO 26262 [57]. Une analyse de sécurité est faite afin d'identifier les évènements indésirables liés aux warnings. Cette analyse est faite en parallèle de la conception logicielle et c'est elle qui permet de déterminer les ASIL des différents éléments. La perte de signalisation est le seul évènement redouté pour les warnings. Cet évènement représente un ASIL QM (Quality Management), *i.e.* nous considérons qu'il n'a pas d'impact sur la sécurité.

6.3.3.3 Conception avancée

Après avoir traité séparément les deux cas d'utilisation, nous déterminons les éléments communs aux deux chaines de calcul pour les regrouper en une chaine de calcul plus complexe.

La figure 6.6 présente une première approche pour rassembler les deux chaines de calcul présentées précédemment. Cette chaine rassemble donc les capteurs pour les clignotants et pour les warnings, ainsi que les actionneurs, qui sont identiques dans les deux cas. Concernant le traitement des informations, un nouvel élément "Logic" a été ajouté pour traiter les deux sources d'informations. Les éléments "ToggleBulb" et "SelectBulb" qui ont été identifiés pour la chaine de calcul des clignotants sont conservés et leur comportement doit être adapté pour le cas où les warnings sont activés.

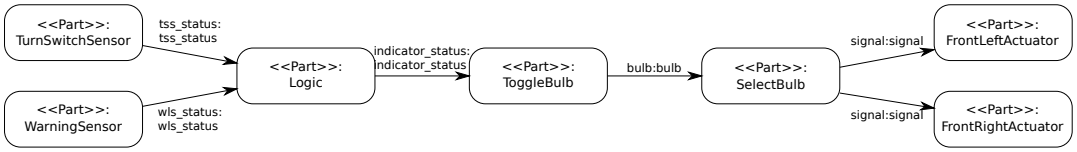


FIGURE 6.6 – Chaine de calcul intermédiaire

Enfin, la conception finale avec l'ensemble des éléments de la chaine de calcul est présentée sur la figure 6.7. Cette figure présente également le regroupement des éléments en runnables : ce regroupement est représenté par des rectangles en pointillés avec le nom des runnables associés. Le seul regroupement qui est fait dans le cas de cette application est celui des éléments "ToggleBulb" et "SelectBulb" dans le runnable "Toggle". Cela permet de regrouper des fonctionnalités proches qui autrement pourraient mener à des runnables de taille réduite. Nous avons ici également supprimé les parties "TurnSwitchSensor" et "WarningSensor" car il s'agit de capteurs, qui sont par conséquent des éléments du Basic Software. Nous avons donc créé ici à la place des runnables "TssRunnable" et "WlsRunnable" qui permettent de récupérer, au niveau applicatif, les informations fournies par les capteurs. Le même processus est appliqué pour les actionneurs.

Nous ajoutons également des éléments de pré-traitement des informations, à savoir "TssPreprocessing" et "WlsPreprocessing". Ces derniers ne sont pas forcément nécessaires dans le cas que nous traitons ici, mais ils peuvent se révéler utiles dans le cas où

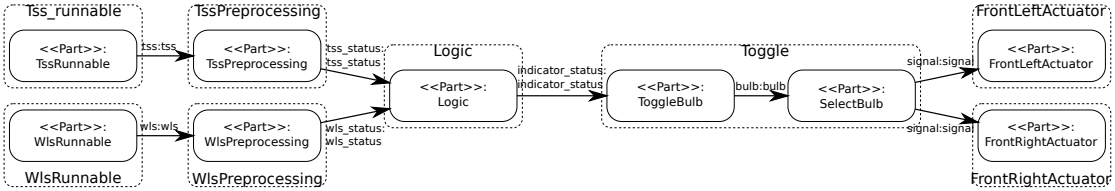


FIGURE 6.7 – Chaine de calcul consolidée

l'information peut être issue de sources différentes, par exemple dans le cas où les warnings doivent être actionnés autrement que par un appui sur le bouton correspondant.

6.3.4 Modèle AUTOSAR

Pour cet exemple, nous ne gérons pas l'aspect distribué de l'application dans la mesure où tous nos composants logiciels sont placés sur le même calculateur. Pour cette raison, nous ne présentons pas ici la vue VFB, mais uniquement la vue RTE.

Nous présentons donc le regroupement des runnables identifiés lors de la phase de conception. Tous les runnables spécifiques qui récupèrent les informations issues des capteurs ("TssRunnable" et "WlsRunnable") ou qui envoient des informations aux actionneurs ("FlaRunnable" et "FraRunnable") sont placés dans des SWCs séparés. Il y a plusieurs raisons pour cela, tout d'abord parce que ces éléments ne sont pas réutilisables : en effet les capteurs et les actionneurs sont spécifiques à une cible matérielle donnée et ne peuvent pas être facilement réutilisés. L'autre raison pour laquelle ces runnables sont placés dans des composants séparés est liée à l'emplacement de ces composants : généralement ils sont placés dans des ECUs spécifiques en fonction de la topologie du réseau. Pour plus de flexibilité, il est donc intéressant de placer les capteurs et actionneurs séparément. Les autres runnables de traitement sont tous placés au sein du même composant.

La figure 6.8 présente le regroupement qui a été fait pour les différents runnables de l'application. Cette figure présente également les évènements qui déclenchent les différents runnables, les ports d'entrée/sortie des composants et les IRV. Les flèches bleues dans ce modèle permettent de montrer par quels runnables les données qui transitent par les ports sont utilisées. Au sein d'un même SWC, les flêches bleues avec une annotation de variable en bleu également représentent les IRV. Dans ce cas, des canaux du RTE ne sont pas nécessaires pour gérer ces variables partagées par des runnables d'un même composant logiciel. Les ports hachurés représentent des communications avec le Basic Software. Enfin les flèches en pointillés représentent les évènements qui déclenchent l'exécution des runnables.

Les runnables chargés du traitement des données issues du capteur pour les warnings ("WlsRunnable" et "WlsPreprocessing") sont exécutés à la réception de données : il s'agit donc de runnables évènementiels. Les autres runnables sont tous périodiques.

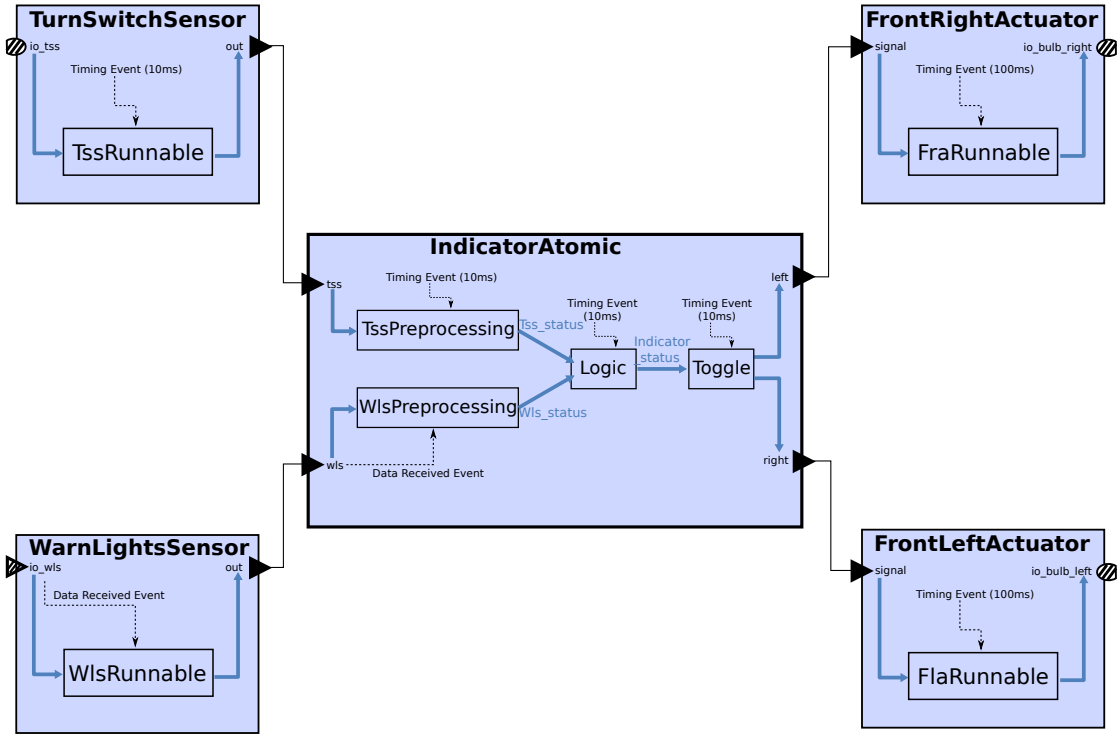


FIGURE 6.8 – Vue AUTOSAR des runnables

6.3.5 Allocation des runnables sur les tâches et modèle de tâche associé

Les runnables évènementiels sont placés chacun seuls dans une tâche étendue pour attendre l'évènement qui déclenche leur exécution. Les runnables qui gèrent les actionneurs doivent s'exécuter à une période de 100 ms. Le traitement préalable des informations envoyées aux actionneurs se fait à 10 ms pour la chaine de calcul qui gère les clignotants et est gérée par des évènements pour les warnings.

Ainsi, nous avons besoin de 4 tâches pour allouer tous les runnables : une tâche à 100 ms pour les actionneurs, deux tâches évènementielles pour la chaine de traitement des warnings et une tâche à 10 ms pour les runnables de traitement des clignotants.

Les tâches et leurs caractéristiques sont présentées dans la table 6.2. Les tâches évènementielles ont ici des priorités plus élevées que les tâches périodiques. Pour les tâches périodiques, plus la période est faible, plus la priorité est élevée. Nous rappelons que dans AUTOSAR OS, une priorité plus élevée est caractérisée par un nombre plus grand. Cette table donne également l'allocation de chacun des runnables dans les 4 tâches du système.

6.3.6 Modélisation temps-réel

Cette section décrit les contraintes temps-réel pour notre application, en particulier les contraintes de précédence et l'analyse d'ordonnancement. Nous présentons également

Tâches	Caractéristiques	Runnables
Task_100ms	Priorité : 1, Full-Preemptive, Période : 100ms	FlaRunnable, FraRunnable
Task_10ms	Priorité : 2, Full-Preemptive, Période : 10ms	TssRunnable, TssPreprocessing, Logic, Toggle
Task_Wls	Priorité : 4, Non-Preemptive, Évènementielle	WlsRunnable
Task_WlsPreprocessing	Priorité : 3, Full-Preemptive, Évènementielle	WlsPreprocessing

TABLE 6.2 – Caractéristiques et allocation des runnables sur les tâches

les résultats de l'analyse de sensibilité pour vérifier qu'il y a suffisamment de place dans l'application pour pouvoir ajouter de nouvelles fonctionnalités. Dans les faits, l'application que nous utilisons dans cette preuve de concept est relativement simple. Par conséquent, le système est assez peu chargé.

6.3.6.1 Contraintes de précédence

Le modèle fonctionnel logiciel présenté précédemment permet de déterminer la matrice de précédence pour le système présenté. Il faut vérifier que les contraintes de précédence peuvent être respectées et que les données sont transmises uniquement des tâches de priorités hautes vers les tâches de priorité faibles. Pour cela, nous commençons par déterminer la matrice de précédence pour les runnables.

Pour simplifier l'écriture de la matrice de précédence pour les runnables, nous attribuons des numéros à chacun des runnables. La table 6.3 donne la correspondance entre les runnables définis dans le modèle et les numéros qui leurs sont attribués.

TssRunnable	R1	WlsRunnable	R2
TssPreprocessing	R3	WlsPreprocessing	R4
Logic	R5	Toggle	R6
FraRunnable	R7	FlaRunnable	R8

TABLE 6.3 – Correspondances runnables et numéros

À partir de la table 6.3, nous donnons la matrice de précédence pour les runnables, présentée par la table 6.4. Puis en utilisant ces informations ainsi que l'allocation des runnables sur les tâches, il est possible de déduire la matrice de précédence pour les tâches.

La matrice de précédence pour les tâches est donnée par la table 6.5. Nous avons ordonné les tâches par ordre de priorité : la tâche ayant la priorité la plus grande est la plus à gauche et la plus haute. Ainsi, la matrice obtenue possède bien les caractéristiques nécessaires pour que les contraintes de précédence soient respectées : elle est triangulaire inférieure, ce qui signifie que les seuls transferts de données se font des tâches de priorité la plus grande vers les tâches de priorité plus faible. Nous pouvons noter qu'il est tout

	R1	R2	R3	R4	R5	R6	R7	R8
R1	0	0	0	0	0	0	0	0
R2	0	0	0	0	0	0	0	0
R3	**1**	0	0	0	0	0	0	0
R4	0	**1**	0	0	0	0	0	0
R5	0	0	**1**	**1**	0	0	0	0
R6	0	0	0	0	**1**	0	0	0
R7	0	0	0	0	0	**1**	0	0
R8	0	0	0	0	0	**1**	0	0

TABLE 6.4 – Matrice de précédence pour les runnables

à fait possible d'avoir des "1" sur la diagonale : cela signifie que plusieurs runnables au sein d'une même tâche communiquent entre eux.

	Task_Wls	Task_WlsPreprocessing	Task_10ms	Task_100ms
Task_Wls	0	0	0	0
Task_WlsPreprocessing	**1**	0	0	0
Task_10ms	0	**1**	**1**	0
Task_100ms	0	0	**1**	0

TABLE 6.5 – Matrice de précédence pour les tâches

6.3.6.2 Analyse d'ordonnancement et de sensibilité

Maintenant que nous avons vérifié que les contraintes de précédence peuvent être satisfaites pour le système avec les caractéristiques des tâches, nous vérifions que le système peut être ordonnancé. Pour cela, nous considérons que les tâches évènementielles du systèmes peuvent être ramenées à des tâches périodiques. Nous considérons que le temps minimum qui peut s'écouler entre deux arrivées du signal les déclenchant est de 10 ms. Ainsi, les tâches "Task_Wls" et "Task_WlsPreprocessing" sont traitées comme des tâches périodiques avec une période de 10 ms.

Les clignotants sont habituellement placés dans le calculateur qui gère la BCM (Body Control Module). Ce calculateur est de type PowerPC, et pour une BCM complète, sa fréquence interne peut être fixée à 64 MHz. Cependant, nous n'avons pas besoin d'un calculateur aussi puissant pour notre application. Nous fixons donc la fréquence interne du calculateur à 200 kHz. Cette valeur nous permet de déterminer les valeurs des WCETs pour les runnables sur notre calculateur. Ces valeurs sont celles obtenues avec le logiciel Otawa [19] (voir section 6.2.3).

La charge maximum d'un système à 4 tâches tel que celui-ci, d'après la formule de Liu et Layland [64], est de 75,68%. Dans les faits, la charge du système est de 81,1 %. L'analyse d'ordonnancement de Bini et Buttazzo permet de déterminer que le système présenté reste cependant ordonnançable (puisque la condition d'ordonnancement de Liu

Tache	Runnables ($WCET_{Runnable}$)	$WCET_{tache}$
Task_Wls	WlsRunnable (2,45 ms)	2,45 ms
Task_WlsPreprocessing	WlsPreprocessing (0,15 ms)	0,15 ms
Task_10ms	TssRunnable (2,8 ms), TssPreprocessing (0,075 ms), Logic (1,075 ms), Toggle (1,425 ms)	5,375 ms
Task_100 ms	FlaRunnable (0,675 ms), FraRunnable (0,675 ms)	1,35 ms

TABLE 6.6 – WCET pour les runnables et pour les tâches sur la cible PowerPC

et Layland est suffisante mais non nécessaire). Notons que la tâche "Task_Wls" n'est pas pré-emptable par les autres tâches du système : cela n'a pas vraiment de conséquence sur l'analyse d'ordonnancement car dans notre système il s'agit de la tâche de niveau de priorité le plus haut, donc aucune tâche du système présenté ici ne peut préempter cette tâche.

L'analyse de sensibilité permet de déterminer les différents facteurs ΔC_k^{max} pour le système, ainsi que le facteur λ^{max}. Pour les tâches à 10 ms, étant donné qu'elles sont toutes les trois considérées à la même période, leur ΔC_k^{max} est identique. Nous avons donc $\Delta C_{Task\_Wls}^{max} = \Delta C_{Task\_WlsPreprocessing}^{max} = \Delta C_{Task\_10ms}^{max} = 1.89$. et pour la tâche a 100 ms $\Delta C_{Task\_100ms}^{max} = 18.9$. Enfin, le facteur $\lambda^{max} = 0,233$. Nous pouvons noter que ce dernier facteur est positif, ce qui confirme que le système est bien ordonnançable.

6.4 Étapes de préparation de l'application

Après avoir décrit l'application initiale, nous développons ici les modifications faites sur le processus de développement pour permettre les mises à jour ultérieures de l'application.

6.4.1 Ajout de méta-données

Dans le modèle fonctionnel logiciel que nous avons mis en place, l'outil que nous utilisons permet d'avoir un UUID correspondant à notre modèle. Ainsi nous avons un identifiant unique pour notre modèle initial. Lorsque nous passons au modèle AUTOSAR, il est nécessaire de conserver cet UUID afin de créer un lien entre le modèle fonctionnel et le modèle AUTOSAR.

Les étapes de configuration du Basic Software n'ont pas besoin de méta-données spécifiques car ces éléments ne sont pas modifiés lors du processus de mise à jour.

Il faut également ajouter les méta-données correspondant aux éléments présents dans l'application et au projet associé au compilateur et au linker. Cela doit permettre de retrouver tous les fichiers correspondants et de fixer les adresses mémoires des objets déjà présents dans l'application.

6.4.2 Ajout de mécanismes pour la gestion des mises à jour

Les modules permettant de gérer les mises à jour doivent également être ajoutés dans l'application. Ces modules sont des modules de bas niveau. La figure 6.9 présente une vue des différents modules impliqués pour gérer les mises à jour.

Les modules "Update Services", "Container Manager" et "Flash Manager" doivent être ajoutés dans l'application. Étant donné qu'il s'agit de modules du Basic Software non standardisés, ils sont placés dans un "Complex Device Driver" [14]. Le module 'FLS" appartient à la MCAL, par conséquent, il s'agit d'un module standard avec des interfaces connues et auxquelles il est nécessaire de se conformer. Le module "Transport Layer" appartient au Basic Software. Enfin le module "Communication Stack" correspond à la pile de communication. Le rôle de chacun des éléments permettant de gérer la mise à jour est présenté dans la section 3.4.3 du Chapitre 3.

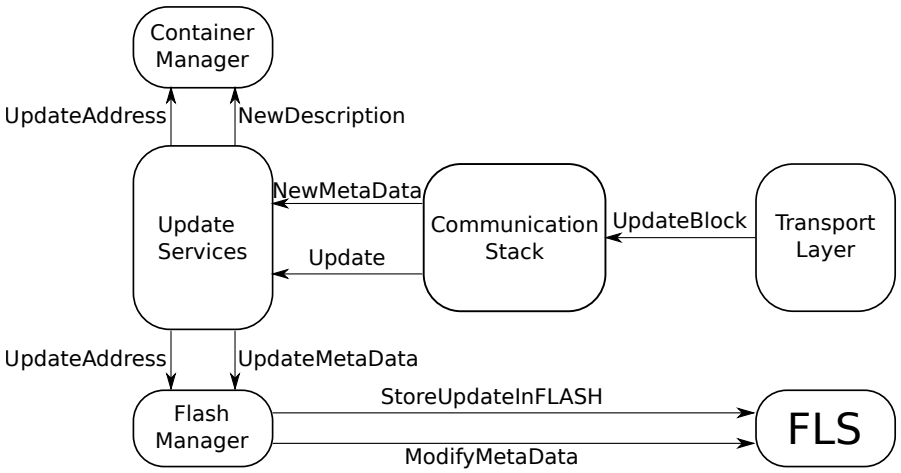


FIGURE 6.9 – Schéma du gestionnaire de mises à jour

Ces modules sont donc ajoutés dans le modèle AUTOSAR. En effet, le Basic Software n'apparait pas dans le modèle fonctionnel haut niveau (seule la partie applicative est modélisée). Le code correspondant fait ensuite partie des couches bas niveau et est compilé comme n'importe quel autre module.

6.4.3 Ajout d'un niveau d'indirection et de *containers*

Lorsque tous les modules de l'application sont prêts et avant l'étape de cross-compilation, il est nécessaire de faire des modifications sur le code pour intégrer un niveau d'indirection supplémentaire. Cela signifie que nous utilisons l'outil présenté dans la figure 5.10 du Chapitre 5. Cet outil va permettre d'utiliser des tableaux plutôt que des appels direct aux runnables au sein de chacune des tâches.

A cette étape sont également ajoutées des cases supplémentaires dans le tableau qui servent de *container* pour les futures mises à jour. La description de chacune des cases doit également être complétée.

Ces informations sont stockées dans le fichier `.arxml` qui accompagne les SWCs. Les figures 6.10 et 6.11 donnent des extraits du fichier arxml associé au SWC "IndicatorAtomic" (voir figure 6.8, page 115). Ainsi, ce fichier nous permet de connaitre les informations temporelles associées aux runnables. Par exemple la figure 6.10 montre l'évènement temporel qui doit permettre de déclencher l'exécution du runnable "Logic". Cela permet de savoir que la période d'exécution de ce dernier est de 10 ms. Les informations concernant certaines contraintes temporelles se retrouvent bien entre le modèle fonctionnel global et le modèle AUTOSAR. La figure 6.11, quant à elle, présente les caractéristiques d'un runnable dans un fichier `.arxml`.

Il est important de noter que chaque élément présenté dans le fichier `.arxml` possède un identifiant propre (UUID), et qu'il est possible de l'utiliser pour identifier de façon unique chacun des éléments présents dans l'application. Cependant, une version compressée de ces données doit être utilisée afin de ne pas utiliser une trop grande quantité de mémoire dans le calculateur.

```
<TIMING-EVENT UUID="3fd780be-f451-4e7b-ac89-6d843b27afde">
      <SHORT-NAME>LogicCyclic10ms</SHORT-NAME>
      <START-ON-EVENT-REF DEST="RUNNABLE-ENTITY">/IndicatorAtomic
          /IB_IndicatorAtomic/Logic</START-ON-EVENT-REF>
      <PERIOD>0.01</PERIOD>
</TIMING-EVENT>
```

FIGURE 6.10 – Extrait du fichier `.arxml` : évènement périodique déclencheur pour le runnable Logic"

Le symbole associé à chacun des runnables correspond à son nom. Cette donnée permet d'identifier le runnable car c'est cette désignation qui est utilisée lors de la génération du RTE, et, par conséquent, c'est également ce nom qui apparait dans le corps des tâches pour invoquer le runnable. A partir de l'ensemble des informations fournies par le fichier `.arxml`, il est donc possible de construire le tableau d'appel indirect des runnables dans les tâches ainsi que sa description (ID du runnable, caractéristiques temporelles, valeur initiale du pointeur).

L'outil que nous avons développé en langage python permet de récupérer les informations concernant les runnables et de modifier le corps des tâches en conséquence. Il est important de noter que les *containers* ne peuvent pas simplement rester vide dans la mesure où ils sont appelés au cours du flot d'exécution de la tâche. Ainsi, nous devons ajouter dans le programme une fonction vide, et qui permet de pointer sur une valeur non nulle dans la mémoire. Étant donné que la fonction est vide, le temps d'exécution de la fonction est nul.

La figure 6.12 présente le code de la tâche "Task_10ms" qui contient quatre runnables, à gauche dans sa version initiale, *i.e.* générée par les outils AUTOSAR, et à droite dans sa version modifiée pour permettre les mises à jour. Nous n'avons pas remis sur cette exemple les différentes initialisations nécessaires pour le bon fonctionnement du code, mais elles sont strictement identiques à celles de la figure 5.8 du Chapitre 5. Il

```
<RUNNABLE-ENTITY UUID="d8c28134-d3c8-45e2-a2aa-276ea583a8e1">
      <SHORT-NAME>Logic</SHORT-NAME>
      <CAN-BE-INVOKED-CONCURRENTLY>false</CAN-BE-INVOKED-
          CONCURRENTLY>
      <READ-VARIABLE-REFS>
        <READ-VARIABLE-REF DEST="INTER-RUNNABLE-VARIABLE">/
            IndicatorAtomic/IB_IndicatorAtomic/tss_status</READ-
            VARIABLE-REF>
        <READ-VARIABLE-REF DEST="INTER-RUNNABLE-VARIABLE">/
            IndicatorAtomic/IB_IndicatorAtomic/wls_status</READ-
            VARIABLE-REF>
      </READ-VARIABLE-REFS>
      <SYMBOL>Logic</SYMBOL>
      <WRITTEN-VARIABLE-REFS>
        <WRITTEN-VARIABLE-REF DEST="INTER-RUNNABLE-VARIABLE">/
            IndicatorAtomic/IB_IndicatorAtomic/indicator_status</
            WRITTEN-VARIABLE-REF>
      </WRITTEN-VARIABLE-REFS>
</RUNNABLE-ENTITY>
```

FIGURE 6.11 – Caractéristiques du runnable “Logic”

est important de noter que dans le cas des appels aux runnables via une table d'indirection, une initialisation complète de cette table est nécessaire. Pour cette raison, une tâche d'intialisation contenant les initialisations de chacune des cases du tableau doit être ajoutée. Cette tâche ne doit s'exécuter qu'une seule fois au démarrage de l'OS pour charger en mémoire toutes les valeurs des fonctions à exécuter. Toutefois, le code de la tâche présenté dans la figure 6.12 est uniquement donné à titre d'exemple. En effet, dans la pratique, il serait préférable de stocker dans un espace mémoire FLASH réservé toutes les adresses des runnables qui doivent être exécutés et de charger ces adresses mémoires au démarrage. Pour des raisons de simplicité et de lisibilité, nous donnons ici un exemple simple qui ne prend en compte que les valeurs initiales.

6.5 Mise à jour de l'application

Cette section décrit deux mises à jour possibles pour l'application, à la fois du point de vue structurel et architectural et du point de vue temps-réel. Nous décrivons également les différentes versions possibles pour l'application et comment gérer toutes les versions possibles.

6.5.1 Description des mises à jour

Nous avons décidé de nous concentrer sur deux mises à jour possibles (une pour chacune des fonctionnalités représentée dans notre application) :

— Une mise à jour concernant les warnings, qui permet d'allumer automatiquement les warnings lors d'un freinage d'urgence.

```
TASK(Task_10ms)
{
   /* start runnable */
   TssRunnable();

   /* start runnable */
   TssPreprocessing();

   /* start runnable */
   Logic();

   /* start runnable */
   Toggle();
   TerminateTask();
}
```

```
TASK(Task_Init)
{
 int j = 0;

 tab_Task_10ms[0].desc.ID_runnable =
      0xc6e31b0d;
 tab_Task_10ms[0].desc.empty = 0;
 tab_Task_10ms[0].ptr = TssRunnable;

 tab_Task_10ms[1].desc.ID_runnable =
      0xc7439e0f;
 tab_Task_10ms[1].desc.empty = 0;
 tab_Task_10ms[1].ptr =
     TssPreprocessing;

 tab_Task_10ms[2].desc.ID_runnable =
      0xd8c28134;
 tab_Task_10ms[2].desc.empty = 0;
 tab_Task_10ms[2].ptr = Logic;

 tab_Task_10ms[3].desc.ID_runnable =
      0x1fc71bd9;
 tab_Task_10ms[3].desc.empty = 0;
 tab_Task_10ms[3].ptr = Toggle;

 for (j=4;j<TABLE_SIZE; j++)
 {
  tab_Task_10ms[j].desc.ID_runnable
      = 0;
  tab_Task_10ms[j].desc.empty = 1;
  tab_Task_10ms[j].ptr = Empty;
 }
}

TASK(Task_10ms)
{
 int i=0;

 for (i=0;i<TABLE_SIZE; i++)
 {
  /*start runnable */
  tab_Task_10ms[i].ptr();
 }
}
```

FIGURE 6.12 – Code original de la tâche “Task_10ms” (à gauche) et code modifié par notre outil (à droite)

— Une mise à jour concernant les clignotants (qui a déjà été brièvement discutée dans le Chapitre 4), permettant d'avoir des clignotants impulsionnels, c'est-à-dire que lors d'un appui bref sur le comodo, l'indicateur de changement de direction correspondant va clignoter 3 fois, par exemple pour indiquer un changement de file.

De la même manière que pour la conceptions préliminaire, la conception des mises à jour se fait en deux étapes. Une première étape dans laquelle la fonctionnalité est conçue seule. Une seconde dans laquelle cette fonctionnalité est intégrée dans la chaine de calcul existante.

6.5.1.1 Freinage d'urgence

La première étape dans la conception de mises à jour est de déterminer les cas d'utilisation. Nous avons donc identifié ici le cas d'utilisation du freinage d'urgence. La figure 6.13 présente ici le diagramme de cas d'utilisation pour cette mise à jour.

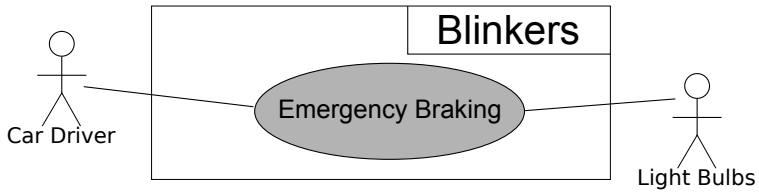


FIGURE 6.13 – Use case freinage d'urgence

Pour ce cas d'utilisation des clignotants, nous modélisons donc la chaine de calcul spécifique (figure 6.14) ainsi que le diagramme de séquence qui décrit le comportement associé (figure 6.15).

La chaine de calcul pour le freinage d'urgence, présenté sur la figure 6.14, est proche de celles pour les warnings. Cependant, l'information ne provient pas cette fois d'un capteur spécifique, mais d'informations fournies par d'autres runnables présents dans le système, à savoir le capteur de vitesse et l'accéléromètre. Bien que ces deux éléments n'aient pas été présentés précédemment dans la chaine de calcul, il s'agit d'éléments présents de façon standard dans un véhicule.

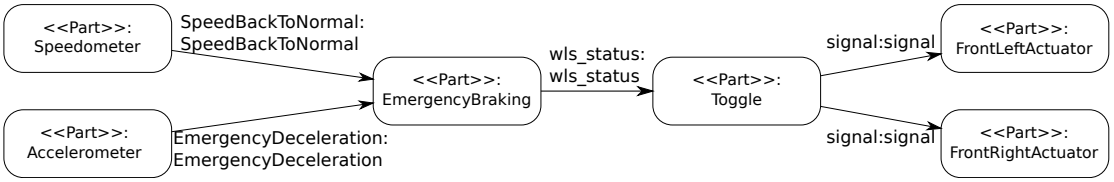


FIGURE 6.14 – Chaine de calcul préliminaire pour le freinage d'urgence

La figure 6.15 présente le comportement attendu pour le composant qui s'occupe du freinage d'urgence. Un élément "EmergencyBraking" doit permettre de vérifier si une décélération importante du véhicule a lieu, et dans ce cas, allumer automatiquement

les warnings. Lorsque le véhicule reprend une vitesse normale (*i.e.* qu'il recommence à accélérer), les warnings doivent s'éteindre automatiquement.

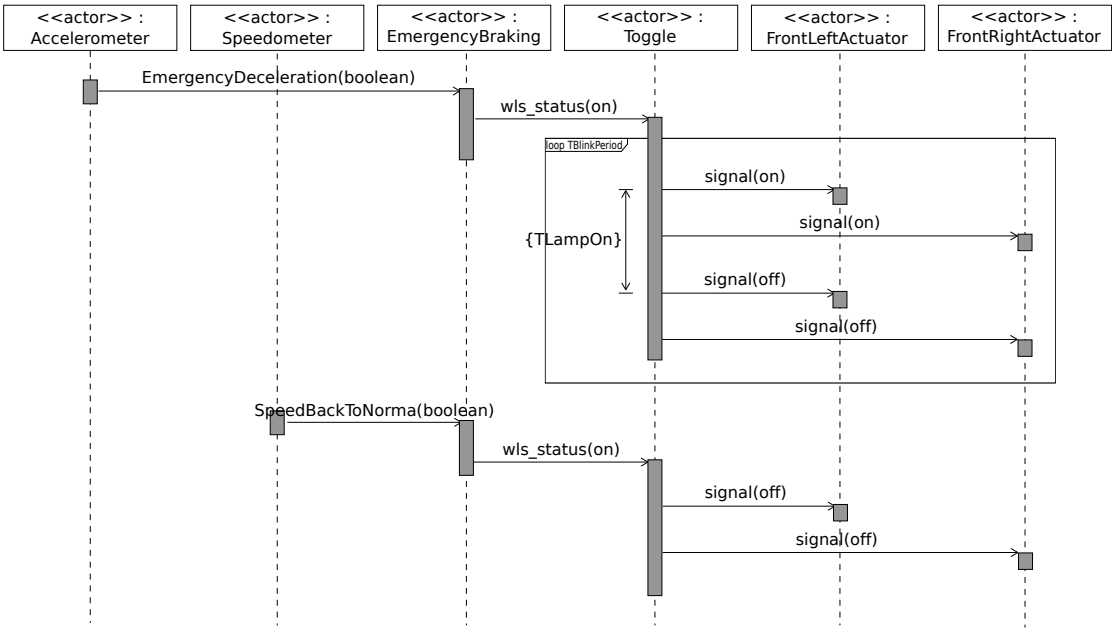


FIGURE 6.15 – Diagramme de séquence pour le freinage d'urgence

Concernant les contraintes temporelles liées au freinage d'urgence, il faut vérifier régulièrement si le freinage a été activé. Nous considérons donc que l'élément "EmergencyBraking" doit être exécuté toutes les 10 ms. L'ASIL pour cette chaine de calcul est le même que pour les warnings, à savoir QM.

6.5.1.2 Clignotants impulsionnels

De la même façon que pour le freinage d'urgence, nous commençons par décrire individuellement les caractéristiques de la mise à jour qui permet de faire fonctionner les clignotants de façon impulsionnelle en plus du comportement nominal pour cette fonctionnalité. L'objectif est donc de pouvoir détecter une impulsion sur le capteur plutôt que de simplement vérifier quelle est la valeur du capteur. Cette mise à jour a déjà été partiellement décrite du point de vue comportemental à la section 4.6 du Chapitre 4.

La figure 6.16 présente ici le diagramme de cas d'utilisation pour les clignotants impulsionnels. Il est très proche de celui pour le freinage d'urgence dans la mesure où les acteurs qui agissent sur le système ou bien qui reçoivent des instructions du système sont les mêmes.

Nous concevons ensuite individuellement la chaine de calcul correspondant aux clignotants impulsionnels. La figure 6.17 présente cette chaine de calcul. Ainsi, si nous la comparons à la chaine de calcul initiale pour gérer simplement les clignotants (voir figure 6.2), nous pouvons voir qu'un élément supplémentaire est nécessaire, à savoir "Tss-

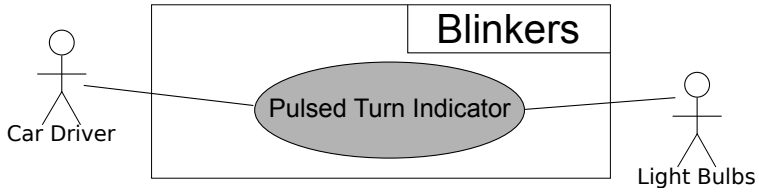


FIGURE 6.16 – Cas d'utilisation des clignotants impulsionnels

Preprocessing". Cet élément va être utilisé pour détecter l'impulsion sur le capteur, et s'assurer que les indicateurs clignotent bien 3 fois suite à cette impulsion (et seulement 3 fois). Si le comportement du reste de la chaine reste identique, c'est donc l'élément "TssPreprocessing" qui va se charger d'envoyer les signaux nécessaires.

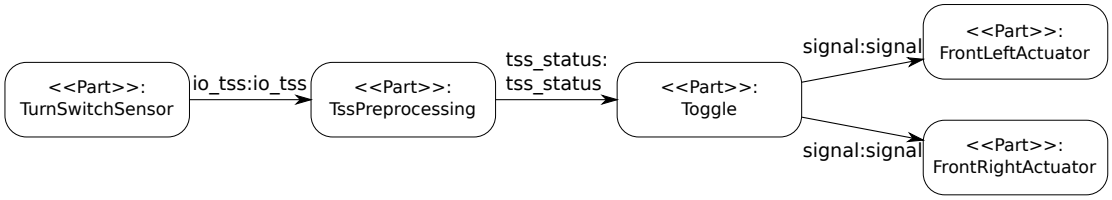


FIGURE 6.17 – Chaine de calcul préliminaire pour les clignotants impulsionnels

L'échange de signaux entre les différents éléments est matérialisé par le diagramme de séquence de la figure 6.18.

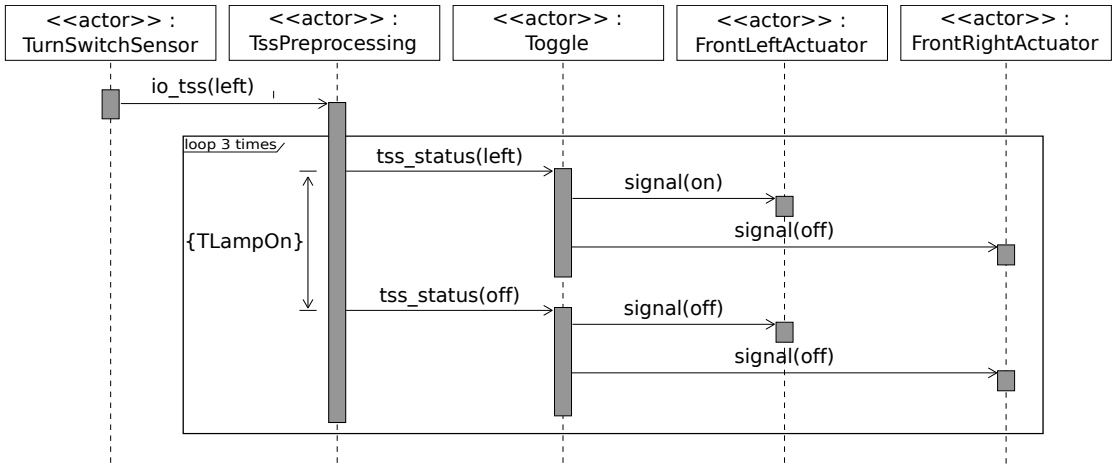


FIGURE 6.18 – Diagramme de séquence pour les clignotants impulsionnels

Les contraintes temporelles ainsi que les contraintes de safety concernant cette chaine de calcul sont strictement identiques à celles pour la chaine qui gère les clignotants.

6.5.1.3 Conception Avancée

Dans cette section nous décrivons comment les mises à jour peuvent être intégrées à la chaine de calcul finale de l'application initiale, représentée par la figure 6.7. La figure 6.19 présente les deux mises à jour insérées dans la chaine de calcul.

Freinage d'urgence Pour le freinage d'urgence, il est nécessaire de récupérer des données qui n'étaient pas présentes auparavant dans la chaine de calcul et d'actionner les warnings en conséquence. De nouveaux canaux de communication doivent donc être ajoutés ainsi qu'un nouvel élément pour gérer les informations issues des capteurs d'accélération et de vitesse.

Les modifications faites à la chaine de calcul pour le freinage d'urgence sont présentées au bas de la figure 6.19 :

- le runnable "EmergencyBraking" doit être ajouté,
- trois canaux de communication doivent être ajoutés,
- le runnable "Logic" doit être modifié pour prendre en compte la nouvelle donnée fournie par le runnable "EmergencyBraking".

Nous ne modifions pas le runnable "WlsPreprocessing" car il s'agit d'un runnable évènementiel qui s'active uniquement lorsqu'il reçoit une donnée du runnable WlsRunnable. Or, nous ne pouvons ni ajouter des runnables évènementiels, ni modifier l'évènement sur lequel le runnable est exécuté. Une autre solution serait d'ajouter un runnable intermédiaire qui centraliserait les données entre "WlsRunnable" et "EmergencyBraking" avant d'envoyer la donnée à "WlsPreprocessing".

Les éléments "Speedometer" et "Accelerometer" sont ici présentés en gris car il s'agit d'éléments déjà présents dans le logiciel du véhicule mais qui n'intervenaient pas précédemment dans cette chaine de calcul.

Clignotants Impulsionnels Pour les clignotants impulsionnels il s'agit uniquement de mettre en place la modification d'une fonctionnalité qui existait déjà dans l'application initiale.

Ainsi cette mise à jour correspond à une *upgrade*. Le seul élément qui est modifié dans ce cas est le runnable "TssPreprocessing".

En effet, son rôle dans le cas d'un appui court sur le capteur est d'envoyer les messages nécessaires pour que les indicateurs de changement de direction clignotent 3 fois.

6.5.2 Modèle AUTOSAR pour les mises à jour

Tous les composants de l'application qui gère les clignotants ainsi que les mises à jour décrites précédemment sont localisés dans un même ECU pour notre étude de cas. Nous ne présentons donc pas la vue VFB pour le modèle AUTOSAR.

Le capteur de vitesse et l'accéléromètre ces éléments peuvent soit être placés au sein du même calculateur que celui qui accueille notre application, soit être placé sur un

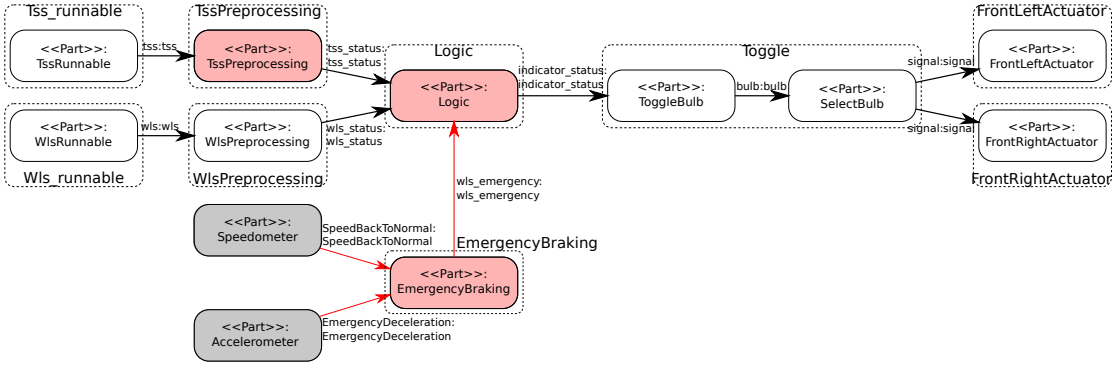


FIGURE 6.19 – Chaine de calcul incluant les deux mises à jour (en rouge)

calculateur différent. Dans le second cas, il est nécessaire qu'un message CAN existe déjà entre le calculateur qui exécute l'application et celui sur lequel les composants de capteur de vitesse et l'accéléromètre. Dans ce cas, il est nécessaire d'ajouter un canal dans le RTE pour récupérer les informations issues de la pile de communication. Aucune modification de la messagerie ou de la pile de communication n'est alors nécessaire.

La figure 6.20 présente ici le modèle AUTOSAR qui correspond aux deux mises à jour possibles. Étant donné que les clignotants impulsionnels correspondent uniquement à un "upgrade" il n'y a pas de modification apparente à ce niveau. Cependant, un nouveau composant apparait pour gérer le freinage d'urgence. Nous avons également fait apparaitre ici le composant "Speedometer" qui gère tout ce qui est lié à la vitesse. Les caractéristiques de ce composant ne sont pas complètes car nous considérons qu'il s'agit d'un composant déjà existant ailleurs dans l'architecture. Dans le cadre de cette preuve de concept, nous n'avons pas développé le modèle complet de la gestion de la vitesse.

Des modifications sont également nécessaires sur le composant "IndicatorAtomic" : un nouveau port doit être ajouté pour gérer la communication avec "EmergencyBraking".

6.5.3 Temps-réel

Du point de vue temps-réel, pour chacune des mises à jour possible, il faut mesurer le WCET des runnables ajoutés et/ou modifiés et, en fonction de la tâche dans laquelle ces mises à jour sont placées, vérifier que le système est toujours ordonnançable. Dans le cas de notre preuve de concept, étant donné que le système est très peu chargé initialement, il sera nécessairement toujours ordonnançable après les mises à jour que nous proposons.

6.5.3.1 WCET

Les valeurs des WCETs pour les mises à jour sont données à partir des valeurs des runnables initialement présents dans le système.

Le runnable "EmergencyBraking" est un runnable qui doit vérifier si un freinage d'urgence a lieu ou non. Le traitement fait sur les variables d'entrée est rapide. Son

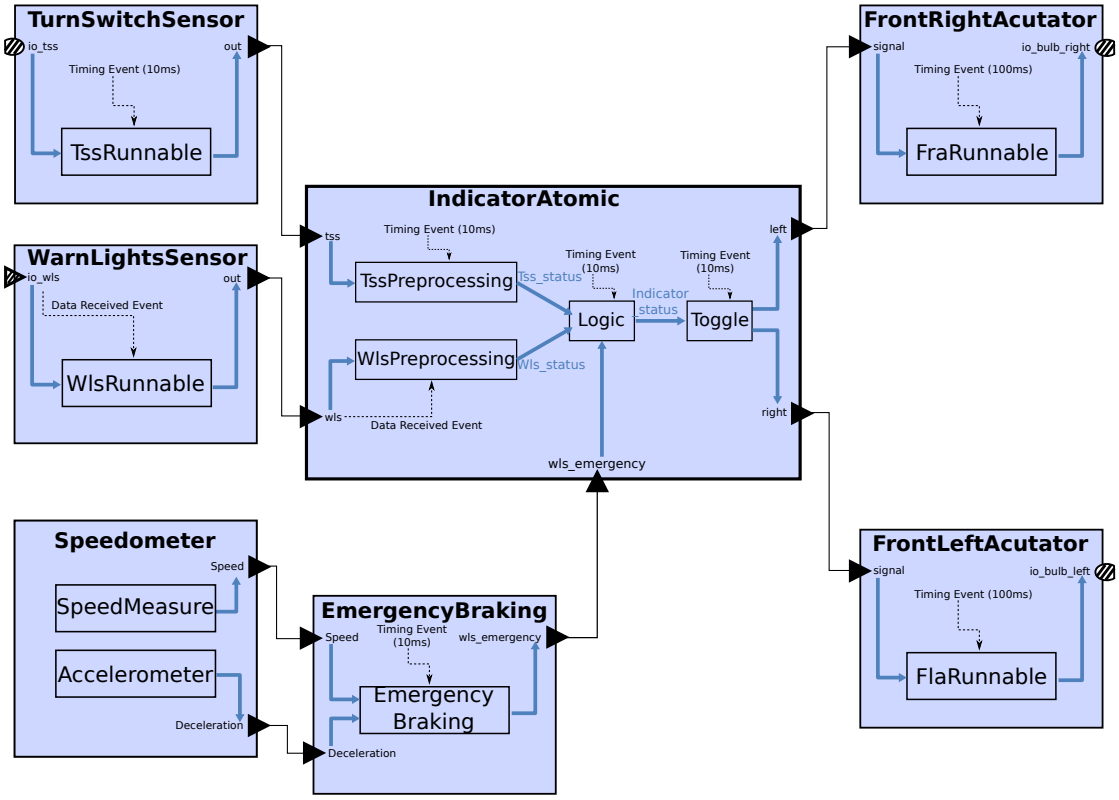


FIGURE 6.20 – Modèle AUTOSAR pour les mises à jour

WCET est proche de celui du runnable “Logic”. Nous estimons que son WCET est d’environ 150 cycles. Cela représente un WCET de 0,75 ms sur notre cible matérielle. Ce runnable est périodique à 10 ms. Il doit donc être placé dans la tâche “Task_10ms”. Le runnable “Logic” est également modifié, et son WCET augmente légèrement pour prendre en compte la nouvelle donnée d’entrée. Le WCET du runnable modifié est donc de 245 cycles, ce qui représente 1,225 ms.

Concernant le runnable “TssPreprocessing” qui est modifié pour prendre en compte le clignotant impulsionnel, l’algorithme est plus complexe que celui du runnable initial. En effet, dans la version initiale, ce runnable copie simplement une donnée. Pour la version mise à jour il doit détecter l’impulsion et envoyer 3 fois le signal pour faire clignoter les actionneurs. Son WCET est donc très nettement augmenté. Il est proche de celui de FraRunnable. Nous estimons que le WCET de la version mise à jour de “TssPreprocessing” est d’environ 100 cycles, ce qui correspond à 0,5 ms à 200 kHz. La nouvelle version reste placé dans la même tâche, à savoir Task_10ms.

6.5.3.2 Contraintes de précédence

Pour le freinage d'urgence, nous ne pouvons pas vérifier toutes les contraintes de précédences car nous n'avons pas l'implémentation complète des éléments liés à la vitesse et à l'accélération. Cependant, "EmergencyBraking" et "Logic" sont placés dans la même tâche. Il faut donc modifier l'ordre d'exécution pour qu'il permette de respecter les précédences.

En ce qui concerne les clignotants impulsionnels, aucun changement n'est nécessaire sur les contraintes de précédences car le runnable qui est modifié garde la même interface.

6.5.3.3 Analyse de sensibilité

Trois cas de figure doivent être traités :
— le cas où seul le freinage d'urgence est ajouté,
— le cas où seuls les clignotants impulsionnels sont ajoutés,
— le cas où les deux mises à jour sont ajoutées ensemble.

Freinage d'urgence Le WCET de la tâche "Task_10ms" croît puisque le WCET du runnable "Logic" augmente et qu'un runnable supplémentaire est ajouté. Ainsi le nouveau WCET pour la tâche, lorsque cette mise à jour est ajoutée, est de 6,275 ms. Il est alors possible de vérifier que le système est toujours ordonnançable dans ces conditions, en calculant le facteur λ^{max} : nous obtenons $\lambda_{max} = 0,1099$. Le système étant toujours ordonnançable, nous vérifions également les nouvelles valeurs pour les ΔC_k^{max} afin de connaitre la flexibilité de chacune des tâches. Nous obtenons $\Delta C^{max}_{Task\_Wls} = \Delta C^{max}_{Task\_WlsPreprocessing} = \Delta C^{max}_{Task\_10ms} = 0.99$. et pour la tâche a 100 ms $\Delta C^{max}_{Task\_100ms} = 9.99$. Nous remarquons alors que la nouvelle valeur pour $\Delta C^{max}_{Task\_10ms}$ est égale à la valeur précédente moins l'augmentation du WCET. Si nous notons $\Delta C1$ la valeur initiale et $\Delta C2$ la valeur après mises à jour, nous avons : $\Delta C2 = \Delta C1$ - (WCET(EmergencyBraking) + (New WCET(Logic) - Old WCET(Logic))).

Clignotants impulsionnels L'impact sur l'ordonnancement du système dans le cas de l'ajout de la fonctionnalité de clignotants impulsionnels est assez minime. En effet, seul le WCET d'un runnable dans une tâche existante est modifié.

Le WCET de la tâche Task_10ms passe alors à 5,8 ms. Cette mise à jour a un impact comparable à celle pour le freinage d'urgence. Nous obtenons pour cette mise à jour $\lambda_{max} = 0.1716$ (donc le système est toujours ordonnançable). Nous vérifions également les nouvelles valeurs pour les ΔC_k^{max} afin de connaitre la flexibilité de chacune des tâches. Nous obtenons $\Delta C^{max}_{Task\_Wls} = \Delta C^{max}_{Task\_WlsPreprocessing} = \Delta C^{max}_{Task\_10ms} = 1.4650$. et pour la tâche a 100 ms $\Delta C^{max}_{Task\_100ms} = 14.650$.

Combinaison des deux mises à jour Dans le cas où les deux mises à jour sont ajoutées dans le système, seul le WCET de la tâche Task_10ms est modifié : sa valeur est alors de 6.7 ms. Lorsque nous combinons les deux mises à jour, le système est toujours

ordonnançable puisque nous avons $\lambda_{max} = 0.0599$. Cependant, sa valeur étant très faible, nous déconseillons de faire des mises à jour supplémentaires.

Nous obtenons également $\Delta C^{max}_{Task\_Wls} = \Delta C^{max}_{Task\_WlsPreprocessing} = \Delta C^{max}_{Task\_10ms} = 0,565$. et $\Delta C^{max}_{Task\_100ms} = 5,65$.

6.5.4 Gestion de version

Dans notre exemple, nous présentons deux mises à jour possible. La combinatoire est donc réduite, et la taille de l'arbre de mise à jour est, par conséquent, simple à gérer.

L'arbre de mise à jour pour les deux mises à jour possibles est proche de ce qui a été présenté dans le Chapitre 5. La figure 6.21 présente l'arbre de mises à jour pour l'application qui gère les clignotants. Les notations que nous avons utilisées pour les runnables sont celles de la table 6.3. Les versions correspondantes aux mises à jour sont également indiquées sur cette figure. Il existe donc quatre versions possibles pour l'application selon l'ordre dans lequel les mises à jour sont faites.

Il est important de noter que l'arbre des mises à jour dépend des caractéristiques temps-réel des mises à jour. En effet, il est nécessaire de vérifier au préalable l'ordonnançabilité du système lorsque chacune des mises à jour est ajoutée et pour chaque combinaison de mise à jour. Dans le cas où le système n'est plus ordonnançable après une mise à jour, alors cet état ne doit pas apparaitre dans l'arbre de mise à jour.

Nous considérons que la version initiale de l'application est la version 1.0. En fonction de la première mise à jour qui est faite, nous passons ensuite, soit en version 1.1 (application initiale avec la mise à jour de freinage d'urgence), soit en version 1.2 (application initiale avec la mise à jour de clignotant impulsionnel). Lorsqu'une mise à jour a été faite, une branche spécifique de l'arbre est choisie et il n'est plus possible de changer de branche, à moins de revenir d'abord à la version initiale. Nous ne considérons pas ici la possibilité de revenir en arrière après qu'une mise à jour est intégrée à la voiture : cela signifie qu'il faudrait effacer une partie de la FLASH, procédé qui n'est pas simple à mettre en place. Cependant, il est possible de ne pas exécuter une mise à jour en cas de problème. Cela ne change pas pour autant la version du code chargé dans le calculateur et les emplacements mémoires correspondant à chaque fonctionnalité.

6.6 Synthèse

Dans ce Chapitre, nous avons montré comment appliquer l'ensemble des concepts définis précédemment pour permettre de faire des mises à jour dans une application embarquée automobile concrète. La préparation de l'application est en conformité avec le processus de développement AUTOSAR standard et les modifications sont minimales (voir Chapitre 3).

Toutes les phases du processus pour l'ajout des mises à jour sont présentées dans ce Chapitre, depuis la préparation de l'application initiale, jusqu'au déploiement effectif de la mise à jour et à la vérification des propriétés temporelles.

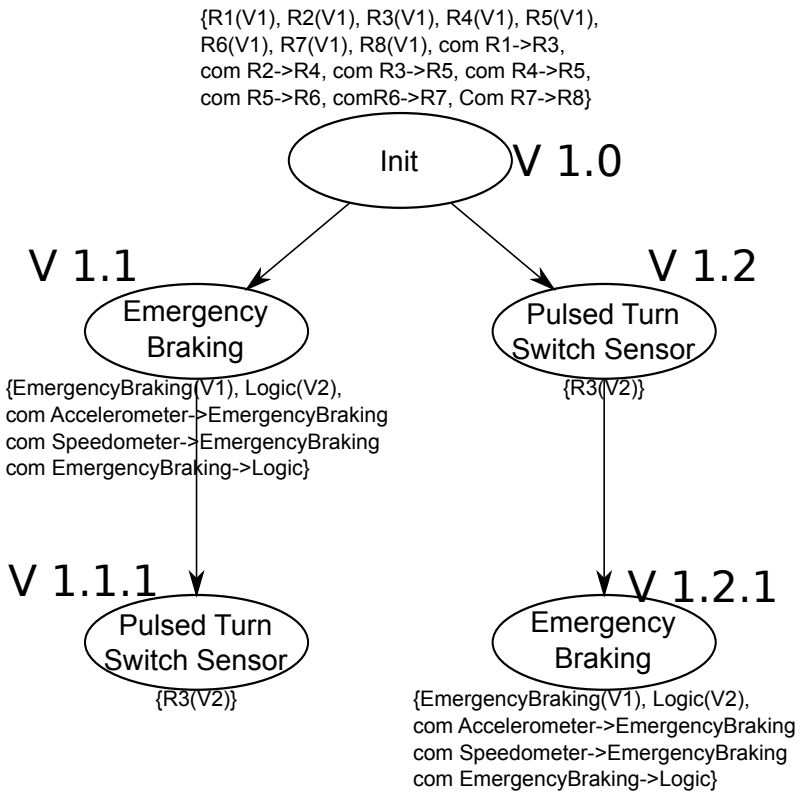


FIGURE 6.21 – Arbre de mises à jour pour les clignotants

Il est donc possible d'ajouter des fonctionnalités concrètes et utiles pour l'utilisateur de la voiture sans surcoût important. Cependant, il est préférable de prévoir suffisamment de marge dans le système si des mises à jour ultérieures sont prévues.

Du point de vue temporel, les systèmes embarqués automobiles ont assez peu d'espace libre disponible. Cependant, même lorsque le taux d'utilisation du système est supérieur à la limite théorique fixée par Liu et Layland[64], nous avons montré qu'il est tout de même possible de faire des mises à jour. De plus, la méthode que nous proposons pour les mises à jour permet de ne pas avoir à refaire une analyse d'ordonnancement intégrale, mais d'utiliser uniquement une méthode différentielle grâce à l'analyse de sensibilité.

Nous avons donc montré ici qu'il est possible de faire des mises à jour efficaces, sans avoir à reflasher l'intégralité du calculateur pour des modifications minimales et tout en assurant le bon fonctionnement du système du point de vue temps-réel.

7

Conclusion

La mise à jour partielle dans les systèmes embarqués automobiles correspond à une étape importante afin, d'une part d'améliorer l'assistance aux conducteurs et le confort des passagers, et d'autre part de faciliter les modifications logicielles pour le constructeur automobile. Les mises à jour traitées ici se concentrent sur le logiciel applicatif des calculateurs enfouis, qui peut être considéré comme critique. Le problème est donc double : d'une part, être en mesure de faire les mises à jour en ajoutant tous les mécanismes nécessaires, et d'autre part, s'assurer que ces mises à jour n'impactent pas négativement le fonctionnement du système.

Ainsi, il s'agit d'un problème complexe qui influence de nombreux aspects du développement logiciel. Le contexte dans lequel cette mise à jour doit se faire est celui du standard AUTOSAR. Cela impose donc un certain nombre de contraintes, étant donné la structure de l'architecture et la méthodologie qui doivent être respectées.

Pour améliorer la flexibilité des systèmes de type AUTOSAR, nous avons donc proposé une approche qui repose sur l'implantation de *containers* dans l'architecture afin d'aménager des espaces pour les futures mises à jour. Nous proposons également une méthode qui permet de vérifier que dans ce contexte, les propriétés temps-réel (qui sont critiques dans le cas d'un système embarqué automobile) ne sont pas violées.

Les travaux présentés ici se fondent sur les hypothèses suivantes :

1. Le contexte dans lequel les mises à jour ont lieu est celui du standard AUTOSAR et les solutions proposées sont adaptées spécifiquement à ce cadre.
2. Aucune modification n'est faite sur la messagerie CAN car cela demanderait de modifier la façon de gérer la configuration de cette messagerie.
3. Il n'y a pas de matériel spécifique rajouté : les seules mises à jour possibles sont logicielles.
4. Les calculateurs dans lesquels les mises à jour sont intégrées sont de type mono-cœur.

Le problème est complexe et donc nous avons posé ces hypothèses pour restreindre le périmètre du travail réalisé dans la thèse.

Dans le périmètre défini par les hypothèses de travail, nous avons montré qu'il est possible de faire des mises à jour partielles sûres de fonctionnement sur des systèmes embarqués critiques comme ceux utilisant une architecture de type AUTOSAR. De plus, ces mises à jour sont possibles sans perturber le processus de développement propre au standard et avec des modifications minimales.

Ces mises à jour passent par la préparation de l'application et la vérification de propriétés temps-réel. Cela permet d'envisager de nouvelles perspectives quant à l'utilisation de systèmes embarqués automobiles. En effet, mettre en œuvre à grande échelle les méthodes que nous avons développées dans cette thèse signifierait créer une bibliothèque d'applications pour les calculateurs automobiles enfouis dans laquelle chaque client peut déterminer quelles fonctionnalités il souhaite ajouter à son véhicule.

Dans cette thèse, ont été étudiés tous les aspects liés à la mise à jour partielle pour le calculateur embarqué. Ce travail a tout d'abord permis de définir plus précisément le processus de développement lié au standard AUTOSAR. Nous avons ajouté une étape de développement qui consiste à mettre en place un modèle fonctionnel logiciel global qui contient toutes les caractéristiques de l'application. Nous avons également donné un certain nombre de critères pour allouer les runnables dans les SWCs, placer les différents composants sur les ECUs disponibles et allouer les runnables de chacun des composants au sein d'un ECU dans les tâches de l'OS. Les concepts d'*espace d'adaptation* et de *container* ont également été définis. Ces concepts ont pour but d'ajouter de la flexibilité au sein de l'architecture AUTOSAR. Les modifications nécessaires concernant le processus de développement, pour inclure les concepts que nous avons définis, sont également décrites.

Les aspects liés à l'ordonnancement temps-réel pour la mise à jour sont également développés ici. En effet, il est nécessaire de vérifier que l'ajout de mises à jour ne perturbe pas le bon fonctionnement du système. Pour cela, nous avons défini un modèle de tâche lié à AUTOSAR en prenant en compte les spécificités liées au standard. Les contraintes de précédences, liées aux flots de données, sont également prises en compte en utilisant un graphe de précédence et une matrice d'adjacence. Enfin, en utilisant des résultats issus de la théorie de l'ordonnancement temps-réel, nous avons montré que la flexibilité opérationnelle des systèmes pouvait être quantifiée. En effet, nous utilisons une analyse de sensibilité pour déterminer l'espace restant disponible dans le système et effectuer une analyse d'ordonnancement simplifiée pour vérifier que la mise à jour peut être ajoutée dans le système.

Ainsi, nous combinons plusieurs points de vue sur l'application pour allouer de la place aux mises à jour et vérifier que les contraintes temps-réel sont respectées. Ces deux aspects de la mise à jour sont donc complémentaires et cette perspective dans les systèmes embarqués automobiles est innovante.

Des aspects d'implémentation sont également importants et ont été étudiés dans ce travail. Tout d'abord la gestion de version avec la notion d'arbre de mise à jour qui permet de déterminer, à partir d'une version donnée d'un système, quelles sont toutes les mises à jour possibles. Nous traitons également la question de la création des mises à jour par rapport à un contexte d'exécution donné, *i.e.* à partir d'une version donnée

de l'application et de tous les modèles correspondants, comment créer une mise à jour. La gestion de la mémoire pour le système embarqué est aussi une question importante. En effet, les systèmes embarqués ont des ressources limitées, et la mémoire fait partie de ces ressources. De plus, écrire dans de la mémoire FLASH n'est pas trivial et le nombre de cycles de lecture/écriture est limité. Ces questions sont également abordées dans ce travail. Enfin, nous traitons les aspects liés à la sûreté de fonctionnement pour les mises à jour, et en particulier les mécanismes qui peuvent être mis en place à chacune des étapes pour assurer que le système est toujours sûr de fonctionnement.

Pour finir, nous appliquons tous les concepts précédemment développés dans une preuve de faisabilité utilisant une application embarquée gérant les clignotants et les warnings, développée suivant le standard AUTOSAR. Nous définissons deux mises à jour possibles pour cette application (un *upgrade* et un *update*), et décrivons du point de vue architectural et temps-réel, non seulement l'application initiale, mais également chacune des mises à jour. L'arbre de mise à jour est également présenté ici, ainsi que les mécanismes bas niveaux qui nous permettent de prendre en charge les mises à jour. L'analyse de sensibilité pour l'application ainsi que les mises à jour est également présentée ici pour montrer que du point de vue temps-réel, ces mises à jour ne perturbent pas le bon fonctionnement du système.

Ainsi, il est possible de faire des mises à jour partielles, sans ajouter de nouveau matériel, sans avoir à reflasher l'intégralité de la mémoire. De nombreuses fonctions peuvent être ajoutés en utilisant ce processus, ce qui permet de gagner en temps, en bande passante, en personnalisation pour le client et en flexibilité globale.

Cette étude nous a amené à deux types de réflexions : d'une part concernant l'architecture AUTOSAR et les technologies logicielles utilisées et d'autre part en identifiant les limites de notre travail et les perspectives d'extension.

Tout d'abord, AUTOSAR est un système qui a par essence peu de flexibilité. Ainsi, nous avons ajouté dans l'architecture des éléments qui permettent d'ajouter de la flexibilité, mais cela n'est pas suffisant lorsque les mises à jour de taille importante doivent être faites. Dans ce cas, le rechargement complet du calculateur apporte une plus grande fiabilité qu'un grand nombre de petites mises à jour successives.

D'autre part, un grand nombre de travaux existent concernant les modèles à composants et l'agilité de ces systèmes. Une solution qu'il pourrait être intéressant d'explorer serait de mettre en place une vraie architecture à composants reposant sur un middleware qui soit capable d'assimiler par nature de nouveaux éléments. Toutefois, cela reviendrait à sortir du cadre classique d'AUTOSAR, ce qui, du point de vue industriel est difficilement envisageable. Une vraie alternative consisterait à concevoir un enrichissement du modèle de programmation AUTOSAR pour introduire de nouveaux modules permettant de supporter nativement les mises à jour. Une refonte du modèle AUTOSAR permettrait de relâcher des hypothèses sur lesquelles s'appuie la méthodologie proposée dans cette thèse, pour prendre en compte les potentielles modifications des messageries sur les bus de communication.

Deuxièmement, seul le bus CAN a été proposé au cours de ce travail, dans la mesure où c'est celui qui reste aujourd'hui le plus utilisé dans le contexte de l'automobile et chez

Renault. Il serait intéressant d'étudier plus en détail les autres bus de communication (FlexRay ou Ethernet par exemple) pour déterminer les possibilités de modification dans la configuration des messages envoyés. En outre, ces bus ont des débits plus importants que celui du CAN, ce qui pourrait permettre à terme à des mises à jour partielles de tailles plus importantes d'être véhiculées rapidement. Concernant les communications, le Chapitre 5 présente comment il est possible d'ajouter de nouveaux canaux RTE. Cependant, cette méthode n'est pas non plus optimale, mais est nécessaire à cause de la nature même du RTE. Une fois encore, avoir un modèle d'architecture plus flexible par essence, avec un middleware capable de prendre en charge les nouvelles communications nécessaires pour les mises à jour serait beaucoup plus optimal.

A l'heure actuelle, le cas d'ajout de matériel spécifique en plus des mises à jour logicielles n'est pas traité. En effet, les mises à jour proposées dans le cadre de ce travail sont uniquement des mises à jour applicatives. Or, ajouter du matériel nécessiterait d'ajouter également dans le ou les ECUs concernés des drivers bas niveaux pour récupérer les informations de ces capteurs ou actionneurs. Toutefois, à long terme, il sera nécessaire de pouvoir ajouter des capteurs (et/ou actionneurs) spécifiques, non initialement présent, puis mettre à jour les calculateurs pour ajouter des options logicielles qui utilisent ces capteurs (et/ou actionneurs). Un exemple de fonctionnalité qui pourrait bénéficier de ce type de mise à jour est le mécanisme de radars de recul. Il faut alors ajouter dans la voiture les capteurs de recul, et mettre à jour le logiciel pour qu'il puisse afficher les informations de proximité lorsque le véhicule recule. L'architecture AUTOSAR se prête mal à l'intégration de nouveaux drivers *a posteriori*.

Enfin, les aspects temps-réel doivent être étudiés de près lorsque des mises à jour sont faites dans un système embarqué critique comme ceux des automobiles. Ainsi, quel que soit le modèle d'architecture utilisé, une analyse d'ordonnancement doit avoir lieu. Toutefois, de nos jours, de plus en plus de systèmes utilisent des processeurs multicœurs, pour lesquels les analyses d'ordonnancement sont plus complexes que dans le cas monocœur que nous avons étudié ici. Dans l'optique de faire des mises à jour dans les systèmes automobiles, il est donc nécessaire de mettre en place une méthode d'analyse de sensibilité dédiée aux systèmes multicœurs qui puisse prendre en compte les problématiques spécifiques (comme le partage de données entre deux cœurs ou la synchronisation de l'exécution).

Les questions de sécurité, au sens de la confidentialité ne sont pas traitées de manière approfondie dans le cadre de ce travail. Cependant, les voitures à l'heure actuelle possèdent de plus en plus de canaux de communications ouverts vers le monde extérieur. Ce sont ces canaux qui sont susceptibles d'être utilisés lors des mises à jour. Ainsi, il est important de travailler sur ces aspects de sécurité des logiciels embarqués afin de s'assurer qu'aucun logiciel malveillant ne peut être installé. Un grand nombre de travaux ont été faits concernant la détection d'intrusion, en particulier avec Renault et le LAAS-CNRS [98].

Bibliographie

[1] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2006.

[2] J. Alat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, and D. Powell. *Encyclopédie de l'informatique et des systèmes d'information*, chapter Tolérance aux fautes, pages 240–270. J. Akoka and I. Comyn-Wattiau, 2006.

[3] F. André, E. Daubert, G. Nain, B. Morin, and O. Barais. F4Plan : An Approach to build Efficient Adaptation Plans. In *7th International ICST Conference on Mobile and Ubiquitous Systems (MobiQuitous)*, Sydney, Australie, Decembre 2010.

[4] C. Angelov, K. Sierszecki, and N. Marian. Component-based design of embedded software : An analysis of design issues. In N. Guelfi, G. Reggio, and A. Romanovsky, editors, *Scientific Engineering of Distributed Java Applications*, volume 3409 of *Lecture Notes in Computer Science*, pages 1–11. Springer Berlin / Heidelberg, 2005.

[5] J. Appavoo, K. Hui, C. A. N. Soules, R. Wisniewski, D. Da Silva, O. Krieger, M. Auslander, D. Edelsohn, B. Gamsa, G. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1) :60–76, 2003.

[6] N. Arshad, D. Heimbigner, and A. L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Control*, 15(3) :265–281, Septembre 2007.

[7] AUTOSAR. AUTOSAR virtual functional bus, 2010. `http://www.autosar.org/download/R4.0/AUTOSAR_EXP_VFB.pdf`.

[8] AUTOSAR. Technical safety concept status report (release 4.0), 2010. `http://www.autosar.org/download/R4.0/AUTOSAR_TR_SafetyConceptStatusReport.pdf`.

[9] AUTOSAR. General requirements on basic software modules (release 4.0). 2011. `http://www.autosar.org/download/R4.0/AUTOSAR_SRS_BSWGeneral.pdf`.

[10] AUTOSAR. Software component template (release 4.0). 2011. `http://www.autosar.org/download/R4.0/AUTOSAR_TPS_SoftwareComponentTemplate.pdf`.

[11] AUTOSAR. Specification of operating system (release 4.0), 2011. `http://www.autosar.org/download/R4.0/AUTOSAR_SWS_OS.pdf`.

[12] AUTOSAR. Specification of RTE. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_RTE.pdfs, 2011.

[13] AUTOSAR. Virtual functional bus (release 4.0). 2011. http://www.autosar.org/download/R4.0/AUTOSAR_EXP_VFB.pdf.

[14] AUTOSAR. Complex driver design and integration guideline, january 2013.

[15] AUTOSAR. Methodology, March 2014.

[16] AUTOSAR Development Cooperation. htt://www.autosar.org.

[17] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1) :11–33, 2004.

[18] J. Axelsson and A. Kobetski. On the conceptual design of a dynamic component model for reconfigurable autosar systems. In *5th Workshop on Adaptive and Reconfigurable Embedded Systems*, April 2013.

[19] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Otawa : an open toolbox for adaptive wcet analysis. In *IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, October 2010.

[20] F. Banno, D. Marletta, G. Pappalardo, and E. Tramontana. Handling consistent dynamic updates on distributed systems. In *Computers and Communications (ISCC), 2010 IEEE Symposium on*, pages 471–476, June 2010.

[21] S. K. Baruah, R. R. Howell, and L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2 :301–324, 1990.

[22] P. Baufreton, J. Derrien, B. Ricque, J. Blanquart, J. Boulanger, H. Delseny, J. Gassino, G. Ladier, E. Ledinot, M. Leeman, and P. Quere. Multi-domain comparison of safety standards. In *ERTS 2010*, ERTS '10, 2010.

[23] J.-L. Béchennec, M. Briday, S. Faucou, and Y. Trinquet. Trampoline - an open-source implementation of the OSEK/VDX RTOS specification. In *IEEE EFTA*, 2006.

[24] B. Becker, H. Giese, S. Neumann, M. Schenck, and A. Treffer. Model-based extension of AUTOSAR for architectural online reconfiguration. In S. Ghosh, editor, *Models in Software Engineering*, volume 6002 of *Lecture Notes in Computer Science*, pages 83–97. Springer Berlin Heidelberg, 2010.

[25] A. Bertout, J. Forget, and R. Olejnik. Research report- automated runnable to task mapping, 2013.

[26] E. Bini and G. Buttazzo. Schedulability analysis of periodic fixed priority systems. *Computers, IEEE Transactions on*, 53(11) :1462–1473, Novembre 2004.

[27] E. Bini, M. Di Natale, and G. Buttazzo. Sensitivity analysis for fixed-priority real-time systems. In *Real-Time Systems, 2006. 18th Euromicro Conference on*, pages 10 pp.–22, 2006.

[28] T. Bloom and M. Day. Reconfiguration and module replacement in argus : theory and practice. *Software Engineering Journal*, 8(2) :102–108, March 1993.

[29] U. Brockmeyer. Challenges and solutions for developing safety critical automotive software in compliance with ISO 26262. http ://www.slideshare.net/ibmrational/innovate-2012-conference-guide, 2014. http ://fr.scribd.com/doc/226379382/ibmautomtive.

[30] M. Broy. Automotive software and systems engineering. In *Formal Methods and Models for Co-Design, 2005. MEMOCODE '05. Proceedings. Third ACM and IEEE International Conference on*, pages 143–149, July 2005.

[31] M. Broy. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 33–42, New York, NY, USA, 2006. ACM.

[32] S. Burmester, H. Giese, M. Hirsch, D. Schilling, and M. Tichy. The Fujaba Real-time Tool Suite : Model-driven Development of Safety-critical, Real-time Systems. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 670–671, New York, NY, USA, 2005. ACM.

[33] Car 2 Car Communication Consortium. `http://www.car-to-car.org/`.

[34] H. Cervantes and R. S. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 614–623, Washington, DC, USA, 2004. IEEE Computer Society.

[35] J. Chen, L. Huang, S. Du, and W. Zhou. A formal model for supporting frameworks of dynamic service update based on OSGi. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 234–241, Nov 2010.

[36] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3) :181–194, 1990.

[37] T. I. S. Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, May 1995.

[38] S. Cotard. *Contribution à la robustesse des systèmes temps réel embarqués multi-cœur automobile*. PhD thesis, l'unam, 2013.

[39] S. Cotard, S. Faucou, J.-L. Bechennec, A. Queudet, and Y. Trinquet. A data flow monitoring service based on runtime verification for autosar. In *IEEE 14th International Conference on High Performance Computing and Communication and IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS),*, pages 1508–1515, 2012.

[40] L. Cucu, R. Kocik, and Y. Sorel. Real-time scheduling for systems with precedence, periodicity and latency constraints. In *Proceedings of Real-time and Embedded Systems*, pages 26–28, 2002.

[41] J.-C. Fabre, M.-O. Killijian, and F. Taiani. Robustness of automotive applications using reflective computing : lessons learnt. In W. C. Chu, W. E. Wong, M. J. Palakal, and C.-C. Hung, editors, *SAC*, pages 230–235. ACM, 2011.

[42] M. Felser, R. Kapitza, J. Kleinöder, and W. Schröder-Preikschat. Dynamic software update of resource-constrained distributed embedded systems. In *Internation Embedded Systems Symposium 2007*, 2007.

[43] L. Feng, D. Chen, and M. Torngren. Self configuration of dependent tasks for dynamically reconfigurable automotive embedded systems. In *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*, pages 3737–3742, Decembre 2008.

[44] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti. Scheduling dependent periodic tasks without synchronization mechanisms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 301–310, April 2010.

[45] Freescale. *MPC5510 Microcontroller Family Reference Manual*, chapter 22 Flash Array and Control, pages 434–461. 2014.

[46] G. Gracioli and A. Frohlich. Elus : A dynamic software reconfiguration infrastructure for embedded systems. In *Telecommunications (ICT), 2010 IEEE 17th International Conference on*, pages 981–988, April 2010.

[47] J. Gray. The transaction concept : Virtues and limitations (invited paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154. VLDB Endowment, 1981.

[48] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transaction Software Engineering*, 22(2) :120–131, 1996.

[49] C.-c. Han, R. Kumar, R. Shea, and M. Srivastava. Sensor network software update management : A survey. *Journal of Network Management*, 15 :283–294, 2005.

[50] H. Hansson, M. Akerholm, I. Crnkovic, and M. Torngren. Saveccm - a component model for safety-critical real-time systems. In *Proceedings of 30th Euromicro Conference*, pages 627–635, August 2004.

[51] K. Heckemann, M. Gesell, T. Pfister, K. Berns, K. Schneider, and M. Trapp. Safe automotive software. In *Proceedings of the 15th international conference on Knowledge-based and intelligent information and engineering systems - Volume Part IV*, KES'11, pages 167–176, Berlin, Heidelberg, 2011. Springer-Verlag.

[52] C. A. R. Hoare. Communicating sequential processes. *Communication, ACM*, 21(8) :666–677, Aug. 1978.

[53] T. Hoppe and J. Dittman. Sniffing/replay attacks on can buses : A simulated attack on the electric window lift classified using an adapted cert taxonomy. In *Proceedings 2nd Workshop on Embedded Systems Security (WESS)*, Salzburg, Austria, 2007.

[54] P. Hosek, T. Pop, T. Bures, P. Hnetynka, and M. Malohlava. Comparison of component frameworks for real-time embedded systems. In L. Grunske, R. Reussner, and F. Plasil, editors, *Component-Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 21–36. Springer Berlin Heidelberg, 2010.

[55] K. Hui, J. Appavoo, R. W. Wisniewski, M. A. Auslander, D. Edelsohn, B. Gamsa, O. Krieger, B. S. Rosenburg, and M. Stumm. Position summary : Supporting hot-swappable components for system software. In *HotOS'01*, pages –1–1, 2001.

[56] C. Insaurralde, M. Seminario, J. Jimenez, and J. Giron-Sierra. Model-based development framework for distributed embedded control of aircraft fuel systems. In *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*, pages 6.E.2–1–6.E.2–14, Octobre 2010.

[57] ISO TC22/SC3/WG16. Iso 26262 - road vehicles, functional safety, november 2011.

[58] G. Karsai, F. Massacci, L. Osterweil, and I. Schieferdecker. Evolving embedded systems. *Computer*, 43(5) :34–40, May 2010.

[59] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Transaction Software Engineering*, 11(4) :424–436, Apr. 1985.

[60] J. Kramer and J. Magee. The evolving philosophers problem : dynamic change management. *IEEE Transactions on Software Engineering,*, 16(11) :1293–1306, Nov 1990.

[61] J.-C. Laprie. Dependability evaluation of software systems in operation. *Software Engineering, IEEE Transactions on*, SE-10(6) :701–714, Nov 1984.

[62] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm : exact characterization and average case behavior. In *Proceedings Real Time Systems Symposium*, pages 166–171, Dec 1989.

[63] Y. Li, F. Wang, F. He, and Z. Li. Osgi-based service gateway architecture for intelligent automobiles. In *Intelligent Vehicles Symposium, 2005. Proceedings. IEEE*, pages 861 – 865, june 2005.

[64] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1) :46–61, Jan. 1973.

[65] C. Lu. *Robustesse du logiciel embarqué multicouche par une approche réflexive : application à l'automobile.* PhD thesis, Université de Toulouse, 2009.

[66] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu. Version-consistent dynamic reconfiguration of component-based distributed systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 245–255, New York, NY, USA, 2011. ACM.

[67] H. Maaskant. A robust component model for consumer electronic products. In P. Stok, editor, *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, volume 3 of *Philips Research*, pages 167–192. Springer Netherlands, 2005.

[68] L. Mangeruca, A. Ferrari, and A. Sangiovanni-Vincentelli. Uniprocessor scheduling under precedence constraints. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, 2006.*, pages 157–166, April 2006.

[69] D. McKinney. Impact of commercial off-the-shelf(cots) software and thechnology on systems engineering. http ://www.incose.org/northstar/2001Slides/McKinneyAugust 2001.

[70] A. Mok and D. Chen. A multiframe model for real-time tasks. In *17th IEEE Real-Time Systems Symposium, 1996.,*, pages 22–29, Dec 1996.

[71] C. Murray. Automakers opting for model-based design. *Design News*, May 2010.

[72] J. Mäki-Turja and M. Nolin. Efficient implementation of tight response-times for tasks with offsets. *Real-Time Systems*, 40(1) :77–116, 2008.

[73] T. Nakanishi, H.-H. Shih, K. Hisazumi, and A. Fukuda. A software update scheme by airwaves for automotive equipment. In *International Conference on Informatics, Electronics Vision (ICIEV), 2013*, pages 1–6, May 2013.

[74] D. Nilsson, P. Phung, and U. Larson. Vehicle ecu classification based on safety-security characteristics. In *Road Transport Information and Control - RTIC 2008 and ITS United Kingdom Members' Conference, IET*, pages 1–7, May 2008.

[75] D. K. Nilsson and U. E. Larson. Simulated attacks on can buses : Vehicle virus. In *Proceedings of the Fifth IASTED International Conference on Communication Systems and Networks*, AsiaCSN '08, pages 66–72, Anaheim, CA, USA, 2008. ACTA Press.

[76] Object Management Group. CORBA Component Model Specifications, July 1999.

[77] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[78] OSEK Group. Osek/vdx operating system (release 2.2.3), 2005. `http://portal.osek-vdx.org/`.

[79] OSEK Group. OSEK/VDX Operating System (Release 2.2.3), 2005. `http://portal.osek-vdx.org/`.

[80] J. C. Palencia and M. González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, pages 26–, Washington, DC, USA, 1998. IEEE Computer Society.

[81] M. Papazoglou. Service-oriented computing : concepts, characteristics and directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering, 2003. WISE*, pages 3 – 12, Decembre 2003.

[82] D. A. Patterson and C. H. Sequin. Risc i : A reduced instruction set vlsi computer. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 443–457, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[83] L. Patzina, S. Patzina, T. Piper, and P. Manns. Model-based generation of runtime monitors for autosar. In P. Gorp, T. Ritter, and L. Rose, editors, *Modelling Foundations and Applications*, volume 7949 of *Lecture Notes in Computer Science*, pages 70–85. Springer Berlin Heidelberg, 2013.

[84] R. Pellizzoni and G. Lipari. Feasibility analysis of real-time periodic tasks with offsets. *Real-Time Systems*, 30(1-2) :105–128, May 2005.

[85] T. Piper, S. Winter, P. Manns, and N. Suri. Instrumenting autosar for dependability assessment : A guidance framework. In *DSN*, pages 1–12, 2012.

[86] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP : Architecture for component trading and dynamic updating. In *ourth International Conference on Configurable Distributed Systems, 1998. Proceedings. F*, pages 43–51, 1998.

[87] P. Prisaznuk. ARINC 653 role in Integrated Modular Avionics (IMA). In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 1.E.5–1–1.E.5–10, Oct 2008.

[88] J. Reedy, S. Lunzman, and B. Mekari. Model-based design accelerates development of mechanical locomotive controls. *SAE International*, 2011.

[89] J. S. Rellermeyer and G. Alonso. Concierge : A service platform for resource-constrained devices. *SIGOPS Operating Systems Revue*, 41(3) :245–258, March 2007.

[90] Research and innivative Technology Administration (RITA). Vehicle-to-infrastructure (v2i) communications for safety. `http://www.its.dot.gov/research/v2i.htm`.

[91] D. Salomon. *Data Compression : The Complete Reference*, pages 850–940. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[92] L. Sha, R. Rajkumar, and M. Gagliardi. Evolving dependable real-time systems. In *Proceedings of Aerospace Applications Conference, 1996*, volume 1, pages 335 –346 vol.1, February 1996.

[93] A. C. Shaw. *Real Time Systems and Software*. 2002.

[94] J. Shen, X. Sun, G. Huang, W. Jiao, Y. Sun, and H. Mei. Towards a unified formal model for supporting mechanisms of dynamic component update. *SIGSOFT Softw. Eng. Notes*, 30(5) :80–89, Septembre 2005.

[95] E. Simon. Osgi en bref. http ://membres-liglab.imag.fr/simon/doc/OSGi-en-bref-v1.0.2.pdf, Septembre 2011.

[96] M. Spuri and J. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. *IEEE Transactions on Computers,*, 43(12) :1407–1412, Decembre 1994.

[97] M. Stoicescu, J.-C. Fabre, and M. Roy. From design for adaptation to component-based resilient computing. In *Proceedings of the 18th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2012)*, pages 1–10, 2012.

[98] I. Studnia, V. Nicomette, E. Alata, Y. Deswarte, M. Kaaniche, and Y. Laarouchi. Security of embedded automotive networks : state of the art and a research proposal. In M. ROY, editor, *Proceedings of Workshop CARS (2nd Workshop on Critical Automotive applications : Robustness & Safety) of the 32nd International Conference on Computer Safety, Reliability and Security*, page NA, Toulouse, France, Sept. 2013. Rapport LAAS n ° 13313 Rapport LAAS n ° 13313.

[99] C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[100] M. Trapp, R. Adler, M. Förster, and J. Junger. Runtime adaptation in safety-critical automotive systems. In *Proceedings of the 25th conference on IASTED International Multi-Conference : Software Engineering*, SE'07, pages 308–315, Anaheim, CA, USA, 2007. ACTA Press.

[101] UNECE. E/ece/trans/505. `http://www.unece.org/fileadmin/DAM/trans/main/wp29/wp29regs/r48r6e.pdf`.

[102] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3) :78–85, March 2000.

[103] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility : A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *IEEE Transactions on Software Engineering,*, 33(12) :856–868, Decembre 2007.

[104] B. Y. Vandewoude Yves. An overview and assessment of dynamic update methods for component-oriented embedded systems,. In *proceedings of The international Conference on Software Engineering Research and Practice*, pages 521–527, Las Vegas USA, 2002.

[105] Vector Informatik GmbH. The universal gateway ecu. `https://www.vector.com/portal/medien/cmc/press/Vector/AUTOSAR_Gateway_ElektronikAutomotive_200710_PressArticle_EN.pdf`.

[106] S. Voget and P. Favrais. AUTOSAR and the Automotive Tool Chain. *Methodology*, 2010.

[107] R. Warschofsky. AUTOSAR Software Architecture. Technical report, Hasso-Plattner-Institute fur Softwaresystemtechnik, 2009.

[108] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-case Execution-time Problem : Overview of Methods and Survey of Tools. *ACM Transaction Embedded Computer System*, 7(3) :36 :1–36 :53, May 2008.

Résumé

Dans le contexte automobile actuel, le standard pour les calculateurs enfouis est AUTOSAR. L'un des inconvénients majeurs de cette architecture est son manque de flexibilité. Cependant, les mises à jour et la personnalisation des systèmes embarqués sont de plus en plus, non seulement plébiscités, mais également nécessaires. En effet, la complexité grandissante des systèmes exige à présent de déployer des moyens supplémentaires pour permettre leur maintenance et leur évolution de manière plus aisée.

Ainsi, partant de ces constats, ce travail étudie les possibilités de faire des mises à jour dans le contexte d'AUTOSAR. Les modifications nécessaires se retrouvent non seulement dans l'architecture, mais également au sein du processus de développement et des considérations temps-réel. Tous ces aspects sont donc regardés en détails pour permettre les mises à jour partielles dans le cadre du standard AUTOSAR.

Cette thèse décrit donc le processus de développement logiciel AUTOSAR et propose certaines améliorations mises en place au cours de ce travail. Un certain nombre de concepts sont également définis, afin d'aménager des espaces d'adaptation logiciels. Ces espaces sont ensuite utilisés pour intégrer des mises à jour partielles dans le calculateur embarqué. Le processus de développement est également modifié pour intégrer ces concepts ainsi que les mécanismes nécessaires à la mise à jour.

Les aspects temps-réel concernant la mise à jour partielle dans les systèmes embarqués automobiles sont également traités ici. Un modèle de tâches approprié est mis en place dans le cadre d'AUTOSAR. De plus l'analyse de sensibilité est utilisée spécifiquement pour déterminer la flexibilité disponible dans un système donné.

Les aspects d'implémentation sont également détaillés. En particulier, la création de mises à jour dans un contexte donné, la gestion des différentes versions possibles pour une application, l'utilisation et l'écriture dans la mémoire embarquée et enfin, les moyens nécessaires à la prise en compte des aspects de sûreté de fonctionnement.

Pour terminer, tous les concepts développés dans ce travail sont appliqués à une preuve de concept reposant sur une application embarquée fournie par Renault. L'approche proposée est donc appliquée de manière pratique.

Mots-clés: Mise à jour, Automobile, Système Embarqué, Temps-réel

Abstract

Currently the standard for embedded ECUs (Electronic Control Unit) in the automotive industry is AUTOSAR. One of the drawbacks of this architecture lies in its lack of flexibility. However, updates and customization of embedded systems are increasingly

demanded and necessary. Indeed, systems are more and more complex and therefore require new methods and means to ease maintenance.

Thus, from these observations, we study the possibilities for updates and resulting modifications (both on an architectural level and within the development process, and from a real-time point of view) in order to integrate within the AUTOSAR standard partial updates.

This PhD thesis describes the software development process in an AUTOSAR context with a number of improvement we designed in this work. We also define concepts that allow to introduce placeholders for further updates within the embedded ECU. The development process has to be subsequently modified for integrating spaces for the updates along with the necessary mechanisms.

Real-time problematic regarding partial updates in automotive systems is also considered here. In particular, we deal with sensitivity analysis that helps determine flexibility within the system.

A number of implementation aspects are also detailed. In particular, the creation of the updates, versions management, use of embedded memory and dependability.

All these concepts are finally applied on a proof of concept using an embedded application from Renault. We present here in details how the proposed approach can be used in practice.

Keywords: Update, Automotive, Embedded System, Real-time