l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Sylvain COTARD

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
Label européen
sous le label de l'Université de Nantes Angers Le Mans

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique, automatique et traitement du signal
Unité de recherche : Institut de Recherche en Communications et Cybernétique de Nantes (IRCCyN)

Soutenue le 12 décembre 2013
Thèse n° : ED 503-208

Contribution à la robustesse des systèmes temps réel embarqués multicœur automobile

JURY

Rapporteurs :	**M. Jean-Charles FABRE**, Professeur, INP de Toulouse, France
	M. Luis Miguel PINHO, Professeur, Politécnico do Porto, Portugal
Examinateurs :	**M^me^ Isabelle PUAUT**, Professeur, Université de Rennes, France
	M^me^ Audrey QUEUDET, Maître de conférences, Université de Nantes, France
	M. Jean-Luc BÉCHENNEC, Chargé de recherche, CNRS, France
	M. Sébastien FAUCOU, Maître de conférences, Université de Nantes, France
	M. Gaël THOMAS, Maître de conférences (HDR), Université Pierre et Marie Curie, France
Invités :	**M. Rémy BRUGNON**, Ingénieur logiciel, Renault Guyancourt, France
	M. Sébastien LE NOURS, Maître de conférence, Université de Nantes, France
Directeur de thèse :	**M. Yvon TRINQUET**, Professeur, Université de Nantes, France

À ma famille, À Julie, À mes amis

Remerciements

Je tiens en premier lieu à remercier Frédérique Pasquier, René Aubrée et Damien Grolleau, membres du laboratoire *Génie Electrique et Automatique* de l'ICAM de Nantes, pour m'avoir conseillé et soutenu dans mon choix de poursuite d'études.

Je remercie mon directeur de thèse, M. Yvon Trinquet, pour m'avoir fait confiance et donné l'opportunité d'effectuer mon doctorat dans l'équipe *Système Temps Réel* de l'IRCCyN. Je remercie l'ensemble des membres permanents et j'exprime ma plus forte gratitude à mon équipe encadrante, Yvon Trinquet, Sébastien Faucou, Audrey Queudet et Jean-Luc Béchennec, pour leur disponibilité, pour m'avoir soutenu, guidé et conseillé tout au long du déroulement de mes travaux. Leurs qualités humaines et leurs compétences scientifiques m'ont permis d'acquérir les aptitudes essentielles pour mener à bien mon projet de thèse.

Je remercie M. Philippe Quéré, chef de l'équipe *Logiciel Embarqué Temps Réel* chez Renault, pour m'avoir accueilli dans son équipe, et M. Rémy Brugnon pour m'avoir encadré durant ces trois ans. Leurs expériences et leurs expertises ont été essentielles pour le bon déroulement de mes travaux.

J'exprime ma profonde gratitude à Mme Isabelle Puault, maître de conférence à l'IRISA de Rennes pour m'avoir fait l'honneur de présider mon jury de thèse. Je remercie également M. Jean-Charles Fabre, professeur à l'INP de Toulouse, et M. Luis Miguel Pinho, professeur à l'institut polytechnique de Porto, pour avoir rapporté mon manuscrit de thèse. Enfin, je remercie également

- M. Gaël Thomas, maître de conférence (HDR) à l'université de Pierre et Marie Curie ;
- Mme Audrey Queudet, maître de conférence à l'université de Nantes ;
- M. Jean-Luc Béchennec, chargé de recherche CNRS ;
- M. Sébastien Faucou, maître de conférence à l'université de Nantes ;
- M. Yvon Trinquet, professeur à l'université de Nantes ;
- M. Sébastien Le Nours, maître de conférence à l'université de Nantes ;
- M. Rémy Brugnon, ingénieur logiciel chez Renault.

pour avoir accepté de participer à mon jury de thèse.

Une pensée particulière est dédiée à l'équipe *Logiciel Embarqué Temps Réel* de Renault, à

savoir Joris, Patrick, Pascal, Régis, Youssef, Rémy et Philippe. Leurs compétences techniques et leurs retours constructifs ont facilité le passage de concepts théoriques dans un contexte industriel concret et réaliste. Leur bonne humeur permanente a également permis de faciliter mon intégration dans l'équipe.

Mes années de thèse se sont déroulées dans une ambiance très agréable. Sans les citer tous, j'associe à mes remerciements l'ensemble des doctorants qui m'ont supporté durant ces trois ans. En particulier, un grand merci à Jonathan, Maissa, Nadine, Aleksandra, Céline, Charbel, Denis, Thomas, Dominique, Inès, Saab, Lauriane, Julien, Adrien, Thomas, Antoine, Hélène, Ivan, ... Que ceux que je n'ai pas mentionnés me pardonnent.

Enfin, puisque l'environnement familial a été, est, et sera toujours essentiel dans la réussite de mes projets, je pense très fort à mes parents, mon frère Nicolas et mon amie Julie. Les encouragements et le soutien dont ils ont fait preuve au quotidien ont énormément contribué à l'accomplissement de mon travail.

Table des matières

Introduction générale

Depuis quelques décennies, l'électronique envahit notre vie quotidienne. Son émergence est accompagnée de celle de l'informatique embarquée. Il s'agit d'un outil qui regroupe les aspects logiciels permettant à un équipement ou une machine de remplir une fonction spécifique. Un tel équipement est qualifié de *système embarqué*. Du four à micro-ondes aux fusées en passant par la téléphonie, la télévision, les satellites, les appareils électroménagers, l'automobile, l'avionique ou encore les transports urbains, les systèmes embarqués sont de nos jours incontournables.

Parmi les domaines où ils sont très utilisés, celui de l'automobile nous intéresse particulièrement. En effet, les travaux de thèse présentés ici sont le résultat d'une collaboration CIFRE entre l'équipe *Logiciel Embarqué Temps Réel – LETR, DE-SFF6* du Technocentre Renault à Guyancourt et l'équipe *Systèmes Temps Réel – STR* de l'Institut de Recherche en Communications et Cybernétique de Nantes (IRCCyN).

L'électronique dans les véhicules prend une place de plus en plus importante et ouvre la porte à une nouvelle méthode de conception basée sur la *mécatronique*. Il s'agit d'un savant mélange coopératif de l'électricité, l'électronique et l'informatique, pour le pilotage intelligent des organes mécaniques du véhicule. Cette méthode de conception permet de repousser les limites que l'on connaissait jusqu'alors avec les solutions purement mécaniques et hydrauliques. La première utilisation d'un système embarqué dans un véhicule a été proposée dès les années 1960 pour assister le pilotage des injecteurs et optimiser le rendement moteur. Depuis, de nombreux organes du véhicule reposent sur ce type de système. Par exemple, l'ABS (*Anti Braking System*) ou l'ESP (*Electronic Stability Program*) permettent d'accroître la sécurité. L'industrie automobile observe depuis une vingtaine d'années une constante augmentation de la part d'électronique dans les véhicules. Cette tendance est dictée par l'évolution du marché (contexte concurrentiel), des contraintes sécuritaires, des réglementations ou encore des contraintes environnementales (réduction des émissions de CO_2). Actuellement, une grande part des innovations apportées aux véhicules est lié à l'électronique.

L'accroissement du nombre de prestations et de la complexité des systèmes embarqués amènent les principaux acteurs du secteur à se poser des questions relatives à leur sûreté de fonctionnement, qui est aujourd'hui un axe de travail majeur. En effet, il faut s'assurer que les prestations offertes soient conformes aux spécifications et qu'aucun évènement indésirable au vu du client ne puisse se produire. La norme ISO 26262 (ISO 26262, 2011) est depuis 2011 la

référence dans le domaine automobile. De plus, pour rationaliser la conception des systèmes logiciels, des standards tels OSEK/VDX (OSEK/VDX, 2005b) puis AUTOSAR (AUTOSAR, 2013) ont été mis en place.

AUTOSAR offre depuis la révision 4.0 la possibilité de développer des applications logicielles sur des architectures multicœur, c'est-à-dire pour lesquelles il est possible d'effectuer plusieurs traitements en parallèle. Cette évolution est nécessaire de par l'augmentation de la puissance de calcul requise par l'augmentation du nombre de prestations, sous la contrainte du maintien du nombre de calculateurs embarqués dans un véhicule. Le revers de la médaille est que l'utilisation des architectures multicœur apporte un lot de problèmes spécifiques et inhérents aux interactions intercœur qu'il convient d'inventorier, et d'apprendre à maîtriser.

Nos travaux visent dans un premier temps à caractériser ces problèmes. En effet, dans ce contexte, la considération du parallélisme et son exploitation sûre demande beaucoup de maîtrise. Par exemple, la mise au point de l'ordonnancement temps réel dans un contexte multicœur est difficile puisque la plupart des résultats d'optimalité établis dans le contexte monocœur ne sont plus valables. De plus, la coopération des cœurs (partage des caches, partage des bus, partage des périphériques, etc.) est une source d'indéterminisme pour le calcul du WCET. Enfin, les données stockées dans la mémoire commune peuvent être corrompues en cas d'accès parallèle par différents cœurs. En observant l'absence d'outils dédiés à l'exploitation sûre de ces architectures, nous proposons dans un second temps des solutions permettant d'accroître la sûreté de fonctionnement d'une application logicielle multicœur.

Le document est divisé en trois parties. Dans la partie I, nous proposons une étude préliminaire permettant de préciser le périmètre de nos travaux et de présenter la stratégie de résolution retenue. Dans le Chapitre 1, nous présentons le contexte industriel automobile et l'ensemble des raisons qui motivent l'étude des architectures multicœur dans ce domaine. Les spécificités de ces architectures et l'ensemble du contexte scientifique sont abordés dans le Chapitre 2. Nous présentons en particulier une introduction au développement d'un système temps réel sur une architecture multicœur. Dans le Chapitre 3, nous précisons les deux axe d'études choisis pour s'appliquer au modèle de fautes que nous avons retenu.

Dans la Partie II, nous présentons un mécanisme de tolérance aux fautes basé sur la vérification en ligne. Dans le Chapitre 4, nous définissons ce qu'est la vérification en ligne, quelles sont les alternatives, et en quoi cette technique est complémentaire aux autres solutions déjà utilisées dans le processus de développement d'une application. Dans le Chapitre 5, nous présentons l'état de l'art pour la synthèse hors-ligne de moniteurs dédiés à la surveillance d'une propriété exprimée formellement à l'aide d'une formule de logique temporelle. Nous décrivons ensuite notre approche qui consiste à synthétiser un moniteur à partir d'une formule LTL et d'un modèle à évènements discrets du logiciel à observer. Dans le Chapitre 6, nous montrons comment les moniteurs synthétisés hors-ligne par l'outil *Enforcer* (conçu dans le cadre des travaux) sont insérés dans le noyau du système d'exploitation temps réel *Trampoline* (développé par l'équipe STR de l'IRCCyN et conforme au standard AUTOSAR) pour obtenir une implémentation conforme aux contraintes du domaine automobile. Une évaluation temporelle est également proposée. Le service de vérification ainsi mis en place est évalué dans le Chapitre 7 par une analyse hors-ligne de l'impact de la vérification à l'échelle d'un calculateur, dans le cadre d'une application automobile concrète. Pour finir, une discussion est proposée dans le Chapitre 8.

La Partie III concerne le second axe d'étude abordé dans nos travaux. Nous y présentons la conception d'un protocole non bloquant pour le partage de ressources logicielles (pour nous, des données) dans les architectures multicœur. Le protocole ciblé repose sur l'étude des mécanismes à mémoire transactionnelle et des mécanismes wait-free. L'état de l'art associé et les principes théoriques sont détaillés dans le Chapitre 9. Sur ces bases, nous proposons le protocole *STM-HRT – Software Transactional Memory for Hard Real-Time systems*, wait-free et inspiré des principes des mémoires transactionnelles. Ce protocole et les structures de données sur lesquelles il repose sont présentés dans le Chapitre 10. Des éléments de vérification basées sur l'outil de vérification de modèle SPIN sont proposés dans le Chapitre 11 ainsi qu'une analyse temporelle de la solution ainsi qu'un bilan de son empreinte mémoire. Pour finir, une discussion est proposée dans le Chapitre 12.

Pour terminer, la conclusion résumera l'ensemble des travaux et leur adéquation aux contraintes industrielles automobiles.

Première partie

Introduction

Chapitre 1

Contexte industriel automobile

Sommaire

1.1 La mécatronique dans l'automobile

1.1.1 Définition

Depuis la fin des années 1990, l'industrie automobile a adapté sa manière de concevoir un véhicule. Le déferlement des innovations électroniques et informatiques conduit à l'introduction de composants électroniques pilotés par l'informatique, en remplacement de composants mécaniques et hydrauliques : c'est l'émergence de la mécatronique.

Définition 1.1. *La mécatronique est la démarche visant l'intégration en synergie de la mécanique, l'électronique, l'automatique et l'informatique dans la conception et la fabrication d'un produit en vue d'augmenter et/ou d'optimiser sa fonctionnalité (NF E 01-010, 2008).*

L'intérêt de ce domaine d'ingénierie interdisciplinaire est de concevoir des systèmes automatiques pour permettre le contrôle de systèmes complexes. Il permet également de minimiser les coûts de développement et/ou développer rapidement de nouvelles fonctions.

Les domaines ouverts à la mécatronique sont variés. Dans l'automobile, on les retrouve par exemple pour le contrôle du moteur (e.g. injection, turbocompresseur, recyclage des échappements, refroidissement), l'habitacle (e.g. airbags), le freinage (e.g. antiblocage ABS, freinage d'urgence), la direction (e.g. assistance de direction, contrôle de trajectoire ESP).

Plus précisément, la mécatronique est composée de divers constituants qui interagissent :

- Les composants mécaniques, hydrauliques et de câblage sont les entités qui permettent le contrôle physique des différents organes du véhicule ;
- Le système de contrôle électrique/électronique (i.e. le système E/E) regroupe les entités matérielles et logicielles qui permettent de piloter de manière intelligente les composants mécaniques et hydrauliques en vue de remplir les fonctions du cahier des charges (c.f. Section 1.1.2).

Cette nouvelle architecture permet également l'intégration et la mise en réseau des différents composants en vue de multiplexer les fonctions distribuées, de mettre en place des structures de pilotage hiérarchisées, d'intégrer des systèmes d'aide à la conduite ou encore de disposer de systèmes pour les diagnostics. Le nombre de composants électroniques (e.g. le nombre de calculateurs) a beaucoup augmenté ces 20 dernières années.

1.1.2 L'architecture électrique/électronique (E/E)

L'architecture E/E peut être divisée en quatre niveaux, comme illustré sur la Figure 1.1. Parmi ces niveaux, nous retrouvons les architectures *fonctionnelle*, *logicielle*, *matérielle* et *opérationnelle*.

Une architecture E/E automobile s'exécute sur un ensemble de calculateurs appelés ECUs (*Electronic Control Units*), répartis dans le véhicule. Ces ECUs offrent des accès d'entrées/sorties vers les autres composants du système mécatronique en vue d'interagir avec eux. Les *capteurs* récupèrent des signaux logiques ou analogiques à partir de l'environnement pour les fournir aux calculateurs (e.g. valeur d'un régime moteur, niveau des fluides, débit des injecteurs). Les calculateurs déduisent les commandes qu'il faut appliquer aux *actionneurs* pour que la fonction soit remplie.

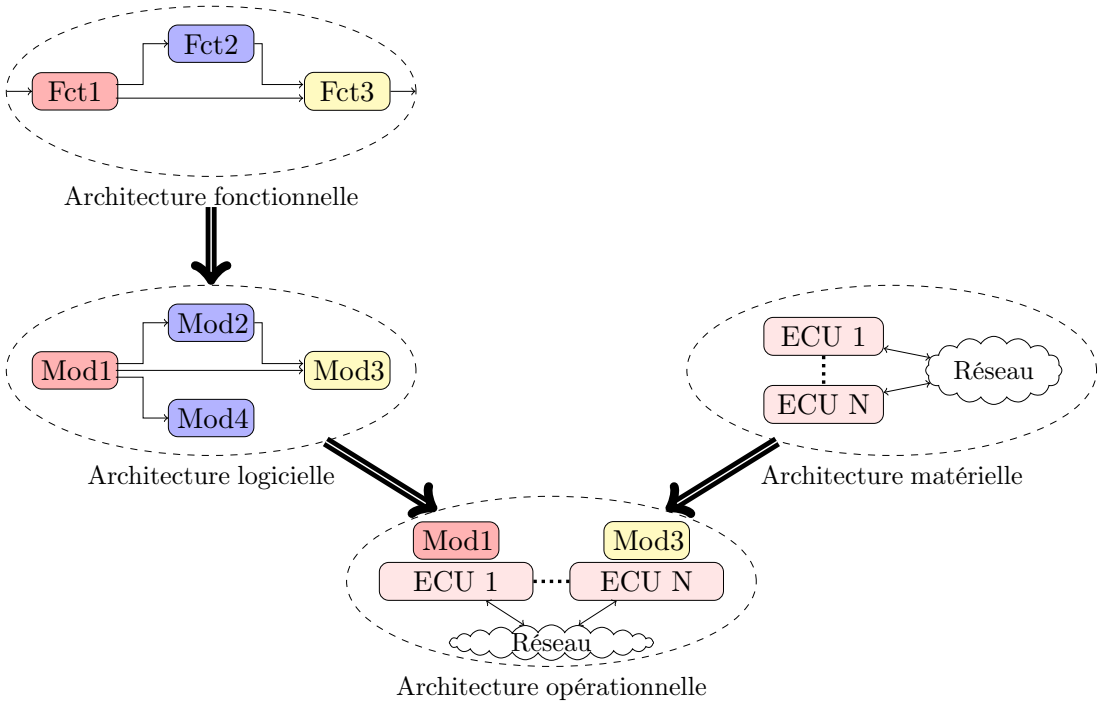


Figure 1.1 – Architecture d'un système E/E

Pour décrire le comportement requis pour l'ensemble des calculateurs, une analyse fonctionnelle de l'application est nécessaire. L'*architecture fonctionnelle* qui en est dérivée est une description à haut niveau des fonctionnalités attendues (fonctions applicatives). Celles-ci sont souvent découpées en blocs fonctionnels, ce qui permet de spécifier chaque composant indépendamment puis de décrire le comportement de l'ensemble. La description de cette architecture est réalisée très tôt dans le processus de développement et se conforme aux spécifications techniques des systèmes.

La connaissance de l'architecture fonctionnelle permet de spécifier l'*architecture logicielle*. Celle-ci décrit les éléments du système, leurs interrelations et leurs interactions. Contrairement à l'architecture fonctionnelle qui décrit ce que le système fait, l'architecture logicielle décrit comment l'application est conçue pour qu'elle puisse réaliser les fonctions. Le découpage prévu doit permettre, après codage, de répondre aux contraintes temps réel dur qui sont requises.

L'*architecture matérielle* est la description des composants matériels utilisés pour que l'application puisse remplir sa mission. Plusieurs critères sont étudiés : le nombre d'entrées/sorties, la puissance calculatoire, le nombre de cœurs, l'adéquation des calculateurs au domaine de fonctionnement, le nombre de contrôleurs CAN, etc. L'architecture matérielle décrit également la manière dont les ECUs communiquent entre eux via les réseaux (e.g. *Control Area Network* – CAN, *Local Interconnect Network* – LIN) dans le cas où des prestations complexes nécessitent l'intervention de plusieurs ECUs.

Enfin, l'application codée et déployée correspond au *niveau opérationnel*. Il est le résultat de la combinaison des niveaux d'architectures précédents.

1.1.3 De l'architecture fédérée à l'architecture intégrée

L'avènement des réseaux multiplexés dans l'automobile a conduit à la mise en place d'une architecture opérationnelle fédérée. Elle limite la complexité de l'implantation du logiciel puisqu'une ECU est dédiée à un type de travail (une ECU = une prestation) et un canal de communication est fourni pour permettre l'échange des messages. Ce type d'architecture facilite la démonstration de la *sûreté* puisqu'il limite naturellement le nombre de canaux de propagation d'erreurs en instaurant une séparation physique entre les prestations. En revanche, l'ajout d'une nouvelle prestation impose l'ajout d'une nouvelle ECU. De plus, la mise en place de prestations complexes peut malgré tout nécessiter le partage d'informations entre les ECUs (e.g. capteurs) et imposer la mise en place d'une infrastructure physique importante pour la communication (e.g. câblage, réseau).

L'évolution a conduit à l'introduction d'un second type d'architecture : l'architecture intégrée. Elle autorise l'implantation de plusieurs sous-systèmes distribués (DAS - *Distributed Application Subsystem*) au sein d'une unique ECU, ce qui a pour effet d'en diminuer le nombre ainsi que la quantité de points de connexion. Il en résulte un coût d'infrastructure moins élevé et une maintenance plus aisée. En revanche, autoriser plusieurs types de prestations à s'exécuter sur une même ECU supprime la barrière physique que l'on avait avec la précédente architecture. Il en résulte un accroissement de la complexité du logiciel, nécessaire pour assurer la maîtrise du comportement et un effort supplémentaire pour répondre aux problématiques de sûreté de fonctionnement est donc nécessaire. Ces objectifs sont concrétisés par l'élaboration du standard de développement logiciel AUTOSAR et de la norme de sûreté de fonctionnement ISO 26262 (c.f. Section 1.3 page 15).

L'industrie automobile utilise depuis quelques années ce dernier type d'architecture pour intégrer des sous-systèmes différents sur le même calculateur.

1.2 Processus de développement d'une architecture E/E

1.2.1 L'ingénierie système

L'ingénierie système, définie par l'Association Française d'Ingénierie Système (AFIS, 2013), est la démarche méthodologique progressive prenant en compte l'ensemble des activités, pour contrôler la conception de systèmes complexes. Il s'agit d'un processus de haut niveau qui englobe l'ensemble du cycle de vie. Le cycle de vie d'un système désigne ses différentes *phases de vie*, regroupées dans les catégories suivantes : conception, production, distribution, exploitation, élimination.

L'ingénierie système se focalise sur la définition des besoins exprimés par le client au début de la phase de conception (i.e. exigences fonctionnelles), et sur les architectures. Une fois les exigences analysées, elle assure le suivi du processus global de mise au point, de la conception à l'intégration : i.e. l'environnement, le contexte, les enjeux, les coûts, la planification et la répartition des tâches, le pilotage du projet, la conception, la maintenance et la fin de vie. En particulier, la conception du système fait appel à l'ensemble des domaines appelés IVVQ (*Intégration, Vérification, Validation, Qualification*), mettant en place les stratégies nécessaires à l'atteinte de la conformité et de la qualité, tout en optimisant au mieux les objectifs QCD (*Qualité, Coût, Délais*). Ce processus nécessite l'utilisation d'outils logiciels (e.g. dSpace, Simulink) et la mise en place d'un atelier de développement spécifique. Ceci a pour but de rationaliser la production d'un système et son suivi.

Chez Renault, le processus de conception est rythmé par une logique de développement visant à gérer les acteurs de l'entreprise de façon coordonnée. Ce processus est décomposable en trois phases : la phase d'avant projet (e.g. étude de faisabilité), la phase projet (i.e. conception) et la phase d'industrialisation. Les spécificités du secteur automobile nécessitent également de gérer les interactions entre OEM (*Original Equipment Manufacturer* – le constructeur) et les équipementiers de rangs 1, 2 et 3 (i.e. respectivement Tier-1, Tier-2, Tier-3). Dans l'ingénierie système (exigences et architectures) mise en œuvre chez Renault, les spécifications des constituants issus de l'architecture des systèmes (i.e. spécifications ECUs) sont confiées aux Tiers-1 pour le développement. Les Tiers-1 peuvent eux-mêmes faire appel à des équipementiers de rang 2 (Tier-2), etc.

La mise en pratique de l'ingénierie système est réalisée tout au long du processus de développement. Le cycle en V décrit ci-après en est un exemple.

1.2.2 Le cycle de développement en V

À partir de maintenant, nous nous focalisons sur le développement du logiciel. Le cycle de développement en V est résumé sur la Figure 1.2.

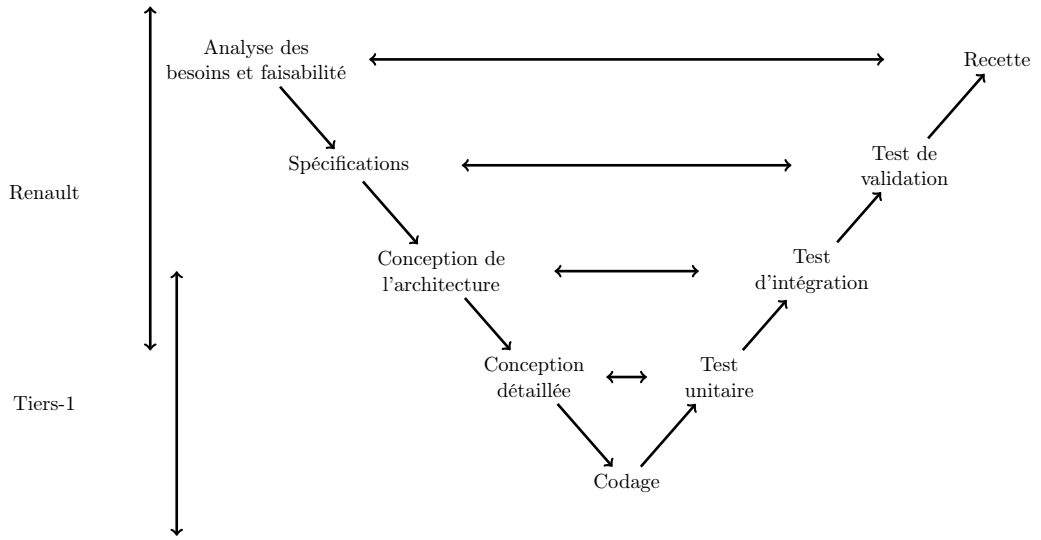


Figure 1.2 – Processus de développement en V

La branche descendante correspond aux étapes d'analyse, de spécification et de conception. Plus on descend, plus le niveau de détails est important. L'étape de codage est le résultat final concret de cette branche descendante. La branche montante correspond aux phases d'intégration. En commençant par les niveaux de détails les plus bas, l'intégration requiert de tester toutes les fonctionnalités du système. Chez Renault, le développement d'une application logicielle suit un cycle en V. Ce développement est le résultat de la coopération avec les Tier-1. La mise en pratique n'est pas toujours aussi statique, du fait d'évolutions de spécifications pouvant survenir pendant le projet (i.e. itérations du processus).

L'accroissement de la quantité de fonctionnalités réalisées par le logiciel pose la question de la sûreté de fonctionnement, c'est-à-dire s'assurer que l'application mise en place respecte ses spécifications dans toutes les configurations possibles, quelles que soient les contraintes

environnementales. La sûreté de fonctionnement du logiciel est un processus accompagnant tout le cycle de développement. Son périmètre est décrit ci-après.

1.2.3 La sûreté de fonctionnement (SdF) logicielle

1.2.3.1 Terminologie de la sûreté de fonctionnement

Définition 1.2. *La sûreté de fonctionnement est la propriété qui permet aux utilisateurs d'un système de placer une confiance justifiée dans le service qu'il leur délivre. (Laprie* et al.*, 1995)*

La sûreté de fonctionnement peut alors être vue comme l'aptitude à éviter les défaillances du service délivré par un système. Le service correspond au comportement du système tel qu'il est perçu par les utilisateurs. Ces utilisateurs sont d'autres systèmes (humains ou physiques) qui interagissent avec celui considéré.

L'arbre de la sûreté de fonctionnement de la Figure 1.3 laisse apparaître trois grandes catégories qui structurent le domaine : les *entraves* à la SdF sont des circonstances indésirables, mais non inattendues, qui sont la cause de la non-sûreté de fonctionnement ; les *moyens* de la SdF sont les méthodes mises en place pour fournir au système l'aptitude à être sûr lors de l'accomplissement de sa mission ; enfin, les *attributs* de la SdF permettent de caractériser la qualité du service délivré.

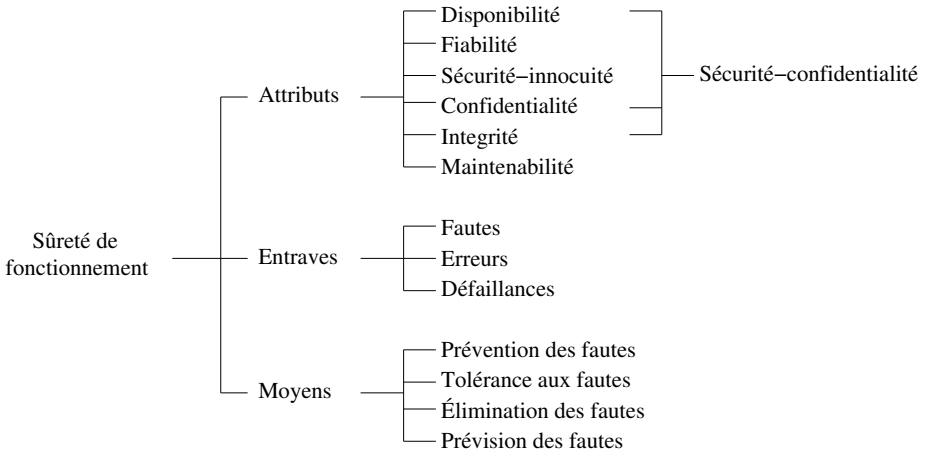


Figure 1.3 – L'arbre de la sûreté de fonctionnement Laprie *et al.* (1995)

Les attributs relatifs à la sécurité-confidentialité ne sont pas considérés dans nos travaux. Ils relèvent de la fragilité du système vis-à-vis des malveillances. La *fiabilité* permet de mesurer la capacité du système à assurer la continuité du service, la *sécurité-innocuité* mesure la capacité du système à éviter les conséquences graves ou catastrophiques, et la *maintenabilité* spécifie la capacité du système à être réparée (maintenance curative) ou à évoluer (maintenance préventive).

En fonction du domaine industriel considéré, les objectifs en termes de sûreté de fonctionnement ne sont pas les mêmes. Par exemple, dans les transports, la fiabilité et la sécurité-innocuité sont privilégiés tandis que pour le e-commerce, c'est la sécurité-confidentialité et la fiabilité qui seront mises en avant (McGregor, 2007).

Les moyens regroupent l'ensemble des techniques qui permettent d'atteindre les exigences en termes de sûreté de fonctionnement. Il en existe quatre : *la prévention des fautes* est une démarche entreprise dès les premières phases du cycle de vie du produit visant à empêcher l'introduction de celles-ci ; *l'élimination des fautes* vise à réduire leur présence en les trouvant et en les supprimant (correction), aussi bien en termes de quantité que de sévérité ; *la prévision des fautes* cherche à estimer leur présence et leurs conséquences (e.g. par l'utilisation d'outils de type *arbre de fautes* ou analyse *AMDE(C) – Analyse des Modes de Défaillance, de leurs Effets (et leurs Criticités)*). Le dernier moyen dénommé *tolérance aux fautes* vise à mettre en place des solutions aptes à tolérer la présence de fautes pendant la phase opérationnelle du système. Elle sera particulièrement étudiée par la suite. L'objectif majeur réside dans le fait que le service soit toujours capable de garantir sa fonction, éventuellement dégradée, même en présence de fautes.

1.2.3.2 Les entraves : fautes, erreurs, défaillances

Définition 1.3. *La défaillance du système survient dès lors que le service délivré dévie de la fonction initiale du système. (Laprie* et al.*, 1995)*

La *défaillance* ne survient pas toujours de la même façon : on parle de *modes de défaillance*. On peut classer selon leurs domaines (en temps ou en valeurs), les perceptions (cohérentes ou incohérentes) qu'en ont les utilisateurs ou encore leurs sévérités (mineures ou catastrophiques). Candea (2007) introduit les notions de défaillances *Heisenbug*, *Bohrbug* et *Schroedingbug* pour distinguer celles qui sont aisément reproductibles et compréhensibles de celles qui ne le sont pas. Il introduit aussi les termes de défaillances *fail stop*, *byzantine* et *fail fast*. Le premier terme correspond à des situations qui conduiront à un arrêt du système sans production de sortie. Le second à la production de sorties erronées, sans aucune maîtrise et le dernier reporte aussitôt sur son interface le fait qu'il est en défaut.

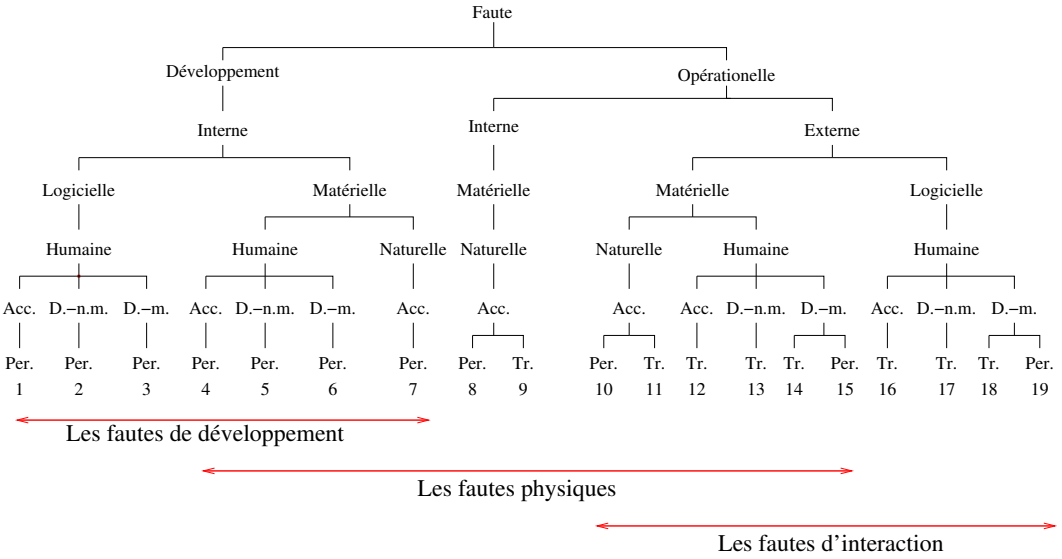
La défaillance est le résultat de la *propagation* d'une *erreur*. Cette erreur résulte elle-même de *l'activation* d'une *faute* (une faute non activée est qualifiée de *dormante*). La faute est souvent visible à l'échelle du matériel ou du logiciel tandis que la défaillance se traduit directement sur le service délivré (au niveau système). Il est également possible qu'une défaillance pour un sous-système soit la *cause* d'une nouvelle faute au niveau du système qui l'englobe. Des défaillances en chaîne peuvent conduire à des évènements indésirables.

La classification des fautes est décrite par l'arbre de la Figure 1.4. Trois groupes apparaissent : les fautes de développement se produisent pendant la phase de développement (e.g. codage), les fautes physiques sont des fautes matérielles et les fautes d'interactions se produisent en phase opérationnelle, lors de l'interaction entre les composants logiciels. Cet arbre est la base pour l'établissement d'un *modèle de fautes*.

1.2.3.3 Mise en pratique dans le contexte industriel automobile

La sûreté de fonctionnement est un objectif majeur pour le développement d'une application ou d'un système et la démarche associée est mise en œuvre tout le long du cycle de développement.

La prévention et la prévision des fautes sont réalisées en instaurant un cadre de développement incluant les démarches de sûreté de fonctionnement (e.g. normes, procédures). En particulier, la prévision vise à déterminer pour une prestation donnée, tous les évènements redoutés. Pour cela, il faut lister les défaillances qui peuvent conduire à une déviation sérieuse



Acc.: Accidentelle, D.–n.m.: Délibérée non malveillante , D.–m.: Délibérée malveillante
Per.: Persistante, Tr.: Transitoire

Figure 1.4 – Modèle de fautes (Arlat *et al.*, 2006)

du comportement du système. Les outils d'analyse de type arbre de défaillance, AMDEC, ou arbre de fautes permettent de caractériser chacune des défaillances et de les classer selon leur gravité. Ces résultats sont une base aux activités visant l'élimination des fautes et la tolérance aux fautes.

L'élimination des fautes peut être associée à l'étape de vérification du système. Pour cela, des outils basés sur des méthodes formelles tels la vérification de modèle ou des protocoles de tests permettent de cibler et de corriger des problèmes. Dans le cas du développement logiciel, des étapes de relecture de spécifications et du code sont des moyens de vérification utilisés.

Des projets ont également été conduits pour mettre en place des stratégies de tolérance aux fautes adaptées aux systèmes automobiles. L'architecture *E-GAS* (EGAS, 2004) vise la mise au point d'une architecture logicielle de surveillance sur 3 niveaux, pour le contrôle des moteurs essence ou diesel. Les trois niveaux sont indépendants. Le premier (L1) est situé au niveau fonctionnel et réalise le contrôle des capteurs et des actionneurs. Le second (L2) comprend des mécanismes logiciels basés sur des contrôles de vraisemblance. Des blocs s'assurent que les calculs effectués par les composants sont exacts. En cas de défaillance, le système est stoppé et reconfiguré. Le troisième niveau (L3) s'assure du fonctionnement du calculateur par un jeu de questions/réponses. Le projet *EASIS* (EASIS, 2006) répond à un manque de standardisation de l'approche visant à concevoir, développer, et maintenir des systèmes critiques ayant des fortes contraintes de sûreté de fonctionnement. Le principal objectif est de concevoir une plateforme logicielle sûre de fonctionnement pour les systèmes électroniques automobiles, fournissant des services sur lesquels les futures applications pourront être construites, à moindre coût. Pour cela, il faut définir des méthodes et des techniques pour gérer les parties critiques du développement au cours du cycle de vie. Enfin, le projet *SCARLET* (SCARLET, 2007) vise l'implémentation de mécanismes de robustesse dans les systèmes multicouches. L'objectif est de proposer une stratégie de configuration et de mise en œuvre permettant d'atteindre des niveaux de robustesse compatibles avec les exigences de sûreté de fonctionnement et de

dimensionnement des futures applications automobiles. L'étude est focalisée sur les composants d'une architecture logicielle embarquée. Des résultats sont présentés dans Lu (2009) et Bertrand *et al.* (2009).

1.3 Contexte « normatif »

1.3.1 Vers la nécessité de standardiser et normaliser

En l'absence de standardisation commune, les équipementiers mettent en pratique leurs propres standards de développement, ce qui contraint les interactions entre les acteurs, lorsque l'on doit intégrer des composants logiciels provenant de différents équipementiers. De plus, les industriels cherchent à favoriser la réutilisation du code des applications (déjà éprouvés et certifiés). Il est alors nécessaire d'instaurer un langage d'échange qui soit commun entre l'OEM et les Tiers et de disposer d'une spécification d'architecture logicielle commune avec des APIs standards.

Tout comme le secteur de l'avionique l'a fait au travers du standard IMA (ARINC, 1997), les constructeurs automobiles ont souhaité mettre au point un standard qui spécifie la manière dont doit être élaboré le logiciel. AUTOSAR (*AUTomotive Open System ARchitecture*) (AUTOSAR, 2013) a été créé en 2003. Historiquement, ce standard se base sur les fondements imposés par l'association de OSEK (Systèmes ouverts et interfaces correspondantes pour l'électronique des véhicules automobiles – constructeurs allemands) et VDX (*Vehicle Distributed eXecutive* – constructeurs français), apparue au début des années 1990, pour former OSEK/VDX (OSEK/VDX, 2005b).

Les systèmes embarqués d'aujourd'hui sont multicritiques, c'est-à-dire qu'ils regroupent des fonctionnalités *critiques* et *non-critiques*. Une fonctionnalité est critique dès lors qu'une défaillance conduit à des évènements indésirables graves (e.g. un accident). La multiplication des intermédiaires et la répartition du travail entre les équipementiers imposent de mettre en place des processus de conception fiables, de tracer et de documenter pour justifier de la sûreté de fonctionnement à toutes les étapes du développement (Sangiovanni-Vincentelli *et al.*, 2007). De plus, les composants logiciels livrés par les équipementiers sont très souvent des *boîtes noires*, c'est-à-dire des briques logicielles pour lesquelles le code source n'est pas accessible. Il faut alors que la démarche de qualification et de validation mise en place par les équipementiers soit compréhensible et admise par le constructeur.

Tour comme le secteur de l'avionique a créé sa propre norme de sûreté de fonctionnement DO 178 (DO-178B, 1992), l'industrie automobile possède aujourd'hui sa propre norme intitulée ISO 26262 (ISO 26262, 2011). Pour favoriser les échanges et s'assurer que tout est mis en œuvre pour assurer la sûreté de fonctionnement des systèmes, elle spécifie pour chaque étape du processus de développement en V, l'ensemble des exigences produit/processus à remplir.

1.3.2 AUTOSAR : le standard pour l'architecture logicielle

AUTOSAR est maintenu par un consortium qui regroupe des constructeurs, des équipementiers, des fournisseurs de logiciels et des sociétés de service. Les *core partners* (Bosh, BMW Group, Continental, Daimler Chrysler Ford, Opel, PSA Peugeot Citroën, Toyota et Volkswagen AG) pilotent le consortium. Une centaine d'autres partenaires appelés *premium members*, dont Renault fait partie, peuvent également participer à l'élaboration des spécifications des

différents modules de l'architecture logicielle. Il existe enfin des *associated member* pouvant utiliser le standard.

AUTOSAR favorise la réutilisation de la partie applicative du logiciel en réalisant l'indépendance entre les *parties applicatives* liées aux fonctions et aux prestations et les *parties services* liées au matériel (e.g. mémoire) et à l'infrastructure embarquée (e.g. réseau CAN). Le portage d'une application peut s'effectuer sans modifications de celle-ci et indépendamment d'une cible. Cette possibilité rend possible l'utilisation de composants sur étagère COTS (*Commercial/Component Off The Shelf*).

1.3.2.1 Architecture logicielle AUTOSAR

AUTOSAR facilite le multiplexage de fonctions sur des calculateurs génériques. Le standard propose une architecture du logiciel à l'aide des trois couches suivantes (voir Figure 1.5) :

- La couche applicative est subdivisée en SWC (*SoftWare Component*). Ce sont des composants logiciels qui sont eux-mêmes constitués de *runnables*, qui contiennent l'ensemble du code réalisant les prestations ;
- Les couches basses BSW (*Basic SoftWare*) peuvent être subdivisées en deux sous-niveaux. Le premier correspond à la MCAL (*MicroControleur Abstraction Layer*), fournis par le fondeur. C'est un ensemble de modules contenant les pilotes spécifiques de la cible. Le second s'interface avec la MCAL et permet de réaliser les fonctionnalités des couches basses (e.g. écriture/lecture d'une donnée en mémoire, envoi/réception d'un message interne ou externe). Le noyau AUTOSAR OS est un composant du BSW. Il est dérivé du noyau OSEK/VDX et permet notamment de gérer l'ordonnancement des tâches ou la prise en compte des ISRs.
- Le RTE (*RunTime Environnment*) est un code glue qui permet d'interconnecter l'appel des services requis par l'application, avec les services correspondants dans le BSW. Ces appels peuvent passer par le système d'exploitation AUTOSAR OS. Le RTE est le point de passage obligatoire de toutes les interactions entre l'application et les couches basses ainsi que toutes les interactions entre les composants de l'application.

1.3.2.2 Configuration statique du logiciel dans AUTOSAR

AUTOSAR offre une stratégie de développement basée sur la configuration statique, c'est-à-dire que toutes les caractéristiques du logiciel doivent être connues par avance.

Les couches configurables correspondent aux modules BSW et MCAL. Dans la pratique, chaque module possède un ensemble de fichiers sources génériques (les APIs) et un ensemble de fichiers de configurations, spécifiques à l'application. La configuration est réalisée dans le langage ARXML (*AutosaR XML*). Elle comprend toutes les spécificités de l'application. Par exemple pour l'OS, nous spécifions le nombre de tâches, les alarmes, les ISRs, etc. et pour la communication, nous spécifions les messages internes, les signaux CAN, etc. Un générateur de code est ensuite utilisé pour synthétiser les modules BSW et MCAL spécifiques à l'application. Pour plus de lisibilité lors de l'utilisation du langage, une surcouche métier peut être utilisée (e.g. Autosar Builder, SystemDesk). L'ensemble des fichiers est ensuite compilé et lié pour former le binaire.

Cette stratégie de développement permet de limiter le temps de codage, mais il reste cependant toujours à la charge de l'intégrateur de vérifier la cohérence globale du système.

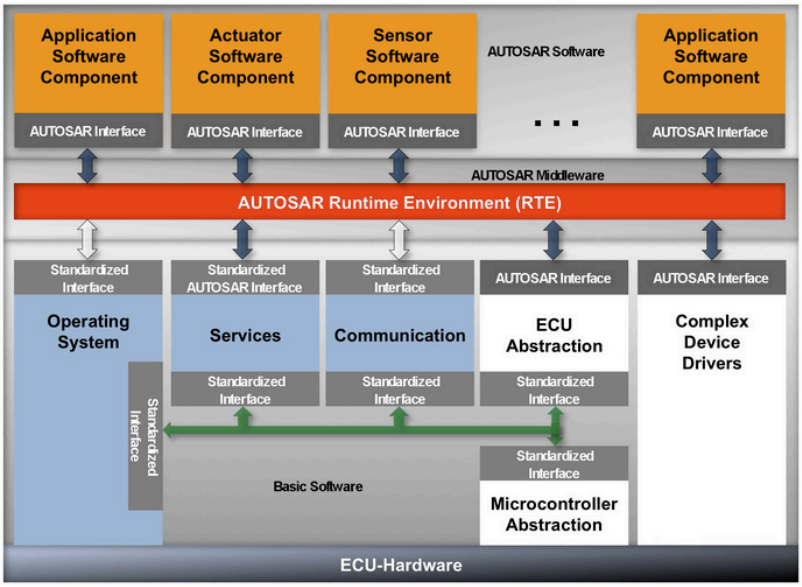


Figure 1.5 – Architecture logicielle (AUTOSAR, 2013)

1.3.2.3 Méthode de développement d'une application AUTOSAR

AUTOSAR propose également une méthode pour décrire l'architecture logicielle d'une application. Elle est résumée sur la Figure 1.6.

La méthode est la suivante : (1) les SWC sont décrits ; chaque SWC est associé à une fonctionnalité du système ; (2) les SWC sont intégrés, c'est-à-dire qu'ils sont interconnectés via le VFB (*Virtual Fonctional Bus*). Le VFB permet la description abstraite de toutes les interconnexions entre les runnables, indépendamment du déploiement qui sera réalisé ; (3) les ECUs sont décrits. La description de haut niveau effectuée jusqu'à présent est indépendante de déploiement des SWC sur les ECUs ; (4) toutes les contraintes liées au système sont décrites (e.g. environnement réseau) ; (5) les SWCs sont alloués aux ECUs ; enfin, en (6), la configuration statique du BSW est effectuée et le RTE est généré.

1.3.2.4 AUTOSAR pour les architectures multicœur

AUTOSAR est applicable aux architectures multicœur depuis la révision 4.0 du standard (AUTOSAR, 2013b). Les architectures matérielles visées sont à mémoire partagée. La mémoire peut éventuellement être partitionnée entre les cœurs. Mis à part l'OS, les BSW sont alloués à un cœur, c'est-à-dire que l'appel d'un service du BSW sera toujours traité par le même cœur, peu importe la provenance de l'appel. Les cœurs sont hiérarchisés en un maître et des esclaves (le nombre de cœurs est connu au départ). Au démarrage, le cœur maître démarre normalement et donne l'ordre aux autres de démarrer (au sens logiciel). Le multicœur exploite également la notion d'OS-Application. Une OS-Application regroupe un ensemble d'entités de l'OS (e.g. tâches, alarmes). Les objets d'une même OS-Application sont systématiquement allouées statiquement sur le même cœur (mais il peut y avoir plusieurs OS-Application sur un même cœur). Enfin, le partage des ressources est adapté aux ressources intercœurs. Les ressources d'un même OS-Application sont partagées via le protocole iPCP (Sha *et al.*, 1990)

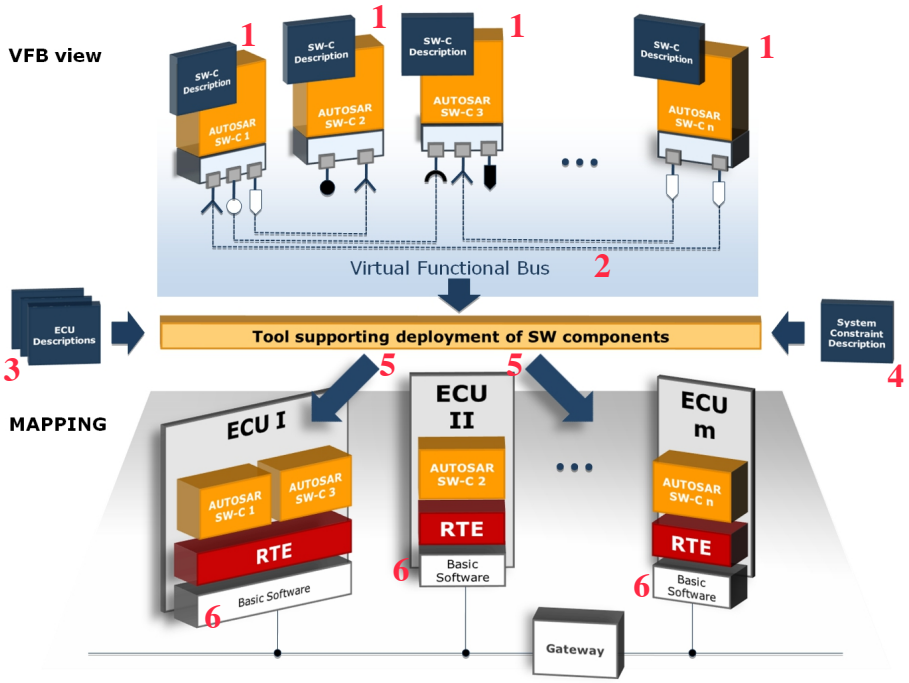


Figure 1.6 – Implantation d'AUTOSAR (AUTOSAR, 2013)

tandis que les ressources intercœurs sont protégées à l'aide de verrous de type spinlock.

On ajoute une couche logicielle supplémentaire dans l'OS, appelée IOC (*Inter-OS Application Communication*), permettant de gérer les communications entre différentes OS-Application.

1.3.3 De la CEI 61508 à l'ISO 26262, la norme de référence pour la sûreté de fonctionnement automobile

L'élaboration de normes pour le secteur du nucléaire a été amorcée dès le début des années 1980 par l'élaboration de *safety guides* (e.g. 50-SG-D3 en 1981 et 50-SG-D8 en 1984). Dans les années 1980, le secteur aéronautique réfléchit à son tour à l'élaboration de normes destinées à fixer les conditions de sécurité des logiciels critiques dans les avions (e.g. ED-12/DO-178 (1981), ED-12A/DO-178A(1985) et ED-12B/DO-178B (1992) (ED-12B, 1999; DO-178B, 1992)).

Les normes de sûreté de fonctionnement destinées aux autres types de systèmes embarqués apparaissent une dizaine d'années plus tard avec la norme CEI 61508 (CEI 61508, 1999). Elle fut développée en version européenne par le CENELEC (*European Committee for Electrotechnical Standardization*) en 2002 sous la dénomination EN 61508. La CEI 61508 propose une approche générique pour tous les domaines d'ingénierie. Elle se focalise sur l'ensemble du processus de développement du produit pour définir et décrire les différentes étapes à respecter pour assurer la sûreté de fonctionnement. En particulier, elle définit les objectifs et les exigences à atteindre pour la spécification, le développement, la mise en œuvre et l'évaluation des systèmes Électriques/Électroniques/Électroniques Programmables (E/E/EP).

La norme vise l'atteinte d'un niveau de confiance qui caractérise l'aptitude du système à remplir ses fonctions correctement. Elle définit cinq *niveaux de criticité* appelés SIL (*Safety*

Integrity Level) de 0 à 4, de criticités croissantes. La criticité d'un système correspond à la sévérité d'une défaillance de ce système. Ce niveau est défini selon quatre paramètres : *les conséquences sur les personnes, la fréquence d'occurrence de l'incident, la contrôlabilité du système après incident et la probabilité d'occurrence.* Le tableau 1.1 explicite les probabilités de défaillances maximales acceptables (par heure ou sur demande) selon le niveau de SIL considéré. Plus le niveau de SIL est élevé, plus la probabilité de défaillance doit être faible et ceci ne peut être réalisable que si les moyens d'élimination, de prévision et de tolérance aux fautes sont correctement mis en place. On considère alors que plus le niveau de SIL est élevé, plus on pourra accorder notre confiance au service délivré par l'application.

Niveau de SIL	**Probabilité de défaillance par heure (PFH)**	**Probabilité de défaillance sur demande (PFD)**
4	$10^{-9} < PFH \leq 10^{-8}$	$10^{-5} < PFD \leq 10^{-4}$
3	$10^{-8} < PFH \leq 10^{-7}$	$10^{-4} < PFD \leq 10^{-3}$
2	$10^{-7} < PFH \leq 10^{-6}$	$10^{-3} < PFD \leq 10^{-2}$
1	$10^{-6} < PFH \leq 10^{-5}$	$10^{-2} < PFD \leq 10^{-1}$
0	Pas d'exigences particulières	

Table 1.1 – Classification quantitative des niveaux de criticité SIL CEI 61508 (1999)

La CEI 61508 est générique et son utilisation peut alors se révéler difficile. C'est pourquoi elle représente une norme *mère* pour toute une série d'autres normes *filles*, dont les attributions sont fixées pour un secteur donné.

Pour le domaine automobile, c'est le dérivé ISO 26262 qui est utilisé depuis 2011. Cette norme a pour objectif de proposer des méthodes pour l'atteinte de la sécurité durant toutes les phases du processus de développement du véhicule. Elle spécifie les bonnes pratiques en matière de documentation, d'interaction entre les acteurs et des moyens mis en place pour justifier de la sûreté de fonctionnement des systèmes conçus en interne ou sous-traités aux équipementiers. L'ensemble des acteurs (OEM et Tiers) est donc lié par cette norme, ce qui facilite les échanges.

La norme définit quatre niveaux de criticité croissants que l'on nomme ASIL (*Automotive SIL*) : de ASIL A à ASIL D (le plus critique). Un cinquième niveau, noté QM (*Quality Management*) n'est associé à aucune exigence particulière. Lors de l'étude du système et de l'attribution de ces niveaux de criticité, trois paramètres sont à prendre en considération (c.f. Table 1.2) : la sévérité qui se base sur le nombre de blessés ou de tués par l'incident ou l'accident (*S1 : pas de blessés, S2 : faiblement blessé, S3 : blessé grave ou décès*) ; la fréquence d'occurrence : *E1 : évènement rare, E2 : quelquefois, E3 : assez souvent, E4 : souvent* ; et la contrôlabilité. Ce dernier paramètre est une notion subjective qui se base sur les aptitudes du conducteur (*C1 : contrôlable, C2 : normalement contrôlable, C3 : incontrôlable*).

Cette norme n'est pas une procédure à suivre strictement. La norme recommande (sans imposer, car ce n'est pas une réglementation) des méthodes ou des mécanismes pour respecter les exigences. Les recommandations sont associées à chaque étape du processus de développement en V, en fonction du niveau d'ASIL ciblé. Certains constructeurs peuvent décider de mettre en pratique toutes les recommandations comme d'autres peuvent appliquer des méthodes différentes. Dans ce dernier cas, une preuve supplémentaire est à fournir pour justifier que les techniques employées répondent bien aux exigences. Au final, elle impose au fournisseur

Sévérité	Fréquence	Contrôlabilité		
		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	ASIL A
	E4	QM	ASIL A	ASIL B
S2	E1	QM	QM	QM
	E2	QM	QM	ASIL A
	E3	QM	ASIL A	ASIL B
	E4	ASIL A	ASIL B	ASIL C
S3	E1	QM	QM	ASIL A
	E2	QM	ASIL A	ASIL B
	E3	ASIL A	ASIL B	ASIL C
	E4	ASIL B	ASIL C	ASIL D

Table 1.2 – Définition des niveaux de criticité ASIL ISO 26262 (2011)

d'un système d'accompagner la réalisation d'un système par la réalisation d'un ensemble de documentations permettant de justifier de sa sûreté de fonctionnement.

1.4 Tendances de développement des systèmes E/E

1.4.1 Accroissement des besoins en ressources CPUs

Avec l'émergence des systèmes mécatroniques, de plus en plus de fonctions mécaniques sont contrôlées par une assistance électronique, nécessitant du logiciel. Cette approche permet de mettre en place des produits plus compétitifs, moins encombrants et plus performants.

Le modèle économique du secteur automobile pousse sans cesse à l'ajout de prestations supplémentaires dans les véhicules haut de gamme. Par exemple, en plus d'utiliser des systèmes embarqués pour contrôler les organes vitaux du véhicule (contrôle moteur, systèmes d'aide au freinage, etc.), on retrouve des services annexes destinés à accroître le confort de l'utilisateur (systèmes de navigation, systèmes multimédias, etc.).

Pour respecter les exigences en termes de sûreté de fonctionnement imposées par les normes, il est nécessaire d'ajouter des mécanismes permettant de garantir le bon fonctionnement pendant l'exécution d'une application. Par exemple, les mécanismes de contrôle de vraisemblance, la réplication des fonctionnalités, etc. engendrent une augmentation du besoin en ressources CPU et mémoire requis sur les calculateurs.

Enfin, le logiciel prend une part importante dans l'optimisation et la réduction de l'empreinte environnementale des véhicules. Par exemple, le domaine automobile est contraint par des normes environnementales de type $EURO$, pour les émissions de CO_2 des moteurs thermiques. L'introduction des véhicules hybrides ou tout électriques conduit également à une utilisation massive de l'électronique.

La mouvance actuelle requiert donc d'accroître toujours plus la quantité de ressources disponibles sur les ECUs. Pour répondre à ce besoin, trois possibilités peuvent être étudiées :

- Profiter de l'amélioration des performances offertes par les architectures monocœur ;
- Accroître le nombre d'ECUs basés sur des architectures monocœur ;

– S'orienter vers de nouvelles architectures permettant, de par leur nature, d'accroître les performances : les architectures multicœur.

1.4.2 Limitations de l'évolution des architectures monocœur

L'augmentation des performances nécessite un accroissement de la puissance du CPU (i.e. augmentation du nombre d'instructions exécutées par unité de temps). Pour cela, les fondeurs agissent sur deux facteurs : la miniaturisation des transistors (i.e. une augmentation de la densité de transistors sur la puce) et l'augmentation de la fréquence.

Les processeurs ont bénéficié au cours des 40 dernières années, de la croissance considérable des performances décrites par Moore (1965) au milieu des années 1960 : *tous les deux ans, le nombre de transistors embarqués dans une puce peut doubler*. Cette loi s'est révélée exacte durant de nombreuses années, mais de nos jours les évolutions, bien que toujours croissantes, tendent à s'en éloigner. De plus, la loi de Moore tend inexorablement à devenir caduque puisque les architectures actuelles sont déjà gravées à quelques dizaines de nanomètres (90/65 nm en 2013). En effet, nous nous rapprochons d'un mur physique qui ne pourrait être surmonté qu'en considérant l'empilement des transistors ou par la généralisation de l'informatique quantique. La miniaturisation des transistors implique également une plus forte consommation puisque les courants de fuite deviennent de plus en plus importants (en-deçà de 100 nm, ces courants de fuites commencent à devenir problématiques). Une solution consiste à réduire la tension d'alimentation de la puce (beaucoup de composants sont maintenant alimentés en 3,3 V). On réduit donc l'écart entre les niveaux logiques 1 et 0, ce qui favorise les bit-flip.

Une autre source d'évolution des architectures monocœur consiste à travailler sur la fréquence de fonctionnement, ce qui permet d'exécuter plus d'instructions en un temps donné. Malheureusement, le gain de fréquence est aussi la source de nouveaux problèmes qu'il faut pouvoir gérer. En effet, la puissance d'un processeur est une fonction de la fréquence de fonctionnement et de la tension d'alimentation ($P(W) = k * C * v^2 * f$ avec C, la capacité (commutation des transistors), v, la tension d'alimentation et f, la fréquence de fonctionnement du processeur) (Freescale, 2009). Il résulte donc que plus la fréquence du cœur est élevée plus il dissipera de chaleur (f augmente $\Rightarrow$ v et P augmentent). Dans le monde de l'informatique embarqué, les contraintes thermiques sont très importantes, non seulement pour le bon fonctionnement du composant, mais aussi pour maîtriser sa durée de vie. Par exemple l'automobile opère sur une plage de températures allant de -40 à 160 degrés Celsius. Si on augmente la fréquence des processeurs, il faudra alors mettre en œuvre des solutions pour la dissipation de la chaleur, qui peuvent être coûteuses (e.g. en termes de prix de revient, de place nécessaire, de poids de l'infrastructure).

S'appuyer exclusivement sur l'amélioration des architectures actuelles rendra donc difficile l'atteinte des objectifs en termes de performance. Une autre solution est d'augmenter le nombre d'ECUs dans les véhicules. Cette solution permet de favoriser la répartition du logiciel dans différents emplacements physiques de la voiture. L'implantation des mécanismes pour la tolérance aux fautes peut alors bénéficier du fait que les réplicats ne soient pas soumis aux mêmes variations de l'environnement. En revanche, l'augmentation du nombre de fonctionnalités conduit à un accroissement de la complexité du logiciel, aussi bien pour la mise en place de prestations complexes nécessitant l'intervention d'un grand nombre de calculateurs, que pour la maîtrise du comportement du réseau (accroissement du nombre de nœuds sur les bus CAN et/ou LIN). Enfin, le poids des véhicules est un paramètre critique, qu'il faut minimiser

pour réduire la consommation de carburant. Dans les véhicules haut de gamme qui intègrent de nombreuses fonctionnalités, le poids de l'électronique peut devenir important.

Cette solution n'est donc pas considérée comme viable à long terme et il faut évoluer dans la manière de penser l'architecture. Pour cela, les fondeurs proposent des solutions multicœur pour les systèmes embarqués.

1.4.3 L'étude des architectures multicœur

Les architectures multicœur disposent d'au moins deux unités de calculs qui permettent d'exécuter du code en parallèle sur une même puce. Les fondeurs comme Freescale ou Infineon proposent des architectures de 2 à 5 cœurs permettant d'exécuter du code selon deux modes : le mode *lockstep* ou le mode *découplé*.

Le *mode lockstep* consiste à exécuter les mêmes instructions sur tous les cœurs puis à comparer les résultats produits. Il permet la sûreté de fonctionnement de fonctions critique (e.g. ASIL D), en offrant une bonne tolérance aux fautes matérielles transitoires. En revanche, une faute permanente dans le code serait dupliquée et ne serait donc pas tolérée.

Le *mode découplé* permet d'exécuter des instructions différentes sur les cœurs et donc, d'exécuter des tâches en parallèle. Il s'agit d'un mode d'utilisation permettant d'exploiter les nouvelles ressources disponibles. Ces ressources peuvent être utilisées à des fins de gain de performance et/ou pour implanter des mécanismes assurant la sûreté de fonctionnement. Notons cependant que doubler le nombre de cœurs ne permet pas de doubler les performances du CPU puiqu'il faut prendre en compte le taux de code parallélisable (Amdahl, 1967). Rappelons que notre objectif n'est pas d'accélérer un logiciel, mais de tirer profit de la capacité de calcul supplémentaire offerte par l'ajout de cœurs.

De plus, la performance du processeur est directement proportionnelle à la racine carrée de sa surface (évalué par la densité de transistors) tandis que sa consommation est directement proportionnelle au carré de la surface ($P_{perf} = K \times \sqrt{surface}$ et $P_{conso} = K \times surface^2$) (Toshinori et Funaki, 2008). Une faible augmentation de la surface conduit alors à une faible augmentation de la puissance mais à une forte augmentation de la consommation. À surface identique, il semble intéressant d'ajouter des cœurs, sans augmenter la fréquence de calcul.

1.4.4 Conclusions sur les besoins industriels

Les tendances de développement des architectures E/E contraignent aujourd'hui les constructeurs automobiles à augmenter la part de l'électronique dans les véhicules. Il devient alors nécessaire d'augmenter la quantité de ressources CPU disponibles pour les calculs. Parmi les axes d'études possibles, l'étude des architectures matérielles multicœur est une piste qui semble justifiée. Ce changement de concept et le manque d'expérience pour leur utilisation imposent d'adapter les processus de développement existants. Mais *comment développer une application sûre de fonctionnement sur ce type d'architecture ?*

Pour répondre à cette question, nous devons d'abord prendre conscience des difficultés introduites pour développer une application sur ce type d'architecture (i.e. parallélisme et ressources partagées). Nous chercherons à cibler les problèmes inhérents à ces architectures (matériel et logiciel) pour en déduire le modèle de fautes. Enfin, nous cherchons à proposer des solutions compatibles AUTOSAR, permettant le respect des exigences de la norme ISO 26262.

Chapitre 2

Contexte scientifique

Sommaire

2.1 Taxonomie des architectures matérielles multicœur

2.1.1 Les architectures homogènes, uniformes et hétérogènes

Trois types d'architectures matérielles existent : les architectures homogènes, uniformes et hétérogènes.

Définition 2.1. *Une architecture est homogène si tous les cœurs présents sur la puce sont identiques, utilisant le même jeu d'instructions et offrant la même capacité de calcul. Ce type d'architecture peut être noté SMP (Symetric MultiProcessing).*

Définition 2.2. *Dans une architecture uniforme, tous les cœurs utilisent le même jeu d'instructions, mais chaque cœur est caractérisé par sa propre capacité de calcul. Ce type d'architecture peut être noté UMP (Uniform MultiProcessing).*

Balakrishnan *et al.* (2005); Wolf *et al.* (2008) proposent une étude comparative de différentes asymétries liées aux capacités de calcul de chacun des cœurs, selon les critères *d'extensibilité* (i.e. quel comportement est observé si l'on ajoute des cœurs ?) et de *répétitivité* des exécutions (i.e. a-t-on toujours le même comportement ?). L'étude repose sur un benchmark incluant de nombreuses études de cas issues du monde informatique (réseaux, traitement d'image, etc.). On observe que des architectures fortement asymétriques sont adéquates pour des applications du type *traitement de l'image* tandis que les architectures homogènes sont adaptées pour des applications du type *réseaux*.

Définition 2.3. *Une architecture est hétérogène si les cœurs sont indépendants et ne possèdent pas le même jeu d'instructions. Une capacité de calcul peut donc être assignée à chacun des cœurs. Ce type d'architecture peut être noté AMP (Asymetric MultiProcessing).*

En particulier, ces dernières architectures sont adaptées aux cas pour lesquels on dispose de cœurs spécialisés, ne pouvant réaliser que certains traitements.

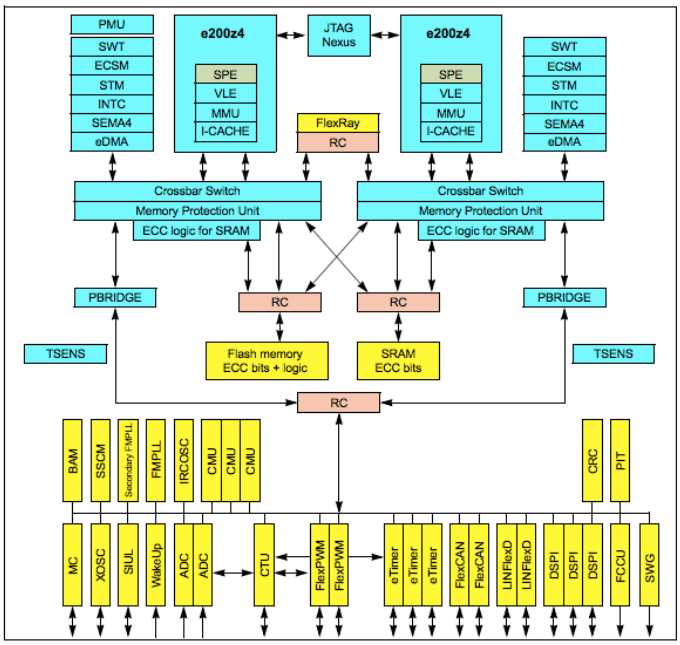
Enfin, on distingue aussi les architectures multicœur sans mémoire commune des architectures possédant une mémoire commune. Dans le premier cas, les interactions entre les cœurs sont réduites (i.e. pas de contentions sur le bus d'accès à la mémoire, pas de mémoires cache partagées). Dans le second cas, des interactions entre les cœurs existent et il est nécessaire de les prendre en considération lors de la conception d'une application.

2.1.2 Un exemple d'architecture : le Leopard MPC 5643L

Le microcontrôleur Leopard MPC 5643L (Freescale, 2010) est un exemple d'architecture homogène à mémoire commune. Celle-ci est représentée sur la Figure 2.1.

Le Leopard est composé de deux cœurs e200z4 (c.f. classe d'architecture des microcontrôleurs Freescale) de fréquence maximale 120 MHz. Chaque cœur possède son propre cache d'instructions. Il n'y a pas de cache de données. La mémoire est partagée et chaque cœur possède des entités dédiées à sa gestion (*Memory Management Unit* – MMU, 16 pages) et à sa protection (*Memory Protection Unit* – MPU, 16 zones par cœur). Il y a 1 MB de mémoire Flash (code et données) et 128 kB en SRAM. Des mécanismes de vérification de la SRAM à base de ECC (*Error Checking and Correcting*) sont présents. Les périphériques d'entrées/sorties (PWM, CAN, LIN) sont partagés.

Le Leopard peut fonctionner en lockstep ou en mode découplé.



ADC – Analog-to-Digital Converter
BAM – Boot Assist Module
CMU – Clock Monitoring Unit
CRC – Cyclic Redundancy Check unit
CTU – Cross Triggering Unit
DSPI – Serial Peripherals Interface
ECC – Error Correction Code
ECSM – Error Correction Status Module
eDMA – Enhanced Direct Memory Access controller
FCCU – Fault Collection and Control Unit
FlexCAN – Controller Area Network controller
FMPLL – Frequency Modulated Phase Locked Loop
INTC – Interrupt Controller
IRCOSC – Internal RC Oscillator
JTAG – Joint Test Action Group interface
LINFlexD – LIN controller with DMA support
MC – Mode Entry, Clock, Reset, & Power
PBRIDGE – Peripheral bridge
PIT – Periodic Interrupt Timer
PMU – Power Management Unit
RC – Redundancy Checker
RTC – Real Time Clock
SEMA4 – Semaphore Unit
SIUL – System Integration Unit Lite
SSCM – System Status and Configuration Module
STM – System Timer Module
SWG – Sine Wave Generator
SWT – Software Watchdog Timer
TSENS – Temperature Sensor
XOSC – Crystal Oscillator

Figure 2.1 – Architecture du Leopard (Freescale, 2010)

2.2 Complexité des systèmes temps réel multicœur

2.2.1 Ordonnancement temps réel

Un système embarqué peut être défini comme un système électronique et informatique autonome dédié à l'exécution de tâches. Ce type de système dispose souvent de ressources limitées (taille de la mémoire, fréquence du CPU). Les systèmes temps réel appartiennent à la classe des systèmes réactifs, souvent critiques, dont le comportement est assujetti à l'évolution dynamique du procédé qu'ils doivent piloter. Un système temps réel doit réagir à tous les changements d'état du procédé, dans une fenêtre temporelle adéquate vis-à-vis des contraintes de l'environnement dans lesquelles il est intégré.

Dans un système multitâches, l'ordonnanceur temps réel a pour rôle de choisir à tout instant la tâche qui doit s'exécuter parmi l'ensemble de celles qui sont prêtes. Ce choix repose sur la mise en œuvre d'une stratégie d'ordonnancement (politique d'ordonnancement). De nombreuses politiques d'ordonnancement existent, chacune ayant une influence directe sur le comportement du logiciel, tant du point de vue des performances que de sa réactivité. Parmi les classes de systèmes temps réel, nous ciblons le temps réel dur, c'est-à-dire que tous les traitements doivent être assurés en respectant les contraintes temporelles.

2.2.1.1 Modèles de tâches temps réel

L'ordonnancement temps réel appuie ses décisions sur un modèle de tâches (Figure 2.2). Une tâche τ_i est caractérisée par les paramètres suivants :

– L'instant de réveil r_i, c'est-à-dire la date à laquelle elle peut commencer son exécution. L'instant où elle démarre réellement peut être différent ;

– Le pire temps d'exécution C_i, c'est-à-dire la durée maximale que prendra τ_i pour s'exécuter. On parle aussi de WCET – *Worst Case Execution Time* ;
– Si τ_i est périodique, T_i définit l'intervalle de temps séparant deux réveils successif de τ_i ;
– Le délais critique D_i définit l'intervalle de temps dont τ_i dispose pour s'exécuter complètement. Si $D_i = T_i$, on parle d'échéance sur requête. L'échéance absolue d_i définit l'instant auquel τ_i devra avoir terminé son exécution ($d_i = r_i + D_i$) ;

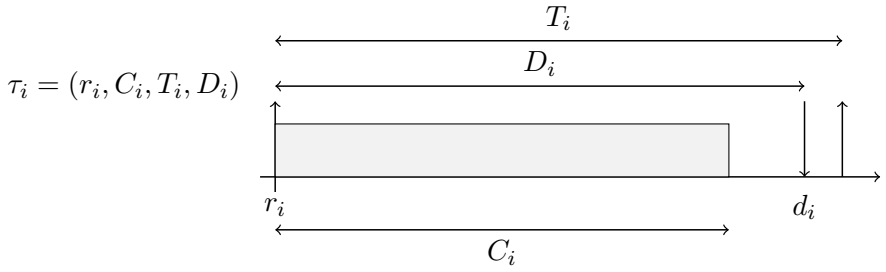


Figure 2.2 – Modèle de tâche

D'autres paramètres peuvent également être définis :
– La laxité de τ_i peut être définie à chaque instant comme la différence entre l'instant présent et l'échéance absolue, à laquelle est retranché le temps d'exécution restant ;
– Le temps de réponse d'une tâche τ_i correspond à l'intervalle de temps qui sépare son réveil et la fin de son exécution.

2.2.1.2 Rappels sur les politiques d'ordonnancement monocœur

Historiquement, toutes les tâches qui devaient s'exécuter dans un système monocœur étaient lancées de manière cyclique. Le modèle d'application n'était donc pas un modèle de tâches concurrentes comme on peut le concevoir maintenant, mais plutôt un modèle avec un séquenceur chargé d'enchaîner l'exécution des fonctions (les routines d'interruption s'exécutant par ailleurs au rythme des interruptions). Ce modèle d'application et le modèle d'exécution qui en découle étaient avant tout guidés par un souci de maîtrise des exécutions, notamment dans les systèmes présentant un certain niveau de criticité. Au fur et à mesure de l'adoption du modèle multitâche conventionnel, des politiques d'ordonnancement plus poussées ont émergé, sur la base des travaux de Liu et Layland. Le modèle d'application, sur lequel des résultats analytiques sont connus, est constitué de tâches périodiques (activation à intervalles de temps réguliers), sporadiques (la durée minimale séparant deux activations consécutives de la tâche est alors connue) ou apériodiques (leur date d'activation n'est pas connue *a priori*). On distingue les ordonnancements basés sur des priorités fixes et ceux basés sur des priorités dynamiques. Parmi les politiques d'attribution des priorités pour les systèmes à priorités fixes, nous pouvons citer *Rate Monotonic* (RM) ou *Deadline Monotonic* (DM) (Liu et Layland, 1973). Parmi les politiques d'attribution des priorités pour les systèmes à priorités dynamiques, nous pouvons citer *Earliest Deadline First* (EDF) (Liu et Layland, 1973) ou *Least Laxity First* (LLF) (Leung et Whitehead, 1982).

L'algorithme RM consiste à assigner les priorités dans l'ordre inverse des périodes. L'algorithme DM consiste à assigner les priorités dans l'ordre inverse des échéances relatives. L'algorithme EDF consiste à assigner les priorités dans l'ordre inverse des échéances absolues.

L'algorithme LLF consiste à attribuer les priorités dans l'ordre inverse des laxités.

L'implantation d'un ensemble de tâches sur un cœur, tout en exploitant au maximum les ressources mises à disposition par la puce, nécessite qu'un test d'ordonnançabilité soit mené hors-ligne. Ce test permet de déterminer si l'implantation considérée permet au système de respecter l'ensemble de ses contraintes temporelles. Le résultat d'une telle analyse dépend d'une multitude de facteurs (politique d'ordonnancement et caractéristiques des tâches). Les modèles pour lesquels des tests d'ordonnançabilité existent sont souvent assez restrictifs : tâches indépendantes (pas de synchronisation intertâches), synchrones (dates d'activations initiales identiques), à échéances contraintes (l'échéance relative est inférieure à la période). Selon le cas, on dispose de Conditions Suffisantes (CS) ou de Conditions Nécessaires (CN) d'ordonnançabilité (e.g. RM ou DM) ou dans le meilleur des cas, de Conditions Nécessaires et Suffisantes, dites Exactes (e.g. EDF). On dispose également de preuves d'optimalité. Ainsi, RM est optimal pour la classe des systèmes de tâches périodiques ou sporadiques, indépendantes, synchrones, à échéances sur requêtes, avec un ordonnancement préemptif non-oisif (l'ordonnanceur fonctionne sans insertion de temps creux) et à priorités fixes. EDF est optimale pour la classe des systèmes à ordonnancement non-oisif. Enfin, des techniques d'analyse plus complexes existent pour les systèmes qui ne répondent pas aux hypothèses ci-dessus. Si l'on s'intéresse au domaine automobile, on trouvera dans Hladik *et al.* (2007) des algorithmes d'analyse adaptés aux systèmes AUTOSAR.

40 ans de travaux ont permis d'aboutir aujourd'hui à un domaine bien cartographié. Un bon panorama peut être trouvé dans Déplanche et Cottet (2006) ainsi que dans Sha *et al.* (2004). Les résultats théoriques sont matures et leur intégration dans les processus industriels est en cours.

2.2.1.3 Généralités sur l'ordonnancement multicœur

Dans le cas des architectures multicœur, on distingue trois classes de politiques d'ordonnancement (Liu, 2000) : les politiques global, partitionné et semi-partitionné.

Définition 2.4. *Avec une politique d'ordonnancement globale, les décisions prises par l'unique ordonnanceur du système permettent l'attribution d'une tâche à n'importe lequel des cœurs disponibles.*

Une politique d'ordonnancement globale permet de raisonner sur l'ensemble du parc de cœurs. Une tâche peut migrer d'un cœur à un autre en fonction des besoins. On distingue la migration à l'échelle de la tâche (i.e. la tâche peut commencer son exécution sur un cœur et poursuivre sur un autre) ou à l'échelle d'une instance (i.e. la tâche ne peut migrer qu'entre deux de ses instances). Un seul ordonnanceur dit global est en charge d'aiguiller les tâches vers les différents cœurs. Les résultats obtenus en monocœur ne sont alors plus valables. En effet, les algorithmes d'ordonnancement se révèlent souvent plus complexes à mettre en œuvre et des surcoûts d'exécution assez importants peuvent apparaître (e.g. coûts des préemptions et/ou des migrations des tâches d'un cœur à l'autre). En 1969, Liu constate que le simple fait qu'une tâche ne puisse utiliser qu'un cœur alors que d'autres sont disponibles ajoute de nombreuses difficultés pour la mise en place de politiques d'ordonnancement optimales en multicœur (Liu, 1969).

Définition 2.5. *La politique d'ordonnancement partitionnée suppose que toutes les tâches sont statiquement allouées sur un cœur. Il y a autant d'ordonnanceurs que de cœurs, chacun agissant sur la liste des tâches en local.*

Dans le cas d'un ordonnancement partitionné, chaque cœur est traité de manière indépendante et possède son propre jeu de tâches géré par un ordonnanceur local monocœur. La difficulté inhérente à cette approche concerne la manière dont les tâches sont assignées aux différents cœurs. Ce problème est NP-complet. Il est donc est traité par des heuristiques de partitionnement (e.g. First-Fit, Best-Fit) reposant sur l'évaluation d'un critère d'ordonnancabilité pour l'allocation des tâches. Dans le cas d'un système temps réel, ce partitionnement est calculé hors-ligne. Une fois l'allocation faite, les politiques d'ordonnancement déjà étudiées dans le cas monocœur peuvent être utilisées indépendamment sur chacun des cœurs (un cœur ordonnance sa propre liste de tâches). Les propriétés établies en monocœur restent alors valables sur chacun des cœurs, bien que le calcul du WCET doive tout de même prendre en compte les interférences.

Définition 2.6. *Avec la politique d'ordonnancement semi-partitionnée, un groupe de tâches est statiquement alloué aux différents cœurs. Les autres tâches peuvent migrer en fonction des choix de l'ordonnanceur.*

La politique d'ordonnancement semi-partitionnée est un compromis entre les politiques globales et partitionnées permettant d'obtenir une bonne utilisation des resources du système tout en générant assez peu de migrations.

Toutes les politiques d'ordonnancement ne s'appliquent pas à chacune des architectures matérielles évoquées. Les associations possibles sont représentées sur la Figure 2.3. Nous constatons que des politiques d'ordonnancement partitionnées sont les plus adaptées pour les architectures matérielles hétérogènes tandis que toutes les politiques d'ordonnancement peuvent s'appliquer sur des architectures matérielles homogènes et uniformes.

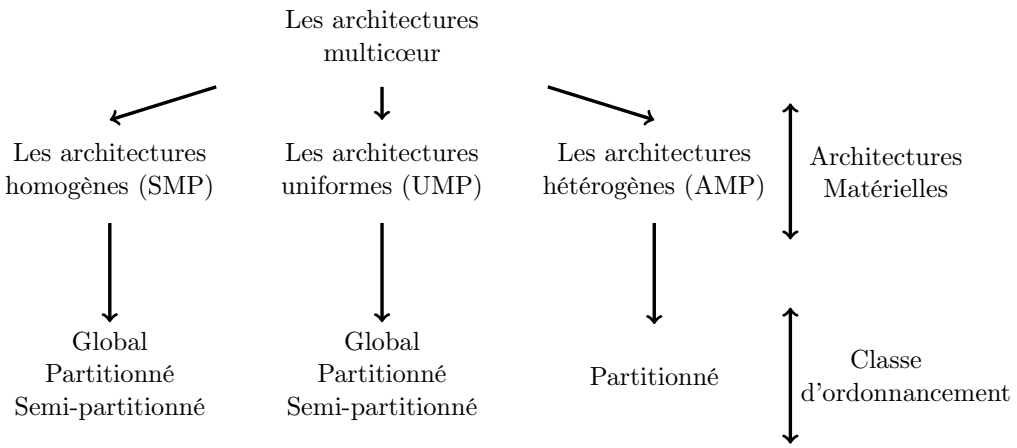


Figure 2.3 – Les Architectures matérielles et logicielles

Historiquement, les travaux de Dhall et Liu à la fin des années 70 ont influencé les axes de

recherche en ordonnancement multiprocesseur, pour les diriger principalement sur des ordonnancements partitionnés. En effet, les résultats de leurs travaux montraient que l'application des politiques d'ordonnancement utilisées en monocœur pour un ordonnancement global peut conduire à de moins bonnes performances qu'en considérant un ordonnancement partitionné. Il s'agit de l'effet Dhall (Dhall et Liu, 1978).

Les algorithmes d'ordonnancement bien maîtrisés en monocœur ont néanmoins été transposés au cas multicœur dans le cas des politiques globales (*GlobalRM* (Andersson *et al.*, 2001), *GlobalEDF* (Srinivasan et Baruah, 2002)), même s'ils perdent alors les bonnes propriétés qu'ils avaient, notamment l'optimalité (sous certaines conditions). Des CS d'ordonnançabilité ont été établies, caractérisées par une borne supérieure du facteur d'utilisation, et les efforts de recherche dans ce domaine ont cherché à repousser cette borne en faisant des hypothèses sur la composition de la configuration de tâches (*RM-US* (Andersson *et al.*, 2001), *EDF-US* (Srinivasan et Baruah, 2002)).

Au cours des années 90, l'ordonnancement global a connu un sérieux regain d'intérêt, notamment par le résultat d'une autre catégorie de travaux, consistant à adopter le principe de « fairness », visant à établir une certaine équité dans l'allocation du temps processeur aux différentes tâches. C'est l'algorithme PFair proposé par Baruah (Baruah *et al.*, 1996) et ses dérivés basés sur le modèle de *Deadline Partitionning* (DP-Fair) qui sont les premiers algorithmes optimaux, au sens où ils permettent d'exploiter la capacité complète de calcul d'un système ; la condition d'ordonnançabilité est en effet $U \leq m$ (avec $U = \Sigma \frac{C_i}{T_i}$, le facteur global d'utilisation et m le nombre de cœurs). Néanmoins, ces algorithmes engendrent, de par leur principe même, beaucoup de préemptions et de migrations de tâches. Les efforts de recherche actuels visent à proposer des techniques de réduction des migrations et préemptions. Notons toutefois que dans l'état actuel des connaissances, ces algorithmes optimaux ne le sont que pour des configurations de tâches périodiques et indépendantes.

2.2.1.4 Anomalies d'ordonnancement multicœur sous une politique globale

L'utilisation des architectures multicœur souffre aussi de nombreux problèmes d'interférences et d'anomalies d'ordonnancement. Une anomalie d'ordonnancement survient quand un changement intuitivement positif conduit à des effets potentiellement négatifs sur l'application.

En monocœur et sous certaines conditions, le taux de tâches qui manquent leurs échéances dépend de la charge du système. Cela suppose qu'en augmentant les périodes, les tâches auront plus de chance de respecter leurs échéances. Cette hypothèse peut se révèler fausse en multicœur. En effet, pour ce type d'architecture, augmenter la période d'une tâche de forte priorité peut augmenter les interférences sur les tâches moins prioritaires. De la même manière, augmenter la période d'une tâche peut impacter la tâche elle-même puisque la modification de sa date d'occurrence peut occasionner plus d'interférences.

Enfin, toute analyse d'ordonnancement repose également sur l'estimation des pires temps d'exécutions des tâches (WCET). Cette estimation, déjà très compliquée en monocœur, est largement complexifiée en multicœur. En effet, ce paramètre dépend non seulement des tâches qui vont s'exécuter sur le même cœur, mais aussi des tâches qui vont s'exécuter sur les autres cœurs (accroissement des interférences).

Une mauvaise estimation de ce paramètre peut être très problématique. En effet, une sous-estimation de la pire durée d'exécution conduira à une analyse d'ordonnançabilité non réaliste, qui ne prouvera en rien la conformité du système, vis-à-vis du comportement attendu.

A contrario, une surestimation de ce paramètre peut être la cause d'anomalies d'ordonnancements. En effet, si une tâche (ou une instance de celle-ci) se révèle plus rapide, l'ordonnancement global risque d'en être impacté et sous ces nouvelles conditions, des tâches peuvent éventuellement manquer leurs échéances. L'illustration de ce phénomène a été faite par Ha et Lui (Ha et Liu, 1994).

En pratique une surestimation des pires temps d'exécution peut être due au fait que l'analyse des architectures multicœur conduit à une analyse pessimiste (surévaluation de la charge des cœurs). Par exemple, la contention sur les bus et les caches conduit à supposer au pire-cas qu'une tâche perd toujours l'arbitrage sur le bus et qu'elle ne subit que des défauts de cache (Joseph et Pandya, 1986; Kumar *et al.*, 2005).

2.2.1.5 Ordonnancement temps réel multicœur dans AUTOSAR

La politique d'ordonnancement spécifiée dans la révision 4.0 du standard AUTOSAR est partitionnée. Toutes les tâches sont statiquement et définitivement allouées sur chacun des cœurs. Chaque cœur possède son propre ordonnanceur à priorité fixe. L'intégrateur est en charge de définir la stratégie de partitionnement la mieux adaptée à l'application qu'il faut mettre en place.

2.2.2 Partage des ressources

La gestion de l'accès aux données partagées repose sur la mise en place de protocoles de contrôle de concurrence permettant d'assurer la cohérence des données sans impacter la sûreté (pas d'interblocage), la progression (pas de famine), ou encore les contraintes temporelles des tâches (respect des échéances). À l'heure actuelle, plusieurs protocoles de synchronisation temps réel ont été définis à la fois pour les environnements monocœur et multicœur.

2.2.2.1 Protocoles de synchronisation monocœur

Les protocoles de synchronisation monocœur se basent généralement sur l'utilisation de verrous pour prendre des ressources. La tâche possédant la ressource est alors dite *en section critique*, les ressources sont dites en *exclusion mutuelle*. On parle ici de synchronisation bloquante.

Un certain nombre de protocoles de synchronisation temps réel monocœur ont été définis. Ils permettent de résoudre notamment le problème dit « d'inversion de priorité ». L'inversion de priorité survient lorsqu'une tâche de haute priorité, souhaitant utiliser une ressource déjà prise, est retardée par une tâche moins prioritaire, n'utilisant pas la ressource, mais plus prioritaire que celle possédant la ressource à un instant donné. Le protocole PIP (Priority Inheritance Protocol) (Sha *et al.*, 1990) a été proposé pour éviter ce phénomène. Lorsqu'une tâche de haute priorité est bloquée par une tâche de plus faible priorité, la tâche responsable du blocage hérite temporairement de la priorité de la tâche qu'elle bloque. Cependant, PIP ne permet pas d'éviter les interblocages et génère parfois des blocages en chaîne. C'est pourquoi d'autres mécanismes plus efficaces ont été proposés.

Le protocole PCP (Priority Ceiling Protocol) (Goodenough et Sha, 1998; Sha *et al.*, 1990) permet d'éviter les situations d'interblocages. Il suppose que chaque tâche possède une priorité fixe (i.e. politique RM ou DM) et que toutes les ressources utilisées sont connues avant le début de l'exécution. Une priorité plafond est assignée à chaque ressource ($P_{plafond}(R_i)$ est

la priorité plafond associée à la ressource R_i). La valeur de $P_{plafond}(R_i)$ correspond à la plus haute priorité parmi celles des tâches pouvant acquérir R_i. Le plafond de priorité courant du système, noté $P'_{plafond}$, est égal au maximum des plafonds de priorité utilisés si au moins une ressource est utilisée, ou à une priorité quelconque plus basse que toutes les priorités plafonds si aucune ressource n'est utilisée. En pratique, quand la tâche τ_i prend R_i, sa priorité est élevée au maximum des priorités des tâches qu'elle bloque (i.e. les tâches en attente de R_i) et quand elle libère la ressource, elle reprend sa priorité initiale. Pour prendre une ressource, il faut que la ressource soit libre et que $Priorité(\tau_i) > P'_{plafond}$. Si cette dernière condition n'est pas respectée, la ressource peut être prise si $P_{plafond}(R_i) \geq P'_{plafond}$.

Avec le protocole SRP (Stack Resource Policy) (Baker, 1990), le blocage des tâches ne peut avoir lieu qu'au début de leur exécution, lorsque l'on identifie que les ressources qu'elles utilisent ne sont pas disponibles. Le type d'ordonnancement ciblé est une politique à priorité dynamique (e.g. EDF). À chaque tâche τ_i est associé un niveau de préemption P_i (une tâche τ_j ne peut préempter τ_i que si $P_j > P_i$), qui est déterminé en-ligne en fonction de sa priorité (sous EDF, $d_i < d_j \Rightarrow P_i > P_j$). À chaque ressource R_i est associé un plafond de préemption $P_{plafond}(R_i)$, défini comme le maximum des niveaux de préemption des tâches qui peuvent l'acquérir. Enfin, le système possède un plafond de priorité $P'_{plafond}$ définit comme le maximum des $P_{plafond}(R_i)$. En pratique, les tâches prêtes à s'exécuter sont placées dans une pile par ordre décroissant de leur priorité et un test de préemption est effectué sur le sommet de la pile. Pour que la tâche τ_i puisse préempter la tâche τ_j, il faut que son niveau de préemption P'_i soit supérieur au plafond de préemption du système $P'_{plafond}$. Le test recommence à chaque fois que $P'_{plafond}$ change de valeur.

Ces protocoles de synchronisation sont maîtrisés et leur utilisation se généralise. Par exemple, dans le domaine automobile, c'est le protocole iPCP (Immediate PCP, plus simple que PCP, mais avec les mêmes performances dans le pire cas) qui a été retenu pour la gestion des ressources partagées par les standards OSEK/VDX OS (OSEK/VDX, 2005b) et AUTOSAR OS (AUTOSAR, 2013a).

2.2.2.2 Protocoles de synchronisation multicœur bloquants

La programmation parallèle offerte par les architectures multicœur complexifie la gestion des ressources partagées. En termes de conception et de codage, le programmeur doit veiller à l'entrelacement des actions sur les ressources partagées, pouvant conduire à la production de résultats non cohérents. De plus, l'accroissement de l'espace d'états rend difficile la détection et la correction des bogues lors de la mise en place des synchronisations. Des protocoles de contrôle de concurrence sont là aussi nécessaires pour maîtriser les effets du parallélisme.

Le partage des ressources dans une architecture multicœur nécessite de distinguer deux types de ressources : les *ressources locales* et les *ressources globales.*

Définition 2.7. *Les ressources locales ne peuvent être manipulées que par des tâches d'un même cœur.*

Définition 2.8. *Les ressources globales sont partagées entre les cœurs. Leur protection nécessite des protocoles spécifiques.*

Les protocoles que nous avons évoqués dans le paragraphe 2.2.2.1 nécessitent d'être adaptés pour prendre en compte le parallélisme des architectures multicœur et limiter les interférences entre les tâches qui souhaitent accéder aux mêmes ressources (Easwaran et Andersson, 2009).

En effet, les blocages (i.e. attente de la libération d'une ressource globale) vont faire intervenir un *blocage local* et un *blocage distant* (dû aux autres cœurs). En fonction des mécanismes de synchronisation utilisés, ces temps de blocage peuvent être importants. De plus, une attention particulière est là aussi nécessaire pour éviter les problèmes d'interblocage et de famine.

Parmi les protocoles bloquants, ceux définis en monocœur ont été étendus pour le multi-cœur.

MPCP (Multiprocessor PCP) (Rajkumar, 1991) est l'extension multicœur du protocole PCP. Les tâches possèdent une priorité de base (obtenue selon la politique d'ordonnancement choisie) et héritent de la priorité plafond d'une ressource à chaque fois qu'elles entrent dans une section critique associée à cette ressource. Comme avec PCP, la priorité plafond d'une ressource locale est définie comme la plus haute priorité de toutes les tâches qui peuvent l'utiliser. Pour les ressources globales, la priorité plafond doit être supérieure à toutes les priorités de toutes les tâches du système. Les tâches qui échouent lors de la prise d'une ressource globale sont ajoutées à une file globale ordonnée selon les priorités des tâches. Une tâche qui est bloquée pour l'obtention d'une ressource globale n'empêche pas des tâches (potentiellement de plus faible priorité) de s'exécuter et de rentrer dans une section critique. Ce protocole n'accepte pas l'imbrication des accès aux ressources (globales et locales). Enfin, au vu de la complexité du protocole, il est recommandé de minimiser le nombre de ressources globales, pour limiter l'accroissement des attentes induites par la gestion de ce type de ressources.

MSRP (Multiprocessor SRP) (Gai *et al.*, 2001) est l'extension multicœur du protocole SRP. Pour les ressources locales, la gestion est la même que celle définie en monocœur. En revanche, pour les ressources globales, une tâche qui ne peut pas accéder directement à la ressource est insérée dans une file d'attente et effectue une attente active (spinning) au lieu d'être simplement bloquée (i.e. suspendue), ce qui implique un blocage actif du cœur. Afin de minimiser ce temps d'attente active, les tâches qui entrent en section critique sur une ressource globale ne peuvent pas être préemptées. La ressource est libérée le plus tôt possible. Pour les tâches qui échouent lors de la prise d'une ressource globale, MSRP utilise une file d'attente ordonnée en FIFO. Ce protocole accepte la prise de ressources locales pendant l'accès à une ressource globale, mais n'accepte pas l'imbrication des ressources globales à cause du risque d'interblocage.

L'alternative FMLP (Flexible Multiprocessor Locking Protocol) (Block *et al.*, 2007) est une combinaison de MPCP et MSRP qui a la particularité de fonctionner pour les politiques d'ordonnancement P-EDF et G-EDF. Le protocole classe les ressources en deux catégories. Les *ressources longues* sont gérées via MPCP et les *ressources courtes* sont gérées avec MSRP. C'est à la charge de l'utilisateur de définir la frontière entre les deux types de ressources. Ce choix d'implémentation vise à améliorer le débit en accélérant les accès aux ressources courtes (en minimisant leur temps de blocage) et en autorisant la suspension des tâches accédant des ressources longues (ce qui réduit les périodes de temps où le CPU est en attente active). Les imbrications d'accès aux ressources sont possibles à condition qu'une ressource longue ne soit pas requise pendant l'accès à une ressource courte. L'imbrication de ressources permet de créer des groupes de ressources. Néanmoins, toutes les ressources d'un même groupe sont du même type (longues ou courtes).

Une implémentation des protocoles MPCP et FMLP est proposée par Brandenburg et Andersson dans Brandenburg et Anderson (2008). Cette implémentation est mise en œuvre sous $LITMUS^{RT}$ (Calandrino *et al.*, 2006), une extension temps réel souple du noyau Linux pour les systèmes multicœur.

2.2.2.3 Partage de ressources globales dans AUTOSAR

La révision 4.0 du standard AUTOSAR multicœur propose l'utilisation de spinlocks pour le partage des ressources globales. Ce type de synchronisation suppose qu'un cœur en attente d'une ressource *attend activement* la libération de celle-ci.

2.3 Modèle de fautes pour les systèmes temps réel multicœur

Nous nous intéressons désormais aux fautes spécifiques aux architectures multicœur, principalement celles dues aux mécanismes utilisés pour la coopération des cœurs. Notons que le modèle de fautes suivant s'applique aussi bien pour une politique d'ordonnancement globale que pour une politique d'ordonnancement partitionnée.

2.3.1 Activation des fautes liées aux accès à la mémoire commune

Le premier canal de propagation des fautes est induit par les blocs matériels communs aux cœurs ; en particulier, nous ciblons ici la mémoire commune.

2.3.1.1 Le problème des accès illégaux en mémoire

Un accès mémoire illégal dû à une faute de codage peut survenir en cas de mauvaise configuration des mécanismes de protection mémoire. Considérons deux exemples de fautes de codage qui seraient masquées dans un contexte monocœur et activées dans un contexte multicœur.

Premièrement, considérons le mauvais dimensionnement d'une pile. Pour illustrer ce phénomène considérons deux tâches τ_i et τ_j, s'exécutant sur une architecture monocœur et pour lesquelles les piles sont contigues en mémoire. Un mauvais dimensionnement de la pile de τ_i conduit à un dépassement sur celle de τ_j, par exemple suite à l'occurrence d'une interruption qui s'exécute dans le contexte de τ_i. Puisque les exécutions sont séquentielles, τ_j s'activera totalement avant (i.e. avant la corruption) ou après τ_i. Dans ce dernier cas, aucun problème ne survient puisque τ_i a déjà dépilé. Sur une architecture multicœur, considérons que τ_i et τ_j sont sur des cœurs distincts. Si les deux tâches s'exécutent en même temps, la corruption de la pile de τ_j par τ_i peut détruire le contexte de τ_j. La faute est donc activée en multicœur.

En second lieu, considérons les problèmes liés à la configuration des mécanismes de protection mémoire et à la complexité de leur mise en place. L'étape de *memory mapping* effectuée durant l'édition des liens peut conduire à un entrelacement des zones mémoires accessibles par les différents cœurs. Même en validant la configuration individuelle des MPUs, la configuration globale peut conduire à des situations problématiques. Le test exhaustif de la configuration est rendu difficile par le trop grand nombre de configurations d'accès à la mémoire possibles. Pour illustrer ce problème, considérons deux tâches τ_i et τ_j dont les configurations des MPUs sont entrelacées, suite à une erreur de configuration. Si les deux tâches s'exécutent sur le même cœur, l'écriture par τ_i dans la zone commune avec τ_j ne pose pas de problème quand les deux tâches s'exécutent l'une après l'autre. Dans un contexte multicœur, l'entrelacement de l'exécution des deux tâches peut conduire à une corruption mutuelle des données stockées dans la zone commune par l'une ou par l'autre. La faute est également activée en multicœur.

2.3.1.2 Le problème du partage des données

Dans un contexte multicœur, les données partagées en mémoire peuvent souffrir de conflits dits de lecture/écriture ou écriture/écriture. Pour illustrer ce problème, considérons deux tâches τ_i et τ_j situées sur deux cœurs différents. τ_i lit une donnée A tandis que τ_j écrit cette même donnée. Si l'écriture de A n'est pas atomique ou réalisée en exclusion mutuelle, la valeur lue par τ_i peut être inconsistante. La mise en place d'un protocole de synchronisation est essentielle.

Dans les protocoles de synchronisation multicœur (c.f. Section 2.2.2 page 30), la gestion des synchronisations intercœur passe par l'utilisation de verrous de type spinlocks. Ces mécanismes doivent être gérés de manière correcte, c'est-à-dire qu'il convient de prendre des précautions avant d'acquérir un verrou (e.g. interrompre les interruptions sur le cœur prenant le verrou), maintenir le verrou le moins de temps possible et le relâcher correctement. La complexité de l'application fait qu'il est difficile de s'assurer que toutes les synchronisations seront effectuées correctement. De plus, des situations de famines ou d'interblocages peuvent survenir en cas de méconnaissance ou de défaillance des mécanismes de verrous matériels. Par exemple, considérons deux tâches τ_i et τ_j allouées à deux cœurs différents et souhaitant accéder à une ressource R. Si τ_i verrouille R et que τ_j souhaite également la verrouiller, τ_j attend. Dans le cas où τ_i subit une défaillance (ou que le verrou de R subit une défaillance), le problème sera propagé sur τ_j, entité pourtant saine à l'origine.

Enfin, le partage de ressources requiert parfois de maintenir la cohérence d'un groupe de données (*coherency group* illustré dans AUTOSAR (2013c)). Pour illustrer ce besoin, considérons un algorithme de régulation. Pour que la consigne calculée soit valide, il faut que toutes les données d'entrées soient issues de la même instance. Si elles sont modifiables par des tâches allouées à des cœurs différents, le parallélisme peut entrainer leur incohérence. Pour éviter cela, l'intégrateur doit mettre en œuvre tous les moyens permettant d'isoler les actions sur ce groupe de données.

De tout cela, il résulte la présence de *fautes de codage en phase de développement inhérentes à la complexité de l'application.*

2.3.2 Activation des fautes liées à la non-maîtrise de l'ordonnancement multicœur

Il faut souvent respecter des relations de dépendance fonctionnelle entre les tâches en veillant à ce que leur exécution conduise à l'observation d'un flot de données qui respecte les spécifications. Si ce n'est pas le cas, la cohérence des données n'est plus assurée. La maîtrise du comportement d'un système multitâche impose de se conformer strictement à une architecture opérationnelle relativement simple. Cela peut être un problème lors de la traduction de l'architecture fonctionnelle en paramètres logiciels, où des dépendances fonctionnelles doivent être traduites en termes de dates d'activation, de priorités de tâche, de ressources partagées, etc.

Sur des architectures monocœur et pour des architectures fonctionnelles simples, la gestion des interrelations entre les tâches peut être rendue déterministe puisque qu'elles s'exécutent les unes après les autres. Dans une telle situation, des tâches fonctionnellement dépendantes peuvent être rendues indépendantes après l'implémentation grâce aux choix des paramètres logiciels. Pour s'y fier, il faut néanmoins bénéficier de preuves permettant d'assurer le respect

des exigences (e.g. analyse d'ordonnancement). Mais l'architecture fonctionnelle d'un système électronique embarqué moderne est complexe. Elle comporte des fonctions de différentes natures, soumises à des contraintes temps réel de différents ordres de grandeur (de quelques microsecondes à quelques millisecondes). La prise en compte de ces contraintes temps réel impose souvent une rupture entre les architectures fonctionnelle, logicielle et opérationnelle de sorte que même des entités fonctionnellement indépendantes peuvent se retrouver liées au niveau opérationnelle.

La gestion des interrelations entre des tâches pouvant s'exécuter en parallèle sur une architecture multicœur n'est pas aisée. La complexité induite par ce parallélisme rend difficile le respect des dépendances fonctionnelles et accroît le nombre de paramètres intervenant dans la maîtrise du bon fonctionnement de l'application. À titre d'exemple, comme introduit dans la Section 2.2.1.4 (page 29), la détermination du WCET est rendu difficile à cause des contensions sur les *bus*, les *crossbars* ou les caches. Même si une analyse basée sur le WCET permet de justifier le respect de certaines exigences, la forte variabilité du temps d'exécution réel des tâches, dû à la surestimation du WCET, peut remettre en cause cette analyse : e.g. si une tâche termine trop tôt, elle va produire ses données en avance. Il est alors plus difficile pour l'intégrateur de maîtriser et de garantir l'absence de fautes dans le logiciel embarqué pour toutes les configurations possible de l'espace d'états.

Notons enfin que le pire-cas en monocœur pris en compte lors des analyses d'ordonnançabilité correspond au moment où toutes les tâches sont actives en même temps, ce qui n'est pas systématiquement le pire-cas dans un contexte multicœur.

De tout cela, il résulte la présence de *fautes de codage en phase de développement inhérentes à la non-maîtrise du comportement de l'application.*

2.4 Vers la tolérance aux fautes

L'occurrence des fautes présentées précédemment conduit à un certain nombre d'erreurs qu'il faut pouvoir gérer. Les fautes qui n'auraient pas pu être traitées par la prévision ou la prévention peuvent toujours se produire pendant la phase opérationnelle du système. Pour traiter les fautes résiduelles, des solutions de tolérance aux fautes peuvent être mises en place. Nous proposons à présent une classification générique des mécanismes de ce type.

2.4.1 Généralités sur la tolérance aux fautes

Les mécanismes de tolérance aux fautes s'interrogent sur la manière de fournir un service à même de remplir la fonction du système en dépit des fautes (Arlat *et al.*, 2006).

Le traitement des fautes nécessite de déterminer leurs causes du point de vue de leur localisation et leur nature (diagnostic) puis de prévenir une nouvelle occurrence (passivation).

Le traitement de l'erreur vise à l'éliminer avant que la défaillance ne survienne. Ce traitement passe systématiquement par trois étapes : la *détection*, le *confinement* et le *recouvrement*. La *détection* utilise des mécanismes spécifiques pour mettre en évidence l'occurrence d'une faute engendrant une erreur dans l'application. Afin de l'isoler et de limiter la contagion des autres parties de cette application, un mécanisme de *confinement* doit être mis en place. Enfin, le *recouvrement* consiste à reconstituer un état exempt de fautes à partir duquel il sera possible de poursuivre l'exécution.

Les mécanismes de tolérance aux fautes peuvent être caractérisés par les deux critères suivants : la *transparence* et la *flexibilité* (Taiani *et al.*, 2006). Dans un souci de *séparation des préoccupations* (*Separation of Concerns*) (Hursch et Lopes, 1995), la *transparence* consiste à employer des techniques de tolérance aux fautes non intrusives, qui ne nécessitent pas de modifications de l'application de base. Un mécanisme *flexible* facilite son portage sur d'autres types de système. Dans l'idéal, le mécanisme de tolérance aux fautes doit être portable et être développé sans se soucier de l'application. Par exemple, des travaux basés sur le tissage d'aspects répondent à ce besoin (Alexandersson et Ohman, 2010).

Nous distinguons les solutions matérielles des solutions logicielles. Dans le premier cas, les mécanismes évoqués nécessitent l'ajout de composants matériels supplémentaires, même si leur utilisation peut être réalisée au niveau du logiciel. Ces solutions reposent donc sur une conception spécifique du matériel. Dans le second cas, les solutions logicielles sont élaborées indépendamment des contraintes matérielles. Seul ce dernier point sera étudié.

2.4.2 Taxonomie des mécanismes logiciels pour la tolérance aux fautes

Certaines solutions sont basées sur la diversification fonctionnelle (Laprie *et al.*, 1995; Laprie, 1995; Torres-Pomales, 2000; Scott *et al.*, 1987; Randell et Xu, 1995). Elle peut être effectuée aussi bien au niveau des spécifications (e.g. mécanisme autotestable) qu'en termes d'équipes de développement ou de langages de programmation (e.g. programmation N-version). Bien que ce dernier niveau soit plus complet, il est aussi plus cher à mettre en place et ne doit être réservé qu'aux fonctionnalités les plus critiques. Dans tous les cas, l'objectif est d'obtenir suffisamment de diversité pour permettre non seulement de traiter des fautes qui surviennent durant la phase opérationnelle, mais aussi celles qui sont issues du développement du logiciel.

Une autre manière de faire repose sur la réplication. Par réplication, nous entendons la *copie* du code ou d'une partie du code. Les versions sont alors toutes identiques, ce qui ne permet pas de lutter contre les fautes de développement. C'est pourquoi ce mécanisme est plutôt adapté pour les fautes transitoires ou permanentes qui surviennent durant la phase opérationnelle.

Une classification des mécanismes logiciels pour la tolérance aux fautes dérivé de Laprie *et al.* (1995) est présentée dans la Table 2.1.

Un *composant autotestable* regroupe le composant logiciel en charge de réaliser un service donné ainsi que les mécanismes de détection d'anomalies permettant de statuer sur la validité du service rendu. Le bloc de détection (aussi appelé détecteur) est synthétisé à partir des spécifications du système (e.g. les mêmes ayant servi au codage de l'application). On peut également se baser sur une description du comportement dysfonctionnel, c'est-à-dire des comportements indésirables et redoutés. Dans la pratique, le détecteur est intégré au composant à surveiller, comme illustré par la Figure 2.4. À partir des entrées utilisées par l'entité fonctionnelle puis des sorties calculées, l'état du système est déduit. En cas d'erreur, l'information est signalée et le composant logiciel fautif est isolé pour éviter la propagation des fautes : c'est le confinement. De par leur nature, ces mécanismes ne permettent que la détection et des solutions spécifiques doivent être mises en place pour le recouvrement.

Le recouvrement nécessite de sauvegarder régulièrement l'état du système (i.e. *checkpoint*). Le choix des sauvegardes peut respecter un planning défini hors-ligne ou être calculé en-ligne.

<table>
<tr><th>Nom
du mécanisme</th><th>Détection</th><th>Confinement</th><th>Recouvrement</th><th>Modèle
de fautes</th></tr>
<tr><td>Composant
autotestable</td><td rowspan="3">Tests d'acceptation</td><td rowspan="3">Suspension du service</td><td>Reprise, poursuite
ou compensation</td><td rowspan="4">Fautes de
développement
et
opérationnelles</td></tr>
<tr><td>Blocs
de recouvrement</td><td rowspan="2">Changement de
variante</td></tr>
<tr><td>Programmation
N-autotestable</td></tr>
<tr><td>Programmation
N-versions</td><td rowspan="2">Comparaison
des résultats</td><td rowspan="2">Transparent</td><td rowspan="2">Vote</td></tr>
<tr><td>Réplication</td><td>Fautes
opérationnelles</td></tr>
</table>

Table 2.1 – Techniques pour la tolérance aux fautes (Laprie *et al.*, 1995)

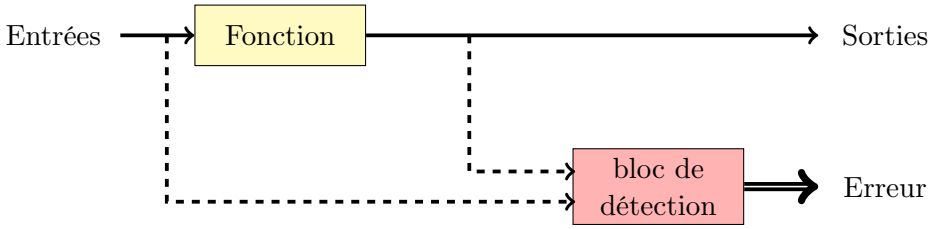


Figure 2.4 – Illustration d'un composant autotestable

Il existe trois types de recouvrement : 1 – *la reprise* est effectuée après la détection d'une anomalie. Elle impose le retour à un état stable à partir d'un précédent checkpoint ; 2 – *la reprise (nommée ici poursuite)* consiste à continuer l'exécution du programme (éventuellement en fournissant un service dégradé) en dépit de fautes ; enfin, 3 – *la reprise (nommée ici compensation)* consiste à rechercher parmi les précédentes exécutions, suffisamment d'éléments qui permettent de continuer l'exécution du programme.

Dans Rashid *et al.* (2010a,b), Pattabiraman *et al.* (2005), des informations sont fournies concernant la stratégie pour le placement des détecteurs : la construction d'un graphe de flot de données associé à l'établissement de certaines métriques prédisant les crashs, permettent de déterminer les nœuds de l'application où il est le plus intéressant de les placer. Les solutions présentées dans Li *et al.* (2008); Hari (2009); Hari *et al.* (2009) proposent des stratégies de détection et de diagnostic sur les architectures monocœur et multicœur.

Les autres solutions exploitent la redondance du code à protéger. Si le nombre de répliques est inférieur à deux, on ne peut faire que de la détection tandis que lorsque le nombre de versions dépasse deux, le recouvrement devient possible. En ce sens les mécanismes à base de *blocs de recouvrement* contiennent plusieurs versions d'une même fonction. Elles sont exécutées de manière séquentielle, c'est-à-dire que dès lors que la version primaire échoue, une version secondaire prend le relais. Le service fourni d'une version à une autre peut potentiellement être dégradé.

La *programmation N-autotestable* consiste à utiliser N composants autotestables. La détec-

tion est toujours effectuée à l'aide de blocs de détection, mais le recouvrement est effectué en utilisant les autres de variante. Il est possible d'exécuter les versions de manière séquentielle, mais pour éviter les contraintes temporelles imposées par la séquentialité, il est également possible d'exécuter toutes les versions en parallèle. Pour détecter les anomalies, la comparaison des résultats entre les répliques peut être effectuée. Le recouvrement quant à lui peut être réalisé par un système de vote, quand le nombre de répliques est suffisant pour pouvoir prendre une décision. En cas de détection d'une erreur, l'impact temporel est réduit au temps nécessaire au vote. La *programmation N-versions* et les macanismes de redondance du code fonctionnent sur le même principe.

Des exemples concrets basés sur la réplication et la diversification fonctionnelle sont respectivement fournis dans Shye *et al.* (2009) et Kantz et Koza (1995). Un exemple de réplication à très fine granularité est rapporté dans Reis *et al.* (2005a,b). Une classification détaillée des différents mécanismes pour la tolérance aux fautes est disponible dans Laprie *et al.* (1995).

Chapitre 3

Contributions de la thèse

Sommaire

3.1 Hypothèses et problématique

3.1.1 Architectures ciblées

Les choix d'architectures sont dictés par le contexte automobile que nous visons. Nous devons donc choisir les architectures les mieux adaptées aux besoins industriels exprimés dans la Section 1.4.4 (page 22).

3.1.1.1 Architecture matérielle

L'architecture matérielle doit être adaptée au domaine d'utilisation (Blake *et al.*, 2009). Les applications basées sur la *manipulation des données* (e.g. applications pour la gestion d'un réseau, la gestion du multimédia) sont fortement parallèles. En revanche, les applications basées sur le *contrôle de processus* sont axées sur la coopération des tâches (e.g. dépendances entre les tâches pour les transferts de données).

Notre domaine d'application correspond à la seconde classe d'application. Les constructeurs cherchent à accroître l'utilisation des ressources CPU, c'est-à-dire exploiter au maximum la performance des microcontrôleurs. Nous choisissons donc de considérer deux types d'architecture : les architectures homogènes utilisées en mode découplé (e.g. de type Leopard MPC 5643L de Freescale) et les architectures uniformes (e.g. de type Bolero MPC 5644C de Freescale). Après analyse des roadmaps des différents fondeurs, des puces à 2, 3 et 5 cœurs peuvent être considérées. Ce choix permet également d'utiliser les architectures multicœur pour faciliter la mise en place de mécanismes de tolérance aux fautes.

3.1.1.2 Architecture logicielle

La spécificité du domaine automobile nous amène à chercher la compatibilité avec AUTOSAR. Les solutions mises en œuvre doivent prendre en compte toutes les hypothèses issues du domaine.

3.1.2 Modèle de fautes considéré

Nous considérons le modèle de fautes présenté dans la Table 3.1.

Fautes logicielles liées au développement	**Exemples de fautes**	**Exemples d'erreurs**	**Traité dans la thèse**
Activation des fautes liées à la non-maîtrise de l'ordonnancement	Mauvaise configuration des paramètres logiciels	- Violation des spécifications en termes de flots de contrôle et/ou données	√
	Sur-estimation du WCET	- Sur/sous production ou consommation de données	√
	Sous-estimation du WCET	- Violation des contraintes temporelles	∅
Activation des fautes liées aux accès à la mémoire commune	Mauvaise configuration	- Configuration des mécanismes de protection mémoire	∅
	Fautes de codage	- Accès illégaux en mémoire (e.g. dépassement de pile)	∅
		- Conflits lecture/écriture ou écriture/écriture	√
		- Interblocage, famine	√
		- Corruption d'un groupe de données	√

Table 3.1 – Modèle de Fautes considéré dans nos travaux

La colonne de droite permet d'identifier le périmètre de ce qui est traité par nos travaux. Le modèle de fautes correspond à la classe de fautes de développement, internes, logicielles, humaines et accidentelles ou délibérées non malveillantes. Ceci correspond aux classes 1 et 2 de l'arbre de fautes de la Figure 1.4. Les erreurs qui en résultent se produisent en phase opérationnelle, pendant l'exécution de l'application.

Le choix du modèle de fautes ciblé est également lié au contexte industriel dans lequel nous évoluons. Nous traitons toutes les fautes liées à la non-maîtrise de l'ordonnancement à l'exception des problèmes de sur-estimation ou sous-estimation du WCET des tâches qui conduirait à des violations de contraintes temporelles. En effet, il existe déjà des mécanismes de protection temporelle dans AUTOSAR pour vérifier ce type de contraintes. Dans les travaux de Bertrand (2011), des stratégies pour le calcul du budget des tâches sont proposées de telle sorte que l'application respecte les contraintes d'ordonnançabilité, tout en relâchant au maximum les contraintes sur les tâches.

La propagation des fautes liées aux accès à la mémoire commune conduit à un certain nombre d'erreurs. Nous choisissons de nous focaliser sur les erreurs provoquant des conflits entre les écrivains et les lecteurs sur des données dans la mémoire, des corruptions de groupes de données et des situations d'interblocage ou de famine. Ces erreurs résultent de la mise en place de la synchronisation des données par l'intégrateur. Nous ne ciblons pas non plus les fautes liées à une mauvaise configuration des mécanismes de protection mémoire qui sont responsables en partie des accès illégaux en mémoire.

3.1.3 Problématique et approche suivie

La problématique est la suivante :

Comment améliorer la robustesse des applications s'exécutant sur des architectures multicœur ?

On entend par *robustesse*, la capacité à satisfaire les spécifications en dépit de la présence de fautes. L'étude du modèle de fautes motive deux axes de travail :

- Maîtriser l'impact de la non-maîtrise de l'ordonnancement par l'ajout d'un mécanisme de tolérance aux fautes basé sur la vérification en-ligne des propriétés intertâches ;
- Minimiser l'occurrence des conflits de lecture et d'écriture sur la mémoire commune en proposant une alternative aux protocoles de synchronisation bloquants, pour les données partagées entre des cœurs. Cette alternative est un protocole non bloquant.

L'ensemble de nos travaux est intégrable dans un système AUTOSAR et nos solutions sont une réponse au respect de la norme de sûreté de fonctionnement ISO 26262.

3.2 Vérification en-ligne de propriétés intertâches

Les fautes liées à la non-maîtrise de l'ordonnancement conduisant au non-respect des contraintes sur les flots de données et de contrôle peuvent être traitées par des mécanismes de tolérance aux fautes. Nous choisissons de mettre en œuvre un mécanisme de détection d'erreurs (au sens de la tolérance aux fautes) reposant sur l'approche de la vérification en-ligne avec une surveillance des flots de données.

Nous souhaitons nous intégrer au processus de développement industriel. Les exigences sont connues dès la phase de conception de l'architecture fonctionnelle. La synthèse des mo-

niteurs (i.e. logiciel de surveillance) utilisés pour la vérification en ligne est donc effectuée en parallèle de ce processus en suivant l'approche suivante : (1) hors-ligne, nous construisons des moniteurs à partir des exigences. Ces moniteurs sont compilés avec le reste de l'application ; (2) en-ligne, un service de vérification surveille le système en temps réel et s'assure du respect des propriétés. La violation ou la satisfaction d'une propriété est notifiée. Par exemple en cas d'erreurs, des stratégies permettant de mettre en œuvre un recouvrement (e.g. mode dégradé, reconfiguration) sont disponibles.

L'architecture d'un tel mécanisme est présentée sur la Figure 3.1. Un exemple d'architecture logicielle considérée est composé de trois tâches (T_0, T_1 et T_2) qui communiquent entre elles par l'intermédiaire de deux buffers (b_0 et b_1). s_{ij} signifie que la tâche T_i envoie un message dans b_j et r_{ij} signifie que la tâche T_i reçoit un message en provenance de b_j. Ainsi, l'évènement s_{00} est produit lorsque T_0 dépose un message dans b_0.

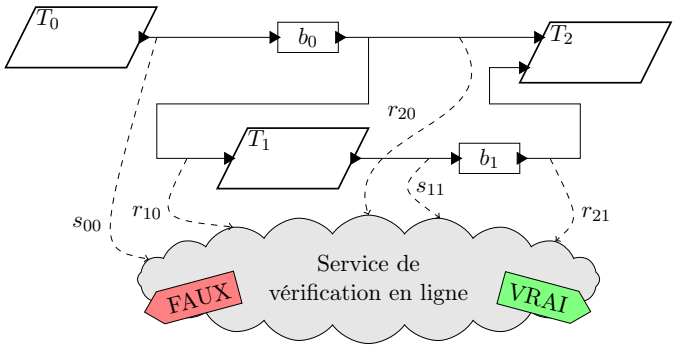


Figure 3.1 – Architecture d'un mécanisme de vérification en ligne

Considérons le problème suivant : T_2 effectue un contrôle de vraisemblance sur les opérations effectuées par T_1. Pour que le contrôle ait un sens, il faut que les messages reçus par T_2 soient cohérents, c'est-à-dire que le message lu à partir de b_1 soit issu de celui lu à partir de b_0. Lorsque cette condition est respectée, on dit que les deux buffers sont *synchronisés*. L'exigence peut s'écrire en langage naturel comme suit :

Exigence 3.1. *Les valeurs lues par T_2 doivent être cohérentes.*

L'exigence peut être décomposée en deux scénarios. Si T_2 lit b_0 en premier, T_2 ne doit pas lire b_1 tant que la synchronisation des deux buffers n'est pas réalisée. Si T_2 lit b_1 en premier, les deux buffers doivent déjà être synchronisés et le rester jusqu'à ce que T_2 lise b_0.

L'analyse de l'architecture permet de définir l'ensemble des évènements, noté Σ^S, qu'il sera nécessaire d'intercepter : $\Sigma^S = \{s_{00},\ s_{11},\ r_{10},\ r_{20},\ r_{21}\}$. La séquence d'évènements est ensuite analysée par le moniteur. Quand une propriété est évaluer à $vrai$ ou $faux$, la surveillance est terminée. Si on ne peut pas conclure, la surveillance doit continuer (c.f. Section 5.2.1, page 66).

À titre d'exemple, la trace d'évènements $s_{00},\ r_{20},\ r_{10},\ s_{11},\ r_{21}$ est conforme à l'exigence tandis que la trace $s_{00},\ r_{20},\ r_{10},\ r_{21}$ ne l'est pas.

Nous montrons dans la Partie II comment synthétiser les mécanismes de détection à partir des exigences, puis nous évaluons leur impact en termes de surcoût d'exécution et d'occupation

mémoire. Un cas d'étude sur une application industrielle automobile encourage la poursuite de leur étude dans le contexte embarqué temps réel.

Notons enfin que la mise en place d'un tel service n'est pas nouveau dans AUTOSAR. En effet, le standard intègre déjà un service de vérification en ligne permettant de surveiller la violation des contraintes temporelles. En ce sens, notre approche est complémentaire et cible une autre catégorie de fautes mise en évidence par l'introduction des architectures multicœur.

3.3 Synchronisation non bloquante pour le partage de données

Des travaux ont déjà été menés sur le partage des ressources mémoires dans une architecture multicœur. Ils ont conduit à l'élaboration d'un ensemble de protocoles, dont certains ont été évoqués dans la Section 2.2.2.2 (page 31). Quelques difficultés apparaissent :

- Les approches bloquantes conduisent à un abaissement du parallélisme (les blocages sur ressources viennent réintroduire de la séquentialité dans les fils d'exécutions parallèles) et entraînent une baisse des performances. Cette chute du parallélisme est d'autant plus visible quand le nombre de cœurs augmente et lorsque la synchronisation intercœur est gérée par des mécanismes basés sur l'attente active (cas d'AUTOSAR) ;
- La nécessité de maintenir la cohérence de groupes de données (*coherency group*), tel que spécifié dans AUTOSAR (2013c), découle des besoins fonctionnels du domaine (e.g. lecture cohérente des entrées d'un algorithme de régulation). La complexité de ce type d'applications rend difficile la mise en place des protocoles de synchronisation. Pour une large section critique, l'utilisation d'un verrou gros-grain n'est pas une solution optimisée. Pour être plus fin, on peut choisir d'utiliser plusieurs *petits* verrous, ce qui complexifie la phase d'implémentation (dépendante de l'application) pour veiller à maintenir la cohérence des accès aux données. Des fautes liées à l'ordre de prise et de relâchement des verrous peuvent notamment être introduites au moment du codage ;
- Les approches à base de verrous sont soumises aux fautes matérielles ou logicielles. Par exemple, une faute matérielle qui conduirait à l'impossibilité de relâcher un spinlock le rendrait indisponible pour toutes les tâches en attente active sur ce verrou.

Les travaux ciblés par la thèse n'ont pas vocation à traiter le premier point. Nous nous intéressons aux problèmes liés à la difficulté d'implémentation de la synchronisation et à la robustesse, tout en conservant le parallélisme d'exécution offert par la plateforme sous-jacente. Pour cela, nous proposons d'étudier une alternative basée sur des protocoles de synchronisation non bloquants. Ces approches possèdent des avantages qui justifient leur étude. Le partage de ressources avec cette approche est géré par le protocole de manière transparente pour l'utilisateur. Ce dernier doit seulement identifier les sections critiques de code et encapsuler les accès aux ressources dans un bloc logiciel pris en charge par le protocole. Cela permet directement de supprimer les fautes de codage au niveau applicatif et ainsi d'améliorer la robustesse du système. L'analyse temporelle peut également être dissociée de la politique d'ordonnancement mise en œuvre sous certaines conditions. Dans un contexte de sûreté de fonctionnement, le protocole doit être certifié, puis peut être réutilisé.

Le protocole proposé permet de garantir la progression de toutes les tâches d'une application AUTOSAR. Une modélisation et une vérification partielle à l'aide d'un outil de vérification de modèle ont été réalisées. Les détails sont disponibles dans la Partie III.

Deuxième partie

Vérification en-ligne de propriétés inter-tâches

Chapitre 4

Introduction à la vérification en ligne

Sommaire

4.1 Vers la vérification en ligne des systèmes

4.1.1 Le problème de la vérification des systèmes

Tout système doit être validé et vérifié.

Définition 4.1. *Un système est dit valide si l'on dispose de preuves tangibles que les exigences décrivant l'utilisation prévue sont satisfaites (ISO 9000, 2005).*

Valider un système revient à se demander si l'on construit le bon système pour respecter les spécifications.

Définition 4.2. *Un système est dit vérifié si l'on dispose de preuves tangibles que les exigences sont satisfaites pour que le système soit sûr (ISO 9000, 2005).*

Vérifier un système revient à s'assurer que le système fonctionne correctement (i.e. de manière sûre) vis-à-vis de ses spécifications.

Plusieurs approches permettent de vérifier un système. Nos travaux se focalisent sur la vérification en ligne. Cette solution est située entre la vérification de modèles et les tests. Ces deux approches sont présentées et comparées dans les sections suivantes.

4.1.1.1 La vérification de modèles (model checking)

Les outils de vérification de modèles ont été développés pour vérifier le bon fonctionnement des systèmes discrets. Les travaux préliminaires ont été menés par Emerson et Clarke (1980); Clarke et Emerson (1982); Queille et Sifakis (1982). Le prix Turing 2007 a été décerné à Clarke, Emmerson et Sifakis pour leurs travaux sur la vérification de modèles. Nous présentons ici les fondements théoriques posés dans les années 1980.

La vérification de modèle, ou *model checking*, est une technique de vérification formelle applicable aux systèmes possédant un nombre fini d'états. L'objectif est de vérifier algorithmiquement qu'un (modèle du) système satisfait des spécifications exprimées le plus souvent en termes de logique temporelle (Pnueli, 1977), pour *tous les chemins d'exécutions possibles*. Pour réaliser cela, cette technique repose sur 3 éléments illustrés sur la Figure 4.1. Tout d'abord, il faut modéliser le système à vérifier (M). Ce modèle est une structure de Kripke (Kripke, 1963), qui décrit les états possibles du système, ses évolutions (relation états/transitions), et la liste des propositions atomiques vérifiées à chaque état. Une propriété (notée ϕ) est écrite à l'aide d'une formule de logique temporelle sur les propositions atomiques qui caractérisent l'état du système. Pour les besoins liés à la vérification, la propriété ϕ est traduite en un automate de Büchi reconnaissant le même langage (c.f. Section 5.1, page 62). Enfin, l'algorithme de vérification compose le modèle et les automates des propriétés pour calculer si elles sont vérifiées (on note $M \models \phi$ si la propriété ϕ est satisfaite par M). Dans la pratique, nous n'utilisons pas directement la propriété à vérifier, mais la *négation de cette propriété*. Cette astuce permet de faciliter l'analyse puisqu'elle se résume alors à décider si le langage de l'automate composé est vide. Si oui, la négation de la propriété ne survient jamais, donc la propriété est vraie. Dans le cas contraire, un contre-exemple est fourni (Figure 4.1).

La vérification de modèles peut être appliquée à plusieurs niveaux du processus de développement. Le plus souvent, le modèle du système à vérifier peut être dérivé de l'analyse de

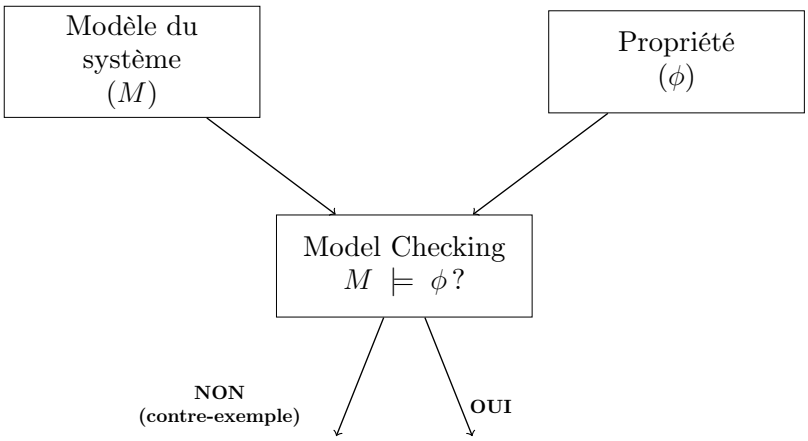


Figure 4.1 – Approche Model-Checking

l'architecture fonctionnelle, de l'architecture logicielle ou encore être synthétisé à partir du code source.

Des outils comme SPIN (Holzmann, 1997) ou encore SMV et NuSMV (McMillan, 1993) trouvent peu à peu leur place dans l'industrie.

4.1.1.2 Le test

Dans la procédure de vérification d'un système, les tests interviennent dans la seconde moitié du cycle de développement en V, c'est-à-dire juste après la phase de codage. L'objectif est d'identifier les comportements qui ne respectent pas la spécification. Une méthode consiste à fournir au système une séquence d'entrées pour laquelle on connaît déjà le résultat attendu. Une comparaison entre ce que l'on attend et ce que l'on obtient permet de conclure si *le système respecte la spécification pour le jeu d'entrées choisi* (e.g. test par oracle (Weyuker, 1980)).

En fonction du niveau d'accessibilité au code des composants, il existe des tests de type *boîte blanche* ou *boîte noire*. Le premier cas permet de concevoir des protocoles de tests fondés sur l'analyse de la structure interne du composant. Dans le second cas, seules les interfaces d'entrées/sorties sont disponibles pour la mise en place des tests.

La facilité de mise en œuvre du test fait de cette solution une pratique appréciée et répandue dans l'industrie. Dans le cadre de la vérification de propriétés de sûreté de fonctionnement, l'établissement de protocole de test résulte d'une analyse préalable, basée sur la recherche des évènements indésirables. La couverture se limite donc au champ de fautes et d'erreurs ciblé par cette analyse ; c'est pourquoi, elle se doit d'être la plus exhaustive possible. Dans (Myers, 1979; Beizer, 1990), des outils pour aider à la spécification des protocoles de tests sont présentés.

De nos jours, les méthodes de tests, bien que toujours requises, ne permettent pas à elles seules de répondre aux exigences de sûreté de fonctionnement exprimées dans les standards en usage.

4.1.1.3 La vérification en ligne

La vérification en ligne (Havelund et Rosu, 2001b; Havelund et Goldberg, 2008; Leucker et Schallhart, 2009) permet de détecter des erreurs pendant le fonctionnement d'un système.

Tout comme la vérification de modèles, elle appartient à la classe des méthodes formelles. Le principe de fonctionnement est similaire à celui décrit pour la vérification de modèles (Figure 4.1), à la différence que *seule l'exécution en cours (i.e. une trace d'exécution notée σ) est vérifiée* (i.e. pour une propriété ϕ, on cherche à décider si $\sigma \models \phi$). La vérification en ligne est réalisée pendant l'exécution du système et requiert l'introduction de composants logiciels supplémentaires appelés *moniteurs*.

Définition 4.3. *Un moniteur observe le comportement du système par l'interception d'évènements de la trace d'exécution courante. Il détermine ensuite si cette observation répond favorablement à l'ensemble des spécifications (Peters, 1999; Leucker et Schallhart, 2009).*

Un moniteur au sens de la vérification en ligne satisfait les propriétés d'*impartialité*, c'est-à-dire qu'une exécution n'est pas déclarée vraie ou fausse s'il existe une suite qui contredira le résultat, et d'*anticipation*, c'est-à-dire qu'une fois que toutes les suites possibles d'une exécution conduisent au même verdict, ce verdict doit être obtenu au plus tôt. En ligne, les moniteurs déduisent du comportement de l'application les valeurs de vérité de l'ensemble des propositions atomiques utilisées pour spécifier les propriétés, et statuent donc sur le respect de ces dernières. Après la détection d'une erreur, le système doit être notifié et réagir en conséquence via un mécanisme dédié à l'isolation et au recouvrement. En ce sens, la vérification en ligne peut être vue comme un détecteur d'erreurs, intégré à un mécanisme de tolérance aux fautes tel que présenté dans la Section 2.4 (page 35).

4.1.2 Positionnement de la vérification en ligne

La vérification en ligne n'est pas une alternative aux tests ou à la vérification de modèle (Leucker et Schallhart, 2009). Nous présentons dans les prochaines sections une comparaison entre ces approches, et en quoi elles sont complémentaires.

4.1.2.1 La vérification en ligne versus la vérification de modèles

Sur le plan théorique, le problème de la vérification de modèles et celui de la vérification en ligne sont proches puisqu'ils relèvent tous deux de la théorie des langages formels. La vérification de modèles se réduit au problème de l'inclusion de langages (le langage obtenu par le produit du modèle et de la propriété est-il vide ?), tandis que la vérification en ligne se réduit au problème de l'appartenance d'un mot à un langage.

La vérification de modèles est réalisée hors-ligne pendant la phase de développement, tandis que la vérification en ligne est utilisée pendant la phase opérationnelle (même si la synthèse des moniteurs relève d'une étude réalisée dès la phase de développement). On observe quelques différences dans leur couverture. En effet, la vérification de modèles vérifie que tous les comportements possibles du (modèle du) système satisfont les propriétés, alors que la vérification en ligne vérifie uniquement que le comportement observé satisfait les propriétés. Par conséquent, la vérification en ligne se limite aux propriétés pour lesquelles on peut décider à partir d'une observation finie si elles sont violées ou satisfaites, alors que la vérification de modèles peut traiter des propriétés qui nécessitent d'analyser des traces infinies. En revanche, la vérification de modèles couvre uniquement les fautes se produisant pendant la phase de développement, tandis que la vérification en ligne couvre également des fautes d'intégration et opérationnelles.

Enfin, sur le plan pratique, la vérification de modèles étant réalisée pendant la phase de développement, elle requiert une bonne connaissance a priori du système. Il est alors inévitable

de faire des hypothèses sur le comportement réel du système. Il faut donc veiller à ce que ces hypothèses soient réalistes afin que l'écart entre le modèle et l'implémentation finale soit acceptable, et que la vérification soit valable. La vérification en ligne, quant à elle, analyse le système pendant sa phase opérationnelle, ce qui lui permet de prendre en compte tous les aspects non décrits dans les spécifications. Il est même parfois possible d'appliquer cette solution sur des systèmes en *boîte noire* sur lesquels nous avons peu de connaissances. Pour cela, il faut au minimum pouvoir intercepter des évènements entrants ou sortants des composants à surveiller.

4.1.2.2 La vérification en ligne versus le test

Les différences entre la vérification en ligne et les techniques de tests ne sont pas très importantes. La vérification en ligne peut être considérée comme une technique particulière de test par oracle dans laquelle les relations entre les entrées et les sorties du système sont formalisées sous la forme de propriétés, et où l'oracle est construit à partir de ces propriétés. À l'instar de la vérification en ligne où l'on génère des moniteurs, le test par oracle consiste à construire un composant logiciel (l'oracle), que l'on attache au système.

Dans la pratique, la vérification en ligne s'applique souvent à des propriétés invariantes du logiciel, devant être vérifiées quelle que soit la séquence d'entrée. La vérification en ligne est donc une forme de test pour lequel l'unique entrée correspond à la trace observée. En ce sens, elle peut être considérée comme complémentaire au test. À titre d'exemple, dans Leucker et Schallhart (2009), on parle de la vérification en ligne comme une forme de *test passif*. Ce propos est également illustré dans Artho *et al.* (2005) où les auteurs montrent comment on peut combiner les méthodes de génération de tests avec la vérification en ligne.

Enfin, sous certaines conditions, les deux solutions peuvent également être appliquées dans les phases du cycle de vie ultérieure au développement. La vérification en ligne est proposée pour être utilisée durant la phase opérationnelle du système. De la même façon, des composants de tests BiT (*Built in Test*) peuvent également servir pour des diagnostics en-ligne.

4.1.2.3 Bilan

Comme illustré sur la Figure 4.2, les trois mécanismes présentés sont utilisés dans différentes phases du cycle de développement. La vérification en ligne est complémentaire à la vérification de modèles et aux méthodes de test à plusieurs niveaux :

- Même si la vérification de modèles est prouvée correcte pendant la phase de développement (effectuée en phase descendante du cycle en V), des erreurs peuvent être introduites lors des phases ultérieures : implémentation, intégration, production, exploitation ;
- Le plan de test permet de démontrer l'absence des évènements redoutés selon certaines configurations d'entrées prédéfinies tandis que la vérification en ligne couvre les propriétés quelle que soit la séquence d'entrée observée ;
- Dans la démarche de mise en œuvre, la vérification de modèles et les tests permettent de s'assurer que le système issu de la phase de conception est conforme aux spécifications tandis que la vérification en ligne permet d'être averti lorsqu'il ne se comporte plus conformément aux spécifications.

La vérification en ligne est également une réponse aux contraintes liées à la sûreté de fonctionnement. C'est un mécanisme de détection intégré à la démarche de tolérance aux fautes

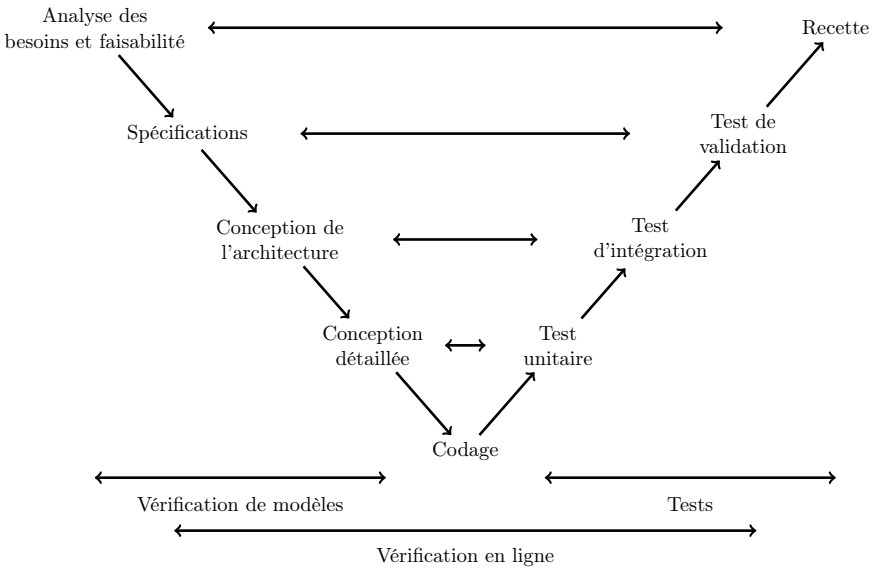


Figure 4.2 – La vérification dans le processus de développement en V

qui contribue à ce que l'application atteigne le niveau de robustesse désiré.

4.2 Architecture d'un mécanisme de vérification en ligne

L'architecture générique d'un mécanisme de vérification en ligne est illustrée sur la Figure 4.3. Il s'agit d'un service ayant pour rôle d'observer une partie du système et de décider la violation ou le respect des spécifications qui lui sont associées.
Le service de vérification en ligne peut s'intégrer à plusieurs niveaux. Par exemple, on peut choisir de le mettre en place au niveau de l'application, du système d'exploitation, ou réparti sur les niveaux.

Le service est composé de trois blocs :

- L'*observateur* capture pas-à-pas l'activité du système pour en déduire son état courant. Pour cela, une légère instrumentation est requise afin que le service puisse intercepter les évènements nécessaires à l'observation. Ceci est réalisé par l'ajout de sondes branchées directement sur les composants à surveiller et connectées au service de vérification en ligne. Il est également possible d'insérer une étape d'identification et de filtrage, ce qui permet d'affiner le tri des évènements ;
- L'*analyseur* prend en entrée les informations provenant de l'observateur. En fonction de l'image de l'état du système extraite par l'observateur et grâce à la connaissance qu'il a sur les évolutions passées, le bloc d'analyse décide du respect des exigences ;
- Le bloc de *traitement* récupère les résultats fournis par l'analyseur et applique le traitement qui en découle. Quand une propriété est satisfaite, il est possible de déclencher un post-traitement spécifique (e.g. changement de mode de l'application). Quand une propriété est violée, ce bloc analyse l'erreur et transmet ses résultats au composant en charge de mettre en œuvre le post-traitement adéquat (i.e. confinement puis diagnostic et/ou recouvrement). Pour faciliter l'analyse post-mortem de la trace ayant conduit à

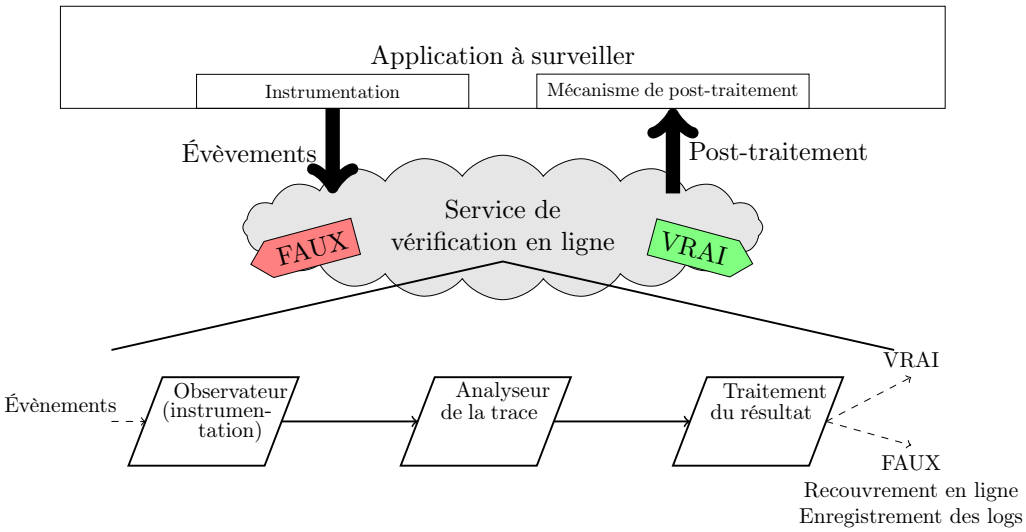


Figure 4.3 – Architecture d'un mécanisme de vérification en ligne

l'erreur, il est parfois possible de la sauvegarder (logs).

L'analyse est effectuée sur un modèle abstrait, dérivé de l'analyse *pas-à-pas* du système réel. La richesse de la trace extraite doit être suffisante pour que le système puisse être analysé ; c'est la raison pour laquelle la définition des *pas* est très importante. L'instrumentation du code de l'application qui permet d'atteindre le niveau de richesse nécessaire à l'analyse de la trace est donc une étape clé.

La vérification en ligne n'est qu'un mécanisme de *détection d'erreurs*. Une détection survient si le comportement observé du système est un comportement *redouté*. La définition exacte des comportements redoutés permet de construire une classification des erreurs et de proposer ensuite les stratégies de confinement et de recouvrement adaptées aux différents types d'erreurs. Rappelons que ces stratégies s'inscrivent dans le processus classique de la tolérance aux fautes.

4.3 État de l'art des mécanismes pour la vérification en ligne

4.3.1 Classification des mécanismes pour la vérification en ligne

Les principes de la vérification en ligne ont déjà été mis en pratique dans de nombreux projets. Dans Watterson et Heffernan (2007), les auteurs présentent les problématiques liées à la mise en place de la vérification en ligne pour les systèmes embarqués. Plusieurs approches existent pour observer et analyser un système. La diversité de ces approches est inhérente à la volonté de mettre en place des mécanismes de vérification en ligne les moins intrusifs possibles et pour lesquels l'impact en termes de surcoûts d'exécution est le plus faible possible. Une première classe de *solutions* est *matérielle*. Par exemple, Tsai *et al.* (1990) ont mis en place un mécanisme de surveillance d'un bus (comportement d'un bus interne) à l'aide de composants matériels. En réduisant l'intrusion des mécanismes d'observation, on vise à maintenir le niveau de performance d'origine du système. Cependant, peu d'architectures matérielles sont prévues

pour autoriser l'ajout de sondes matérielles, et la complexité de ces architectures rend encore plus difficiles les observations. De plus, ce type de solution est dédié à de petits systèmes et ne permet pas de vérifier des propriétés du logiciel à haut niveau.

Nous choisissons donc de nous focaliser sur des *solutions logicielles*, qui sont flexibles puisqu'elles autorisent des interactions directes avec l'application. En revanche, l'impact sur le système initial doit être maîtrisé pour éviter les interférences, notamment quand l'exécution du système de base est perturbée par l'exécution du service de surveillance.

Dans toutes ces solutions, le principe consiste à synthétiser des moniteurs, à les intégrer à l'application, et à s'en servir pour statuer sur l'état du système. Il existe plusieurs approches pour la synthèse des moniteurs. La première est basée sur l'*utilisation de propriétés écrites en logique temporelle* type LTL. Elle permet de synthétiser automatiquement des moniteurs dédiés à ces formules. Le plus souvent, les moniteurs ainsi générés sont sous la forme de machine à états finis. En analysant les formules, l'outil synthétisant les moniteurs déduit la liste des évènements nécessaires à l'observation du système. Un algorithme en-ligne permet, à partir des observations, de faire évoluer les machines à états. La seconde approche s'appuie sur l'*utilisation d'un langage dédié*. L'expressivité du langage facilite la description des moniteurs en permettant d'exprimer clairement toutes les étapes liées à la vérification (évènements à intercepter, variables surveillées). Cette approche permet également d'agir plus finement dans le système, par exemple en déclarant des assertions qui seront intégrées au code source. Il est également possible d'intégrer au langage un noyau de logique temporelle pour faciliter la mise en œuvre de l'algorithme de décision. En ce sens les deux approches peuvent se compléter.

4.3.2 Mécanismes logiciels pour la vérification en ligne basés sur des propriétés écrites en LTL

Les solutions présentées ici reposent sur l'utilisation de la logique LTL pour exprimer les propriétés et synthétiser les moniteurs.

Dans Havelund et Rosu (2001a,b) puis dans Havelund et Rosu (2002), *Havelund* et *Rosu* présentent *JavaPAX*. Il s'agit d'un outil pour la vérification en ligne se basant sur une description des propriétés avec l'outil Maude (Clavel *et al.*, 1999). Les spécifications sont utilisées pour exprimer des propriétés écrites en logique temporelle LTL. Les moniteurs (appelés ici observateurs) sont générés sur la base de ces propriétés. Pour que le système puisse être observé, il faut l'instrumenter. Ceci est réalisé par l'ajout de portions de code qui émettent des évènements à destination du logiciel PAX. Ceci permet d'analyser les erreurs liées à la violation des propriétés issues des spécifications fonctionnelles (le système réagit comme prévu) et également des erreurs de programmation indépendantes (interblocage, famine). Un module d'analyse des chemins ayant conduit à l'erreur permet de faciliter le diagnostic.

Dans Bucur (2012), *Bucur* présente un service pour la vérification en ligne de systèmes utilisant le système d'exploitation *TinyOS*. Là encore, la solution se base sur une instrumentation du code permettant de signaler des évènements au composant de vérification. Les moniteurs sont des machines à états finis, dérivées des propriétés LTL. Les surcoûts temporels et mémoires sont également évalués. Les résultats sur une plateforme expérimentale montrent que l'impact est négligeable sur la RAM et sur le temps CPU, mais que l'impact en ROM représente $5,5\%$ de l'espace total.

4.3.3 Mécanismes logiciels pour la vérification en ligne basés sur l'utilisation de langages dédiés

Les travaux suivants sont basés sur la synthèse d'un moniteur à partir d'une description hors-ligne de son comportement. Cette description peut être effectuée par l'utilisation d'un langage dédié (*Domain Specific Language* – DSL).

Définition 4.4. *Un DSL est un langage de programmation ou de spécification qui offre au travers de notations appropriées une puissance expressive axée sur, et limitée à, un domaine spécifique (Deursen* et al.*, 2000).*

Ce type de langage est donc conçu pour s'adapter exclusivement aux contraintes du domaine d'application ciblé. L'objectif est de faciliter la description et la spécification de propriétés du domaine.

Dans (Rosenblum, 1995), les auteurs insèrent des assertions dans le programme par le biais d'un outil appelé *APP – Annotation PreProcessor*, développé en C. Cet outil permet de déclarer des assertions, qui sont ensuite automatiquement insérées dans le code par le préprocesseur. Quatre types d'assertions peuvent être utilisées : *Vérification des entrées (assume)* est utilisée pour spécifier des préconditions d'une fonction ; *Vérification des sorties (promise)* est utilisée pour spécifier des postconditions d'une fonction ; *Assertions de retour (assert)* est utilisée pour spécifier des contraintes sur les valeurs retournées et *Assertion intermédiaire (intermediate)* est utilisée pour spécifier des contraintes sur des valeurs intermédiaires. Les erreurs peuvent être rapportées par l'utilisation de macros. Cette approche est particulièrement adaptée pour effectuer des contrôles sur l'algorithme exécuté par une fonction. En revanche, elle n'est pas adaptée à la surveillance des flots de contrôle ou de données dans une architecture à composants et/ou multitâche.

Le système MaC (*Monitoring and Checking*, Kim *et al.* (2002)), a pour but de surveiller des systèmes codés en Java. La spécification des règles est écrite dans le langage *MEDL – Meta Event Definition Language*. Ce langage est une extension de la logique LTL, c'est-à-dire qu'il rend plus facile l'écriture des propriétés en LTL tout en gardant l'expressivité. En particulier, les utilisateurs peuvent définir des variables auxiliaires qui permettent de représenter l'état du système. L'écriture des propriétés peut reposer directement sur ces variables, parfois qualifiées de *variables d'états*. Le moniteur quant à lui est décrit dans le langage *PEDL – Primitive Event Definition Language* – qui définit quels sont les évènements capturés et la manière dont ils sont interprétés par le moniteur. Les informations recueillies par le moniteur peuvent être sauvegardées pour être analysées hors-ligne. Dans la pratique, aucun couplage n'existe entre les scripts MEDL et PEDL.

Dans (Pike *et al.*, 2010), les auteurs présentent *copilot*, un langage embarqué en Haskell. Les travaux présentés ciblent les systèmes embarqués temps réel critiques. Les propriétés exprimées dans *copilot* permettent de synthétiser des moniteurs, à intégrer dans l'application. Concrètement, des variables globales sont périodiquement lues puis analysées afin de déterminer si les propriétés exprimées sont correctes ou non. Un des principaux problèmes mis en avant dans l'article repose sur l'ordonnancement. En effet, pour éviter que l'introduction du moniteur ne perturbe le comportement temporel du système, le moniteur possède son propre ordonnanceur (aussi généré via la description *copilot*). Cet ordonnanceur peut être utilisé pour exploiter les temps creux du processeur ou peut être mis en œuvre sur un autre processeur. Dans le dernier cas, des mécanismes de synchronisation doivent être mis en place.

L'article Bauer *et al.* (2006b) introduit un langage de spécification temporelle appelé *SALT*,

très proche de la logique temporelle LTL, et destiné à l'écriture des propriétés que le système doit vérifier. Le moniteur ainsi généré est exécuté en parallèle du système selon une approche réflexive (Maes, 1987). Dans le cas d'une détection de faute, la solution proposée permet également d'assurer les étapes de diagnostic et de confinement dans un contexte automobile AUTOSAR. Contrairement à notre approche, les propriétés sont écrites à haut niveau sur des variables *fonctionnelles* de l'application pour vérifier son comportement, mais elles ne permettent pas de prendre en compte des évènements issus du noyau.

Dans D'Angelo *et al.* (2005), les auteurs présentent *LOLA*, un langage de spécification pour la surveillance en ligne et hors-ligne de systèmes embarqués conçus selon l'approche synchrone. Une spécification *LOLA* décrit l'exécution d'un flux de sortie à partir d'un ensemble de flux d'entrée (par l'utilisation d'un ensemble d'équations où les sorties dépendent des variables d'entrées). Les algorithmes sont en charge de résoudre en ligne les équations liées aux spécifications.

4.4 Périmètre de l'étude

4.4.1 Objectifs et contraintes

Notre étude vise à construire un mécanisme à même de détecter les fautes activées par la non-maîtrise de l'ordonnancement dans un système multicœur. En particulier, nous cherchons à vérifier que les exigences sur les flots de données inter-tâches sont toujours satisfaites (c.f. Section 3.1, page 40).

La surveillance des propriétés inter-tâches sur les flots de données peut être résolue par l'utilisation d'une approche formelle. Nous souhaitons mettre en place un service de vérification en ligne adapté aux systèmes embarqués temps réel multicœur. En particulier, nous ciblons une preuve de concept dans le contexte embarqué automobile.

Ce contexte d'étude nous impose de respecter les nombreuses contraintes qui y sont liées :

- Le service de vérification en ligne doit avoir le moins d'impact possible sur l'ordonnancement afin que le système soit toujours en mesure de respecter les contraintes temporelles. L'interception des évènements et l'analyse du chemin d'exécution doivent être réalisées par des algorithmes efficaces (le plus rapide possible, en temps constant ou linéaire) et de manière déterministe ;
- L'empreinte mémoire du service de vérification en ligne doit être compatible avec les contraintes du domaine (peu de RAM et ROM) ;
- Le service de vérification en ligne doit pouvoir s'interconnecter avec des composants applicatifs fournis sous forme de code objet, provenant par exemple de sous-traitances diverses ;
- Dans un contexte industriel, chaque composant doit être qualifié (dans certains cas, certifiés) en référence à un standard. Dans notre cas, le service de vérification doit pouvoir s'insérer dans une architecture logicielle AUTOSAR (AUTOSAR, 2013) et répondre aux exigences de la norme de sûreté de fonctionnement ISO 26262 en vigueur dans l'automobile (ISO 26262, 2011) ;
- Enfin, notre approche doit s'insérer dans les processus de développement industriel.

4.4.2 Approche suivie

4.4.2.1 Intégration dans le processus de développement

Pour intégrer la vérification en ligne dans le processus de développement, nous suivons l'approche décrite par la Figure 4.4. Sur cette figure, la branche classique de développement se trouve en dehors de la zone rectangle pointillée tandis que notre domaine d'étude est dans cette zone.

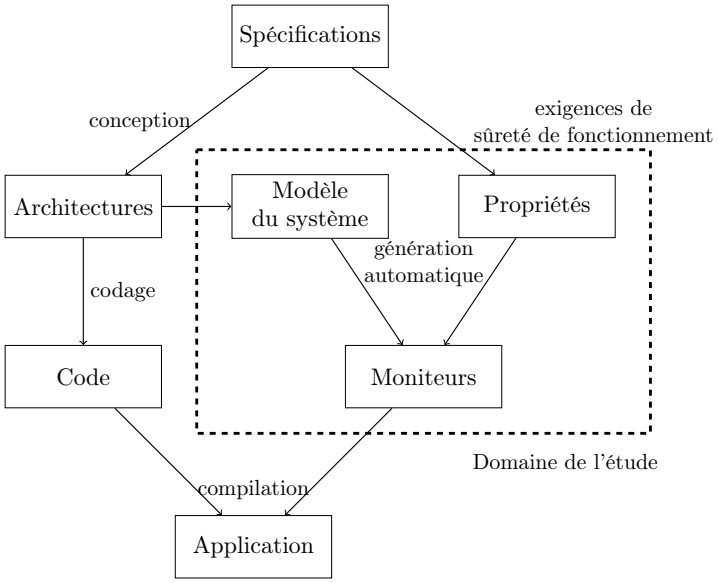


Figure 4.4 – Synthèse des moniteurs dans le cycle de développement

Les spécifications déjà utilisées pour construire les architectures fonctionnelle et logicielle sont également utilisées pour extraire les exigences de sûreté de fonctionnement. Le modèle du système, quant à lui, peut être dérivé des architectures (fonctionnelle, logicielle et opérationnelle). Pour permettre la synthèse des moniteurs, nous avons créé l'outil *Enforcer*. Pour l'utiliser, nous créons un DSL nous permettant de formaliser les deux entrées précédentes. La synthèse des moniteurs est effectuée hors-ligne en deux temps :

- Une propriété exprimée en logique LTL permet de construire un moniteur intermédiaire à même de la surveiller (c.f. Section 5.2, page 66) ;
- Nous composons le moniteur intermédiaire avec un modèle du système pour obtenir le moniteur final sous la forme d'un automate à états fini (i.e. une machine de Moore) (c.f. Section 5.3, page 72). Cet automate est utilisé en-ligne pour la vérification.

Finalement, la vérification en ligne que nous décrivons peut être vue comme un détecteur d'erreurs au sens de la tolérance aux fautes. En particulier, notre approche pour la détection d'érreurs est basée sur la diversification des spécifications.

4.4.2.2 Choix d'implantation du service de vérification en ligne

Pour atteindre nos objectifs et en particulier maîtriser le surcoût temporel induit, nous choisissons d'implanter le service de vérification en ligne dans le noyau du système d'exploita-

tion temps réel.

Il serait également possible d'implanter le service au niveau du RTE, point de passage obligé de toutes les interactions entre les couches basses et l'application. Les travaux présentés dans (Lu *et al.*, 2009; Patzina *et al.*, 2013) mettent en œuvre des mécanismes à ce niveau. Ce choix a quelques inconvénients. D'une part, il n'est pas systématique de disposer d'une zone mémoire protégée à ce niveau ; d'autre part, le surcoût temporel est plus important qu'avec une insertion dans le noyau.

Pour l'illustrer, considérons une implantation du service dans le RTE. Pour contrôler la concurrence de l'accès aux moniteurs, des sections critiques doivent être mises en place. Dans AUTOSAR OS, les mécanismes d'exclusion mutuelle intracoeur (protocole iPCP ou masquage des interruptions) et intercoeur (spinlock) sont différents. La mise en place de cette exclusion mutuelle coûte donc 4 appels systèmes. Selon le type d'évènement à analyser, des appels systèmes supplémentaires peuvent s'ajouter. Ainsi, pour identifier un évènement de communication, il faut identifier l'émetteur et le buffer cible, ce qui nécessite 2 appels systèmes supplémentaires (c.f. Figure 4.5 (a)).

Si le mécanisme est intégré au noyau (cf. Figure 4.5 (b)) :

- Le contrôle de concurrence se fait à l'aide des mêmes mécanismes, mais sans aller-retour entre le mode noyau et le mode utilisateur ;
- L'interrogation des structures de données du noyau par le bloc d'identification des évènements se fait sans aller-retour entre le mode noyau et le mode utilisateur.

Ce choix permet donc d'obtenir une implantation plus efficace. Il peut être réalisé de deux manières :

- En créant un appel système dédié ;
- En instrumentant le code des appels systèmes existant qui correspondent à des occurrences d'évènement intéressant.

Puisque nous nous intéressons aux évènements inter-tâches qui se traduisent tous par des appels systèmes, nous choisissons la seconde approche.

```
GetTaskID()
GetBufferID()
DisableInterupt()
  GetTheLock()
    /* monitoring */
  ReleaseTheLock()
EnableInterupt()
```

(a) Hors du noyau (6 appels systèmes)

```
StartOSservice()
  getTaskID()
  getBufferID()
  /* monitoring */
```

(b) Dans le noyau (1 appel système)

Figure 4.5 – Deux approches pour l'implémentation du service de vérification en ligne

La mise en œuvre est effectuée dans *Trampoline*, un système d'exploitation temps réel (RTOS) libre compatible AUTOSAR (Béchennec *et al.*, 2006). Ce RTOS est développé dans l'équipe *Système Temps Réel* de l'IRCCyN. Un outil nommé *Enforcer* a été conçu pour la synthèse des moniteurs (c.f. Section 6.2.1, page 79) que l'on injectera dans *Trampoline*.

4.4.2.3 Classe d'évènements à intercepter

Nous avons développé un prototype qui permet de surveiller les flots de données inter-tâches dans une architecture multicœur AUTOSAR. Ce prototype intercepte les évènements du système liés à la communication (e.g. une lecture ou une écriture dans un buffer de communication). Tous les autres évènements envoyés vers le système d'exploitation pourraient être surveillés de la même manière. À titre d'exemple, l'activation ou la terminaison d'une tâche pourrait être intercepté de la même manière, ce qui permettrait alors de surveiller des propriétés relatives aux flots de contrôle du système.

Puisque la configuration d'un système d'exploitation temps réel de type AUTOSAR est statique, les objets système (tâches, messages, etc.) sont créés à la compilation. Les informations nécessaires à cette description peuvent donc être exploitées pour la synthèse des moniteurs. Il est alors possible d'identifier hors-ligne les évènements qu'il faudra intercepter, ce qui permet de générer un filtre facilitant, en-ligne, l'étape d'analyse des évènements interceptés.

Les évènements sont interceptés et traités au niveau des appels systèmes qui correspondent aux services de communication présents dans le système d'exploitation. Quand une propriété est jugée $vrai$ (resp. $fausse$), des mécanismes peuvent être utilisés pour exécuter du code utilisateur (e.g. une stratégie de recouvrement dans le cas $faux$). Ce code est accroché à un *hook* similaire à ceux déjà proposés par AUTOSAR.

Chapitre 5

Synthèse de moniteurs à partir d'une formule LTL

Sommaire

5.1 Fondements théoriques

5.1.1 Alphabets, mots et langages

La théorie des langages a pour objectif de décrire et manipuler les langages formels.

Définition 5.1. *Un alphabet est un ensemble fini non vide de symboles.*

Définition 5.2. *Une séquence ordonnée de symboles appartenant à un alphabet Σ est appelée mot.*

La séquence finie $w = (a_1, a_2, ..., a_n) \in \Sigma^n$ de n symboles est un mot de longueur n aussi noté $a_1a_2....a_n$. Ce mot peut également être vu comme une fonction $w : [1..n] \to \Sigma$. Le symbole ϵ est utilisé pour dénoter le mot vide, de longueur 0. L'ensemble des mots définis sur Σ est noté Σ^*.

On définit également la notion de mot infini, ou ω-mot sur Σ : un ω-mot est une fonction de $\mathbb{N}$ sur Σ. L'ensemble des ω-mots définis sur Σ est noté Σ^ω.

Pour la suite de ce chapitre, quelques définitions supplémentaires sont nécessaires :

Définition 5.3. *Soit $u \in \Sigma^*$ et $v \in \Sigma^\omega$. Le mot $w \in \Sigma^\omega$ est la concaténation de u et v ssi $\forall i, i \in [1..|u|] \to w(i) = u(i)$ et $\forall j, j > |u| \to w(j) = v(j - |u|)$. On note $w = u.v$.*

Définition 5.4. *Soit $w \in \Sigma^\omega$ et $u \in \Sigma^*$. On dit que u est un préfixe de w ssi $\exists v \in \Sigma^\omega$ tel que $w = u.v$*

Définition 5.5. *Un langage L sur Σ est une partie de Σ^*.*

Définition 5.6. *Un ω-langage L sur Σ est une partie de Σ^ω.*

5.1.2 Définitions de base sur les automates

Définition 5.7. *Soit un automate A et un langage $\mathbb{L}$. On note $\mathbb{L}(A)$, l'ensemble des mots reconnus par l'automate A.*

Il existe plusieurs types d'automates. Avant de les définir, nous présentons les systèmes de transitions et les structures de Kripke.

Définition 5.8. *Un système de transitions ST, est défini par le tuple $ST = (\Sigma, Q, q_0, \delta)$ tel que :*
- *Σ est un alphabet ;*
- *Q est un ensemble non vide (potentiellement infini) d'états de ST ;*
- *$q_0 \in Q$ est l'état initial ;*
- *$\delta \subseteq Q \times \Sigma \times Q$ est l'ensemble des transitions.*

Une structure de Kripke (Kripke, 1963) est un système de transitions enrichi d'un ensemble de propositions atomiques dont la valeur change selon l'état.

Définition 5.9. *Une structure de Kripke KS, est définie par le tuple $KS = (\Sigma, Q, q_0, \delta, \lambda)$ tel que :*
- *Σ est un alphabet ;*
- *Q est un ensemble non vide d'états de KS ;*

- $q_0 \in Q$ *est l'état initial ;*
- $\delta \subseteq Q \times \Sigma \times Q$ *est l'ensemble des transitions ;*
- $\lambda : Q \longrightarrow 2^{AP}$ *est la fonction injective d'étiquetage qui donne l'ensemble des propositions atomiques vraies à un état donné.*

Parmi les différents automates, nous distinguons les automates finis non déterministes et déterministes, pouvant agir sur des mots finis ou infinis. Nous introduisons également les automates de Büchi et les machines de Moore.

Définition 5.10. *Un automate fini non déterministe (NFA) A est un automate fini défini par le tuple* $A = (\Sigma, Q, Q_0, \delta, T)$ *tel que :*

- Σ *est un alphabet ;*
- Q *est un ensemble non vide d'états de A ;*
- $Q_0 \subseteq Q$ *est l'ensemble des états initiaux ;*
- $\Delta \subseteq Q \times \Sigma \times Q$ *est la relation de transition de A. On peut avoir* $q \xrightarrow{\epsilon} q'$ *;*
- $T \subseteq Q$ *est l'ensemble des états finaux.*

Un mot $v = v_1....v_n \in \Sigma^*$ est une séquence d'actions. La succession des actions et des états est un chemin défini par $\rho = q_0 v_1 q_1 ... q_{n-1} v_n q_n$, où q_0 est un état initial et $\forall i \in \mathbb{N}$, $q_i \xrightarrow{v_i} q_{i+1} \in \Delta$. Un mot est accepté si $q_n \in T$.

L'automate fini déterministe est un cas particulier de l'automate fini non déterministe, pour lequel la relation de transition devient une fonction de transition (i.e. $\forall q \in Q, v \in \Sigma, |\delta(q, v)| = 1$ et q_0 est unique).

Définition 5.11. *Un automate fini déterministe (DFA) A, est un automate fini défini par le tuple* $A = (\Sigma, Q, q_0, \delta, F)$ *tel que :*

- Σ *est un alphabet ;*
- Q *est un ensemble non vide d'états de A ;*
- $q_0 \in Q$ *est l'état initial ;*
- $\delta : Q \times \Sigma \longrightarrow Q$ *est la fonction de transition de A. Si* $q \in Q$, $q' \in Q$ *et* $\sigma \in \Sigma$, $\delta(q, \sigma) = q'$. *on note* $q \xrightarrow{\sigma} q'$. *En particulier,* $q \xrightarrow{\epsilon} q$ *;*
- $T \subseteq Q$ *est l'ensemble des états finaux.*

Les NFA et DFA permettent de reconnaître une sous-classe des langages qu'on appelle les *langages réguliers*. Il existe un algorithme de complexité exponentielle qui permet de passer d'un NFA à un DFA, équivalent en termes d'égalité de langage.

L'automate de Büchi est un automate fini qui accepte des mots infinis.

Définition 5.12. *(Büchi, 1962) Un automate de Büchi non déterministe (NBA) A est défini par le tuple* $A = (\Sigma, Q, Q_0, \delta, T)$ *tel que :*

- Σ *est un alphabet fini défini sur* (AP) *;*
- Q *est un ensemble non vide d'états ;*
- $Q_0 \subseteq Q$ *est un ensemble d'états initiaux ;*
- $\delta \subseteq Q \times \Sigma \times Q$ *est la relation de transition finie de A ;*
- $T \subseteq Q$ *est l'ensemble des états terminaux (dits acceptants).*

Considérons le chemin $\rho = q_0 v_1 q_1 ...$, où q_0 est l'état initial. On note $Inf(\rho)$, les états visités infiniment souvent. ρ est acceptée si $Inf(\rho) \bigcap T \neq 0$. Si au moins un chemin de ce type existe, le langage reconnu par l'automate de Büchi est dit *non vide*.

Les automates de Büchi permettent de reconnaître une sous-classe des ω-langages qu'on appelle ω-langage régulier, qui sont la limite des langages réguliers (le langage des préfixes finis d'un ω-langage régulier est un langage régulier).

Une machine de Moore est un automate fini qui émet des sorties dont les valeurs dépendent exclusivement de l'état courant. À chaque état, on fait correspondre une étiquette de sortie qui sera émise quand l'état sera atteint. Ce type d'automate est bien adapté pour la vérification en ligne, la sortie étant utilisée pour communiquer le statut courant de la vérification. Formellement, une machine de Moore est un automate fini (déterministe) auquel on ajoute une fonction de sortie λ.

Définition 5.13. *Une machine de Moore A est définie par le tuple $A = (\Sigma, S, Q, q_0, \delta, \lambda)$ tel que :*
- *Σ est l'alphabet ;*
- *S est l'alphabet de sortie ;*
- *Q est l'ensemble non vide d'états de A ;*
- *$q_0 \in Q$ est l'état initial ;*
- *$\delta : Q \times \Sigma \longrightarrow Q$ est la fonction de transition ;*
- *$\lambda : Q \longrightarrow S$ est la fonction de sortie.*

Travailler sur des automates impose de réaliser certaines opérations. Parmi ces opérations, on dispose de tests d'accessibilité d'états.

Définition 5.14. *Un état $q_j \in Q$ est accessible à partir d'un état $q_i \in Q$ s'il existe au moins un chemin dans l'automate qui mène de l'état q_i à q_j.*

En particulier, l'état q_j est *accessible* s'il est accessible depuis un état initial.

Enfin, les produits cartésien et synchonisé d'automates sont également utilisés dans nos travaux.

Définition 5.15. *Soit $A' = (\Sigma, Q', Q'_0, \Delta', T')$ et $A'' = (\Sigma, Q'', Q''_0, \Delta'', T'')$, deux automates à états finis définis sur le même alphabet. Le produit cartésien de $A' \times A''$ est l'automate $A = (\Sigma, Q, Q_0, \Delta, T)$ tel que*
- *$Q = Q' \times Q''$;*
- *$Q_0 = Q'_0 \times Q''_0$;*
- *$T = T' \times T''$;*
- *$\Delta = \{((e'_i, e''_j), v, (e'_k, e''_l)) \mid (e'_i, v, e'_k) \in \Delta'$ et $(e''_j, v, e''_l) \in \Delta''\}$.*

Définition 5.16. *Soit $A' = (\Sigma, Q', Q'_0, \Delta', T')$ et $A'' = (\Sigma, Q'', Q''_0, \Delta'', T'')$, deux automates à états finis définis sur le même alphabet. Le produit synchonisé de $A' \times A''$ est l'automate $A = (\Sigma, Q, Q_0, \Delta, T)$ tel que*
- *$Q = Q' \times Q''$;*
- *$Q_0 = Q'_0 \times Q''_0$;*
- *$T = T' \times T''$;*

– $\Delta = \{((e'_i, e''_j), v, (e'_k, e''_l)) \mid (e'_i, v, e'_k) \in \Delta' \text{ et } (e''_j, v, e''_l) \in \Delta''\} \cup$
$\{((e'_i, e''_j), v, (e'_k, e''_j)) \mid (e'_i, v, e'_k) \in \Delta' \text{ et } \forall (e''_j, u, e''_l) \in \Delta'',\ u \neq v\} \cup$
$\{((e'_i, e''_j), v, (e'_i, e''_l)) \mid (e''_j, v, e''_l) \in \Delta'' \text{ et } \forall (e'_i, u, e'_k) \in \Delta',\ u \neq v\}.$

5.1.3 La logique temporelle LTL

Les logiques temporelles (ou logiques modales) sont une extension de la logique propositionnelle. Elles intègrent des nouveaux opérateurs qui permettent de raisonner sur le comportement (passé et futur) d'un système discret.

En 1977, Amir Pnueli introduit l'utilisation de ces logiques pour la vérification formelle des programmes (Pnueli, 1977) : une formule est alors utilisée pour décrire les règles auxquelles le programme doit se conformer.

La logique de temps linéaire (LTL – *Linear Temporal Logic*) permet d'exprimer des propriétés portant sur des chemins individuels (issus de l'état initial) du programme. Puisque la vérification en ligne ne s'intéresse qu'à l'exécution courante d'un système, c'est-à-dire à un chemin, la logique LTL est adaptée. Une propriété de ce type est définie sur un ensemble de *propositions atomiques* que l'on note AP, combinées à l'aide d'un ensemble d'opérateurs modaux (*toujours, un jour, immédiatement après, ...*) et de connecteurs de logiques (*non, et, ou, implique*).

Une formule établie sur l'ensemble AP prend ses valeurs dans l'ensemble $\{vrai, faux\}$.

5.1.3.1 Syntaxe de LTL

Considérons un ensemble de propositions atomiques AP. L'ensemble des formules LTL sur AP est défini inductivement comme suit :

– Tout élément p de AP est une formule LTL ;
– $\top$ (i.e. vrai) et $\bot$ (i.e. faux) sont des formules LTL ;
– Si ϕ et ψ sont deux formules LTL, alors, $\neg\phi$, $X\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi U \psi$ sont aussi des formules LTL.

5.1.3.2 Sémantique de LTL

Une formule LTL est évaluée sur un mot de $\Sigma = 2^{AP}$, c'est-à-dire l'ensemble des parties de AP. On note $\sigma = s_0 s_1 s_2 s_3 .... s_i .... \in \Sigma^*$ (resp. Σ^ω), une séquence finie de taille $|\sigma|$(resp. séquence infinie). On pose $\sigma, i = s_i$ le i^{me} élément de σ. $\sigma_i = s_i s_{+1} ...$ est la séquence finie (resp. infinie) commençant par le i^e élément. Considérons maintenant ϕ et ψ, deux formules LTL sur AP. La relation de satisfaction $\sigma, i \models \phi$ (i.e. σ commençant à l'index i satisfait la propriété ϕ) est définie comme suit :

– $\sigma, i \models \top$;
– $\sigma, i \models p$ ssi $p \in \sigma(i)$;
– $\sigma, i \models \neg\phi$ ssi $\sigma, i \not\models \phi$;
– $\sigma, i \models \phi \vee \psi$ ssi $\sigma, i \models \phi$ ou $\sigma, i \models \psi$;
– $\sigma, i \models \phi \wedge \psi$ ssi $\sigma, i \models \phi$ et $\sigma, i \models \psi$;
– $\sigma, i \models X\phi$ ssi $i + 1 \leq |\sigma| - 1$ (pour les mots finis) et $\sigma, i + 1 \models \phi$;
– $\sigma, i \models \phi\, U \psi$ ssi $\exists j$ tel que : $\forall k \mid i \leq k < j$, $\sigma, k \models \phi$ et $\sigma, j \models \psi$.

$\sigma \models \phi$ ssi $\sigma, 0 \models \phi$.

À partir des opérateurs déjà introduits, deux autres peuvent être déduits :
- F (aussi noté *finally*) est l'opérateur *fatalement*. Si ϕ est une propriété LTL définie sur l'ensemble AP, $F\phi = \top\ U\ \phi$ demande que ϕ devienne vraie en un nombre fini d'étapes ;
- G (aussi noté *always*) est l'opérateur *toujours*. Si ϕ est une propriété LTL définie sur l'ensemble AP, $G\phi = \phi\ U\ \bot$ demande que ϕ soit toujours vraie.

Enfin, la combinaison $GF\phi$ traduit que dans un chemin, ϕ sera vraie une infinité de fois et la combinaison $FG\phi$ signifie qu'à partir d'un état, ϕ deviendra vraie et restera toujours vraie.

Il existe des techniques pour traduire une formule LTL en un automate de Büchi non déterministe qui reconnaît le même ω-langage régulier, en passant par la construction d'un automate de Büchi généralisé. Dans nos travaux, nous utiliserons ensuite un DFA reconnaissant le langage des préfixes finis, pour la construction du moniteur.

5.2 Processus de génération d'un moniteur à partir d'une formule LTL

5.2.1 Classification des propriétés LTL pour la vérification en ligne

La traduction d'une propriété LTL en automate de Büchi permet de disposer d'un critère d'acceptation général sur des chemins infinis. En revanche, un moniteur, tel que défini pour la vérification en ligne, doit être capable de conclure $vrai$ ou $faux$ en un temps fini, c'est-à-dire sur un préfixe d'un chemin infini. Il n'est pas toujours possible de statuer sur le respect ou non d'une propriété à partir d'un chemin fini évalué pas-à-pas puisque la suite de l'exécution peut conduire à l'un ou l'autre des verdicts. Bauer *et al.* (2006a, 2007) proposent une sémantique LTL_3, identique à la sémantique LTL, mais permettant d'enrichir la table de vérité de sortie (i.e. l'ensemble des conclusions possibles lors de l'analyse d'une propriété) pour prendre en compte le manque d'information sur le préfixe observé.

La table de vérité de LTL_3 est notée $\mathbb{B}_3 = \{\bot, ?, \top\}$. Une formule sera évaluée à $\top$ ($vraie$) si pour toutes les suites possibles de l'exécution, la propriété est évaluée à $\top$. Une formule sera évaluée à $\bot$ ($faux$) si pour toutes les suites possibles de l'exécution, la propriété est évaluée à $\bot$. Enfin, dans tous les autres cas, la propriété sera évaluée à ? ($inconclusive$), car l'analyse du chemin ne permet pas de conclure sur la satisfaction ou la violation de la propriété.

Dans Falcone *et al.* (2009, 2010), les auteurs proposent la classification des formules LTL illustrée sur la Figure 5.1. Cette classification fait apparaître 3 classes de propriétés.

La première classe comprend les propriétés de *sûreté* et de *garantie*, chacune ayant une table de vérité de type $\mathbb{B}_2$. Pour les propriétés de *sûreté*, c'est la table $\mathbb{B}_2^{\bot} = \{?, \bot\}$. Par exemple, considérons $\phi = G\neg p$, une propriété de *sûreté* qui vise à garantir qu'un évènement redouté p ne se produit jamais. Dès que p survient, la propriété est $\bot$, mais tant que p n'est pas survenu, la propriété est ? puisque nous ne pouvons pas exclure que p survienne dans le futur. De la même manière, les propriétés de *garantie* agissent sur $\mathbb{B}_2^{\top} = \{\top, ?\}$. Par exemple, considérons $\phi = Fp$. Dès que p survient, la propriété est $\top$, mais tant que p n'est toujours pas survenu, la propriété est ?, car il est toujours possible que p survienne dans le futur.

La seconde classe comprend les propriétés de *réponse* et *persistance*. Les propriétés de cette classe agissent sur la table de vérité $\mathbb{B}_3$ définie par $\mathbb{B}_3 = \{\bot, ?, \top\}$. Une propriété de *réponse* vise à vérifier que des propositions atomiques deviendront vraies après une sollicitation (e.g.

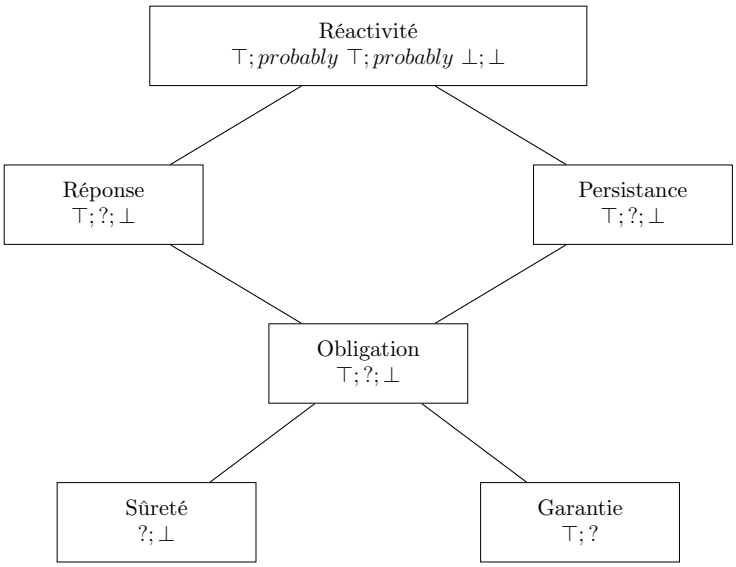


Figure 5.1 – Classification *Safety Progress Falcone* et al. *(2009, 2010)*

$\phi = G(request \rightarrow acknowlegment)$ – une requête est toujours suivie d'une réponse). Une propriété de *persistance* vise à vérifier que des propositions atomiques deviendront vraies et le resteront (e.g. $\phi = FGp$ – *Fatalement, p deviendra vrai et le restera éternellement*).

Les deux dernières classes sont les combinaisons des deux autres. Ainsi, la classe d'*obligation* englobe les propriétés de *sûreté* et de *garantie* et la classe *réactivité* englobe les propriétés de *réponse* et de *persistance*. Par exemple, $\phi = \neg pUq$ (p ne survient pas tant que q n'est pas survenu) est une propriété d'obligation. C'est l'occurrence de p et la position de cette occurrence par rapport à celle de q qui détermine si la propriété est $\top$ ou $\bot$. Si p survient avant q, la propriété est $\bot$, mais si p survient après q, la propriété est $\top$. Tant que p n'est pas survenu, la propriété est ?. D'autre part, les propriétés de *réactivité*, agissent sur la table de vérité définie par $\mathbb{B}_4 = \{\bot, probably\ \bot, probably\ \top, \top\}$. Tant que la valeur finale dépend des évènements futurs, la propriété est évaluée à *probably* $\top$ ou *probably* $\bot$. Par exemple, la propriété $\phi = FGp$ (propriété de persistance) est également une propriété de réactivité, car tant que p reste vraie, la propriété peut être évaluée à *probably* $\top$.

Notons finalement qu'il n'est pas possible de vérifier toutes les propriétés. En effet, seules les propriétés décidables sur un préfixe fini peuvent conduire à un verdict $\top$ ou $\bot$. Une discussion sur ce point est proposée dans la prochaine section.

5.2.2 Génération d'un moniteur à partir d'une propriété exprimée en LTL

Soit ϕ, une formule LTL. Pour vérifier le respect de ϕ, nous devons mettre en place un moniteur sous la forme d'une machine de Moore. Pour chaque état, la fonction de sortie évalue la valeur de la sortie sur l'ensemble défini par la table de vérité $\mathbb{B}_3$. La procédure permettant d'obtenir le moniteur à partir d'une formule LTL est illustrée par la Figure 5.2. Cette procédure est proposée par Bauer *et al.* (2006a, 2011).

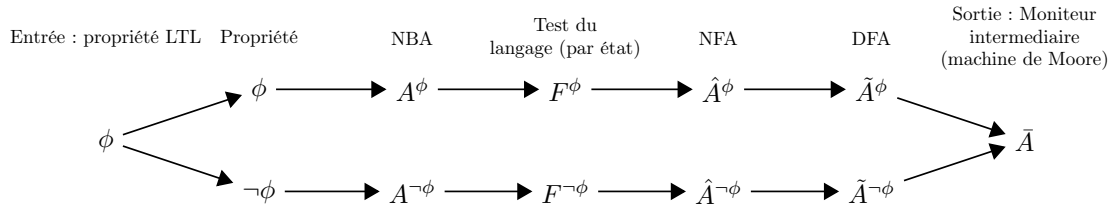


Figure 5.2 – Procédure pour la synthèse des moniteurs (Bauer *et al.*, 2011)

Comme nous le constatons sur la figure, nous considérons à la fois la propriété ϕ et la négation de cette propriété (i.e. $\neg\phi$). Dans le cas général, puisque les tables de vérité de ϕ et $\neg\phi$ sont respectivement sur $\mathbb{B}_2^{\perp}$ et $\mathbb{B}_2^{\top}$, la combinaison des deux branches permet alors d'établir un diagnostic sur $\mathbb{B}_3$. Puisque le traitement est identique pour ϕ et $\neg\phi$, nous ne détaillons qu'une seule de ces branches : celle de ϕ.

À partir de la formule LTL, nous obtenons un automate de Büchi non déterministe $A^{\phi} = (\Sigma^{\phi}, Q^{\phi}, Q_0^{\phi}, \delta^{\phi}, T^{\phi})$ qui reconnait le ω-langage engendré par la formule. L'alphabet du moniteur que l'on souhaite mettre en place est $\Sigma = 2^{AP} = \Sigma^{\phi} \cup \Sigma^{\neg\phi}$. La transformation d'une propriété LTL en un automate de Büchi non déterministe est généralement faite en deux étapes. La première consiste à générer un automate de Büchi généralisé (GBA) à partir de ϕ. La seconde est la traduction du GBA en un automate de Büchi non déterministe.

Dans la littérature, trois techniques permettent de réaliser ces deux étapes. La première solution repose sur la notion de *tableau déclaratif* Fujita *et al.* (1985). Cette solution est simple, mais peu efficace (non-optimisation du nombre d'états) puisqu'elle vise à décomposer la formule en entités élémentaires directement traduites en GBA. Une seconde solution introduite par Gerth *et al.* (1996) est basée sur l'utilisation de tableaux incrémentaux. Une version optimisée est présentée dans Anping *et al.* (2008). Cette version limite la création des états du GBA aux seuls états accessibles, en les ajoutant à l'automate de Büchi au fur et à mesure. À l'état initial, une formule ϕ doit être vraie. S'il existe plusieurs possibilités pour satisfaire ϕ, on introduit plusieurs états. Par exemple, une propriété $\phi = pUq$ sera décrite par deux sous-formules : une devant satisfaire p et l'autre devant satisfaire $X(pUq)$ (i.e. $X(\phi)$). Cette méthode est utilisée par exemple dans le model checker SPIN (Holzmann, 1997). Dans leur solution, Gastin et Oddoux (2001) proposent une autre solution se basant sur l'utilisation des automates alternants. Dans le principe, une formule LTL est d'abord transformée en automate alternant très faible. Cet automate permet ensuite de générer un automate de Büchi non déterministe qui reconnaît le même langage. Le détail de l'implémentation est donné dans Oddoux (2003). Cette solution a conduit à l'élaboration de l'outil *LTL2BA*, que nous utilisons dans nos travaux pour réaliser le passage d'une propriété LTL à un automate de Büchi non déterministe.

L'étape suivante consiste en la synthèse de l'automate fini non déterministe. Ce problème se réduit au test du ω-langage vide appliqué à chaque état du NBA. En pratique, nous regardons pour chaque état q, si au moins un état terminal de l'automate de Büchi peut être atteint une infinité de fois. Nous allons alors chercher à évaluer la fonction intermédiaire $F^{\phi} : Q^{\phi} \longrightarrow \mathbb{B}_2$ définie comme suit :

Définition 5.17. *$F^\phi(q) = \top$ s'il existe un chemin ayant pour origine q, atteignant au moins un état terminal de l'automate A^ϕ, une infinité de fois. Sinon, $F^\phi(q) = \bot$.*

Pour évaluer F^ϕ, nous commençons par rechercher l'ensemble des composantes fortement connexes de l'automate de Büchi. Cela peut être effectué par l'utilisation de l'algorithme de Tarjan (Tarjan, 1971). Parmi cet ensemble, nous ne conservons que les composantes fortement connexes qui possèdent au moins un état terminal de A^ϕ. F^ϕ est ensuite obtenu en effectuant un test d'accessibilité : pour chaque état $q \in Q$, nous recherchons si nous pouvons atteindre au moins une de ces composantes fortement connexes. Si oui, $F^\phi(q) = \top$. Dans le cas contraire, $F^\phi(q) = \bot$ (le langage est vide).

L'automate fini non déterministe qui en résulte est le tuple $\hat{A}^\phi = (\Sigma^\phi, \hat{Q}^\phi, \hat{Q}_0^\phi, \hat{\delta}^\phi, \hat{T}^\phi)$ tel que $\hat{Q}^\phi = Q^\phi$, $\hat{Q}_0^\phi = Q_0^\phi$ et $\hat{\delta}^\phi = \delta^\phi$. $\hat{T}^\phi$ est quant à lui défini comme l'ensemble des états $q \in Q$ tel que $F^\phi(q) = \top$. $\hat{A}^\phi$ est enfin déterminisé pour obtenir $\tilde{A}^\phi = (\Sigma, \tilde{Q}^\phi, \tilde{Q}_0^\phi, \tilde{\delta}^\phi, \tilde{T}^\phi)$. Selon le théorème de Rabin-Scott (Rabin et Scott, 1959), le langage reconnu par $\tilde{A}^\phi$ est le même que celui de $\hat{A}^\phi$. Notons tout de même que puisque l'automate renvoyé par *LTL2BA* n'est pas sur Σ, mais sur Σ^ϕ, il faut au préalable compléter l'automate sur Σ avant de le déterminiser. La relation de transition de l'automate fini déterministe résultant sera donc une fonction totale.

Rappelons que les mêmes étapes ont été effectuées sur $\neg\phi$ en parallèle. Nous avons donc désormais deux automates finis déterministes $\tilde{A}^\phi$ et $\tilde{A}^{\neg\phi}$. Il faut désormais faire le produit cartésien pour obtenir la machine de Moore $\bar{A} = \tilde{A}^\phi \times \tilde{A}^{\neg\phi}$ correspondant au moniteur de ϕ. Cette machine est définie par le tuple $(\Sigma, \bar{Q}, \bar{q_0}, \bar{\delta}, \bar{T}, \bar{\lambda})$ dont les paramètres résultent du produit. En particulier, si q est issue de $\tilde{A}^\phi$ et q' est issue de $\tilde{A}^{\neg\phi}$, la fonction de sortie est définie par $\bar{\lambda} : \bar{Q} \longrightarrow \mathbb{B}_3$, telle que :

- $\bar{\lambda}((q, q')) = \top$ si $q' \notin \tilde{F}^{\neg\phi}$;
- $\bar{\lambda}((q, q')) = \bot$ si $q \notin \tilde{F}^{\phi}$;
- $\bar{\lambda}((q, q')) = ?$ si $q' \in \tilde{F}^{\neg\phi}$ et $q \in \tilde{F}^{\phi}$.

Notons que par construction : $\forall q$, nous n'avons pas $q \in \tilde{F}^\phi$ *et* $q \in \tilde{F}^{\neg\phi}$.

La solution présentée ici permet de générer un moniteur qui réagit sur Σ, c'est-à-dire sur l'ensemble des propositions atomiques qui décrivent l'état du système à un instant donné. Pour générer des moniteurs réagissant sur des évènements du système, il faudra également prendre en compte les informations qui proviennent du système.

Analyse de la complexité

Les algorithmes qui permettent d'obtenir une machine de Moore qui résulte du traitement proposé introduisent une certaine complexité :

- L'obtention d'un automate de Büchi non déterministe à partir d'une formule LTL ϕ de taille $|\phi|$ (nombre de propositions atomiques apparaissant dans ϕ) est de complexité exponentielle en $O(2^{|\phi|})$*danslepirecas* (Oddoux, 2003) ;
- La recherche des composantes fortement connexes a une complexité en O(n) (Tarjan, 1971) ;
- L'obtention des automates finis non déterministes a une complexité en O(n) ;
- La déterminisation d'un automate a une complexité en $O(2^n)$ dans le pire cas ;

– Le calcul du produit cartésien de deux automates de tailles n_1 et n_2 a une complexité en $O(n_1 \times n_2)$.

Globalement, la complexité pire cas de l'algorithme est donc double exponentielle $O(2^{2^{|\phi|}})$.

En pratique, comme nous le verrons dans les sections suivantes, les formules LTL sont petites et les pires-cas sont rarement atteint, ce qui permet d'obtenir des moniteurs de taille acceptable. De plus, la synthèse des machines de Moore est faite hors-ligne et n'ajoute aucune contrainte sur le système.

5.2.3 Applicabilité de la solution

La vérification en ligne des propriétés exprimées en LTL a quelques limites. En effet, il existe des propriétés pour lesquelles il n'est pas possible de statuer sur $\top$ ou $\bot$ en un temps fini. Le moniteur résulte alors en un seul état ?.

Dans Bauer *et al.* (2011), nous pouvons trouver une discussion détaillée sur les classes de propriétés qui peuvent être vérifiées en ligne. Les auteurs présentent un panel de 107 règles réalistes par rapport aux contraintes industrielles. Ces règles correspondent aux patrons présentés dans Dwyer *et al.* (1999). Sur l'ensemble de règles établies, 53 peuvent être vérifiées en ligne. Parmi les propriétés qui ne peuvent pas être vérifiées, plus de la moitié sont des propriétés de la classe *réponse* (Figure 5.1). En effet une propriété du type *À toute requête, nous recevrons une réponse* ne peut pas être vérifiée sur un préfixe fini. Les autres règles ne pouvant pas être vérifiées sont des règles de la classe *garantie*. D'un autre côté, notons que les règles de *sûreté* ont toutes conduit à une synthèse de moniteurs valides. Ce résultat est intéressant puisque la plupart des propriétés requises dans notre étude de cas industriel sont de ce type (c.f. Section 7, page 95).

Enfin, la Table 5.1 montre pour les 53 propriétés pour lesquelles nous pouvons faire de la vérification, le nombre d'états du moniteur.

Nombre de propriétés	Nombre d'états
20	2
20	3
11	4
1	5
1	6

Table 5.1 – Nombre d'états des machines de Moore pour les 53 propriétés vérifiables

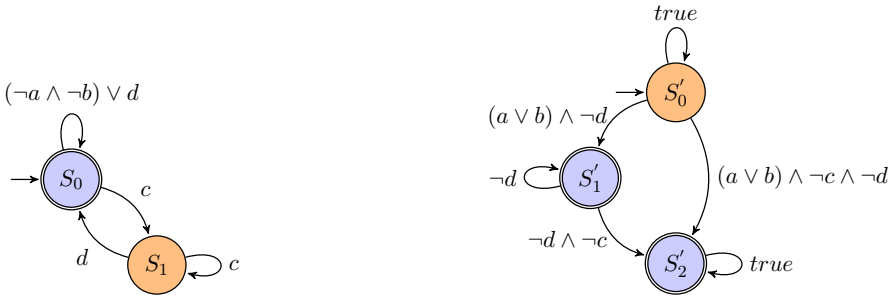
Nous constatons que dans la plupart des cas, les moniteurs sont de petite taille. Dans le chapitre suivant, nous constaterons que le moniteur final que nous injecterons dans un système sera principalement dimensionné par la taille des automates qui modélisent le système. Ces résultats montrent que la vérification en ligne de propriétés LTL est envisageable pour les systèmes temps réel embarqués industriels.

5.2.4 Illustration de la synthèse du moniteur d'une formule LTL par un exemple

Pour illustrer la solution, considérons la formule LTL ϕ définie par $\phi = \mathbf{G}\ ((a \vee b) \implies (c\,\mathbf{U}\,d))$. Une formule de ce type permet de vérifier que dès lors que a ou b devient vrai, c doit être vraie et le rester jusqu'à ce que d devienne vrai à son tour. Pour cet exemple, on définit :

- AP est l'ensemble {a, b, c, d} ;
- $\Sigma = 2^{AP}$ est l'ensemble $\{\epsilon, \{a\}, \{b\}, \{c\}, \{d\}...\{abcd\}\}$.

Les automates de Büchi non déterministes pour ϕ et $\neg\phi$ sont obtenus à l'aide de l'outil *LTL2BA* et sont représentés par la Figure 5.3. Pour simplifier la représentation de l'automate, on étiquette les transitions à l'aide d'une formule de logique des propositions caractérisant l'ensemble des parties de AP qui étiquettent une transition de l'état source vers l'état cible.



(a) NBA pour ϕ (b) NBA pour $\neg\phi$

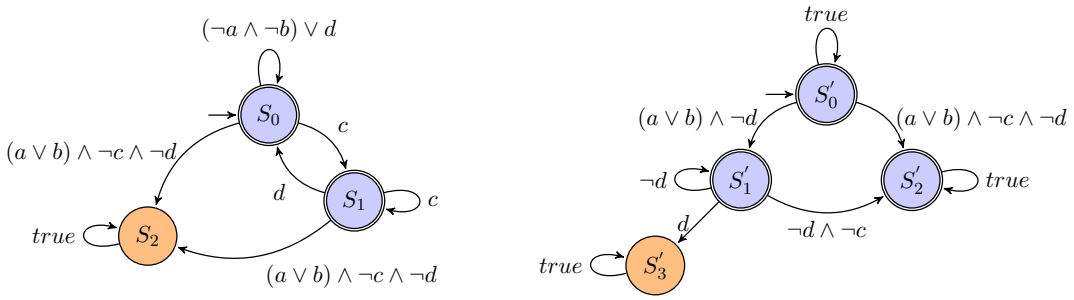
Figure 5.3 – Automates de Büchi non déterministes A pour ϕ et $\neg\phi$

La traduction des automates de Büchi non déterministes en automates finis non déterministes se réduit à un test d'accessibilité. À partir de chaque état q, nous recherchons si au moins une composante fortement connexe des automates de Büchi est accessible. Les composantes fortement connexes contenant un état acceptant permettent ensuite de définir la fonction F. Dans notre cas, il apparait que $F^{\phi}(S_0) = \top$, $F^{\phi}(S_1) = \top$, $F^{\neg\phi}(S'_0) = \top$, $F^{\neg\phi}(S'_1) = \top$ et $F^{\neg\phi}(S'_2) = \top$.

L'ensemble des états acceptants des automates finis non déterministes est alors donné par $\hat{T}^{\phi} = \{S_0, S_1\}$ pour ϕ et $\hat{T}^{\neg\phi} = \{S'_0, S'_1, S'_2\}$ pour $\neg\phi$.

Les résultats précédents nous permettent de déduire les automates finis non déterministes. Les automates finis non déterministes correspondant sont présentés sur la Figure 5.4. Nous observons que $\hat{A}^{\phi}$ a été complété par un état puits appelé S_2 et que $\hat{A}^{\neg\phi}$ a été complété par un état puits appelé S'_3. Ceci est nécessaire pour que les deux automates soit complets et agissent strictement sur le même alphabet (i.e. sur $\Sigma = 2^{AP} = \Sigma^{\phi} \cup \Sigma^{\neg\phi}$) et que le produit soit possible.

La déterminisation de chaque NFA est ensuite effectuée. Enfin, le résultat du produit des automates après minimisation est visible sur la Figure 5.5. On remarque que seuls trois états sont utiles pour décrire la règle. Lorsque a ou b devient vrai et que c est vrai, on passe à l'état $St2$. Si d devient vrai, on retourne à l'état initial. En revanche, si c devient faux avant, on passe dans l'état *bad*.



(a) NFA pour ϕ (b) NFA pour $\neg\phi$

Figure 5.4 – Automates finis non déterministes $\hat{A}$ pour ϕ et $\neg\phi$

Notons enfin que la règle considérée étant une propriété de *sûreté*, la machine de Moore obtenue renvoie les valeurs de sorties sur $\mathbb{B}_2^{\perp} \subset \mathbb{B}_3$.

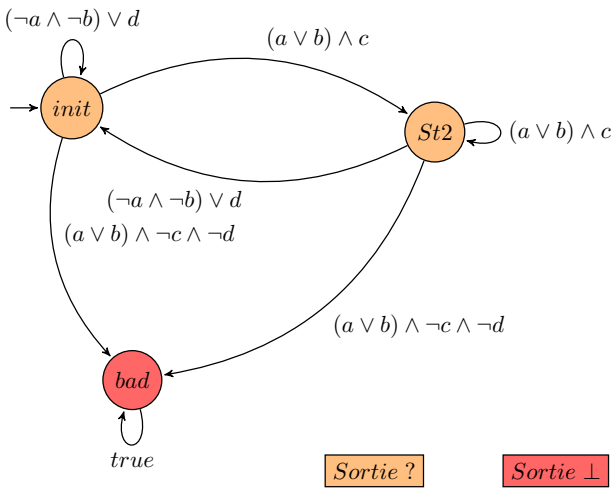


Figure 5.5 – Machine de Moore $\bar{A}$: moniteur de ϕ

5.3 Synthèse du moniteur final

Nous présentons désormais notre approche pour synthétiser le moniteur qui sera finalement utilisé par le service de vérification en ligne. Nous le désignons sous le terme de *moniteur final*. La machine de Moore obtenue dans la Section 5.2.2 réagit aux changement de valeurs d'un ensemble de propositions atomiques, c'est-à-dire sur $\Sigma = 2^{AP}$. Le moniteur que nous utilisons pour la vérification réagit sur les évènements du système. L'ensemble de ces évènement correspond à un alphabet noté Σ^S. Nous choisissons d'effectuer la correspondance entre les deux hors-ligne. À l'aide d'un modèle du système à surveiller, l'occurrence d'un évènement de Σ^S conduit à un changement d'état dans ce modèle et donc à un changement de l'ensemble des propositions atomiques *vraies* à cet état. Concrètement, les propositions atomiques sont écrites directement à partir des états qui représentent le modèle du système.

5.3.1 Topologie du moniteur final

Pour atteindre nos objectifs, nous construisons une machine de Moore notée M' définie par le tuple $M' = (\Sigma^S, Q', q'_0, \delta', \lambda', \mathbb{B}_3)$. Notons que les valeurs de sorties du moniteur final sont issues de la table de vérité $\mathbb{B}_3$. Par la suite, un moniteur de ce type est désigné sous le terme de *table de transitions*. Illustrons la table de transitions d'une propriété de sûreté ϕ dans la Table 5.2. Il y a autant de lignes que d'états dans le moniteur et autant de colonnes que d'évènements à intercepter pour ce moniteur (les évènements sont issus de l'ensemble Σ^S). Au démarrage de l'application, le moniteur est dans l'*état initial*. À chaque fois qu'un évènement de Σ^S est intercepté, nous changeons d'état en parcourant la table (ce parcours est toujours effectué en temps constant). Puisque la configuration est statique, la relation entre un évènement et les moniteurs devant évoluer sous son occurrence est connue hors-ligne. Pour simplifier la représentation, toutes les transitions qui conduisent vers un état dont la fonction de sortie est $\perp$ (ϕ violée) sont renvoyées vers un même état puits nommé $FAUX$.

	δ'	*Évènements* (Σ^S)			λ'
		évènement 1	**évènement 2**	**évènement 3**	
états (Q')	**état initial** q'_0	état 1	état 2	**FAUX**	?
	état 1	état 2	état 3	état initial	?
	état 2	**FAUX**	état 2	état 3	?
	état 3	état initial	**FAUX**	état 3	?
	FAUX				$\perp$

Table 5.2 – Une table de transitions pour une propriété de *sûreté* ϕ

5.3.2 Synthèse du moniteur final sous la forme d'une table de transitions

Pour synthétiser le moniteur final, il est impératif de disposer des deux entrées suivantes :
- Une *description du système* que l'on observe. Pour cela, nous utilisons une description à l'aide d'automates (par exemple une structure de Kripke) ;
- Le moniteur intermédiaire de la formule LTL à vérifier ($\bar{A}$).

Le modèle du système est une structure de Kripke $A^S = (\Sigma^S, Q^S, q_0^S, \delta^S, \lambda^S)$. La fonction représentée par λ^S est décrite dans la Définition 5.18.

Définition 5.18. *La fonction injective $\lambda^S \subseteq Q^S \longrightarrow 2^{AP}$ fait correspondre à chaque état q l'ensemble des propositions atomiques de AP, qui sont vraies dans cet état.*

Le moniteur intermédiaire est la machine de Moore $\bar{A}^M = (\Sigma^M, Q^M, q_0^M, \delta^M, \lambda^M)$. La fonction représentée par λ^M est définie dans la Section 5.2.2.

Le moniteur final est une machine de Moore M' sur Σ^S. La définition formelle de ce moniteur est donnée dans la Définition 5.19.

Définition 5.19. *Le moniteur final d'une propriété P est une machine de Moore définie par le tuple $M' = (\Sigma^{S'}, Q', q'_0, \delta', \lambda', \mathbb{B}_3)$ tel que :*
- *$\Sigma^{S'} \subseteq \Sigma^S$ est l'alphabet comprenant l'ensemble des évènements interceptés dans le système pour la surveillance de P ;*
- *$Q' = Q^S \times Q^M$ est l'ensemble non vide des états de M' ;*

- $q'_0 = (q_0^S, q_O^M)$ *est l'état initial ;*
- $\delta' \subset (Q' \times \Sigma^S) \longrightarrow Q'$ *est la fonction de transition présentée sous la forme* $\delta'((q^S, q^M), \sigma) = (r^S, r^M)$ *ssi*
  - $\delta^S(q^S, \sigma) = r^S$ *et* $\delta^M(q^M, u) = r^M$ *;*
  - $u \subseteq \lambda^S(r^S)$ *et* $\lambda^M(q^M) =?$.
- $\lambda' \subset Q' \longrightarrow \mathbb{B}_3$ *tel que* $\lambda'(q^S, q^M) = \lambda^M(q^M)$.

Les différents paramètres de M' trouvent leurs correspondances dans la table de transitions 5.2. Nous choisissons de bloquer le moniteur quand un état $\top$ ou $\bot$ est atteint. Une action de l'utilisateur est nécessaire pour reprendre la surveillance.

L'algorithme de synthèse construit l'ensemble des états accessibles de Q' par une exploration en profondeur (DFS – *Depth First Search*) commençant à l'état initial q'_0. Le calcul de δ' est issu de cette exploration : chaque état atteint est valide tant que le moniteur intermédiaire reste dans un état ?. La profondeur est bornée par celle des chemins qui conduisent à un état $\top$ ou $\bot$. Cet algorithme termine, car Q' est un ensemble fini.

5.3.3 Synthèse finale d'un moniteur sur un exemple

Nous illustrons maintenant la synthèse d'un moniteur. Pour cela, nous considérons à nouveau l'Exigence 3.2, définie pour l'architecture présentées dans la Figure 5.6.

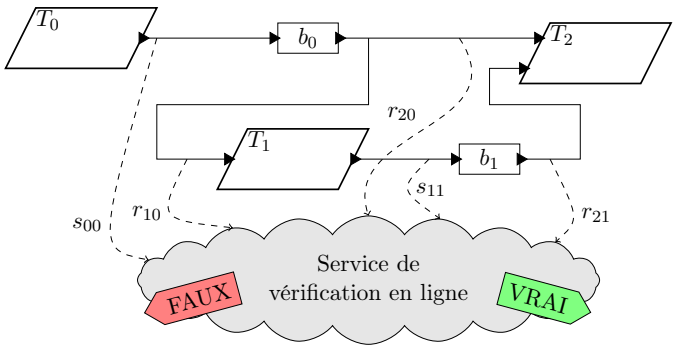


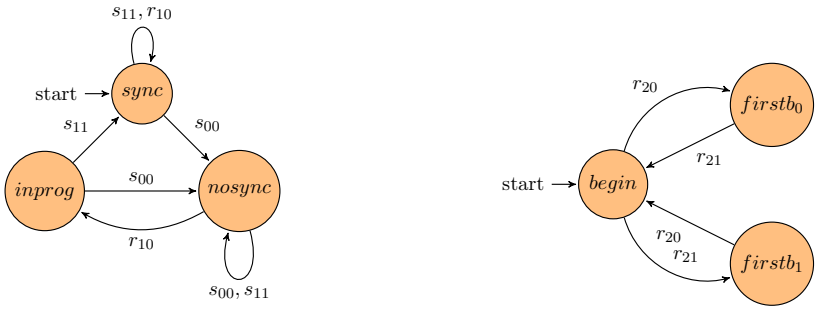
Figure 5.6 – Architecture d'un mécanisme de vérification en ligne

Nous rappelons que cette exigence requiert que les valeurs lues par T_2 soient cohérentes, dans le cas où elle effectue un contrôle de vraisemblance sur les opérations effectuées par T_1. Pour la vérifier, nous devons surveiller la synchronisation des buffers et leurs lectures par la tâche T_2. Nous utilisons les deux automates de la Figure 5.7 pour modéliser le système.

La synchronisation des buffers est illustrée par la Figure 5.7(a). Les buffers sont synchronisés à l'état initial. Dès que T_0 envoie un message dans b_0 (évènement s_{00}), ils se désynchronisent. Pour à nouveau les synchroniser, il faut que T_1 lise le message dans b_0 (évènement r_{10}), effectue son traitement et produise son résultat dans b_1 (évènement s_{11}). Pendant cette phase, T_0 ne doit pas mettre à jour b_0 sous peine de rompre la synchronisation.

Le comportement de T_2 est représenté sur la Figure 5.7(b). Aucune hypothèse n'est faite sur l'ordre de lecture des messages. T_2 peut d'abord consommer le message de b_0 (évènement r_{20}) puis celui de b_1 (évènement r_{21}) ou l'inverse.

Le modèle du système complet A^S est le résultat du produit synchronisé de m_sync et m_t2. L'ensemble des propositions atomiques est implicitement l'union de l'ensemble des états



(a) *m_sync* : Synchronisation des buffers

(b) *m_t2* : Comportement de T_2

Figure 5.7 – Modèle formel du système à surveiller

des automates modélisant le système (i.e. $m\_sync.sync$, $m\_sync.nosync$, $m\_sync.inprog$, $m\_t2.begin$, $m\_t2.firstb_0$ et $m\_t2.firstb_1$).

La traduction de l'exigence en termes de propriété LTL doit distinguer le fait que T_2 lise d'abord b_0 ou b_1. Il faudrait alors deux règles pour surveiller les deux comportements. Pour des raisons de simplicité dans l'exemple, nous restreignons l'Exigence 3.2 et considérons l'Exigence 5.1

Exigence 5.1. *Les buffers b_0 et b_1 doivent être synchronisés et le rester tant que la tâche T_2 est en cours de lecture.*

La formule LTL dérivée de l'Exigence 5.1 est la suivante :

$$\phi \text{ définie par } \phi = \mathbf{G}\ ((m\_t2.firstb0 \vee m\_t2.firstb1) \implies (m\_sync.sync\ \mathbf{U}\ m\_t2.begin))$$

Une formule LTL de ce type a déjà été traitée dans la Section 5.2.4 (page 71). Le moniteur intermédiaire $\bar{A}^M$ est alors identique à celui de la Figure 5.5 tel que $a = m\_t2.firstb0$, $b = m\_t2.firstb1$, $c = m\_sync.sync$ et $d = m\_t2.begin$. Dès que T_2 lit un des deux buffers, le moniteur passe dans l'état $St2$. Il reste dans cet état jusqu'à ce que T_2 finisse les lectures (i.e. retour à l'état *init*) où jusqu'à ce qu'une erreur se produise (i.e. état puits *bad*).

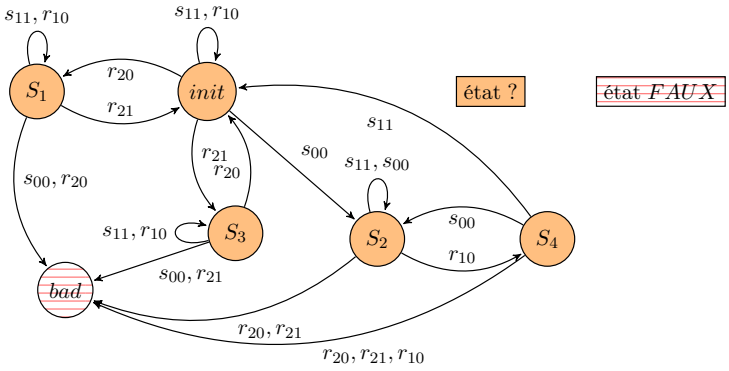


Figure 5.8 – Moniteur final de l'Exigence 5.1

Enfin, la dernière étape consiste à composer le modèle du système A^S avec le moniteur intermédiaire $\bar{A}^M$, selon la démarche proposée dans Section 5.3.2. Le résultat est illustré par la Figure 5.8.

Chapitre 6

Implémentation et évaluation

Sommaire

6.1 Comportement en ligne du service de vérification

Le comportement en ligne est représenté sur la Figure 6.1.

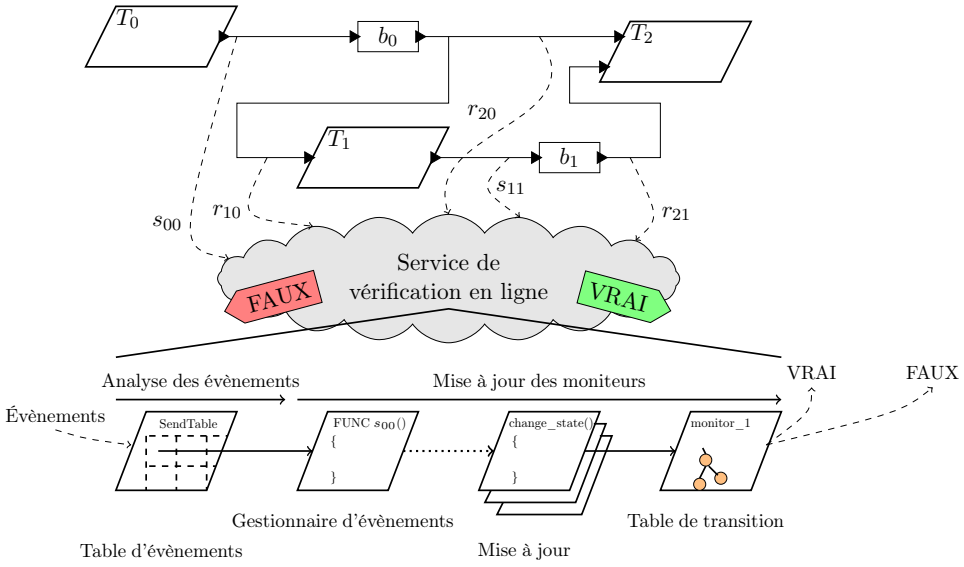


Figure 6.1 – Comportement en ligne du service de vérification

Quand un évènement lié à la communication est intercepté, l'étape d'analyse consiste à parcourir une *table d'évènements* indexée par l'identifiant de la tâche qui initie l'évènement et le message sur lequel on agit (il y a une table par sens de communication). Le champ de la table qui correspond à l'évènement intercepté contient un pointeur vers la fonction correspondante dans le *gestionnaire d'évènements*. Si l'évènement n'est lié à aucun moniteur, la fonction pointée est vide et nous avons perdu un peu de temps. Nous verrons dans la Section 6.4.2 (page 90) comment réduire cette perte de temps.

Pour les autres évènements, la fonction associée à l'évènement appelle de manière séquentielle la fonction de *mise à jour (change_ state())* pour tous les moniteurs qui doivent réagir sur cet évènement. En pratique, la fonction cherche, à partir de l'état courant et de l'évènement, le nouvel état du moniteur dans la table de transition.

Le moniteur statue ensuite sur la validité des propriétés. Si la propriété est ni vraie, ni fausse, le service attend l'occurrence du prochain évènement. Dans le cas contraire, la fonction de hook correspondante est appelée. Cette fonction provient d'un *gestionnaire de hook*, c'est-à-dire d'une source rassemblant toutes les fonctions de hook. En cas de violation, des stratégies de recouvrement peuvent être mises en place. Par exemple, en conformité avec AUTOSAR (AUTOSAR, 2013a), il est possible de redémarrer le système d'exploitation, de redémarrer tout l'ECU ou encore de terminer la tâche fautive. Quelques macros permettent également d'obtenir des détails sur les causes de l'erreur et facilitent l'analyse. Le moniteur peut ensuite être réinitialisé ou désactivé, selon le choix de l'utilisateur.

Dans la suite, nous décrivons comment la chaîne de traitement ainsi illustrée est synthétisée.

6.2 Synthèse des moniteurs avec *Enforcer*

6.2.1 Schéma de principe du fonctionnement d'*Enforcer*

Pour faciliter la synthèse des moniteurs, nous avons construit l'outil *Enforcer* dont le fonctionnement est illustré sur la Figure 6.2.

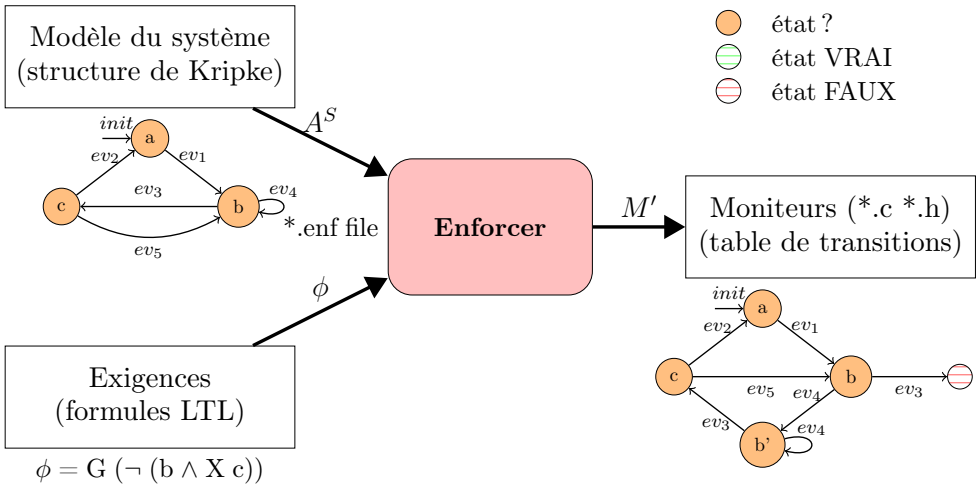


Figure 6.2 – Synthèse hors-ligne des moniteurs avec *Enforcer*

Enforcer est l'outil implémenté dans le cadre de la thèse, pour synthétiser automatiquement les moniteurs. Il est disponible à l'adresse `http://enforcer.rts-software.org`. *Enforcer* prend en entrée le modèle du système A^S et l'ensemble des propriétés ϕ à surveiller en ligne. Le processus de génération est composé de deux étapes. La première consiste à générer le moniteur intermédiaire $\bar{A}^M$ pour chacune des propriétés à surveiller. Pour cela, *Enforcer* implémente les algorithmes décrits dans la Section 5.2.2 (page 67) et utilise l'outil *LTL2BA* [1] pour traduire la propriété LTL en un automate de Büchi non déterministe. La seconde étape consiste à composer ces moniteurs intermédiaires avec le modèle du système qui est fourni en entrée. Pour cela, *Enforcer* implémente les algorithmes décrits dans la Section 5.3.2 (page 73).

En sortie, *Enforcer* fournit le code source des moniteurs sous la forme de tables de transitions complètes ainsi que les descripteurs de moniteurs qui permettent de manipuler ces tables. Le code source généré (fichiers *.c et *.h) est ensuite compilé avec le noyau du système d'exploitation.

1. *LTL2BA* est développé par *Gastin et Oddoux* (Gastin et Oddoux, 2001). Il peut être téléchargé à l'adresse : `http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/download.php`. Une version modifiée pour intégrer la chaîne de développement d'*Enforcer* est disponible dans le package d'*Enforcer*.

6.2.2 Chaîne de compilation d'*Enforcer*

6.2.2.1 Entrées requises

Enforcer est écrit dans le langage *Galgas*[2]. Galgas est un DSL, compilateur pour écrire des compilateurs de DSL. Nous l'avons utilisé pour décrire la syntaxe et la sémantique de l'outil. Le langage *enf* que nous avons créé permet de décrire simplement les automates modélisant le système ainsi que les propriétés. Un exemple illustratif de la syntaxe du langage *enf*, issue de l'Exigence 5.1, est donné dans la Figure 6.3.

```
rule syncho {
  automata {
    m_sync {
      states {sync, nosync, inprog};
      events {s00, s11, r10};
      transitions {
        in sync {
          when s00 then nosync;
          when s11 then sync;
          when r10 then sync;
        }
        in nosync {
          when s00 then nosync;
          when r10 then inprog;
          when s11 then nosync;
        }
        in inprog {
          when s00 then nosync;
          when s11 then sync;
        }
      }
    } // fin de m_sync
    m_t2 {
      ...
      }
    } // fin de m_t2
  }
  property = always ((m_t2.firstb0 or m_t2.firstb1)
      implies (m_sync.sync until m_t2.begin));
  reset = true;
}
```

Figure 6.3 – Description *enf* partielle de l'Exigence 5.1 avec *Enforcer*

Pour la description d'une règle, trois blocs distincts sont présents.

– Le bloc *automata* permet de décrire un ensemble d'automates (autant que nécessaire). Les ensembles Σ^S, Q^S et δ^S de chaque automate sont à écrire explicitement dans les champs *states*, *events* et *transitions*. L'état initial de chaque automate de ce type est implicitement le premier état du champ *states*. Enfin, λ^S est calculé par *Enforcer* : chaque état est associée à une unique proposition atomique qui porte le même nom.

2. *Galgas* est développé au sein de l'équipe *Systèmes Temps Réel* à l'IRCCyN. L'outil et sa documentation sont disponibles à l'adresse : `http://galgas.rts-software.org`.

– Le bloc *property* est utilisé pour écrire les formules LTL correspondant aux propriétés ;
– Le bloc *reset* est un booléen utilisé pour déterminer si la surveillance doit se poursuivre après l'accès à un état $\top$ ou $\bot$. Si la valeur est *true*, le moniteur reprendra son état initial pour poursuivre.

La grammaire BNF du langage est disponible en-ligne dans le package d'*Enforcer*.

6.2.2.2 Sorties générées

En sortie, *Enforcer* fournit quatre types de fichiers, au format *.c et *.h, correspondant à la synthèse :
– du gestionnaire d'évènements ;
– des tables de transitions ;
– des descripteurs de moniteurs ;
– du gestionnaire de hook.

Le gestionnaire d'évènements

Le gestionnaire d'évènements (event handler) est le point d'entrée du service de vérification. À chaque évènement intercepté, une fonction est associée. Elle permet d'appeler la fonction de changement d'état pour tous les moniteurs qui doivent réagir sur cet évènement. Ceci est illustré sur la Figure 6.4 pour un évènement $s00$. Nous pouvons noter que puisque tous les évènements du système ne conduisent pas à l'évolution de tous les moniteurs, un tel évènement doit être traduit en un évènement local spécifique à chaque moniteur. Ceci est réalisé par le gestionnaire d'évènements via l'appel de la fonction *change_ state()*, prenant en paramètres le descripteur du moniteur à faire évoluer et l'identifiant local de l'évènement du système.

```
FUNC(void, OS_CODE) s00()
{
  change_state(&prop1_desc, s00_local_event_prop1);
  ...
  change_state(&propN_desc, s00_local_event_propn);
}
```

Figure 6.4 – Fonction du gestionnaire d'évènements pour $s00$

Les tables de transitions

Une table de transitions est une matrice, stockée en ROM, de taille $N_s \times N_e$ où N_s est le nombre d'états du moniteur et N_e est le nombre d'évènements requis par le moniteur. $m_{i,j}$ est l'image par la fonction de transition de l'état i sous l'occurrence de l'évènement j. Chaque champ $m_{i,j}$ est un mot de 8 bits, qui permet de représenter jusqu'à 256 états différents. La table de transition présentée dans la Table 5.2 peut être écrite sous la forme de la Table 6.1.

Sur la Figure 6.5, nous montrons comment la table de transitions est codée en mémoire. Sur le schéma, une ligne de la mémoire a une largeur de 1 octet, c'est-à-dire que $N_e = 3$ lignes mémoires sont nécessaires pour coder une ligne de la table de transition. Pour des raisons de lisibilité, nous ne représentons que les lignes correspondant aux transitions en partance de

$\delta()$	e_0	e_1	e_2
$q_0 =$ **000**	001	010	**100**
$q_1 =$ **001**	010	011	000
$q_2 =$ **010**	**100**	010	011
$q_3 =$ **011**	000	**100**	011
$q_4 =$ **100**	**100**	**100**	**100**

Table 6.1 – Une table de transitions telle que $N_e = 3$ et $N_s = 5$

q_0 et q_1. Notons qu'à partir d'un état et sous l'occurrence d'un évènement, nous récupérons directement la valeur de l'état courant suivant.

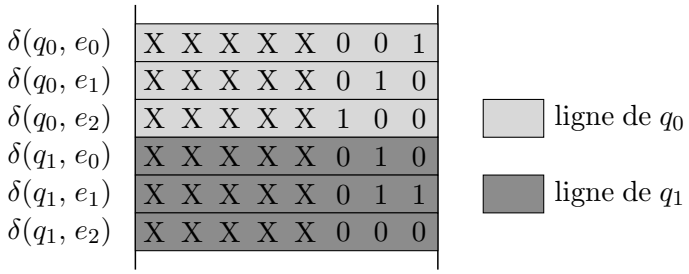


Figure 6.5 – Implémentation de la table de transitions

Seuls trois bits sont nécessaires pour coder un état. Les bits notés X ne portent pas d'informations utiles pour le codage de la table : ce sont des bits perdus. Après une évaluation de l'empreinte mémoire, nous discutons dans la Section 6.4.1 (page 85) d'améliorations permettant de réduire ce nombre de bits perdus.

Les descripteurs de moniteurs

Il y a un descripteur par moniteur, stocké en RAM, et chacun est typé selon la Figure 6.6.

```
struct TPL_FSM_INITIAL_INFORMATION {
  const u8 monitor_id; /* identifiant du moniteur */
  const u8 *automata; /* table de transition */
  var u8 current_state; /* état courant */
  const u8 nb_event; /* nombre d'évènements */
  CONST(tpl_false_func, TYPEDEF) false_function;
    /* fonction appelée sous l'occurrence de "false_state" */
  CONST(tpl_true_func, TYPEDEF) true_function;
    /* fonction appelée sous l'occurrence de "true_state" */
  const u8 false_state; /* identifiant de l'état faux */
  const u8 true_state; /* identifiant de l'état vrai */
  const u8 reset; /* flag de reset */
};
```

Figure 6.6 – Typage du descripteur d'un moniteur

Dans ces descripteurs, nous retrouvons l'identifiant du moniteur, un pointeur vers la table de transition, l'état courant, le nombre d'évènements, le booléen de *reset* ou encore les valeurs des identifiants des états correspondants aux sorties $\top$ ou $\bot$. Ces deux champs peuvent être vides en fonction de la classe de propriétés que l'on considère. Notons enfin la présence des *false_fonction* et *true_function*. Ce sont les fonctions de *hook* qui sont appelées quand le moniteur produit les verdicts $\bot$ ou $\top$.

Le gestionnaire de hook

Le gestionnaire de hook (hook handler) contient toutes les fonctions pointées par les champs *false_function* et *true_funtion*. *Enforcer* génère le squelette de ces fonctions et il reste à la charge de l'intégrateur de coder ces fonctions. Par exemple, dans la Figure 6.7, nous illustrons le squelette de la fonction appelée quand une propriété *prop*1 est violée.

```
FUNC(void, OS_CODE) prop1_falseState()
{
   /* code d'isolation, de diagnostic et recouvrement */
}
```

Figure 6.7 – Fonction du gestionnaire d'évènements pour $s00$

6.3 Intégration du mécanisme de vérification en ligne dans le RTOS Trampoline

6.3.1 Principe de l'intégration dans le noyau de *Trampoline*

Une présentation de *Trampoline* est disponible en Annexe A. La chaîne de compilation enrichie pour intégrer la configuration du mécanisme de vérification en ligne est présentée sur la Figure 6.8. La chaîne de compilation initiale est étendue par l'ajout d'une branche supplémentaire, liée à la configuration du service de vérification en ligne. La synthèse des moniteurs est réalisée avec *Enforcer* à partir d'un fichier d'entrée. *Enforcer* génère le code source des moniteurs et de leurs descripteurs. L'ensemble des fichiers sources est ensuite compilé avec *gcc*, puis lié. Notons que la vérification en ligne peut surveiller des comportements impliquant des tâches dont tout ou partie du code est uniquement disponible sous forme de code objet.

L'utilisation du service de vérification en ligne impose de l'activer dans le noyau. Pour cela, les évènements à intercepter sont identifiés par avance grâce à la description statique. Le code du système d'exploitation est instrumenté sur cette base. L'instrumentation est réalisée par l'ajout de *sondes* dans le noyau, et par l'activation du code correspondant aux fonctions de changement d'état des moniteurs (une fonction générique pour tous les moniteurs).

Pour simplifier le processus de développement d'une application avec le service de surveillance, nous offrons la possibilité de faire générer le fichier de description des moniteurs (entrée d'*Enforcer*) par *goil*. Pour cela, nous étendons le langage de description *OIL*. Cette solution facilite également l'analyse de la configuration préalable à l'instrumentation du noyau.

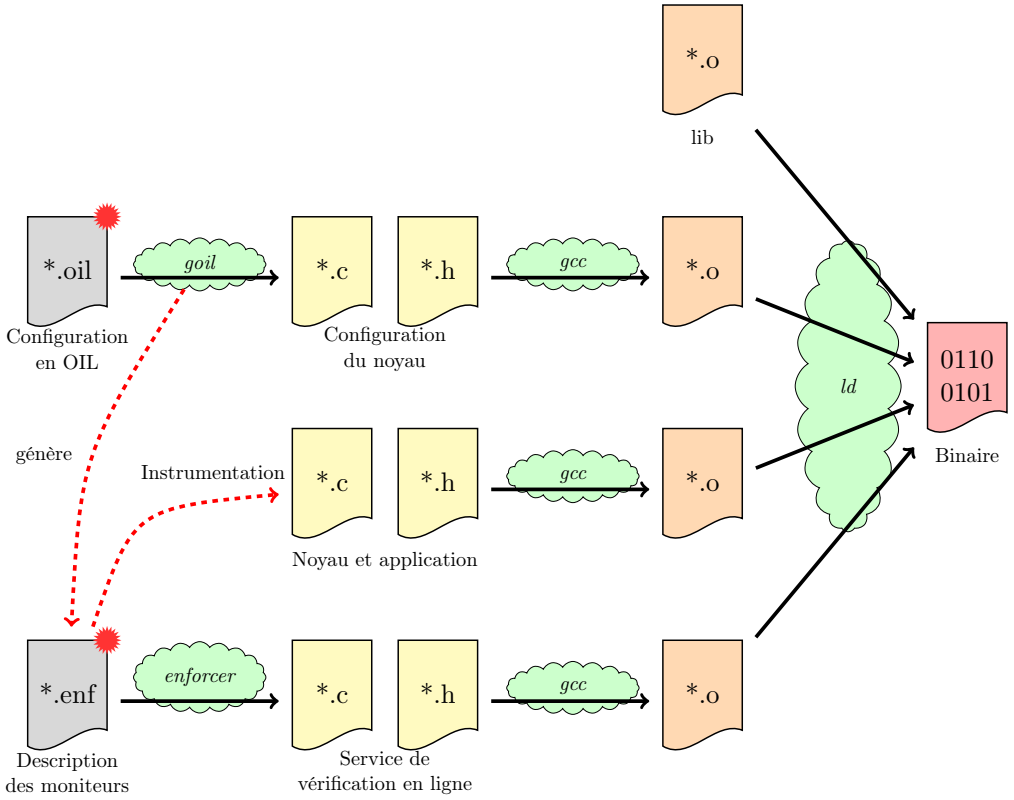


Figure 6.8 – Chaîne de compilation de *Trampoline*, incluant *Enforcer*

6.3.2 Extension du langage OIL pour *Enforcer*

Nous décrivons maintenant l'extension du langage OIL permettant de générer le fichier d'entrée pour *Enforcer* au format *enf*. Trois nouveaux objets doivent être définis en respectant la syntaxe OIL. Sur la Figure 6.9 (a), nous présentons la manière dont est déclaré un évènement lié à *Enforcer*. C'est la combinaison d'une tâche, d'un buffer et d'un sens de transmission (i.e. envoi ou réception d'un message). Notons qu'il existe plusieurs types d'évènements. Les évènements *INTERNAL_ COM* correspondent à ceux résultant de l'envoi ou la réception d'un message. Les évènements *INTERNAL_ COM_ DATA_ EQUAL_ TO* permettent en plus, de tester la valeur du buffer avant de transmettre l'évènement au service (un champ *REFERENCE* permet alors de configurer la valeur à tester). Il existe également des évènements de type *..GREATER_ THAN*, *...SMALLER_ THAN*, etc ... Notons que ces différents types d'évènements sont similaires aux filtres proposés dans la communication interne OSEK/COM OSEK/VDX (2005a).

Sur la Figure 6.9 (b), nous donnons un exemple de déclaration d'un automate. Comme pour le langage *enf*, l'état initial d'un automate est implicitement le premier état de la liste STATE.

Enfin, nous déclarons les formules dans la section *LTL_ RULE* (Figure 6.10). Cette description contient les références aux automates nécessaires pour représenter le modèle du système, puis les formules LTL et enfin la valeur booléenne de *reset*.

```
ENFORCER_EVENT s00 {
  KIND_OF_EVENT = INTERNAL_COM {
    TASK = T0;
    MESSAGEOBJECT = b0;
    ACTION = send;
  };
};
```

(a) Déclaration en OIL des évènements à intercepter

```
AUTOMATA m_sync {
  STATE = sync; // état initial
  STATE = nosync;
  STATE = inprog
  TRANSITION sync_to_nosync {
    FROMSTATE = sync;
    TOSTATE = nosync;
    ENFORCER_EVENT = s00;
  };
  /* etc */
};
```

(b) Description en OIL du modèle du système

Figure 6.9 – Extension du langage OIL pour déclarer des évènements et le modèle du système

```
LTL_RULE rule1 {
  AUTOMATA = m_sync;
  AUTOMATA = m_t2
  PROPERTY = "always((m_t2.firstb0 or m_t2.firstb1)
      implies (m_sync.sync until m_t2.begin))";
  RESET = true;
};
```

Figure 6.10 – Déclaration des règles en OIL

À la compilation, *goil* analyse cette configuration et traduit les informations dans le langage *enf* accepté par *Enforcer*. Un ensemble de vérification permet de tester la cohérence de la configuration vis-à-vis du reste de l'application (e.g. une tâche peut-elle envoyer ou recevoir des messages vers un buffer ?). Seule la formule LTL est recopiée telle quelle. C'est donc *Enforcer* qui vérifie la syntaxe de la formule et qui s'assure que les propositions atomiques qui y sont référencées correspondent à des états du système.

6.4 Évaluation des surcoûts mémoire et temporel

6.4.1 Évaluation et minimisation de l'empreinte mémoire des moniteurs

6.4.1.1 Protocole de mesure

L'empreinte mémoire est obtenue par l'analyse du binaire. Les tailles des descripteurs de moniteurs et des tables de transitions peuvent être déduites directement par l'analyse du code généré en c. La taille du code est obtenue en désassemblant le binaire (avec *arm-elf-objdump*).

6.4.1.2 Empreinte mémoire de la solution

Les résultats concernant l'empreinte mémoire de la vérification en ligne sont donnés dans la Table 6.2. Elle est séparée en trois groupes :

– Les tables de transitions sont stockées en ROM. Nous devons considérer les tables de transitions de tous les moniteurs du système. La taille d'une table de transitions dépend du moniteur. Les résultats fournis dans la table 6.2 concernent l'Exigence 5.1 ;
– Les descripteurs de moniteurs sont initialement stockées en RAM. Il y a un descripteur de ce type par moniteur ;
– Le code nécessaire pour mettre à jour les moniteurs est stocké en ROM (i.e. fonction *change_ state()*). Le code du gestionnaire des évènements et la fonction de changement d'état des moniteurs (i.e. fonction de transition) sont stockés en ROM. Cette partie de l'empreinte mémoire est fixe.

	Table de transitions ROM	Descripteur du moniteur RAM	Taille du code ROM
Méthode Initiale	30 *octets*	12 *octets*	216 *octets*

Table 6.2 – Empreinte mémoire pour l'Exigence 5.1

Il faut également prendre en compte l'empreinte mémoire des deux tables d'évènements. Rappelons que ces tables sont indexées avec l'identifiant des tâches effectuant l'action et l'identifiant du message sur lequel l'action est faite. Il y a une table pour chaque sens de communication. Chacune de ces tables contient $Nb_{taches} * Nb_{messages}$ pointeurs de fonctions, soit une taille de $Nb_{taches} * Nb_{messages} * 4$ octets.

Enfin, la dernière partie de l'empreinte mémoire est due au code du gestionnaire d'évènements. Rappelons que les fonctions du gestionnaire d'évènements permettent d'appeler les fonctions de changement d'état pour tous les moniteurs devant évoluer sur un évènement. Pour un évènement lié à la communication, l'empreinte mémoire est de 12 octets pour chaque appel. Il est aussi possible d'appeler une fonction de mise à jour uniquement si la valeur du message attachée à l'évènement est égal à une valeur prédéfinie. Dans ce cas, l'empreinte mémoire est de 20 octets pour chaque appel.

Les systèmes embarqués automobiles ont une capacité en mémoire (en RAM et ROM) limitée. La taille requise en RAM et en ROM par le service de vérification en ligne croît en fonction du nombre de moniteurs embarqués. Pour maîtriser l'empreinte mémoire, nous proposons des améliorations que nous détaillons dans les deux prochaines sections.

6.4.1.3 Améliorations apportées au codage des descripteurs de moniteurs

Dans ces structures, seul l'état courant est une variable devant nécessairement être stockée en RAM. Les autres champs peuvent être stockés en RAM ou en ROM. Nous proposons à l'utilisateur de choisir la manière dont il souhaite stocker les descripteurs. Pour faire ce choix, il faut trouver le compromis entre les temps d'accès à la mémoire et le gain d'occupation mémoire possible. Notons tout de même que les descripteurs de moniteurs sont petits et n'ont pas autant d'impact que les tables de transitions.

6.4.1.4 Améliorations apportées au codage de la table de transitions en ROM

L'empreinte mémoire de chaque moniteur dépend exclusivement de la propriété qu'il surveille. Dans le pire cas, sa taille est $2^{2^{|\phi|}} * |Q^S|$.

Inconvénients de la méthode initiale

Pour des moniteurs possédant un nombre d'états inférieur à 128, 64, 32, etc, il y a des bits qui n'ont aucune utilité. En effet, pour N_s états, seul $\lceil log_2 N_s \rceil$ bits sont nécessaires. Ces bits perdus occasionnent un gaspillage des ressources en ROM. Dans la Table 6.1, nous présentons le codage des états pour la Table 5.2, en nous limitant à la présentation des bits utiles (i.e. pour 5 états, 3 bits utiles par état).

Les Équations 6.1 et 6.2 représentent respectivement le nombre de bits perdus (N_l) et le nombre d'octets (N_o) nécessaires au codage de la table de transitions selon la méthode initiale. Pour l'exemple considéré dans la Table 6.1, nous perdons 5 bits par état. Puisque dans l'implémentation initiale, 15 états sont codés dans la table de transition ($N_e \times N_s$), $5 \times 15 = 75$ bits sont inutiles.

$$N_l = (8 - \lceil \log_2(N_s) \rceil) \times N_e \times N_s \quad (6.1) \qquad\qquad N_o = N_s \times N_e \quad (6.2)$$

Il est possible d'améliorer ce codage en concaténant les données. Notons que cette modification complexifie le parcours de la table de transitions, ce qui impacte le temps d'exécution.

Deux méthodes d'amélioration ont été implantées. Pour les présenter, nous introduisons des notations supplémentaires. n est la distance en bit entre deux chargements mémoire (i.e. les chargements mémoire sont réalisés à partir d'adresses modulo n). Sur la plupart des cibles, un *fetch* peut récupérer jusqu'à 32 bits. n sera donc choisi parmi les valeurs 8, 16, 24 ou 32 pour minimiser le nombre de bits perdus N_l. Le nombre d'octets N_o, nécessaires au codage d'une table de transition, est un multiple de n. En pratique, c'est *Enforcer* qui choisit ce paramètre en calculant l'empreinte mémoire avec les différentes valeurs possibles et en choisissant la plus performante. Une version des fonctions de changement d'état pour chacune des méthodes est présente dans le système d'exploitation. Les versions nécessaires sont sélectionnées à la compilation.

Dans la suite, nous notons $N_b = N_e * N_s * \lceil log_2 N_s \rceil$, le nombre de bits utiles pour le codage d'une table de transition. L'impact en ROM est évalué par le couple (N_l, N_o).

Méthode 1 : concaténation des champs d'une ligne de la table de transition

La première solution illustrée sur la Figure 6.11 consiste à concaténer tous les champs d'une ligne de la table de transition. En pratique, pour récupérer l'état cible $m_{i,j}$, toute la ligne qui correspond à l'état i est récupérée en un chargement mémoire, et on isole ensuite les bits qui correspondent à l'occurrence de j. Pour réduire le temps nécessaire à l'obtention du résultat, on impose qu'une ligne de la table de transition soit bornée par $\lceil \log_2(N_s) \rceil * N_e \leq 32$. L'étude de cas présentée dans la Section 7 (page 95) montre que cette borne est raisonnable pour la majorité des règles.

D'une ligne à l'autre, nous devons aligner les bits selon un multiple de 8 pour que la ligne suivante de la table de transition débute au même endroit que le prochain chargement mémoire. Dans notre configuration, chaque chargement mémoire est séparé par 16 bits : $n = 16$. Les bits nécessaires à l'alignement sont perdus.

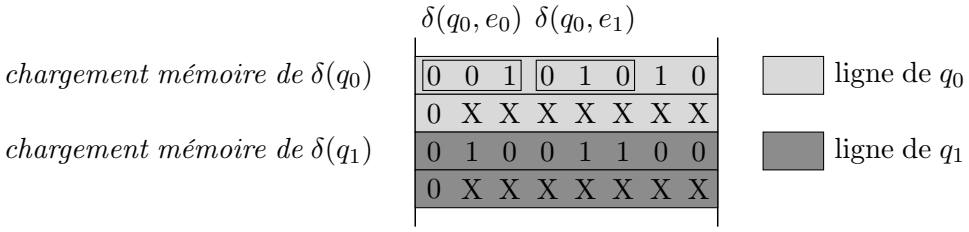


Figure 6.11 – Implémentation de la table de transitions selon la Méthode 1

Les Équations 6.3 et 6.4 représentent respectivement le nombre de bits perdus et le nombre d'octets nécessaires au codage de la table de transition selon cet agencement. Sur notre exemple, le nombre de bits perdus est égal à 35.

$$N_l = (n - N_e \times \lceil \log_2(N_s) \rceil) \times N_s \quad (6.3) \qquad N_o = N_s \times \frac{n}{8} \quad (6.4)$$

Les descripteurs de moniteurs générés par *Enforcer* doivent être adaptées pour ajouter les informations nécessaires à l'extraction du résultat (Figure 6.12). On y trouve le nombre de bits nécessaires pour coder un état $m_{i,j}$, la valeur du masque correspondant et la valeur de n. Ces informations sont calculées par *Enforcer*.

```
struct TPL_FSM_INITIAL_INFORMATION {
  /* ... */
  u8 nb_bit_per_state;
  u8 mask; /* valeur du mask */
  u8 nb_bit_per_line; /* valeur de n */
};
```

Figure 6.12 – Extension du descripteur de moniteurs pour la Méthode 1

Méthode 2 : concaténation de toutes les lignes

La seconde solution est illustrée sur les Figures 6.13. Elle consiste à concaténer toutes les lignes. Seuls les bits nécessaires à l'alignement en fin de tableau sont perdus. Au pire-cas, $N_l = n - 1$, c'est-à-dire 31 si $n = 32$. Là encore, *Enforcer* est en charge de minimiser le nombre de bits nécessaires à l'alignement via le choix de n. En pratique, pour récupérer l'état cible $m_{i,j}$, il faut d'abord rechercher la ligne où se trouve $m_{i,j}$ puis isoler les bits correspondants. Avec cette méthode, il est possible que le codage d'un état soit réparti sur deux mots mémoires distincts : les poids forts sur la fin d'un mot et les poids faibles sur le début du mot suivant. Ceci nécessite plus de temps pour reconstituer l'état, car il faut faire deux chargements mémoire à la suite.

Les Équations 6.5 et 6.6 représentent respectivement le nombre de bits perdus et le nombre d'octets nécessaires au codage de la table de transition selon cet agencement. Sur notre exemple, le nombre de bits perdus vaut 3.

$$N_l = \begin{cases} 0 & si\ (N_b\ mod\ 8) = 0 \\ 8 - (N_b\ mod\ 8) & sinon \end{cases} \quad (6.5) \qquad N_o = \left\lceil \frac{N_b}{8} \right\rceil \quad (6.6)$$

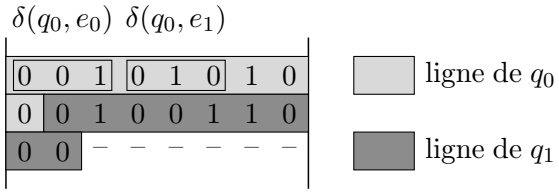


Figure 6.13 – Implémentation de la table de transition selon la Méthode 2

Là encore, les descripteurs des moniteurs générés par *Enforcer* doivent être enrichis avec des données calculées hors-ligne (Figure 6.14). On y trouve le nombre de bits nécessaires pour coder un état, la valeur de division nécessaire pour calculer les adresses des mots à récupérer. La valeur du modulo, la valeur du masque et celle de n sont nécessaires pour cibler les bits à extraire.

```
struct TPL_FSM_INITIAL_INFORMATION {
  /* ... */
  u8 nb_bit_per_state;
  u8 div_value; /* valeur de la division */
  u8 mod_value; /* valeur du modulo */
  u8 mask; /* valeur du mask */
  u8 nb_bit_per_line; /* valeur de n */
};
```

Figure 6.14 – Extension du descripteur de moniteurs pour la Méthode 2

Discussion

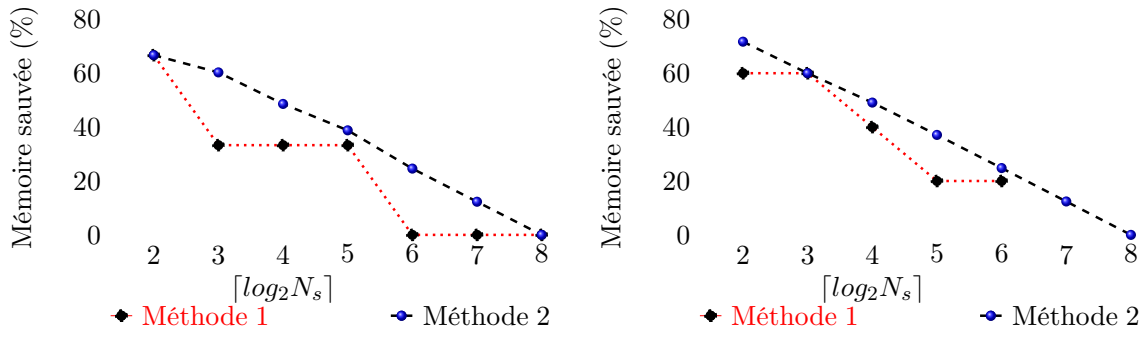
Sur la Figure 6.15, nous présentons des graphiques qui montrent le ratio de bits que l'on sauve par rapport à l'implémentation initiale. Ce ratio est dépendant de N_s et N_e. Sur la Figure 6.15 (a), $N_e = 3$ tandis que sur la Figure 6.15 (b), $N_e = 5$. Sur les deux graphiques, l'axe des abscisses utilise une échelle en $\lceil log_2 N_S \rceil$. Les traits reliant les points n'ont aucune réalité physique, mais ils permettent de mettre en avant l'évolution du gain en mémoire en fonction de N_s.

L'efficacité des améliorations décroit au fur et à mesure que N_s augmente. En effet, quand un état est codé par 7 ou 8 bits, toutes les solutions conduisent aux mêmes résultats et nous finissons par ne plus rien gagner.

La Méthode 2 est la plus performante. Le gain qu'elle apporte décroît quasi linéairement en fonction de $\lceil log_2(N_s) \rceil$. En ce qui concerne la première méthode d'amélioration, le gain qu'elle apporte décroît par paliers. En effet, le gain est fonction de $N_e * \lceil log_2(N_s) \rceil$. Pour tous les points d'un même palier, $N_e * \lceil log_2(N_s) \rceil$ appartient à un intervalle défini par deux multiples de 8 consécutifs, ce qui correspond aux alignements mémoires. Enfin, l'arrêt de la courbe correspondant à la Méthode 1 survient quand on dépasse la limitation $\lceil \log_2(N_s) \rceil * N_e \leq 32$.

Les améliorations du codage de la table de transitions pour l'Exigence 5.1 sont présentées dans la Table 6.3.

Les améliorations permettent de gagner de la place en ROM, au prix de quelques octets pour les descripteurs de moniteur et la taille du code. Pour cet exemple précis, elles permettent



(a) Pourcentage de mémoire sauvée ($N_e = 3$)

(b) Pourcentage de mémoire sauvée ($N_e = 5$)

Figure 6.15 – Améliorations apportées au codage de la table de transitions pour $N_e = 3$ et $N_e = 5$

	Table de transitions ROM	Descripteur du moniteur RAM	Taille du code ROM
Méthode 1	12 *octets* (−60%)	18 *octets* (+20%)	236 *octets* (+9, 2%)
Méthode 2	12 *octets* (−60%)	20 *octets* (+33.3%)	328 *octets* (+51, 8%)

Table 6.3 – Empreinte mémoire pour l'Exigence 5.1

de sauver 60% de l'espace mémoire nécessaire pour stocker la table de transition. Notons que le gain est le même quelle que soit la méthode d'optimisation, car quand $N_e = 5$ et $N_s = 6$, les deux méthodes sont équivalentes (c.f. Figure 6.15(b)). Pour ce cas particulier, *Enforcer* choisit d'utiliser la méthode qui impose le moins de surcoûts temporels, c'est-à-dire la Méthode 1. Avec cette méthode, nous avons augmenté la taille du code et des descripteurs de moniteurs de 20% chacun. À première vue, l'intérêt peut sembler limité, mais avec l'accroissement du nombre de moniteurs, la tendance s'inverse rapidement.

Dans la Section 7 (page 95), nous discuterons des gains mémoires qu'il est possible d'atteindre grâce à ces optimisations sur une étude de cas réelle.

6.4.2 Évaluation et minimisation du surcoût temporel

6.4.2.1 Plateforme matérielle

Les expérimentations ont été réalisées sur une carte de développement olimex lpc2294. Le microcontrôleur LPC2294 est un processeur 32 bits, ARM7TDMI-S, cadencé a une fréquence de 60 MHz. L'OS et l'application ont été compilés avec le compilateur gcc (optimisation -O3). Le programme, les données et les descripteurs de moniteurs sont stockés en RAM externe. Les tables de transitions des moniteurs sont stockées en ROM.

La plateforme matérielle utilisée n'est pas multicœur. Cependant, au vu du contexte AU-

TOSAR, le service de surveillance est une section de code séquentielle dont l'exécution est indépendante du nombre de cœurs. Dans un contexte multicœur, il faut cependant tenir compte de la synchronisation nécessaire à l'utilisation du mécanisme.

6.4.2.2 Protocole de mesure

Les expérimentations sont réalisées pour des propriétés relatives à la communication. Les mesures visent à déterminer le surcoût temporel nécessaire à la surveillance de propriétés. Pour cela, nous mesurons :

- La durée des services de communication utilisés pour envoyer et recevoir des données (*SendMessage()* et *ReceiveMessage()*) sans vérification, sans notification ;
- La durée du service de communication utilisé pour envoyer des données (*SendMessage()*), sans vérification, avec les notifications *ActivateTask* et *SetEvent* (c.f. Annexe A) ;
- La durée des services de communication en intégrant la vérification en ligne.

Le temps nécessaire à la vérification seule est obtenu en effectuant la différence entre les temps d'exécution des services de communication avec la vérification et sans la vérification. Les expériences ne sont faites qu'une seule fois pour chaque valeur mesurée. En effet, en l'absence de cache et de pipeline superscalaire dans le microcontrôleur, le coût induit par le moniteur est constant. Nous rappelons qu'en mode noyau, aucune préemption ne peut survenir. Les valeurs mesurées sont obtenues à l'aide d'un timer, cadencé à la même fréquence que la puce. Le timer est lu à l'entrée du service puis à nouveau à la sortie. La différence nous donne le nombre de cycles requis pour l'exécution du service. Cette valeur est enfin traduite en μs. Le timer fait 32 bits, ce qui permet de mesurer un intervalle d'au plus 71 secondes sans avoir de dépassement à gérer.

Le choix de la propriété à surveiller n'a pas d'importance puisque l'exécution du service est indépendante de la taille de la table de transitions. En effet, ce sont toujours les mêmes opérations qui sont effectuées, à savoir (1) observer et analyser l'évènement (2) parcourir la table de transitions et (3) statuer sur la valeur. En pratique, les expériences ont été menées sur l'Exigence 5.1.

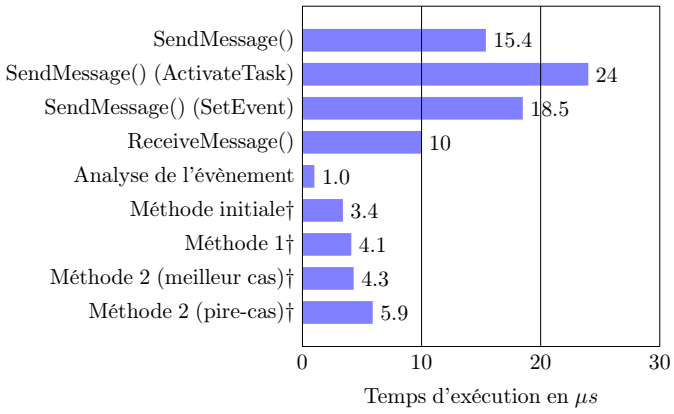
Nous ne mesurons pas l'exécution des fonctions de *hook* appelées lorsqu'une propriété est violée ou satisfaite. En effet, ces fonctions sont spécifiques à chaque application.

6.4.2.3 Surcoût d'exécution induit par la surveillance en ligne

Les résultats de mesure sont illustrés sur la Figure 6.16.

Les quatre premières valeurs sont les durées d'exécution des services de communication seuls, avec ou sans notification. Le service le plus court est celui destiné à la réception des messages, qui s'exécute en $10\mu s$. C'est pour ce service que la surveillance aura l'impact relatif le plus important.

Nous mesurons ensuite les durées d'exécution lorsque le service de vérification en ligne est actif pour une seule règle. Avec l'implémentation initiale, la surveillance prend $3.4\mu s$, ce qui représente un surcoût de 34% pour le service le plus court (i.e. *ReceiveMessage()*), et de 14% pour le service le plus long (i.e. *SendMessage()* avec la notification *ActivateTask*). Les stratégies mises en place pour réduire la taille des tables de transitions ont un impact sur le temps d'exécution. En effet, la Méthode 1 ajoute $0.7\mu s$ pour la surveillance. Ceci est dû au temps du masquage nécessaire pour récupérer l'état cible. La valeur donnée pour la Méthode 2



† Y compris la durée nécessaire à l'analyse de l'évènement

Figure 6.16 – Temps d'exécution de la vérification en ligne pour l'Exigence 5.1

est obtenue dans le pire cas. En effet il faut $5.9\mu s$ pour récupérer un état cible qui serait réparti sur deux lignes. Dans le meilleur cas, la Méthode 2 a des résultats similaires à la Méthode 1.

Nous mesurons enfin uniquement la durée nécessaire à l'analyse d'un évènement intercepté. Cette étape est essentielle pour savoir si l'évènement doit être traité, et si oui, comment le traiter.

À un niveau plus global, le surcoût induit par le service dans une application dépend uniquement du nombre de moniteurs à mettre à jour suite à l'occurrence d'un évènement, et de la fréquence des appels aux services de communications surveillés. Sur la Figure 6.17, nous observons l'évolution de la durée d'exécution du service de vérification en fonction du nombre de moniteurs réagissant à l'évènement.

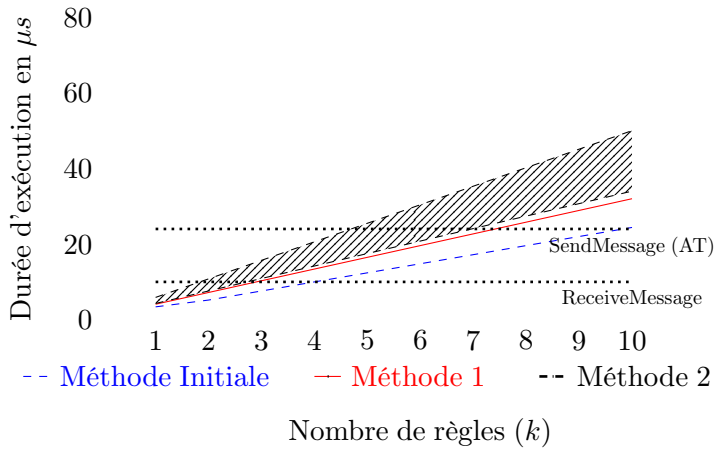


Figure 6.17 – Durée d'exécution de la surveillance pour k règles ($k \in \{1, 2, 4, 6, 8, 10\}$)

Nous constatons une évolution linéaire, ce qui est logique puisque les propriétés sont vérifiées les unes après les autres. Notons tout de même que le temps nécessaire à l'analyse de l'évènement est constant et payé une seule fois. Notons également que pour la Méthode 2,

l'évolution est effectuée au sein de la zone hachurée. En effet, nous avons vu que cette méthode à une durée pire-cas et meilleur-cas. Les bordures hautes et basses correspondent à ces cas limites. Très rapidement, l'accroissement du nombre de règles attachées à un évènement impose un surcoût temporel qui dépasse les 100%. Pour limiter cette contrainte, nous pouvons composer certaines règles. Nous proposons une solution partielle à ce problème dans les sections suivantes.

6.4.2.4 Éviter le surcoût lié à l'interception d'évènements non surveillés

L'instrumentation effectuée dans le système d'exploitation conduit à l'interception de tous les évènements liés à la communication. En effet, l'étape d'analyse effectuée par le service de vérification doit vérifier si l'évènement entraîne la mise à jour d'au moins un moniteur. Cette étape d'analyse qui consiste à parcourir la table des évènements représente $0.28\mu s$ soit 30% du temps nécessaire à la surveillance sur la cible utilisée. Pour les évènements inutiles à la surveillance, c'est du temps gaspillé.

Dans *Trampoline*, l'appel des fonctions liées au service de communication est réalisé par le biais d'un pointeur de fonctions. Le pointeur est initialisé statiquement dans la structure de données de description du message. Ce patron permet au noyau d'appeler la fonction correspondant aux caractéristiques du message. Puisque nous connaissons hors-ligne la liste des évènements qu'il nous faut surveiller, nous pouvons adapter le descripteur du message pour disposer de deux versions pour chaque service de vérification : une version instrumentée pour la vérification et une version non instrumentée. Nous dirigeons alors le pointeur de fonctions vers la bonne version du service lors de la configuration statique du noyau. Avec cette solution, le surcoût lié à l'interception d'évènements non surveillés vaut 0.

Par extension, il est également possible d'envisager cette solution pour les autres services du système d'exploitation (e.g. activation ou terminaison d'une tâche). Comme pour la communication, deux versions des services peuvent être utilisées (une première version instrumentée pour la surveillance, une seconde non instrumentée). Pour réaliser cela, nous ajoutons dans les structures de données correspondantes, un pointeur de fonctions vers la bonne version du service de surveillance. Contrairement aux services de communication, une indirection est tout de même nécessaire puisqu'il faut maintenant rechercher dans les structures de données le service à appeler via le pointeur de fonctions alors qu'avant, nous l'appelions directement. Cette implémentation permet néanmoins de regagner 70% du temps que l'on aurait perdu avec la solution initiale, pour la cible utilisée.

Ces solutions ont un coût en termes d'empreinte mémoire ROM puisqu'elles nécessitent une duplication du code des services concernés.

6.4.2.5 Choix de la granularité de la surveillance

Quand le service de vérification en ligne intercepte un évènement, il est possible que plusieurs moniteurs y soient attachés. Les temps d'observation, d'interception et d'analyse des évènements restent constants, peu importe le nombre de moniteurs attachés à l'évènement. En revanche, le temps nécessaire à l'analyse de la propriété évolue linéairement en fonction du nombre de moniteurs attachés à l'évènement.

Pour réduire ce temps, il est possible de composer des moniteurs. Concrètement, la combinaison de n propriétés LTL conduit à la synthèse d'un unique moniteur au lieu de n. La composition peut être faite selon deux moyens : *par conjonction* ou *par disjonction* des propriétés.

Dans le premier cas, la violation d'une propriété rend difficile la phase de diagnostic puisque nous ne savons pas quelle portion de la propriété est violée. La satisfaction d'une propriété impose également que toutes les propriétés composées aient un verdict $\top$. Dans le second cas, il suffit qu'une seule propriété soit vérifiée $\top$ pour que l'ensemble soit $\top$. Il faut donc composer les moniteurs de manière logique en fonction de l'application que l'on considère.

Pour illustrer la composition par conjonction, considérons deux moniteurs M^1 et M^2 agissant respectivement sur les ensembles d'évènements Σ^1 et Σ^2. Notons également $|Q^1|$ et $|Q^2|$, le nombre d'états respectivement de M^1 et M^2. Dans le cas où $\Sigma^1 \subseteq \Sigma^2$ ou $\Sigma^2 \subseteq \Sigma^1$, la composition résultante est l'intersection des deux moniteurs. Le moniteur composé M aura donc un nombre d'états valant $|Q| \leq max(|Q^1|, |Q^2|)$. Dans ce cas particuliers, en plus de gagner du temps lors de la mise à jour des moniteurs, l'empreinte mémoire est également diminuée. Dans le cas général, seuls certains évènements sont communs. Le résultat de la composition dépend du nombre d'évènements partagés. La borne supérieure du nombre d'états après composition est dans ce cas donnée par $|Q| = |Q^1| \times |Q^2|$. En effet, au pire-cas, la conjonction de deux moniteurs n'ayant pas d'évènement commun revient à effectuer un produit libre entre ces moniteurs. L'empreinte mémoire du moniteur final dépend de la taille de Σ^1 et Σ^2 puisqu'ils doivent être complets pour réagir sur tous les évènements.

Finalement, le résultat est dépendant de l'application et nous discutons dans la prochaine section des possibilités offertes par la composition sur un cas d'étude pratique en Section 7 (page 95).

Chapitre 7

Étude de cas

Sommaire

Nous cherchons à évaluer l'impact du mécanisme à l'échelle d'un calculateur. Pour cela, nous intégrons le projet *PROTOSAR* réalisé au sein de l'équipe *Logiciel Embarqué Temps Réel* chez Renault.

7.1 Le projet *PROTOSAR*

7.1.1 Contexte du projet

Le projet *PROTOSAR* (*Prototype Safe-AUTOSAR*) repose sur le projet *PAMU* (*Plateforme Avancée de Mobilité Urbaine*), réalisé chez Renault. *PAMU* a pour objectif la mise au point, au sein du Technocentre Renault, d'un système global de transport utilisant une flotte de véhicules électriques utilisables en libre accès et ayant des fonctions automatisées. Chaque véhicule possède deux modes d'utilisation : *le mode traditionnel* avec conducteur et *le mode automatisé* dans lequel le véhicule se déplace seul, sans intervention humaine. Dans ce dernier mode, l'utilisateur commande son véhicule pour une mise à disposition à un lieu de rendez-vous. Par exemple, de retour sur le site du Technocentre, le véhicule peut être programmé pour rentrer au garage seul, ayant préalablement déposé le conducteur à un autre endroit. L'autonomie acquise par le véhicule impose de disposer de ressources électriques/électroniques fiables, et sûres de fonctionnement.

Le projet *PROTOSAR* vise à remplacer le système de contrôle du véhicule utilisé dans *PAMU*, par un calculateur prototype basé sur AUTOSAR R 4.0.

7.1.2 Objectifs

Le système *PAMU* est présenté sur la Figure 7.1. Le module de supervision est un module décisionnel de haut niveau. Il permet par exemple de connaître le mode du véhicule. Le système de localisation permet de connaître l'état du véhicule. Le module de perception récupère les informations provenant des capteurs. Le système de navigation est en charge de la planification de la mission (consigne de base, choix de la trajectoire). Enfin, le module de contrôle permet, à partir du plan de la mission de piloter le véhicule. C'est ce dernier module système qui contient les algorithmes de régulation en charge de définir les consignes pour chaque actionneur, en prenant soin de respecter l'ordre de mission.

Le projet *PROTOSAR* ne concerne que le module de contrôle du véhicule, appelé UCM (*U-MAP Control Module – UnManned Autonomous Platform Control Module*). La solution existante doit être remplacée par une implémentation complète respectant le standard AUTOSAR R4.0. La plateforme matérielle choisie pour l'implémentation du démonstrateur est un microcontrôleur Freescale Power PC BOLERO MPC 5644C, multicœur uniforme.

L'UCM assure les fonctionnalités suivantes :

- Le passage entre les modes *traditionnel* et *autonome* ;
- Le suivi d'une trajectoire (à partir d'une consigne) ;
- Le freinage d'urgence ;
- La surveillance des capteurs en mode autonome ;
- La gestion des modules d'alimentation des actionneurs.

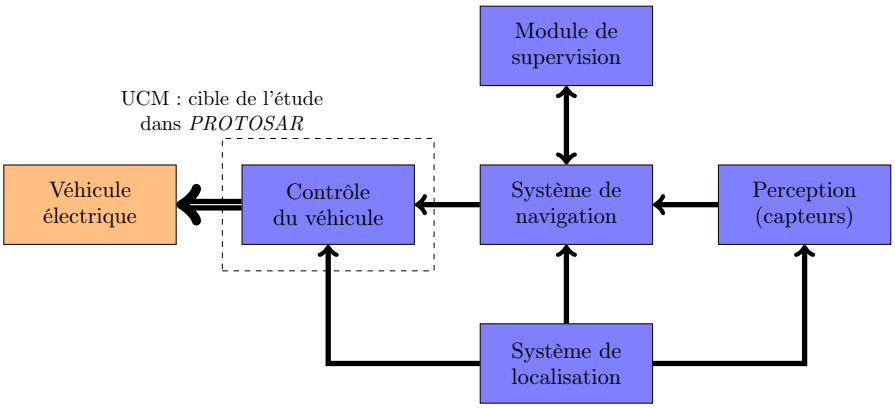


Figure 7.1 – Cible de l'étude *PROTOSAR* dans le projet *PAMU*

7.2 Détails sur la topologie de l'architecture de *PROTOSAR*

Une vue interne de l'UCM est présentée sur la Figure 7.2.

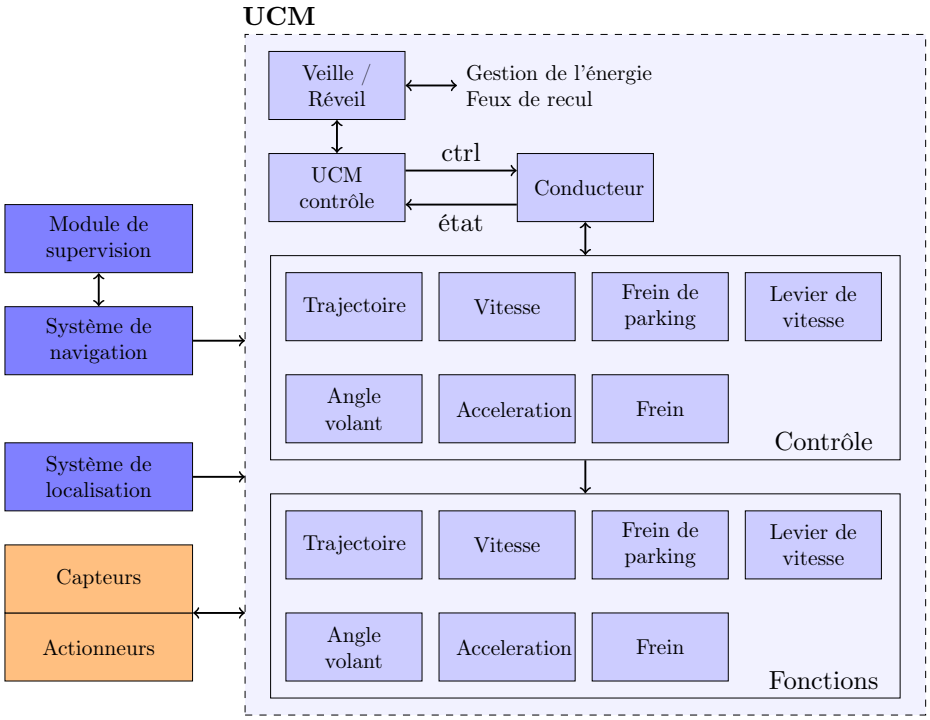


Figure 7.2 – Architecture globale de l'UCM

Le calculateur UCM peut prendre plusieurs états (*Sleep, Standby, Working, Autonomous, Emergency Braking, Exception*). Le passage d'un état à un autre dépend de l'état du véhicule, et de l'état des composants de l'application.

Le module UCM est organisé en blocs hiérarchiques. Le bloc de contrôle transmet au

conducteur les informations nécessaires à la conduite (e.g. mode de conduite, consignes). Le conducteur renvoie en retour les informations relatives à son état. Le conducteur transmet les ordres de contrôle à chaque composant nécessaire à la conduite. Il y a sept composants : le contrôle de la trajectoire, le contrôle de la vitesse, le frein de parking, le levier de vitesse, le contrôle du volant, la gestion de l'accélération et le contrôle du freinage. Chaque composant possède un bloc spécifique au contrôle et un bloc fonctionnel. Le bloc de contrôle permet de définir le mode du composant (machine à états). Les blocs fonctionnels appliquent les algorithmes de conduite en concordance avec le mode sélectionné. Les interactions entre chaque partie fonctionnelle des composants sont décrites par la Figure 7.3. Le cadre de notre étude est limité aux composants eux-mêmes (parties contrôle et fonctionnelle).

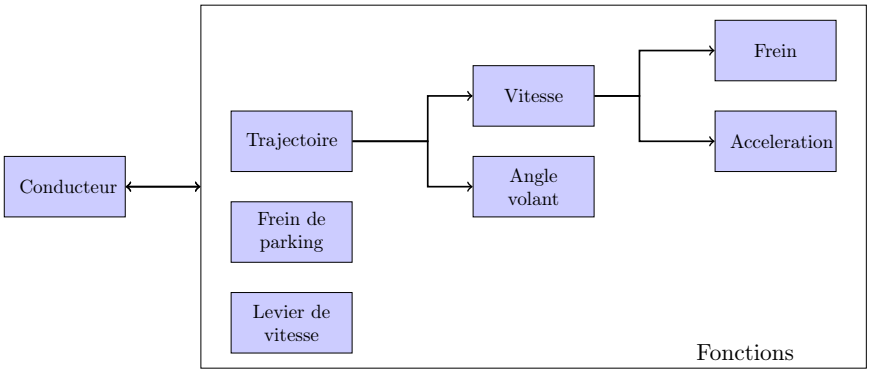


Figure 7.3 – Interactions entre les composants dans *PROTOSAR*

Le pattern relatif à l'architecture fonctionnelle de l'application *PROTOSAR*, présenté sur la Figure 7.4 est répercuté pour tous les composants que l'on vient d'évoquer.

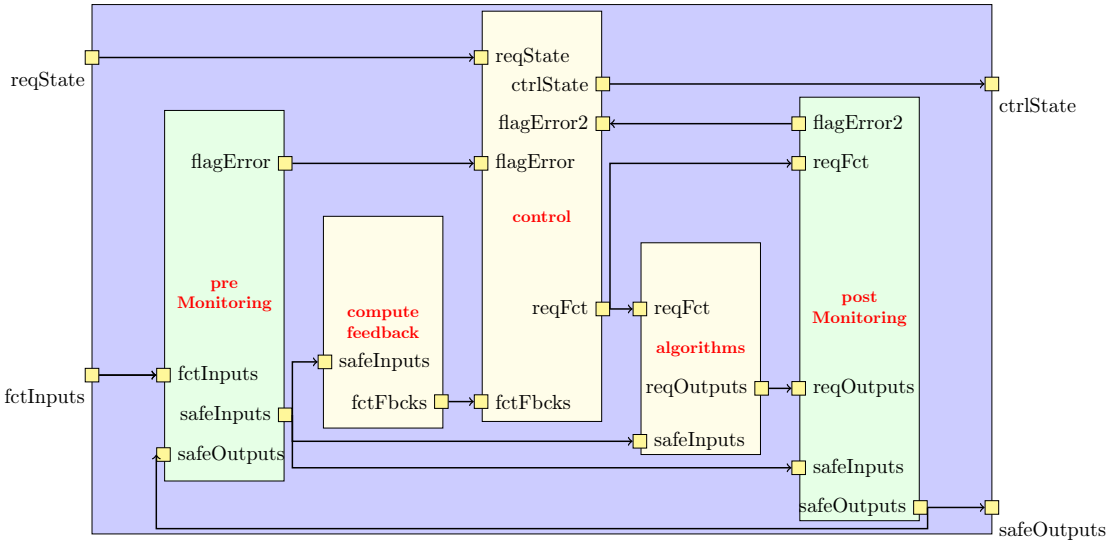


Figure 7.4 – Pattern de l'architecture fonctionnelle appliqué à chaque composant

Les signaux d'entrées/sorties du composant *reqState* et *ctrlState* sont respectivement la

consigne du conducteur (mode, etc.) et le rapport effectué au conducteur. *fctInputs* et *safeOutputs* sont respectivement les entrées provenant de la lecture des capteurs véhicule et les sorties vers les actionneurs ou les composants suivants de la chaîne.

On distingue cinq blocs :

- *preMonitoring* est un bloc de sûreté en charge de vérifier que toutes les entrées lues par le composant sont conformes. En cas de non-conformité, des flags d'erreurs sont transmis au bloc *control*. Sinon, les entrées lues sont fournies au *compute feedback*, à *algorithms* et au *postMonitoring* ;
- *compute feedback* traite les entrées provenant du *preMonitoring* et déduit l'état du véhicule.
- L'état du véhicule et les ordres du conducteur sont utilisés par le bloc *control* pour placer le logiciel dans l'état correspondant à celui du véhicule. Il s'agit d'une machine à états. Le mode choisi est transmis au bloc *algorithms* ;
- En fonction du mode du composant, le bloc *algorithms* applique les algorithmes adéquats. À partir des données lues, les algorithmes de régulation déduisent les nouvelles consignes pour piloter les actionneurs ;
- *postMonitoring* est un bloc de sûreté chargé d'effectuer un contrôle de vraisemblance sur les résultats du bloc *algorithms*. Pour cela, il récupère les mêmes entrées ainsi que la sortie de *algorithms*. En cas de défaillance, le bloc *control* est informé via un flag d'erreur.

Nous ciblons les exigences en termes de flots de données. Dans l'Annexe B.1, vous trouverez pour chacun des sept composants les informations relatives aux signaux partagés entre les blocs.

7.3 Écriture des exigences pour *PROTOSAR*, dans la phase de développement, pour le calculateur

L'écriture des exigences apparaît très tôt dans le processus de développement du système. En effet, l'écriture des exigences en langage naturel est menée en amont, c'est-à-dire avant la traduction de l'architecture fonctionnelle en termes d'architecture logicielle (sur multicœur).

Le périmètre de notre étude se résume aux composants dont l'architecture fonctionnelle a été présentée par la Figure 7.4. L'analyse effectuée sur ce pattern sera répercutée sur l'ensemble des composants à considérer, en tenant compte des signaux correspondant à chaque cas (c.f. Annexe B.1). Pour chacune des exigences ci-dessous, le modèle du système et la formule LTL sont décrits dans l'Annexe B.2.

7.3.1 Exigences sur les productions et consommations de données

Les Exigences suivantes sont liées à la surveillance de la production et la consommation des données.

Dans les Exigences 7.1, 7.2 et 7.3, nous vérifions que toutes les données écrites dans un buffer sont réellement consommées.

Exigence 7.1. *Toutes les sorties* safeInputs *produites par le bloc* preMonitoring *sont lues par le bloc* algorithms.

Exigence 7.2. *Tous les flags* fctFbcks *produits par le bloc* compute feedback *sont lus par le bloc* control.

Exigence 7.3. *Toutes les sorties* reqOutputs *produites par le bloc* algorithms *sont lues par le bloc* postMonitoring.

Dans l'Exigence 7.4, nous présentons une variante en autorisant un nombre maximum de données pouvant être perdues (échantillonnage).

Exigence 7.4. *Au moins une fois sur trois, les sorties* safeInputs *produites par le bloc* preMonitoring *sont lues par le bloc* compute feedback.

Enfin, les Exigences 7.5 et 7.6 sont basées sur le même principe sauf que la valeur de la donnée est prise en compte. En effet, seule la valeur des flags indiquant une erreur (i.e. valeur égale à 1) doit être absolument consommée. Les règles de ce type permettent d'assouplir la surveillance en ne se préoccupant que des cas où il y a une erreur.

Exigence 7.5. *Dès lors qu'une erreur est détectée par le bloc* preMonitoring *(i.e. flagError égal à 1), le flag doit systématiquement être lu par le bloc* control.

Exigence 7.6. *Dès lors qu'une erreur est détectée par le bloc* postMonitoring *(i.e. flagError2 égal à 1), le flag doit systématiquement être lu par le bloc* control.

7.3.2 Exigences sur la propagation des données

Les Exigences 7.7 et 7.8 sont des exigences *temporelles* dans le sens où nous vérifions que l'occurrence d'une erreur conduit bien à l'arrêt du pilotage des sorties dans un temps dicté par le nombre d'activations du bloc *algorithms*.

Exigence 7.7. *Dès lors qu'une erreur est détectée par le bloc* preMonitoring*, le bloc* algorithms *doit stopper définitivement la diffusion de ses sorties* reqOutputs *au bout d'au plus deux activations de ce bloc. Un signal d'état dédié valant 0 en sortie du bloc* algorithms *permet de connaître cette information.*

Exigence 7.8. *Dès lors qu'une erreur est détectée par le bloc* postMonitoring*, le bloc* algorithms *doit stopper définitivement la diffusion de ses sorties* reqOutputs *au bout d'au plus deux activations de ce bloc. Un signal d'état dédié valant 0 en sortie du bloc* algorithms *permet de connaître cette information.*

7.3.3 Exigences sur la cohérence des données

Les Exigences 7.9 et 7.10 imposent qu'un ensemble de données soit lu de manière cohérente, c'est-à-dire que toutes les données doivent avoir la même instance au moment de leur lecture.

Exigence 7.9. *Toutes les données* safeInputs *lues par le bloc* compute feedback *doivent obligatoirement être synchronisées pendant la lecture.*

Exigence 7.10. *Toutes les données* safeOutputs *lues par le bloc* preMonitoring *doivent obligatoirement être synchronisées pendant la lecture.*

L'Exigence 7.11 correspond à la surveillance du contrôle de vraisemblance. En effet, un module de contrôle de vraisemblance doit systématiquement lire les mêmes entrées *safeInputs* que le bloc à contrôler (i.e. *algorithms*) ainsi que les sorties *reqOutputs* produites par ce bloc (à partir de ces entrées).

Exigence 7.11. *Le contrôle de vraisemblance effectué par le bloc* postMonitoring *doit être cohérent.*

Enfin, l'Exigence 7.12 impose que lorsqu'une donnée est lue par un bloc, la même instance doit être lue par un autre. Cette cohérence est essentielle ici, car le bloc *postMonitoring* utilise ces données pour prendre des décisions impactant la sûreté du système.

Exigence 7.12. *Quand les données* reqFct *sont lues par* algorithms*, elles doivent obligatoirement être également lues par* postMonitoring*.*

7.3.4 Nombre de moniteurs à synthétiser

Dans la Table 7.1, nous analysons à partir des messages existant entre chacun des blocs le nombre de moniteurs qu'il faudrait générer pour chacun des composants, pour chacune des exigences. Ces résultats sont déduits de l'analyse par composant, des signaux se trouvant entre chaque bloc.

Exigence	Trajectoire	Vitesse	Frein parking	Levier vitesse	Angle volant	Accelération	Freinage	Total
7.1	2	7	2	3	3	6	29	**52**
7.2	0	2	1	1	1	2	1	**8**
7.3	2	2	2	2	3	2	4	**17**
7.4	0	3	3	3	3	3	3	**18**
7.5	1	1	1	1	1	1	1	**7**
7.6	1	1	1	1	1	1	1	**7**
7.7	1	1	1	1	1	1	1	**7**
7.8	1	1	1	1	1	1	1	**7**
7.9	0	1	1	1	1	1	1	**6**
7.10	0	0	1	1	0	0	1	**3**
7.11	4	14	4	6	9	12	116	**165**
7.12	1	1	1	1	1	1	1	**7**
Total	**13**	**34**	**19**	**22**	**25**	**31**	**160**	**304**

Table 7.1 – Nombre de moniteurs par exigence pour chaque composant

Les exigences précédentes sont déduites de l'architecture fonctionnelle de l'application. Au moment de l'étude, le projet PROTOSAR est en phase de conception de l'architecture logicielle. À ce stade du processus de développement, la répartition des composants fonctionnels dans les tâches ainsi que leur période ne sont pas connues. Pour effectuer une analyse au pire-cas, nous supposons dans un premier temps que pour tous les composants, chaque bloc est associé à une tâche distincte. Les résultats obtenus sont pessimistes et les choix liés à la mise en place de l'architecture logicielle permettront de réduire le nombre de moniteurs à synthétiser.

Pour les mêmes raisons, l'analyse en-ligne de la solution n'est pas effectuée.

7.4 Évaluation du coût associé à la synthèse hors-ligne des moniteurs

7.4.1 Taille des tables de transitions

Les résultats de l'empreinte mémoire requise en ROM pour stocker les moniteurs sont donnés dans le Tableau 7.2. Nous constatons globalement que l'empreinte mémoire par moniteur est faible et dépend principalement de la taille du modèle du système. Notons que le codage des tables de transitions a été effectué avec la méthode d'amélioration n°2 (déduit hors-ligne par *Enforcer*) pour toutes les exigences. Cela permet de gagner 49% de l'espace mémoire qui serait nécessaire sans aucune optimisation. En effet, initialement, il fallait 14.52 ko là où nous en avons besoin réellement de 7.4 ko de ROM pour l'ensemble du calculateur.

Exigence	N_s	N_e	N_s **du moniteur intermédiaire**	**Taille d'une table de transitions**	**Nombre de moniteurs**	**Taille des tables de transitions**
7.1	3	2	2	2 octets	52	104 octets
7.2	3	2	2	2 octets	8	16 octets
7.3	3	2	2	2 octets	17	34 octets
7.4	4	2	2	3 octets	18	54 octets
7.5	5	4	3	8 octets	7	56 octets
7.6	5	4	3	8 octets	7	56 octets
7.7	17	5	3	54 octets	7	378 octets
7.8	17	5	3	54 octets	7	378 octets
7.9	20	6	3	75 octets	6	450 octets
7.10	6	4	3	9 octets	3	27 octets
7.11	14	5	3	35 octets	165	5.775 koctets
7.12	5	3	3	6 octets	7	42 octets
Total	–	–	–	–	**304**	**7.37 koctets**

Table 7.2 – Impact mémoire des moniteurs

Nous observons que l'Exigence 7.11 prend à elle seule 78% de l'espace mémoire requis pour les tables de transitions. Cette exigence vise à vérifier la cohérence des données lues par le bloc *postMonitoring* pour effectuer le contrôle de vraisemblance sur les opérations du bloc *algorithms*. Comme nous le signalons dans l'Annexe B, nous vérifions la cohérence pour chaque paire d'entrées/sorties du bloc *algorithms*. Nous avons donc besoin d'un nombre de moniteurs largement supérieur au nombre de signaux surveillés. Pour un composant donné, le nombre de moniteurs nécessaires pour cette exigence est donné par *nombre d'entrées safeInputs* * *nombre de sorties reqOutputs*.

Notons également que l'occurrence d'un évènement sur la sortie de *algorithms* conduit à l'évolution d'autant de moniteurs qu'il y a d'entrées *safeInputs*, et réciproquement. Ce procédé peut se révéler très problématique dans certains cas. Par exemple, le bloc *algorithms* du composant de *freinage* possède 29 entrées et 4 sorties. Puisque l'évolution de 29 moniteurs sous l'occurrence d'un évènement paraît prohibitif, nous discuterons des possibilités nous permettant d'améliorer ce résultat.

7.4.2 Taille des descripteurs de moniteurs et du code

L'implémentation des tables de transitions est effectuée avec la méthode d'amélioration n°2. Chaque moniteur requiert donc 18 octets pour son descripteur. Pour les 304 moniteurs du calculateur, l'espace mémoire requis pour les descripteurs de moniteurs vaut 5.5 ko. En prenant en compte la répartition RAM/ROM sachant que seul l'état courant nécessite d'être stocké en RAM, l'espace mémoire requis est de 304 octets en RAM et 5.2 ko en ROM.

Enfin, puisque toutes les propriétés utilisent la même méthode d'optimisation, la fonction de mise à jour est unique et occupe 328 octets en ROM.

7.4.3 Taille du gestionnaire d'évènements

La taille du gestionnaire d'évènements dépend du nombre d'évènements à intercepter pour surveiller les exigences. La Table 7.3 donne le nombre d'évènements par exigence pour chaque composant.

Pour les Exigences 7.5, 7.6, 7.7 et 7.8, on peut noter que le nombre d'évènements à intercepté est inférieur au paramètre N_e exprimé dans la Table 7.1. En effet ces règles nécessitent de connaître si le message considéré vaut une certaine valeur ou pas. Un même évènement au niveau du système conduit à plusieurs évènements au niveau du service de surveillance.

Exigence	Trajectoire	Vitesse	Frein parking	Levier vitesse	Angle volant	Accelération	Freinage	Total
7.1	4	14	4	6	6	12	58	**104**
7.2	0	4	2	2	2	4	2	**16**
7.3	4	4	4	4	6	4	8	**34**
7.4	0	6	6	6	6	6	6	**36**
7.5	2	2	2	2	2	2	2	**14**
7.6	2	2	2	2	2	2	2	**14**
7.7	3	3	3	3	3	3	3	**21**
7.8	3	3	3	3	3	3	3	**21**
7.9	0	6	6	6	6	6	6	**36**
7.10	0	0	4	4	0	0	4	**12**
7.11	10	25	10	13	15	22	95	**190**
7.12	3	3	3	3	3	3	3	**21**
Total	**31**	**72**	**49**	**54**	**54**	**67**	**192**	**519**

Table 7.3 – Nombre d'évènements par exigence, pour chaque composant

A priori, un total de 519 évènements devraient être considéré. Cependant, certains de ces évènements sont couplés et peuvent faire évoluer plusieurs moniteurs. Pour prendre en compte ce couplage, nous devons analyser les signaux qui sont utilisés plusieurs fois. Nous observons que :

- Tous les évènements de l'Exigence 7.1 sont réutilisés dans l'Exigence 7.11 ;
- Tous les évènements de l'Exigence 7.3 sont réutilisés dans l'Exigence 7.11 ;
- Tous les évènements de l'Exigence 7.4 sont réutilisés dans l'Exigence 7.11 ;
- Tous les évènements de l'Exigence 7.9 sont réutilisés dans l'Exigence 7.11 ;
- Tous les évènements de l'Exigence 7.5 sont réutilisés dans l'Exigence 7.7 ;

– Tous les évènements de l'Exigence 7.6 sont réutilisés dans l'Exigence 7.8.

Cette analyse permet de réduire le nombre d'évènements à 281. Nous avons donc besoin de 281 fonctions dédiées à leur traitement. Pour connaître l'empreinte mémoire du gestionnaire d'évènements, il faut également tenir compte du nombre de moniteurs à faire évoluer sous l'occurrence d'un évènement. Dans notre cas, 63 événements font évoluer 1 moniteur, 56 en font évoluer 2, 33 en font évoluer 3, 52 en font évoluer 4, 32 en font évoluer 5, 33 en font évoluer 6, 4 en font évoluer 7 moniteurs et 8 en font évoluer 29.

Dans ces conditions, il résulte que le gestionnaire d'évènements prend 13.2 ko, en ROM.

7.4.4 Taille des tables d'évènements

Il faut également prendre en compte la taille des deux tables d'évènements. Pour cela, il faut connaître le nombre total de signaux et le nombre de tâches du système. À cette étape du processus de développement de l'application, la répartition des composants fonctionnels dans les tâches n'est pas connue. Nous ne discuterons donc pas ce paramètre.

7.5 Bilan et améliorations

En additionnant les résultats précédentes, l'empreinte mémoire totale est de 26.1 ko en ROM et 304 octets en RAM pour le calculateur UCM entier. Le codage des moniteurs représente 30% de l'empreinte mémoire, le code et les descripteurs de moniteurs représentent 20% et le gestionnaire d'évènements, 50%. En analysant ce résultat, nous remarquons que la contrainte mémoire en RAM est acceptable. L'empreinte mémoire en ROM est quant à elle beaucoup plus importante. En observant que sur les microcontrôleurs actuels tels le Leopard, on dispose de 1 à 2 Mo en RAM, ce résultat peut être acceptable. On peut cependant améliorer les résultats obtenus. Concernant l'empreinte mémoire, les améliorations apportées au codage de la table de transition ont déjà montré qu'on pouvait sauver 50% d'espace mémoire pour leur stockage. Il est également possible de réduire l'impact des descripteurs de moniteurs en supprimant des champs. Cependant, il faudrait les recalculer en ligne et les performances temporelles du mécanisme seraient pénalisées. Enfin, la moitié de l'empreinte mémoire est due au nombre d'évènements qui doivent être interceptés. *Pour améliorer l'efficacité du service de ce point de vue, nous allons chercher à diminuer le nombre d'évènements du système requis pour l'observation.*

L'analyse de l'impact temporel de la solution sur l'ensemble de l'application relève de plusieurs facteurs : la fréquence d'appel du service et le nombre de moniteurs devant évoluer pour chaque appel. Le premier facteur ne sera pas discuté puisqu'il dépend principalement de l'ordonnancement (e.g. allocation des tâches, choix des périodes), ce qui n'est pas connu à ce stade du déroulement du projet. Pour le second facteur, nos résultats montrent qu'au pire 29 évènements doivent évoluer suite à l'occurrence d'un même évènement. *Pour améliorer l'efficacité du service de ce point de vue, nous chercherons à réduire le nombre de moniteurs devant évoluer sous l'occurrence d'un évènement.*

Deux solutions sont à étudier. Nous discutons dans un premier temps de la mise en œuvre de *moniteurs composés*, puis enfin de l'émission *d'hypothèses sur l'implémentation.*

7.5.1 Composition des moniteurs

La composition des moniteurs a deux avantages.

Premièrement, elle permet de réduire le nombre de moniteurs devant évoluer sous l'occurrence d'un évènement, ce qui a pour effet de réduire le temps CPU nécessaire à la vérification. Pour la mettre en pratique, il faut analyser les exigences pour lesquelles l'intersection des ensembles d'évènements à intercepter est non vide.

Par exemple, l'Exigence 7.11 est bien adaptée pour la composition. En effet, si on choisit de synthétiser un unique moniteur pour cette règle, nous pouvons diviser le temps CPU requis pour la vérification de cette exigence par 29 dans le cas du composant de freinage. Notons que cela permet également de diminuer le nombre de moniteurs de plus de 70%, ce qui réduit d'autant l'espace mémoire réservé au stockage des descripteurs de moniteurs. En revanche, il n'y a aucun gain sur le gestionnaire d'évènements car même après la composition, il faut encore intercepter tous les évènements. Pour cibler ce problème, il faudra faire des hypothèses sur l'application.

Deuxièmement, elle permet également sous certaines conditions d'économiser de l'espace en mémoire au niveau des tables de transitions. Pour illustrer cela, considérons par exemple le cas des Exigences 7.7 et 7.8, qui manipulent les mêmes données de sortie du bloc *algorithms*. Dans ce cas précis, la composition des exigences 7.7 et 7.8 pour lesquelles nous avons 17 états et 5 évènements chacun (i.e. 108 octets) conduit à la production d'un moniteur de 17 états et 7 évènements (i.e. 75 octets). Sur les 7 moniteurs de ce type, le gain total représente 26%.

7.5.2 Hypothèses liées à l'implémentation

Il est aussi possible de diminuer l'empreinte mémoire en effectuant des hypothèses sur l'implémentation, principalement sur l'ordre d'envoi et réception des messages. Là encore, utilisons l'Exigence 7.11 pour illustrer le gain mémoire offert par ces hypothèses. Comme présentées dans l'Annexe B.2, les propriétés de ce type peuvent être surveillées en interceptant uniquement les évènements sur le premier et le dernier des buffers. Pour chaque composant, nous aurons alors un unique moniteur de 8 états et de 7 évènements, ce qui permet de passer de 5775 octets à 196 octets, soit un gain de 96%. De plus, la surveillance de cette exigence requiert l'interception de 49 évènements à l'échelle du calculateur (au lieu de 190 précédemment). Cependant, le gain apporté sur le gestionnaire d'évènements n'est pas aussi important puisque beaucoup d'évènements non utilisés ici le sont quand même pour d'autres exigences. Au total, ce sont tout de même 52 évènements gagnés.

Un autre exemple est celui des Exigences 7.9, 7.10. En effet, nous pouvons modéliser l'ordre de lecture ou d'écriture des messages. En supposant par avance que nous connaissons cet ordre, le modèle du système est là aussi simplifié. Pour l'Exigence 7.9, nous passons d'un moniteur de 20 états, 6 évènements à un moniteur de 8 états, 6 évènements. Pour l'Exigence 7.10, nous passons d'un moniteur de 6 états, 4 évènements à un moniteur de 4 états, 4 évènements.

7.5.3 Conclusion

Il existe donc des solutions pour réduire l'impact de la surveillance. Pour cela, nous choisissons d'apporter les améliorations suivantes :

– Les Exigences 7.11, 7.9 et 7.10 sont traitées en effectuant des hypothèses sur l'ordre d'émission/réception des messages dans les buffers ;

– Les Exigences 7.7 et 7.8 sont composées.

Il serait possible d'apporter d'autres améliorations. Cependant, nous choisissons de ne pas composer les exigences 7.5 et 7.6 pour pouvoir diagnostiquer rapidement si la cause de l'erreur provient du *preMonitoring* ou du *postMonitoring*. Pour les mêmes raisons, nous choisissons de ne pas composer les Exigences 7.1, 7.3 et 7.11.

Ces améliorations permettent d'économiser 52 évènements et 165 moniteurs. Sur l'ensemble du calculateur, nous constatons les gains suivants :

– La table de transitions passe de 7.4 ko à 2.4 ko en ROM, soit un gain de 68%. Ceci est principalement dû aux hypothèses liées à l'implémentation ;
– La taille des descripteurs de moniteurs passe de 5.2 ko à 2.2 ko en ROM, soit un gain de 58%. Ceci est dû au fait que nous passons de 304 moniteurs à 139. La taille requise en RAM passe de 304 octets à 139 octets ;
– La taille du code reste inchangé en ROM : 328 octets ;
– La taille des gestionnaires d'évènements passe de 13.2 ko à 3.7 ko, soit un gain de 72%. Ce gain est dû au fait que 52 évènements n'ont plus besoin d'être interceptés et qu'au plus un même évènement fera évoluer 4 moniteurs. Plus précisément, 162 évènements ne font évoluer que 1 moniteur, 53 en font évoluer 2 et 14 en font évoluer 4. Cela permet également un gain temporel.

Au total, nous montrons qu'à l'échelle du calculateur, 139 octets sont requis en RAM et 8.7 ko sont requis en ROM (plus de 66% de gain).

Notons que dans la suite du projet, les choix au niveau de l'architecture opérationnelle vont permettre de trier les règles effectivement requises. En effet, si certains blocs sont alloués dans les mêmes tâches, les propriétés entre ces blocs seront implicitement vérifiées par les choix de l'architecture. L'empreinte mémoire finale sera donc plus faible.

Chapitre 8

Discussion

Le développement d'une application de contrôle-commande multitâche temps réel nécessite de maîtriser, entre autres, les flots de données et de contrôle entre les tâches. L'état actuel des connaissances et pratiques de l'ordonnancement temps réel multicœur montrent que pour ces architectures, il est difficile d'atteindre le niveau de maîtrise requis. Il est alors nécessaire d'utiliser les outils de la sûreté de fonctionnement pour assurer l'absence d'occurrence de défaillance. Dans ce contexte, nous nous sommes intéressés à la vérification en ligne, qui permet de construire des moniteurs, éléments de base d'une stratégie de tolérance aux fautes.

Nous avons montré comment utiliser ce mécanisme dans le processus de développement d'une application, afin de vérifier des propriétés inter-tâches. Concrètement, les exigences issues de l'analyse de l'architecture fonctionnelle sont utilisées pour écrire des propriétés à vérifier et un modèle du système (par exemple obtenu par l'analyse des architectures fonctionnelle, logicielle et/ou opérationnelle) est utilisé pour synthétiser des moniteurs. L'outil *Enforcer* conçu dans le cadre de la thèse permet de les synthétiser hors-ligne. Un DSL a été conçu pour fournir à l'outil la description des entrées nécessaires.

Pour démontrer l'utilisation de la vérification en ligne dans le contexte industriel automobile, nous avons intégré *Enforcer* à la chaîne de compilation de *Trampoline*, un RTOS compatible AUTOSAR. En plus d'injecter les moniteurs dans le noyau, cette intégration permet également d'instrumenter le système afin d'intercepter tous les évènements nécessaires à la vérification. Notre étude se focalise sur le contrôle des flots de données, mais peut être étendue à tous les services du noyau.

Enfin, une évaluation en-ligne a été effectuée afin de déterminer le coût temporel de la solution sur les services instrumentés du système d'exploitation. Nos résultats démontrent que nos choix d'implémentation (i.e. service intégré dans l'OS, moniteurs synthétisés hors-ligne) sont compatibles avec les contraintes industrielles. Une étude de cas sur un projet industriel a aussi permis d'évaluer le coût du mécanisme en termes d'empreinte mémoire à l'échelle d'un calculateur. Les résultats obtenus démontrent également que le coût nécessaire à la vérification des propriétés inter-tâches est tout à fait acceptable vis-à-vis du gain de sûreté de fonctionnement obtenu.

Puisque la version multicœur de *Trampoline* n'est pas encore disponible, notre implémen-

tation ne cible pas ces architectures. Pour porter le service de vérification en ligne dans un tel environnement, il faut prendre en compte certains effets inhérents au parallélisme :

- Tenir compte du fait qu'on utilise un moniteur séquentiel pour observer un environnement parallèle ;
- Veiller à la cohérence des descripteurs des moniteurs, en particulier, de l'état courant.

Concernant le premier point, le phénomène majeur à prendre en compte concerne la *simultanéité* des évènements se produisant sur différents cœurs. Dans ce cas, l'ordre d'observation des évènements par le service peut être différent de leur ordre d'occurrence. Concrètement, nous choisissons de privilégier l'ordre d'observation des évènements par le service de vérification en ligne. Il faut cependant veiller à ce que le mécanisme soit robuste aux *faux positifs*. En effet, si un évènement a se produit un très court instant avant un évènement b, le service de vérification en ligne peut potentiellement les percevoir dans l'ordre inverse. Il faut donc veiller à ce que le mécanisme permette de détecter ces faux positifs. Cela implique vraisemblablement d'agir aux différentes étapes : hors ligne lors de l'écriture de la formule et de la génération du moniteur pour permettre de caractériser et permettre le signalement des situations potentiellement problématique, puis en ligne pour interpréter cette remontée d'information supplémentaire.

Concernant le second point, il faut également s'assurer de la cohérence des données partagées. Dans notre cas, seul l'état courant du moniteur (modélisant l'état du système à un instant t) est une variable. On peut alors envisager plusieurs solutions pour la protection du service de surveillance. Premièrement, on peut inclure l'ensemble du mécanisme dans une section critique. Ce verrouillage gros-grain permet de sérialiser tous les accès au service. Une approche avec un verrouillage plus fin est également possible, qui assure l'accès en exclusion mutuelle à la section de code critique, qui comprend la recherche dans la table de transition et la mise à jour de l'état courant. Ce verrouillage peut être réalisé de façon simple avec une boucle incluant la section critique et une mise à jour de l'état avec une instruction atomique type CAS.

Enfin, des perspectives sont également ouvertes concernant l'extension du type de propriétés à vérifier. Nos propriétés sont décrites à l'aide de formules LTL. Les moniteurs réagissent uniquement sur des évènements discrets et sont donc incapables de surveiller des propriétés temps réel. Une suite naturelle à nos travaux serait de s'intéresser à la vérification en ligne de propriétés inter-tâches temps réel. Notons tout d'abord qu'AUTOSAR propose un service de protection temporelle qui permet de s'assurer que le comportement des tâches est conforme au modèle utilisé pour valider l'ordonnancement du système (Bertrand, 2011). Pour les autres types de propriétés, il faudrait se tourner vers les outils de la théorie des systèmes temporisés : langages temporisés, logique temps réel telles que MTL (*Metric Temporal Logic*) ou TLTL (*Timed LTL*), automates temporisés, etc. Les travaux existant dans ce domaine montrent que la synthèse des moniteurs est complexe puisque le moniteur doit prendre en compte tous les futurs possibles. C'est l'écoulement du temps et plus précisément le passage des frontières temporelles qui permet de décider en ligne les chemins qui ne seront pas pris. Ainsi, Robert *et al.* (2010) montrent les différentes difficultés induites pour la vérification en ligne de propriétés temps réel. Ces travaux montrent en particulier que la taille de la table de transition nécessaire à la surveillance d'une propriété peut rapidement devenir un problème dans un contexte industriel embarqué, pour lequel des solutions sont encore à trouver.

Troisième partie

STM-HRT : un protocole logiciel wait-free à base de mémoire transactionnelle pour les systèmes temps réel embarqués multicœur critiques

Chapitre 9

Introduction au partage de ressources non bloquants

Sommaire

Nous nous intéressons aux problèmes liés à la difficulté d'implémentation du partage des données. Nous proposons d'étudier une alternative aux protocoles de synchronisation bloquant, c'est-à-dire une approche non bloquante pour laquelle nous n'avons pas besoin de verrous. Nos objectifs sont de faciliter l'étape d'implémentation des sections critiques, de réduire l'occurrence de fautes de codage, et ainsi d'accroître la robustesse d'une application (c.f. Section 3.3, page 43).

9.1 Garanties de progression des processus manipulant des ressources partagées

On distingue trois types de garanties de progression des transactions : *obstruction-free*, *lock-free* et *wait-free*.

Définition 9.1. *La garantie de progression obstruction-free assure qu'un processus exécuté en totale isolation peut progresser.*

Elle représente la plus faible garantie de progression et n'exclut pas l'occurrence de famines. Cette faible garantie est inappropriée pour des systèmes temps réel.

Définition 9.2. *La garantie de progression lock-free assure la progression d'au moins un processus (malgré la concurrence).*

Pour les autres, il n'est pas possible de définir une borne maximale de leur duré d'exécution. Cette garantie permet d'assurer des contraintes temps réel souples et ces implémentations sont évaluées en termes de qualité de service (i.e. ratio du nombre de processus exécutées dans le respect de leur échéances sur le nombre total de processus exécutées).

Définition 9.3. *La garantie de progression wait-free assure la progression de toutes les transactions.*

Cette garantie se révèle particulièrement bien adaptée aux systèmes temps réel dur.

9.2 Protocoles de synchronisation multicœur non bloquants

Bien que peu mises en œuvre dans un contexte temps réel embarqué, des recherches actuelles visent à considérer des protocoles de synchronisation non bloquants, c'est-à-dire des protocoles qui autorisent l'accès parallèle aux ressources tout en garantissant la cohérence de ceux-ci.

Avec ces approches, aucun verrous n'est utilisé mais il devient nécessaire de passer des tests de validation pour vérifier la présence ou non de conflits. Les notions d'inversion de priorité ou de blocage disparaissent puisqu'un gestionnaire de contention permet de choisir les tâches à privilégier en appliquant des heuristiques de résolution des conflits.

Les contraintes temps réel dur auxquelles sont soumis les systèmes embarqués imposent de garantir une progression wait-free. À ce jour, il existe quelques implémentations de mécanisme adaptées à ce type de garantie de progression. En revanche, elles mettent en jeu des processus parallèles non temps réel.

Dans Herlihy (1993), *Herlihy* décrit une implémentation wait-free d'objets concurrents dans le contexte d'enfiler et défiler une file. L'implémentation proposée permet de simplifier la programmation parallèle en laissant le protocole gérer les conflits. La garantie de progression est assurée par un mécanisme d'*aide*. En pratique, les structures de données (partagées par tous les processus) peuvent identifier à tout moment les objets qui sont en cours d'accès (via une table dans laquelle les processus annoncent les objets manipulés). Pour éviter les conflits, chaque processus aide les processus concurrent avant de réaliser ses opérations. La solution est ensuite évaluée et comparée avec les approches bloquantes traditionnelles telles les spinlocks. Les résultats montrent qu'en présence d'un fort taux de contention, la méthode wait-free est moins performante que l'approche bloquante (ceci est principalement dû au mécanisme d'aide). Seulement, l'augmentation du nombre de cœurs réduit cet écart et permet même d'inverser la tendance. Le protocole est évalué sur des objets de petite taille et de grande taille. Des optimisations ont également été apportées en utilisant des BEB (Binary Exponential Backoff) pour réduire les contentions.

Dans Anderson et Moir (1995) et Anderson et Moir (1999), une implémentation wait-free d'un Multi-Word Compare And Swap (MWCAS) à base d'instructions Load Link/Store Conditional est présentée. L'implémentation proposée est dérivée de celle présentée dans Herlihy (1993). La procédure MWCAS est la procédure de base pour l'implémentation d'opérations accédant à de multiples objets puisqu'elle permet de modifier *atomiquement* l'ensemble des mots mémoire passés en paramètres. Par analogie au concept de transaction que nous définirons dans la Section 9.3 (page 114), le MWCAS semble s'exécuter de manière atomique aux yeux des tâches parallèles, c'est-à-dire que soit *tous* les mots mémoire sont *mis à jour en même temps*, soit aucun ne l'est. Un processus MWCAS $processus_1$ échoue si au moins une des valeurs à mettre à jour a été modifiée par un autre processus. Le protocole présenté est wait-free dans le sens où il est possible de donner un résultat sur le MWCAS (i.e. succès ou échec) en un nombre fini d'étapes. Dans Moir (1997), *Moir* propose une variante intitulée *Conditionally Wait-Free MWCAS*. Cette variante se base sur une garantie de progression lock-free mais elle utilise un mécanisme d'aide externe autorisé à annuler les opérations MWCAS qui ne parviendront pas à terminer. Le comportement global est wait-free.

Contrairement à l'objectif visé dans nos travaux, il n'y a aucune garantie sur le succès des opérations dans chacune de ces implémentations. L'implémentation proposée dans ces travaux est analogue à un protocole à *mémoire transactionnelle*. En effet, des structures de données permettent de décrire l'état d'un processus qui a engagé une opération MWCAS, ou encore de stocker l'état des objets manipulés (i.e. qui manipule l'objet, stockage de l'ancienne et la nouvelle valeur, etc ..). Le protocole quant à lui permet de gérer la consistance de l'ensemble des objets à mettre à jour (i.e. objets ouverts en lecture/écriture dans un contexte transactionnel) à l'image du commit d'une transaction (c.f. Section 9.3, page 114). Comme déjà vu dans les travaux de Herlihy, un mécanisme d'*aide* est essentiel pour assurer la progression wait-free. On distingue trois types d'aide : l'aide incrémentale, l'aide cyclique et l'aide basée sur les priorités (Anderson *et al.*, 1997b,a). Ici, c'est une aide incrémentale qui est utilisée. Elle consiste à aider toutes les transactions avec lesquelles on est en conflit en profondeur, c'est-à-dire que si une transaction ω_i est en conflit avec ω_j et ω_j est elle-même en conflit avec ω_k, ω_i va aider ω_j puis aider ω_k au nom de ω_j, etc... L'*aide cyclique* se base sur l'utilisation d'un compteur qui désigne le processeur sur lequel se trouve le processus à aider. À tout instant, ce compteur correspond à l'identifiant du processeur qui exécute un processus d'un autre processeur qui nécessite de l'aide. Enfin, l'*aide basée sur la priorité* repose sur l'utilisation d'un compteur désignant le processeur sur lequel s'exécute la tâche de plus forte priorité. Cette tâche sera

aidée en premier.

Dans Tsigas et Zhang (1999), les auteurs décrivent une solution non bloquante pour l'accès en lecture et en écriture à un buffer dans des systèmes multicœur sous ordonnancement P-RM. Les opérations en écriture sont wait-free mais les opérations en lecture sont lock-free. Chaque buffer possède au plus (R + W + 1) entrées, R étant le nombre de lecteurs et W le nombre d'écrivains. Pour chaque entrée du buffer, un champ dédié permet de connaître l'état de l'entrée : une valeur positive si l'entrée possède la valeur la plus récente, une valeur négative si la donnée n'est plus fraîche ou si un écrivain travaille dessus.

Enfin, les travaux de *Chen et Burns* (Chen et Burns, 1997) présentent un protocole wait-free pour un schéma de communication 1 : m (*1 écrivain, m lecteurs*). L'algorithme, appelé *algorithme de Chen*, permet de gérer la cohérence des données tout en fournissant aux lecteurs la donnée la plus récente. Pour fonctionner, une table de $m + 1$ copies doit être utilisée par donnée partagée. La valeur en cours de l'objet peut se trouver n'importe où dans ce tableau. Pour mettre à jour une donnée, un écrivain doit donc récupérer un emplacement libre du tableau, c'est-à-dire différent de celui de la copie concurrente (i.e. valeur publique) ou d'un emplacement utilisé par un autre écrivain. La donnée est ensuite copiée à cet emplacement et devient la nouvelle valeur courante de la donnée. Toutes les instructions atomiques sont réalisées à l'aide de *Compare & Swap (CAS)*. D'autres travaux ont repris l'algorithme de Chen dans le but d'améliorer les performances mémoire. Par exemple, dans Song et Choi (2003) et Cho *et al.* (2007) des optimisations basées sur l'analyse de l'ordonnancement et des préemptions permettent d'affiner le nombre réel d'emplacements nécessaires dans la table des copies. Dans les travaux de thèse, nous considèrons nous aussi un schéma de communication 1 : m, mais pour répondre à nos contraintes, nous souhaitons gérer la cohérence d'un groupe de données (c.f. Section 3.3, page 43), ce qui complexifie l'approche. Des expérimentations menées sous $LITMUS^{RT}$ sont disponibles dans Brandenburg *et al.* (2008).

Pour gérer la concurrence d'un groupe de données, d'autres approches non bloquant à base de *mémoire transactionnelle* peuvent être étudiées. Notons que la cohérence d'un groupe de donnée signifie qu'aucune des données du groupe ne doit être modifiée pendant l'accès au groupe.

9.3 Les mécanismes à mémoire transactionnelle

Le concept de *mécanisme transactionnel* apparaît dès la fin des années 1970 (Eswaran *et al.*, 1976). Ces mécanismes sont initialement utilisés pour gérer la cohérence des accès aux données partagées dans une base de données.

Les mémoires transactionnelles désignent plus particulièrement l'accès à des ressources dans une mémoire partagée où les accès sont concurrents. C'est le cas dans les microcontrôleurs multicœur à mémoire commune. Les accès aux ressources partagées sont réalisés à l'intérieur d'une transaction (c.f. Section 9.3.1). Un protocole de contrôle de concurrence non bloquant permet d'assurer la consistance de ces accès. L'implémentation du protocole repose sur l'utilisation de structures de données spécifiques qui sont utilisées lors des phases de détection et de résolution des conflits.
Il faut également disposer d'instructions de base sur la cible pour pouvoir réaliser des opérations atomiques. Une taxonomie de ces instructions est proposée dans la Section 9.4.1 (page 120).

9.3.1 Qu'est-ce qu'une transaction ?

Les concepts présentés ci-après sont issus de (Wang *et al.*, 2010; Marshall, 2005) et de la seconde édition de l'ouvrage *Transactional Memory* (Harris *et al.*, 2010).

Définition 9.4. *Une transaction est une séquence d'actions qui apparaît comme indivisible et instantanée pour un observateur extérieur.*

Une transaction satisfait un ensemble de propriétés, communément référencé par le terme ACID (Atomicité, Cohérence, Isolation, Durabilité) (Haerder et Reuter, 1983).

- L'*atomicité* suppose que les transactions se déroulent en *tout-ou-rien*, c'est-à-dire que soit *toutes* opérations sont réalisées sont rendues visibles (en cas de succès), soit *aucune* opération n'est rendue visible (en cas d'abandon). En cas d'abandon, des mécanismes doivent être mis en œuvre pour assurer un retour en arrière consistant ;
- La *cohérence* est respectée si à partir d'un état cohérent avant l'exécution d'une transaction, les modifications apportées par celle-ci conduisent à un autre état cohérent,une fois la transaction validée ;
- L'*isolation* signifie que le résultat produit par une transaction est indépendant du fait qu'elle s'exécute seule ou en concurrence avec d'autres :
- La *durabilité* concerne le stockage des ressources. Ce stockage doit assurer la disponibilité de la donnée dans le temps. Cet attribut est un héritage des systèmes de gestion de bases de données et peut être relaxé dans le cadre des systèmes embarqués. Pour nous, une donnée doit être accessible sur toute la durée de vie de l'application embarquée.

9.3.1.1 Déroulement d'une transaction

Le déroulement d'une transaction se décompose en trois phases, comme illustré par la Figure 9.1.

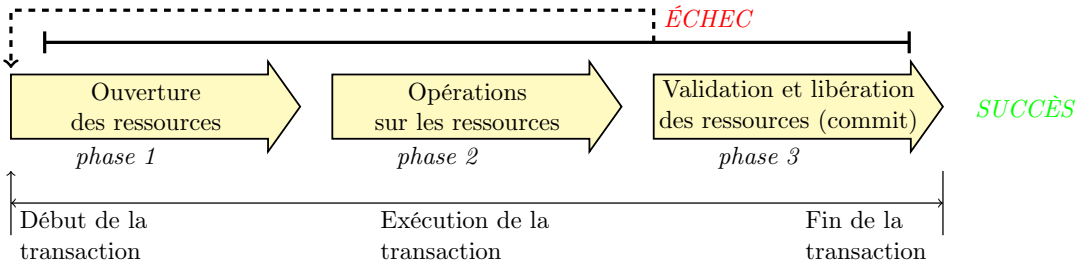


Figure 9.1 – Déroulement d'une transaction

La *phase 1* consiste à *ouvrir* l'ensemble des ressources manipulées au sein de la transaction. Cet ensemble est l'union de deux sous-ensembles : l'ensemble des *ressources manipulées en lecture seule* et l'ensemble des *ressources manipulées en lecture/écriture*. Les ressources ouvertes en lecture seule peuvent uniquement être lues tandis que celles ouvertes en lecture/écriture peuvent également être modifiées.

La *phase 2* consiste à réaliser les opérations sur les ressources. Concrètement, les ressources ouvertes en lecture/écriture sont modifiées selon la politique de mise à jour des ressources associée au protocole.

Enfin, la *phase 3* consiste à mettre à jour toutes les ressources. Cette étape est appelée *commit*. En cas de succès, la cohérence des données est assurée et l'ensemble des opérations effectuées sur les ressources ouvertes en lecture/écriture est rendu visible au reste du système.

Au cours de son exécution, une transaction peut *échouer*. Un échec survient si *au moins une des ressources manipulées par la transaction a été modifiée par une autre transaction depuis son ouverture*. En cas d'échec, une transaction est *abandonnée* et est *re-exécutée depuis le début*. Avant d'être rejouée, la transaction est d'abord remise dans un état consistant. L'échec peut survenir à n'importe quel moment du déroulement de la transaction et une transaction peut rejouer plusieurs fois. La mise en place d'une politique de détection et de résolution des conflits permet de borner le nombre d'échecs et d'assurer la progression. Enfin, en cas de *succès*, la transaction est complètement terminée. Elle est alors dite *validée* ou *commitée*.

Le protocole de concurrence d'un mécanisme à mémoire transactionnelle fait intervenir trois algorithmes : l'ouverture d'une ressource en lecture seule, l'ouverture d'une ressource en lecture/écriture et la mise à jour des ressources manipulées dans la transaction (commit).

9.3.1.2 Déclaration d'une transaction

Lors de la mise en place d'un protocole de synchronisation à base de verrous, l'attention à prêter pour l'utilisation des ressources partagées (e.g. pour éviter les interblocages) est souvent à la charge du développeur. L'un des avantages des protocoles non bloquants est de simplifier l'étape du codage. En effet, une fois clairement identifiées, les manipulations des *ressources critiques* sont encapsulées à l'intérieur de transactions (i.e. représenté par une section atomique *atomic{}*). Le protocole transactionnel est indépendant de l'application et peut se présenter sous la forme d'une bibliothèque ou d'une API du système d'exploitation. Par construction, il est capable de détecter les conflits, d'abandonner puis de redémarrer une transaction, jusqu'à ce que l'on soit certain de la cohérence des données.

Comparons les deux approches sur l'exemple de la Figure 9.2.

```
lock(a)
  lock(b)
    a <= f(a,b)
  unlock(b)
  lock(c)
    a <= g(a,c)
  unlock(c)
unlock(a)
```

(a) Lock-modify-unlocks

```
atomic {
  a <= f(a,b)
  a <= g(a,c)
}
```

(b) Bloc transactionnel

Figure 9.2 – Interfaces programmatiques pour le partage de ressources

Les deux affectations successives d'une variable a doivent être réalisées dans une section critique. Dans le cas où l'on utilise des verrous, la sémantique *lock-modify-unlock* présentée sur la Figure 9.2 (a) laisse entrevoir la difficulté lié à leur utilisation quand leur nombre augmente. L'utilisation d'une mémoire transactionnelle est quant à elle présentée sur la Figure 9.2 (b). Contrairement à la méthode utilisant du verrouillage, aucune réflexion n'est à prévoir pour prévenir les interblocages. Notons que la traduction sémantique du bloc atomique *atomic*

{} correspond à une syntaxe *do ... while(!commit)*, c'est-à-dire que l'on boucle tant que la transaction n'a pas réussi.

9.3.1.3 Imbrication et entrelacement des transactions

Définition 9.5. *Deux transactions d'un même processeur sont imbriquées si l'une s'exécute entièrement dans le corps de la seconde.*

Dans le contexte temps réel, nous excluons l'utilisation de transactions imbriquées. Dans notre contexte applicatif automobile, cette fonctionnalité n'a pas besoin d'être supportée.

Définition 9.6. *Deux transactions sont entrelacées si elles s'exécutent en parallèle et qu'elles manipulent des ressources communes.*

L'entrelacement des transactions est responsable de conflits qui peuvent se produire dans un système. Dans le cas des systèmes temps réel, nous définissons deux formes d'entrelacement.

Définition 9.7. *L'entrelacement local est un entrelacement entre des transactions localisées sur le même cœur.*

Définition 9.8. *L'entrelacement distant est un entrelacement entre des transactions s'exécutant sur des cœurs distincts.*

L'entrelacement distant est inhérent à l'exécution parallèle des tâches dans un système. L'entrelacement local survient uniquement dans un contexte préemptif. Par exemple, en considérant une politique d'ordonnancement P-RM, la tâche au sein de laquelle s'exécute la transaction ω_i peut être préemptée par une tâche du même cœur, plus prioritaire, exécutant une transaction ω_j. On dira alors que ω_j et ω_i sont entrelacées.

9.3.2 Choix de conception d'un mécanisme à mémoire transactionnelle

Puisque nous ciblons uniquement le partage des données, **dans la suite du document le terme partage de ressources désigne le partage des ressources mémoire** (i.e. des données).

La mise en place d'un mécanisme à mémoire transactionnelle peut se faire différemment en fonction des contraintes du système sur lequel on le met en place. En particulier, il convient de choisir :

- La granularité des données manipulées dans la transaction ;
- La façon dont les versions des données sont gérées ;
- La façon dont les conflits sont détectés ;
- La façon dont les conflits sont résolus.

9.3.2.1 Granularité d'une transaction

Les données accédées au sein d'une transaction peuvent avoir deux niveaux de granularité. On parle de *granularité objet* lorsque les accès sont effectués à l'échelle d'un objet, c'est-à-dire lorsque le protocole est indépendant de la taille des données manipulées. Cette approche permet en particulier de manipuler des structures de données complexes, à condition que les valeurs soit contiguës en mémoire. On parle en revanche de *granularité mot* quand les données

manipulées ont une taille d'un mot mémoire.

Dans la suite, les ressources de granularité objet seront désignées par le terme *objets* tandis que les ressources de granularité mot seront désignées sous le terme *données*.

9.3.2.2 Politiques de mise à jour, gestion des versions

Le choix de la politique de mise à jour des ressources définit la manière dont les anciennes et les nouvelles valeurs sont gérées. Les anciennes valeurs doivent rester disponibles dans le cas où la transaction échoue et les nouvelles valeurs doivent être rendues publiques lors du commit de la transaction.

Il existe plusieurs solutions pour la mise à jour des données : la gestion des versions dite *directe* (*eager*), ou la gestion dite *indirecte* (*lazy*).

Définition 9.9. *La politique de gestion de version est dite directe (ou eager), lorsque la transaction modifie directement la donnée en mémoire.*

Définition 9.10. *La politique de gestion de version est dite indirecte (ou lazy), lorsque la transaction modifie une copie locale de la donnée. La mémoire n'est pas mise à jour tant que la transaction n'a pas réussi son commit.*

Dans la première approche, il est nécessaire de sauvegarder la valeur initiale de la donnée avant de la modifier. Ceci permet de revenir en arrière (rollback) si la transaction échoue afin de restaurer un état consistant. Cette méthode a pour avantage de rendre la donnée publique rapidement lorsque la transaction réussit. En revanche en cas d'échec, il faut prendre le temps de reconstruire l'état initial. C'est une approche dite *optimiste* dans le sens où l'on estime que la transaction a de fortes chances de terminer sans conflit.

Avec la seconde méthode, la ressource n'est mise à jour que lorsque l'on est certain du succès de la transaction. En cas de succès, il faut alors rendre publique la nouvelle version. En revanche, en cas d'échec, il n'y a pas besoin de restaurer un état cohérent. C'est une approche dite *pessimiste* dans le sens où l'on estime que l'occurrence de conflit a une probabilité élevé.

La différence entre les deux approches se mesure en termes de performance. Il est compliqué de conclure qu'une approche est meilleure que l'autre, car les résultats sont très dépendants du degré de contention de l'application considérée.

9.3.2.3 Détection des conflits

La détection et la résolution des conflits permettent d'assurer conjointement l'isolation entre les transactions. La politique de détection permet de déterminer l'instant auquel une transaction sera à même de détecter les conflits et de rejouer. En règle générale, les conflits d'accès à une même donnée peuvent être de deux types : lecture/écriture ou écriture/écriture.

Définition 9.11. *Deux transactions sont en conflit si elles ouvrent toutes les deux une même donnée. On parle de contention lorsqu'au moins deux transactions ont ouvert la même donnée et qu'au moins l'une des transactions est en mesure de la mettre à jour.*

Il existe principalement deux façons de gérer la détection des conflit : la détection *au plus tôt* (*eager*) et la détection *au plus tard* (*lazy*).

Définition 9.12. *Le protocole de contrôle de concurrence est dit au plus tôt (eager) si la détection d'un conflit a lieu aussitôt après l'occurrence du conflit.*

Définition 9.13. *Le protocole de contrôle de concurrence est dit au plus tard (lazy) si la détection d'un conflit a lieu après son occurrence. En général, cette vérification est effectuée en phase de commit.*

La détection au plus tôt permet de réduire le temps consacré à l'exécution d'une transaction qui est vouée à l'échec. En revanche, il est possible d'abandonner une transaction qui aurait tout de même pu se terminer avec succès. Pour illustrer ce phénomène, considérons l'exemple suivant. Deux transactions ω_j et ω_k sont en conflit avec une troisième transaction ω_i sur deux données distinctes, respectivement r_1 et r_2. Considérons que la résolution des conflits conduit à abandonner ω_j quand le conflit sur r_1 est détecté avec ω_i. Considérons également que ω_i est abandonnée quand le conflit sur r_2 avec ω_k est détecté. Nous aurions alors pu autoriser ω_j à s'exécuter. La détection au plus tard évite ce problème mais en cas d'échec, le « temps perdu » est plus important.

La comparaison des deux approches en termes de performance ne permet pas d'affirmer que l'une est meilleure que l'autre. Cette évaluation est dépendante de l'application que l'on considère. Pour le lecteur intéressé, des travaux visant à évaluer ces deux approches sont rapportés dans (Belwal et Cheng, 2011).

Afin de détecter un conflit, des étapes de vérification de l'état des données ont lieu. La détection peut être assurée de deux manières distinctes. Soit la valeur de la donnée à la fin de la transaction est comparée avec celle du début de la transaction, soit on compare un numéro de version associé à cette donnée. La seconde méthode permet d'éviter le problème ABA (Dechev *et al.*, 2010) que l'on peut illustrer comme suit : considérons trois transactions (ω_i, ω_j, ω_k) et une donnée partagée de granularité objet ayant pour valeur initiale A. ω_i commence et lit la valeur de l'objet (elle lit A). ω_j démarre ensuite, écrit la valeur B dans l'objet et commit. À ce stade, si ω_i devait lire l'objet, elle lirait B. En troisième lieu, ω_k commence, écrit la valeur A puis commit. En considérant une détection au plus tard, quand ω_i entre en phase de commit, on va comparer la valeur initialement lue (A) avec la valeur actuelle (A). Aucun conflit ne sera détecté ici alors que pourtant, l'objet a subit des modifications.

9.3.2.4 Résolution des conflits

La résolution des conflits répond à des heuristiques particulières en accord avec la garantie de progression. Le *gestionnaire de contention* a pour but de résoudre les conflits une fois qu'ils ont été détectés. Ce mécanisme, non wait-free, décide lors d'un conflit laquelle des transactions doit être privilégiée et la manière dont l'autre (ou les autres) va (ou vont) être abandonnée(s) et redémarrée(s). Nous avons le choix entre plusieurs types de gestionnaire de contention. Parmi les différentes politiques existantes, on peut citer Karma, Polite, Eruption, Polka, Timestamp, etc. Une présentation et des évaluations comparatives de ces différentes heuristiques est disponible dans (Scherer III et Scott, 2004, 2005; Guerraoui *et al.*, 2005b,a).

L'*aide* (helping) est un mécanisme alternatif existant permettant d'aider des transactions qui sont en conflit. Dans ce cas, on ne fait pas référence au gestionnaire de contention. En pratique, une transaction ω_i aide une transaction ω_j en exécutant les opérations de ω_j au nom de ω_j. Après l'aide, ω_i peut poursuivre sans craindre d'être mise en échec par ω_j. Ce mécanisme d'aide se révèle essentiel pour le respect des garanties de progression voulues. À

titre d'exemple, le schéma d'aide décrit dans (Anderson et Moir, 1995) permet d'obtenir une progression wait-free. L'aide doit cependant être maîtrisée pour éviter les phénomènes d'aides récursives, de blocages ou encore de livelocks (Harris et Fraser, 2003).

9.4 Implémentation des mécanismes à mémoire transactionnelle

9.4.1 Taxonomie des instructions matérielles pour l'atomicité dans les cibles embarquées

Nous avons besoin de réaliser des opérations de manière atomique. Pour réaliser cela, il est nécessaire de disposer d'instructions matérielles dédiées. Plusieurs instructions ont été conçues dans ce but.

Read-Modify-Write désigne une classe d'instructions basées la lecture d'une zone mémoire puis sur la mise à jour de celle-ci de manière atomique. Les instructions de cette classe préviennent les contentions entre plusieurs processus parallèles dans une application. En pratique, elles peuvent être aussi bien utilisées pour implémenter des mécanismes d'exclusion mutuelle, des sémaphores ou des algorithmes de synchronisation non bloquants. Les différentes instructions sont classées en fonction de leur capacité à gérer les consensus entre plusieurs processus qui accèdent à un même emplacement mémoire (Herlihy, 1991).

L'instruction *Test & Set* permet d'implémenter l'exclusion mutuelle dans les systèmes multicœur. Elle est utilisée pour écrire atomiquement dans un emplacement mémoire et retourne l'ancienne valeur de la donnée présente dans cet emplacement. Quand un processus accède une donnée en mémoire via un *Test & Set*, une information interne permet de mémoriser que l'adresse de la donnée est *occupée*. Si d'autres processus tentent d'accéder à la même donnée, ils devront attendre et retenter l'accès plus tard, ce qui génèrent de l'overhead.

Dans Herlihy (1991), Herlihy prouve que le *Test & Set* ne permet pas de régler les consensus entre plus de deux processus. L'instruction *Compare & Swap* quant à elle permet de gérer des consensus entre un nombre quelconque de processus. Cette instruction prend 3 paramètres : l'adresse du mot mémoire à modifier (*&word*), l'ancienne valeur (*oldValue*) et la valeur à écrire (*newValue*). Elle permet d'écrire la valeur *newValue* à l'emplacement mémoire *&word* (i.e. instruction *Swap*), à condition que la valeur courante soit égale à *oldValue* (instruction *Compare*). Toutes ces opérations sont effectuées de manière atomique. L'exécution d'un CAS peut donner deux résultats : soit la substitution a eu lieu, c'est-à-dire que la nouvelle valeur écrite est bien cohérente avec l'ancienne, soit la substitution n'a pas lieu, car l'emplacement mémoire a été mis à jour par un autre processus parallèle. Dans le dernier cas, l'opération doit être répétée. *Compare & Swap* ne permet pas d'éviter le problème dit *ABA* présenté dans la section 9.3.2.3 (page 120). Pour éviter ces faux positifs, une instruction double-length CAS peut être utilisée. L'espace supplémentaire est alors utilisé pour maintenir un compteur. En plus de la condition sur la valeur de la donnée, les compteurs doivent également être égaux. Le compteur est incrémenté atomiquement lors du swap des valeurs. Le problème ABA n'est pas réglé si le compteur est incrémenté suffisamment de fois pour effectuer un tour complet. Sur une instruction de CAS de 32 bits, cela représente un multiple de 2^{32} mises à jour de la valeur.

Pour répondre aux besoins des différents algorithmes lock-free et wait-free, des extensions

de l'instruction CAS ont été proposées. L'instruction *Double Compare & Swap* (DCAS) permet de mettre à jour deux emplacements mémoire en même temps. L'instruction *Double-Wide Compare & Swap* (DWCAS) permet de mettre à jour deux emplacements mémoire contigües. *Single Compare, Double Swap* modifie deux emplacements mémoire, mais la comparaison est réalisée sur une seule. Enfin, le *Multi-Word Compare & Swap* (MWCAS) généralise l'instruction CAS à un nombre arbitraire d'enplacements mémoire. Cette dernière repose en revanche sur une implémentation logicielle.

Enfin, le couple d'instructions *Load Link/Store Conditional* (LL/SC) permet également d'assurer l'atomicité lorsqu'on accède à une zone de mémoire partagée. *Load Link* retourne la valeur courante de l'emplacement mémoire passé en paramètre. *Store Conditionnal* le met à jour à condition que la valeur courante n'ait pas été mise à jour depuis le *Load Link* précédent. Le comportement est assimilable à un CAS mais puisque l'opération est réalisée par deux instructions distinctes, LL/SC prévient totalement le problème ABA. En effet, même si la valeur courante au moment du SC a été restaurée à la valeur lue au moment du LL, l'opération va échouer. Il est aussi plus performant qu'un CAS puisqu'il est possible d'exécuter une série d'instructions entre LL et SC.

Comme le CAS, LL/SC permet de gérer les consensus entre un nombre quelconque de processus. Il est également possible d'implémenter CAS ou Test & Set à partir de LL/SC, comme démontré par Anderson et Moir (1995).

9.4.2 Types d'implémentations

Il existe plusieurs techniques d'implémentation des mécanismes à mémoire transactionnelle. Un des objectifs majeurs consiste à gérer des transactions en concurrence, tout en garantissant les meilleures performances possibles (i.e. maximisation de l'utilisation des ressources matérielles offertes par les architectures multicœur).

Les premières implémentations ayant vu le jour étaient *matérielles (HTM – Hardware Transactional Memory)*. Ces techniques se basent sur les protocoles de cohérence des caches. Elles requièrent alors des composants matériels spécifiques pour leur bon fonctionnement. Des exemples de supports matériels nécessaires aux mémoires transactionnelles sont donnés dans Herlihy et Moss (1993). La première implémentation matérielle, de type lock-free, a été réalisée en 1993 par Herlihy *et al.* (1993). Elle repose sur l'utilisation de nouvelles instruction-machines et d'un cache transactionnel permettant de gérer la bufferisation des données ouvertes au sein d'une transaction. Dans Schoeberl *et al.* (2010), les auteurs présentent la *RTTM* (Real-Time Transactional Memory). La solution, entièrement matérielle, permet la synchronisation de données en assurant le déterminisme temporel. Une analyse permet de calculer le pire temps d'exécution d'une transaction, ce qui rend cette solution particulièrement adaptée aux systèmes temps réel dur. Une autre implémentation matérielle temps réel (Meawad *et al.*, 2011) s'appuie sur des micros-transactions pour les opérations sur des files FIFO dans des systèmes embarqués s'utilisant en Java. L'implémentation matérielle permet d'augmenter les performances puisque les accès en mémoire, la gestion des versions, la détection des conflits et leur résolution sont gérés exclusivement par des mécanismes matériels et ne sont pas remontés aux couches logicielles. Par contre, la taille du cache transactionnel peut parfois s'avérer insuffisante si l'application comporte un grand nombre de transactions longues (i.e. qui manipulent beaucoup de données). Pour être efficace, il faut donc que les sections atomiques soient les plus courtes possibles et manipulent peu de données.

La précédente méthode nécessite de disposer d'un microcontrôleur conçu spécifiquement

pour la gestion de systèmes transactionnels. Il existe peu de processeurs offrant de telles possibilités. C'est pourquoi il est intéressant de considérer des techniques d'implémentation entièrement logicielles. On parle alors de *STM (Software Transactional Memory)*. Ces solutions requièrent uniquement de disposer d'instructions atomiques telles que *Compare & Swap (CAS)* ou le *Load Link/Store Conditional (LL/SC)*. Puisque l'ensemble du protocole est géré par le logiciel, ces solutions sont souvent moins performantes que les HTM. Cependant, elles s'adaptent bien aux systèmes possédant un très grand nombre de transactions. La première implémentation logicielle est de type lock-free et a été réalisée par Shavit et Touitou (1995). Cette implémentation suppose une configuration statique de l'application puisqu'aucun objet lié au mécanisme n'est créé en ligne. Plus récemment, d'autres implémentations ciblent des architectures dynamiques. Parmi celles-ci, on peut citer la DSTM (Dynamic STM) (Harris et Fraser, 2003; Herlihy *et al.*, 2003), l'ASTM (Adaptative STM) (Marathe *et al.*, 2005), la TL2 (Transactional Locking II) (Dice *et al.*, 2006), ou encore la OSTM (Object-Based STM) (Fraser, 2004). Les seules implémentation logicielle qui prend en compte les aspects temps réel dur sur multicœur est la solution proposée par Holman et Anderson (2006). Cependant, elle se base sur une politique d'ordonnancement de type Pfair (algorithmes basés sur l'allocation de quantum de temps, mais dont l'overhead d'exécution peut être important du fait du grand nombre de migrations de tâches inhérent à la séquence d'ordonnancement qu'il produit), ce qui la rend inadaptée dans notre contexte industriel (AUTOSAR impose une politique d'ordonnancement partitionnée, à priorité fixe). De son côté, la RTSTM (Sarni *et al.*, 2009a,b) prend en compte des contraintes temps réel souples. Enfin, l'implémentation de la *RobuSTM* (Wamhoff *et al.*, 2010) vise quant à elle à assurer la robustesse du système en termes de tolérance aux fautes, en le rendant résistant au crash d'un cœur ou à la non-progression d'une transaction. Par exemple, une transaction qui ne parvient pas à commiter depuis trop longtemps devient prioritaire en utilisant du verrouillage. Une extension de ces travaux est proposée dans Wamhoff et Fetzer (2011).

L'implémentation d'un mécanisme STM dépend du protocole de contrôle de concurrence à mettre en œuvre. Ce protocole repose à son tour sur des structures de données. Des structures de données complexes et riches en informations vont réduire les temps de calcul nécessaires au protocole, mais vont également imposer une empreinte mémoire plus importante. A contrario, des structures de données minimalistes imposent de calculer en ligne les informations nécessaires au bon fonctionnement du protocole. Les principaux axes de travail résident donc dans la recherche du compromis entre le temps de calcul nécessaire au maintien de la cohérence des données et l'empreinte mémoire des structures de données sous-jacentes.

Enfin, il existe également des solutions hybrides nommées *Hybrid TM – HyTM*. Ces implémentations permettent de tirer profit des avantages des solutions entièrement matérielles et entièrement logicielles. En pratique, les transactions peuvent être initiées par un jeu d'instructions offert par le matériel, mais il reste à la charge du logiciel de gérer la résolution de conflits. Une autre combinaison matérielle/logicielle peut consister à utiliser le cache transactionnel matériel tant que les transactions en cours peuvent y être stockées puis à commuter en logiciel dès lors que la capacité du cache ne suffit plus à consigner les exécutions. Une taxonomie est présentée dans (Harris *et al.*, 2010).

Dans la suite, nous présentons plus précisément des exemples d'implémentation. Bien que ces mécanismes ne soient pas wait-free, le protocole que nous proposons s'appuie sur les résultats obtenus dans ces solutions.

9.4.3 Exemples d'implémentations logicielles de type Lock-Free

Nous présentons ici deux mécanismes lock-free représentatifs de l'état de l'art sur les STM ayant servi de base pour nos travaux. La OSTM (Fraser, 2004) ne permet pas de prendre en compte les aspects temps réel. La RT-STM Sarni *et al.* (2009a,b) permet de prendre en compte l'échéance des tâches comme critère pour la prise de décision en cas de conflits. La RT-STM permet d'assurer des contraintes temps réel souples. La performance est mesurée en termes de qualité de service (QoS).

La OSTM est une implémentation de mémoire transactionnelle logicielle non temps réel proposée par Fraser (Fraser, 2004). Il s'agit d'une implémentation de type *lock-free*, de granularité *objet*, ayant une politique de résolution des conflits *lazy* et une mise à jour des données *lazy*.

Les structures de données sont représentées sur la Figure 9.3. À chaque objet est associé un emplacement mémoire qui désigne la valeur courante (i.e. la valeur visible par toutes les transactions du système) d'un objet : c'est l'*objet concurrent*. Chaque objet possède également sa propre en-tête. L'*en-tête d'un objet* est un pointeur qui désigne à l'état initial l'objet concurrent. À chaque transaction est associé un *descripteur de transaction*. Il s'agit d'une structure comprenant le statut de la transaction (*UNDECIDED*, quand la transaction est en cours ; *COMMITED*, après le succès ; *ABORTED*, en cas d'échec et *READ_ CHECKING*, au commit), la liste des objets manipulés en lecture seule et la liste des objets manipulés en lecture/écriture. Ces deux listes sont des listes chaînées de *méta-objets*. Un seul pointeur vers le premier élément de la liste est présent dans la structure du descripteur de transaction (notons que dans l'illustration, il n'y a aucun objet accédé en lecture seule). Quand un objet est ouvert dans une transaction, on lui associe un *méta-objet* (il y aura autant de méta-objets que de transactions ouvrant l'objet). Il contient un pointeur vers la référence de l'objet (en-tête de l'objet), un autre vers *l'ancienne version de l'objet*, un autre vers *la nouvelle version de l'objet*, puis un dernier vers l'objet suivant dans la liste.

La OSTM est dynamique, c'est-à-dire que ces structures sont créées et détruites en fonction des besoins au cours de l'exécution. Le démarrage d'une transaction conduit à la création d'un descripteur de transaction tandis que l'ouverture d'un objet conduit à la création d'un méta-objet et d'une copie locale. Pour des raisons de performances au niveau de la mémoire, un *garbage collector* doit être utilisé.

Plusieurs algorithmes sont nécessaires au fonctionnement du protocole. À *l'ouverture d'un objet* en lecture seule (resp. en lecture/écriture), un *méta-objet* associé à cet objet est immédiatement créé et ajouté à la fin de la liste chaînée des objets ouverts en lecture seule (resp. en lecture/écriture) par la transaction. Si l'objet est ouvert en lecture/écriture, toutes les modifications apportées à celui-ci sont réalisées dans sa *copie locale*. La phase de *commit* est la plus importante de la transaction. Premièrement, il faut *acquérir* tous les objets manipulés dans la transaction. Ceci se traduit par le fait que *l'en-tête de chaque objet* vient pointer vers le *descripteur de transaction* qui l'acquiert. Toute autre transaction qui souhaite effectuer un commit sera désormais en mesure de détecter les conflits. Si tous les objets ayant été ouverts sont consistants, le commit est un succès et toutes les opérations effectuées sur les objets ouverts en lecture/écriture sont rendues publiques (i.e. tous les objets ouverts en lecture/écriture pointent maintenant vers la copie locale). Les opérations permettant de modifier les pointeurs sur les structures de données sont réalisé à l'aide de l'instruction atomique CAS.

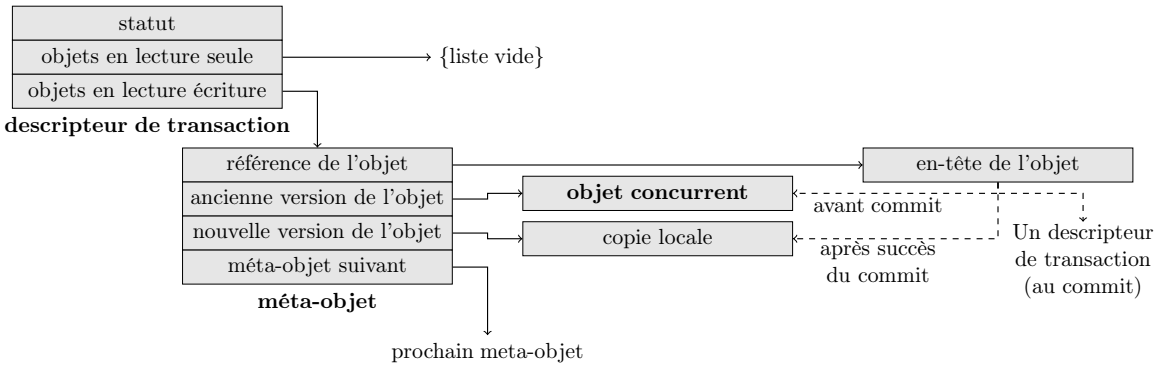


Figure 9.3 – Structures de données dans la OSTM

La garantie de progression est assurée grâce au mécanisme d'*aide*. Lors de la détection d'un conflit, une transaction peut aider une autre en exécutant ses actions en son nom.

La RT-STM est une implémentation de mémoire transactionnelle logicielle conçue pour les systèmes temps réel souples, proposée dans les travaux de Sarni *et al.* (2009a,b). Il s'agit d'une implémentation de type *lock-free*, de granularité *mot*, ayant une mise à jour des données *lazy*.

En plus du mécanisme de base garantissant la progression lock-free déjà utilisée dans la OSTM, la RT-STM impose la prise en compte des contraintes temps réel. Des critères supplémentaires sont alors nécessaires. Pour choisir quelles sont les transactions à aider ou à abandonner, il faut tenir compte non seulement du type de conflit (lecture/écriture ou écriture/écriture), mais aussi des contraintes temporelles. On privilégie ainsi la progression des transactions dont l'échéance est la plus proche du temps courant. Des priorités sont assignées en-ligne aux transactions pour refléter cette urgence.

Les structures de données et le protocole sont assez similaires à ceux utilisés dans la OSTM. Le descripteur de transaction a été enrichi pour intégrer l'échéance de la tâche. L'en-tête de l'objet a été enrichi pour intégrer plusieurs informations supplémentaires : une table qui enregistre les accès en lecture seule sur l'objet permet de gérer les conflits lecture/écriture et un champ *mark* désignant la transaction la plus prioritaire parmi celles qui ont ouvert l'objet en lecture/écriture permet de trier les aides par ordre de priorité des transactions.

Deux protocoles de contrôle de concurrence sont proposés par les auteurs. Le premier est le protocole *1W* (un écrivain, N lecteurs), adoptant une politique pour la détection et la résolution des conflits de type *eager* et le second est le protocole *MW* (*M écrivains, N lecteurs*), adoptant une politique pour la détection et la résolution des conflits de type *lazy*.

9.5 Périmètre de l'étude

9.5.1 Objectifs et contraintes

Nous souhaitons étudier une alternative aux protocoles de synchronisation bloquants dans le contexte embarqué automobile. La conception d'un tel protocole nous impose de respecter un ensemble de contraintes afin de répondre aux objectifs du domaine :

– Nous devons garantir la cohérence de groupes de données partagés entre plusieurs cœurs.

Ce besoin émerge à partir de la révision 4.1 d'AUTOSAR au travers de la notion de *coherency group* ;
- Nous ciblons des systèmes temps réel dur. La progression de toutes les tâches en concurrence sur des transactions doit être assurée sans famine (wait-free) et le coût du protocole doit être parfaitement caractérisé (en termes de surcoût temporel) pour l'intégrer dans l'analyse d'ordonnançabilité ;
- Nous cherchons à minimiser l'empreinte mémoire ;
- Le protocole doit être robuste au crash d'un cœur ou d'une tâche ;
- AUTOSAR repose sur une chaîne de configuration statique. Le protocole STM-HRT devra donc également être configuré statiquement. Un bon compromis entre les structures de données (générées statiquement) et le protocole (exécuté en-ligne) doit être trouvé.

9.5.2 Approche suivie

Notre état de l'art fait apparaitre deux grandes familles de solutions : les mécanismes *wait-free* et les mécanismes *à mémoire transactionnelle*. Sur ces bases, nous proposons le protocole STM-HRT (*Software Transactional Memory for Hard Real-Time systems*).

STM-HRT est un mécanisme wait-free dont la structure et l'utilisation sont inspirées des mémoires transactionnelles. Le protocole est conçu pour gérer la communication interne inter et intra-cœur sans verrous et pour s'interfacer à terme avec les APIs de l'IOC définies par la spécification AUTOSAR multicœur. Il est une alternative aux protocoles préconisés dans ces spécifications.

Dans une démarche de sûreté de fonctionnement, STM-HRT vise les caractéristiques suivantes :
- En l'absence de verrous, l'application est plus résistantes au crash d'un cœur, ou d'une tâche ;
- Toutes les transactions dans STM-HRT sont définies hors-ligne et font partie intégrante de l'étape de configuration statique du système d'exploitation ;
- L'encapsulation des données partagées dans les transactions permet d'assurer leur consistance.

Les difficultés sont désormais reportées sur la conception du protocole ainsi que sur l'outil de génération du code. Dans la suite, nous nous focalisons sur la conception du protocole STM-HRT.

Chapitre 10

STM-HRT : un protocole non bloquant pour le partage de données

Sommaire

10.1 Modèles et hypothèses

La mise en place d'un protocole basé sur les principes des mémoires transactionnelles logicielles pour des systèmes embarqués industriels impose de relever de nombreux défis (Fetzer et Felber, 2011). Nous présentons ici les choix de conception qui permettent de garantir l'ensemble de objectifs définies dans la Section 9.5.1 (page 124).

10.1.1 Modèle du système

Le modèle du système est composé des 4 ensembles représentés sur la Figure 10.1.

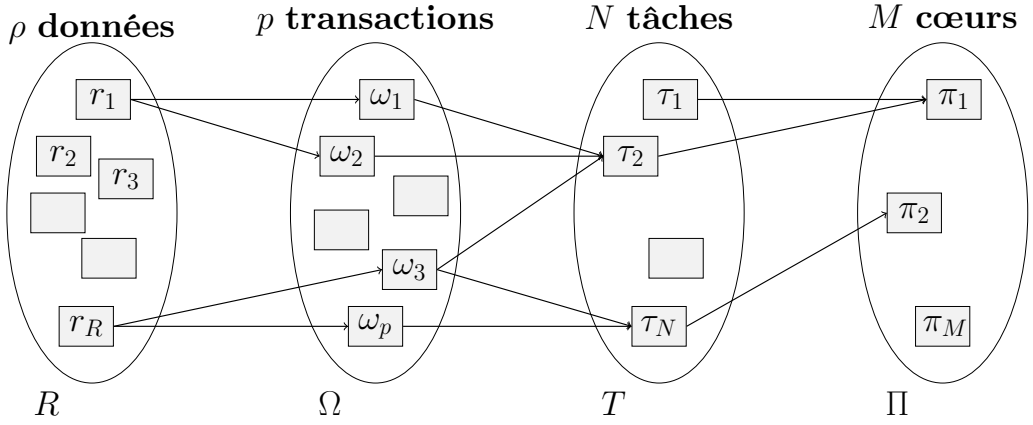


Figure 10.1 – Configuration d'une application utilisant STM-HRT

Nous considérons une plateforme multicœur $\Pi = \{\pi_1, ..., \pi_M\}$ composée de M cœurs identiques. Le système comprend un ensemble $\tau = \{\tau_1, ..., \tau_N\}$ de N tâches, à contraintes temps réel dures allouées sur les M cœurs. Les tâches sont allouées aux différents cœurs conformément à l'heuristique de partitionnement choisie. Les tâches partagent un ensemble $R = \{r_1, ..., r_\rho\}$ de ρ données.

Le mécanisme décrit ici a été nommé STM-HRT. Il permet la synchronisation des données et le maintien de la cohérence de leurs valeurs. Tous les accès aux données partagées sont effectués dans le contexte de transactions. On considère un ensemble $\Omega = \{\omega_1, ..., \omega_p\}$ de p transactions dans le système. Une transaction ω_k est caractérisée par le tuple $(e_{\omega_k}, R_{\omega_k})$ où e_{ω_k} est la durée maximale d'exécution de ω_k en totale isolation (i.e. pas de préemption, pas de conflits) et $R_{\omega_k} \in R$ est l'ensemble des données manipulées par ω_k. La granularité choisie pour STM-HRT est une *granularité objet*. Par la suite, nous considérons qu'à toute données r_j, correspond la manipulation d'un objet noté o_j.

Les transactions sont allouées aux tâches. Dans le système considéré, chaque tâche est caractérisée par le tuple $(r_i, C_i, T_i, D_i, \Omega_{\tau_i})$, où r_i est la date de réveil de τ_i, C_i est son pire temps d'exécution (y compris les transactions en totale isolation), T_i est le temps d'inter-arrivée minimale entre deux activations successives de τ_i, D_i est son échéance relative et $\Omega_{\tau_i} \subseteq \Omega$ est l'ensemble des transactions qu'elle manipule. Ces paramètres sont illustrés sur la Figure 10.2. Chaque activation de la tâche τ_i donne lieu à l'exécution d'une instance de la tâche.

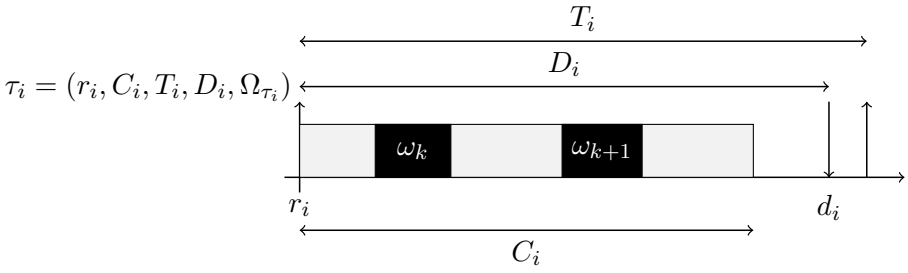


Figure 10.2 – Modèle de transaction considéré dans STM-HRT

10.1.2 Hypothèses considérées

10.1.2.1 Hypothèses liées au domaine d'application

Nos hypothèses nous conduisent à définir au préalable une nouvelle terminologie.

Définition 10.1. *Une transaction ω_k est dite homogène si elle manipule tous ses objets dans un seul mode d'accès ; autrement, la transaction est dite hétérogène.*

Définition 10.2. *Une transaction ω_k est dite homogène en lecture si la totalité des objets manipulés sont ouverts en lecture seule.*

Définition 10.3. *Une transaction ω_k est dite homogène en écriture si la totalité des objets manipulés sont ouverts en lecture/écriture.*

Nous supposons que *toutes les transactions du système sont des transactions homogènes.* Cette hypothèse permet de simplifier la conception du protocole wait-free. Elle est également en phase avec le contexte automobile. En effet, avec une stratégie de communication *implicite* telle que définie dans AUTOSAR (tous les objets sont lus en début d'exécution d'un runnable et tous les objets sont écrits en fin d'exécution). Par conséquent, un ensemble de transactions en lecture sera utilisé en début de tâche et un ensemble de transactions en écriture sera utilisé en fin de tâche.

STM-HRT est un service du système d'exploitation. L'exécution d'une transaction est réalisée en mode noyau et est par conséquent non préemptible. L'accès aux données est indépendant de la politique d'ordonnancement choisie (à condition que l'on soit dans un système partitionné) et le succès des transactions ne dépend que du protocole. De plus, seul un entrelacement distant est possible donc au plus M transactions peuvent être actives en même temps. Dans la suite, le terme *transaction active sur π_i* désigne l'unique transaction actuellement en cours d'exécution sur le cœur π_i.

La dernière hypothèse concerne le modèle de communication utilisé dans notre système. Nous supposons que *tous les écrivains d'un même objet sont localisés sur le même cœur.* Cette hypothèse est réaliste puisque dans les systèmes ciblés, le modèle de communication impose souvent un seul écrivain par objet. De plus, puisqu'une transaction est non préemptible, les conflits écriture/écriture seront impossibles. Seuls les conflits lecture/écriture devront être traités par le protocole.

10.1.2.2 Choix de conception du protocole

Puisqu'une transaction en écriture ne peut pas échouer, le choix de la politique de gestion des versions n'a pas d'importance. Nous avons donc retenu une politique de *gestion des versions différées* (i.e. de type *lazy*).

Une politique de *détection et résolution des conflits au plus tard* (i.e. de type *lazy*) a été choisie. Les transactions de lecture devront attendre le commit pour éventuellement échouer. Même en cas de contention (i.e. une transaction en écriture a déjà ouvert l'objet au moment où la transaction de lecture y accède), il reste toujours une chance pour que la transaction de lecture réussisse. Une politique au plus tôt pourrait occasionner un taux d'échecs des transactions de lecture plus important. Les performances du système seraient alors pénalisées.

Enfin, pour assurer la progression wait-free visée dans nos travaux, nous utiliserons un mécanisme d'*aide* au lieu d'un gestionnaire de contention. Chaque transaction devra s'assurer de la progression des transactions avec lesquelles elle est en conflit. En plus de garantir le progrès, l'aide permet également d'atteindre les contraintes de robustesse visées (i.e. résister au crash d'un coeur ou d'une tâche).

En résumé, STM-HRT est un protocole *robuste*, *non bloquant* offrant une garantie de progression de type *wait-free*. Ces caractéristiques présentent plusieurs avantages :

- Toutes les transactions se termineront systématiquement en un temps borné. En pratique, le nombre d'échecs d'une transaction est borné ;
- Le pire temps d'exécution d'une transaction est indépendant de la politique d'ordonnancement choisie ;
- Le pire temps d'exécution d'une transaction peut être inclus dans le pire temps d'exécution de la tâche comprenant la transaction. Ceci permet de faciliter l'analyse d'ordonnançabilité de l'application.

10.1.2.3 Notations

L'ensemble des notations utilisées dans la suite du document est référencé dans la Table 10.1.

10.1.2.4 Positionnement de STM-HRT vis-à-vis des approches wait-free et à mémoire transactionnelle

STM-HRT est un protocole conçu sur les bases offertes par l'étude des mécanismes à mémoire transactionnelle et des mécanismes wait-free. Le positionnement du protocole vis-à-vis de ces deux approches est *transverse*. Nous pouvons comparer ces deux approches suivant plusieurs critères : les schémas de communication, le type de transaction, la taille des transactions, les conditions de terminaison et les contraintes temps réel.

Les mécanismes à mémoire transactionnelle permettent d'assurer l'intégrité d'un ensemble de données manipulées en lecture et/ou écriture dans des transactions hétérogènes. Elles permettent également de gérer des accès multi-écrivain, multi-lecteur et d'assurer éventuellement le succès de toutes les transactions. Les implémentations logiciels sont souvent de type lock-free car la souplesse de ces mécanismes rend difficile la prise en compte des contraintes temps réel dur, où il faut borner le nombre de rejoues des transactions. Dans Fetzer et Felber (2011),

Symbole	Description
τ_i	Tâche i de l'ensemble de tâches τ
ω_k	Transaction k de l'ensemble de transactions Ω
π_i	Cœur i
$\pi(\omega_k)$	Cœur sur lequel la tâche exécutant ω_k est allouée
Ω_{π_i}	Ensemble des transactions allouées au cœur π_i
Ω^R	Ensemble des transactions homogènes de lecture du système
$\Omega^R_{\pi_i}$	Ensemble des transactions homogènes de lecture du cœur π_i
$\Omega^R_{\tau_i}$	Ensemble des transactions homogènes de lecture de la tâche τ_i
Ω^W	Ensemble des transactions homogènes d'écriture du système
$\Omega^W_{\pi_i}$	Ensemble des transactions homogènes d'écriture du cœur π_i
$\Omega^W_{\tau_i}$	Ensemble des transactions homogènes d'écriture de la tâche τ_i
$\lvert\Omega^W_{\tau_i}\rvert$	Nombre de transactions homogènes d'écriture de la tâche τ_i
O^{ω_k}	Ensemble des objets accédés par la transaction ω_k
o_j	Objet j de l'ensemble des objets O^{ω_k}
$\pi(o_j)$	Cœur autorisé à mettre à jour l'objet o_j
$\Pi(o_j)$	Ensemble des cœurs uniquement autorisés à lire l'objet o_j

Table 10.1 – Notation utilisées

les auteurs montrent qu'il y de nombreux défis pour mettre en œuvre de tels mécanismes dans des systèmes embarqués.

Les protocoles wait-free sont, par construction, mieux adaptés aux systèmes temps réel dur. Sous certaines conditions, il peuvent également assurer l'intégrité d'un ensemble de données manipulées en lecture et/ou écriture dans un contexte multi-écrivain, multi-lecteur. À la différence des mécanismes à mémoire transactionnelle, l'état de l'art sur les mécanismes wait-free montrent que l'objectif n'est pas d'assurer le succès des opérations mais uniquement le fait de pouvoir fournir un verdict (potentiellement un échec) dans un temps borné (e.g. opération MWCAS (Anderson et Moir, 1995)).

STM-HRT doit pouvoir lier en partie les avantages des deux approches. En effet, il est un mécanisme wait-free dans le sens ou toutes les opérations se terminent dans un temps borné. Il peut également être vu comme un mécanisme à mémoire transactionnelle dans le sens où tous les accès aux données sont encapsulés dans des transactions et que toutes les transactions se termineront par un succès. En revanche, pour faciliter la mise en place du mécanisme, nous avons considéré uniquement des transactions homogènes et un schéma de communication 1 : m, ce qui fait qu'STM-HRT appartient à une sous-classe des mécanismes à mémoire transactionnelle. L'étude de son applicabilité dans le contexte automobile révèle que ces hypothèses sont en adéquation avec les besoins exprimés au niveau applicatif dans le domaine.

10.2 Structures de données de STM-HRT

10.2.1 Configuration hors-ligne

De manière à être en adéquation avec AUTOSAR, STM-HRT est configuré hors-ligne. Cette configuration est statique : aucun objet n'est crée en-ligne.

Nous construirons **un en-tête d'objet pour chacune des données utilisées** dans le système. De plus, puisqu'une seule transaction peut être active par cœur à tout instant, seuls M **descripteurs de transaction** seront nécessaires. Plus formellement, le descripteur de transaction du cœur π_i est partagé par toutes les transactions de l'ensemble $\Omega_{\pi_i} = \{\Omega^R_{\pi_i} \cup \Omega^W_{\pi_i}\}$. Il doit donc être en mesure de supporter toutes les transactions pouvant s'exécuter sur π_i.

10.2.2 Descripteur des transactions

La Figure 10.3 décrit le descripteur de transaction. Il est composé de 5 champs. *proc_ id* désigne l'identifiant du cœur auquel est associé le descripteur. *status* contient à la fois le statut de la transaction et un compteur désignant le numéro d'instance du descripteur. *read-set* est un pointeur vers la table des objets ouverts en lecture (read-set table), *write-set* est un pointeur vers la table des objets ouverts en écriture (write-set table). La table des objets ouverts en lecture est utilisée lorsque la transaction active sur π_{proc_id} est une transaction de lecture, tandis que la table des objets ouverts en écriture est utilisée lorsque la transaction active sur π_{proc_id} est une transaction en écriture. Pour une même transaction, une seule de ces tables est donc utilisée à la fois. Enfin, le vecteur d'accès *access_ vector* contient l'ensemble des objets ouverts par la transaction active sur π_{proc_id}.

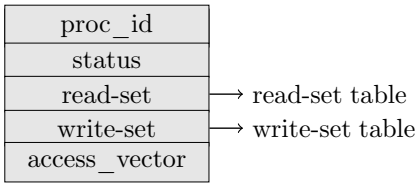


Figure 10.3 – Structure du descripteur de transaction de STM-HRT

Nous détaillons à présent les informations contenus dans les quatre derniers champs.

status est un champ de 8 bits. Les 6 bits de poids fort servent de compteur pour connaître le numéro de l'instance associée au descripteur de transaction. Puisque toutes les transactions allouées à un cœur partagent le même descripteur, ce compteur permet de distinguer les utilisations successives du descripteur par des transactions du même cœur. Ce compteur est nécessaire en cas d'aide pour se prémunir des corruptions des structures de données (c.f. Section 10.3.5, page 142). Les 2 bits de poids faible encodent le statut de la transaction. Les différents statuts sont ceux décrits dans la Figure 10.4.

Puisque nos hypothèses assurent l'absence de conflit écriture/écriture, une transaction en écriture réussira obligatoirement son commit (diagramme d'état de la Figure 10.4(a)). Au démarrage, une transaction devient active et son statut passe à *INPROG* (*in progress*). Puisqu'elle ne peut pas échouer, la transaction passe directement en statut *INACTIVE* au moment de son succès. En revanche, nos hypothèses ne préviennent pas la présence de conflits lecture/écriture. On rappelle qu'une transaction de lecture échoue dès lors qu'au moins un de ses objets

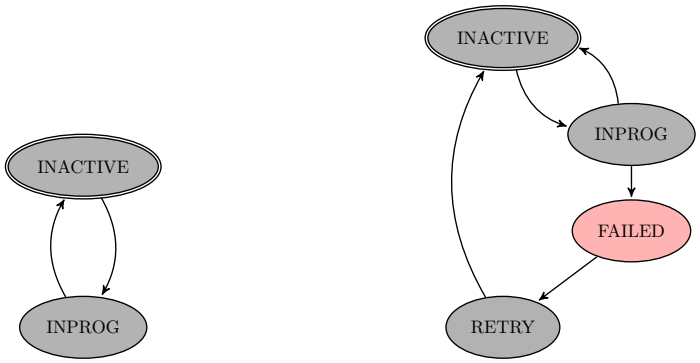


Figure 10.4 – *statuts* pour une transaction (a) d'écriture et (b) de lecture

a été modifié depuis son ouverture. Le protocole mis en œuvre garantie cependant qu'une transaction de lecture ne pourra échouer qu'une seule fois (diagramme d'états de la Figure 10.4(b)). Quand la transaction démarre, elle devient *INPROG*. Après son premier essai, la transaction peut échouer (*FAILED*). Durant sa seconde exécution, la transaction a le statut *RETRY*. La transaction va désormais systématiquement réussir. Dès que le succès est observé, le statut retourne à son état initial (*INACTIVE*).

read-set (resp. *write-set*) est un pointeur vers la table des objets manipulés en lecture (resp. écriture) sur le cœur. Une table de ce type est requise pour chacun des cœurs. La table des objets ouverts en lecture (read-set table) est un tableau de pointeurs contenant autant d'entrées que d'objets manipulés dans le système. Chacune de ces entrées correspond à un objet donné (table indexée par les identifiants des objets). Quand une transaction de lecture ouvre un objet, la ligne correspondante du tableau pointe vers la copie de l'objet lu. De manière similaire, la table d'objets ouverts en écriture contient des pointeurs utilisés pour localiser la copie de la version mise à jour d'un objet avant sa publication.

Enfin, *access_ vector* est le vecteur d'accès, c'est-à-dire le vecteur des objets actuellement ouverts par la transaction. À chaque bit du vecteur est associé un objet. L'identifiant de l'objet permet de pointer le bit correspondant du vecteur. Quand la transaction ouvre un objet, le bit de rang correspondant est mis à 1. Cela permet de connaître l'ensemble des objets ouverts par la transaction active sur π_{proc_id}. Une transaction de lecture parcourt le vecteur pour vérifier au moment du commit la cohérence de l'ensemble des objets ouverts. Une transaction en écriture parcourt le vecteur dans la phase de commit pour savoir quels objets doivent être mis à jour.

Puisqu'à chaque objet du système correspond un bit, la manipulation atomique du vecteur d'accès nous limite le nombre d'objets à 32 pour les machines actuelles classiques.

10.2.3 En-tête des objets

Chaque objet manipulé par différents cœurs dans le système est encapsulé dans un descripteur d'objet que nous appelons *en-tête d'objet*. La Figure 10.5 représente la structure correspondante. Trois champs sont à distinguer. *object_ id* est l'identifiant de l'objet. *copy_ table*

est un pointeur vers une table contenant toutes les versions de l'objet nécessaires au fonctionnement du protocole. Enfin, *concurrency_ vector* représente l'état de la ressource vis-à-vis de toutes les transactions concurrentes qui y accèdent.

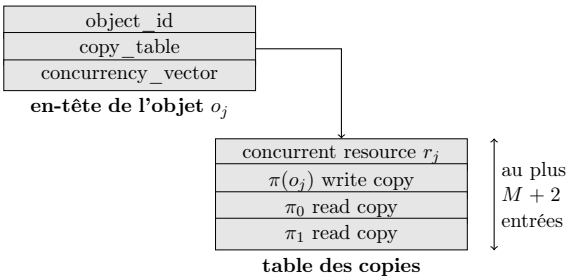


Figure 10.5 – Structure de l'en-tête d'un objet de STM-HRT pouvant être lu par 2 cœurs

Nous détaillons à présent les informations contenus dans les deux derniers champs.

La table présentée par *copy_ table* contient toutes les versions de l'objet. Les deux premières entrées correspondent à la valeur courante de l'objet concurrent et à la valeur de la copie de l'objet quand il est en cours de mise à jour (on rappelle que pour un protocole 1 : m, un seul écrivain à la fois peut mettre à jour l'objet). Un bit dans le *concurrency_ vector* (voir ci-après) permet de connaître à tout instant laquelle de ces deux entrées contient la valeur la plus fraîchement mise à jour. En effet, lors de la mise à jour d'une donnée, ce bit est complémenté. Les autres entrées de la table sont les copies nécessaires aux lecteurs. L'exemple de la Figure 10.5 considère que l'objet peut être lu par des transactions s'exécutant sur les cœurs π_0 et π_1. Il existe deux sortes de lecteurs : **les lecteurs directs** sont des transactions ouvrant directement l'objet en lecture ; **les lecteurs indirects** sont des transactions en écriture qui peuvent être amenées à l'ouvrir en lecture en cas d'aide (voir le détail du protocole dans la Section 10.3, page 135). Seule une copie par cœur qui possède au moins une transaction lisant l'objet est nécessaire. Au pire-cas, on peut ouvrir simultanément un objet en lecture directe ou indirecte sur tous les cœurs. La table des copies contiendra donc au plus $M + 2$ entrées.

concurrency_ vector est un vecteur d'une taille fixe de $2 * M + 2 \leq 32$ bits. La taille du vecteur est bornée de telle manière que l'on puisse effectuer les modifications de manière atomique (mot de 32 bits). Cette borne limite STM-HRT à 15 cœurs. Quatre informations sont présentes dans ce vecteur comme illustré dans la Figure 10.6.

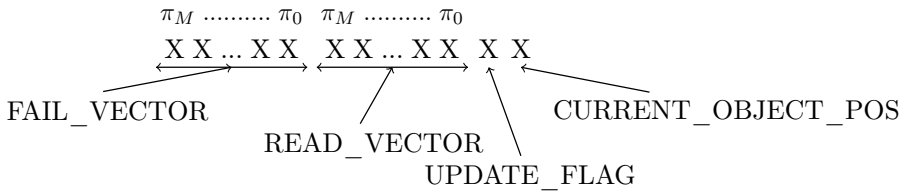


Figure 10.6 – Représentation du *concurrency_ vector*

CURRENT_ OBJECT_ POS est la position de l'objet concurrent à un instant t. Ce bit vaut 0 si l'objet concurrent est le premier élément de la table des copies et vaut 1 si l'objet concurrent est le second élément de la table. *UPDATE_ FLAG* passe à 1 quand une transaction

active d'écriture est en train de modifier l'objet. *READ_ VECTOR* est un vecteur de M bits utilisé pour identifier sur quels cœurs des transactions de lecture ont ouvert l'objet depuis sa dernière mise à jour. Par exemple, si la transaction active sur le cœur π_0 ouvre l'objet en lecture, le bit du *READ_ VECTOR* correspondant à la position de π_0 est mis à 1 (i.e. READ_VECTOR = XX ... X1). Enfin, *FAIL_ VECTOR* est un vecteur de M bits utilisé pour identifier les cœurs sur lesquels les transactions actives ont déjà échoué une première fois. Par exemple, lorsque la transaction active du cœur π_0 échoue, le bit du *FAIL_ VECTOR* correspondant à la position de π_0 est mis à 1. (i.e. FAIL_VECTOR = XX ... X1).

10.2.4 Illustration sur une configuration transactionnelle

Nous illustrons ici les structures de données qui sont statiquement générées à l'initialisation de STM-HRT sur l'exemple donné sur la Figure 10.7, dont la configuration est la suivante :

- $M = 2$. Nous considérons deux cœurs ($\Pi = \{\pi_0,\ \pi_1\}$). Nous aurons donc deux descripteurs de transaction dans le système ; l'un associé à π_0 et le second associé à π_1 ;
- $N = 2$. Nous considérons deux tâches ($T = \{\tau_0,\ \tau_1\}$). La tâche τ_0 est allouée sur π_0 et la tâche τ_1 est allouée sur π_1 ;
- $p = 2$. Nous considérons deux transactions dans le système ($\Omega = \{\omega_0,\ \omega_1\}$). ω_0 est exécutée dans τ_0 (i.e. sur le cœur π_0) et ω_1 est exécutée dans τ_1 (i.e. sur le cœur π_1) ;
- $\rho = 2$. Nous considérons deux données (i.e. 2 objets, $R = \{r_0,\ r_1\}$). Deux en-têtes d'objet seront utilisés.
- ω_0 est une transaction en écriture. Elle modifie r_0 et r_1. La table des objets ouverts en écriture de $\pi(\omega_0) = \pi_0$ a une taille de 2 tandis que la table des objets ouverts en lecture est vide (optimisation des structures de données quand aucune transaction du cœur n'ouvre des objets en lecture) ;
- ω_1 est une transaction de lecture seule. Elle lit r_0 et r_1. La table des objets ouverts en lecture de $\pi(\omega_1) = \pi_1$ a donc une taille de 2 tandis que la taille de la table des objets ouverts en écriture est vide (optimisation des structures de données quand aucune transaction du cœur n'ouvre des objets en écriture) ;
- o_0 et o_1 peuvent être lus en accès direct par ω_1. Les deux objets peuvent également être lus en accès indirect par ω_0 dans le cas où ω_0 aide ω_1. Les tables de copies de o_0 et o_1 ont donc deux emplacements réservés aux lecteurs (4 au total).

Tous les éléments de la table des objets ouverts en écriture ainsi que ceux de la table des objets ouverts en lecture sont initialisés à la valeur *NULL*.

10.3 Algorithmes du protocole STM-HRT

10.3.1 Description de l'API interne

L'accès aux données via le protocole STM-HRT est effectué dans des transactions. L'API externe est l'appel système qui initie la transaction. Dans l'API interne, une transaction est implémenté par une boucle *do ... while*, exécutée tant qu'elle n'a pas réussi. Une description de l'interface interne est proposée dans la Figure 10.8. Une transaction de lecture (lignes 2 à 4) permet de récupérer des données. Par exemple, l'ouverture de l'objet $o1$ à la ligne 4 permet de récupérer la valeur lue dans $inValue$ ($inValue$ est la donnée passé en paramètre par l'API externe pour récupérer la valeur de la donnée dans l'application). De le même manière, une

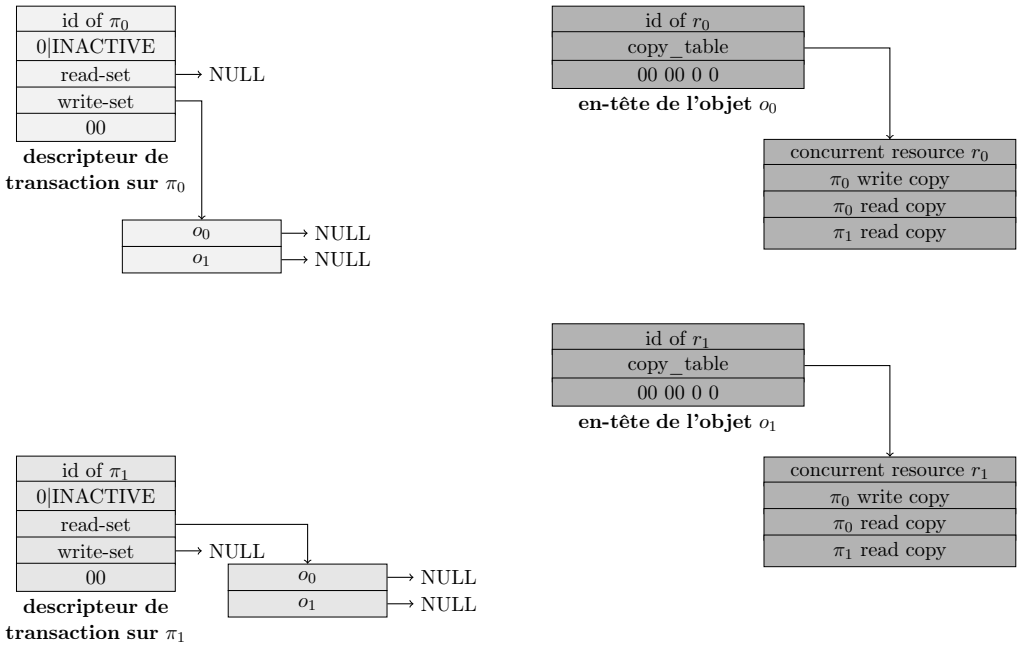


Figure 10.7 – Structures de données générées pour STM-HRT ($M = 2$, $p = 2$, $\rho = 2$)

transaction en écriture permet la mise à jour des objets (lignes 6 et 7). Par exemple, l'objet $o2$ est mis à jour avec la valeur de $outValue$. Notons l'absence de boucle *do ... while* pour la transaction en écriture puisque son succès est garanti par construction.

```
1:   /* transaction homogène en lecture */
2:   do {
3:     open_read_stm_object(o1, &inValue);
       ...
4:   } while (!commit_read_stm_tx())

5:   /*transaction homogène en écriture */
6:   open_write_stm_object(o2, &outValue);
      ...
7:   commit_write_stm_tx()
```

Figure 10.8 – Interface du protocole STM-HRT

STM-HRT repose sur les primitives suivantes :

- *open_read_stm_object()* (et *read_obj()*), pour ouvrir un objet en lecture ;
- *open_write_stm_object()*, pour ouvrir un objet en écriture ;
- *commit_read_stm_tx()*, pour commiter une transaction de lecture ;
- *commit_write_stm_tx()* (et *update()*), pour commiter une transaction en écriture.

10.3.2 Définition des tableaux, des types et des macros

Le protocole requiert l'utilisation de tableaux accessibles uniquement en lecture par les transactions :

- ***transTable*[]** est la table des transactions du système. $transTable[proc\_id] = T_x$ représente l'adresse du descripteur de transaction du cœur π_{proc_id} ;
- ***objectTable*[]** est la table des objets du système. $objectTable[object\_id] = Oh$ représente l'adresse de l'en-tête de l'objet o_{object_id} ;
- ***writerTable*[]** est la table des écrivains. $writerTable[object\_id] = Tx$ représente l'adresse du descripteur de transaction utilisé par les écrivains de l'objet o_{object_id}.

Les différents types utilisés sont référencés dans la Table 10.2 et les macros sont représentées dans la Table 10.3. Ces dernières permettent de simplifier l'écriture des algorithmes et d'abstraire les manipulations sur les structures de données. Les détails sont donnés en Annexe C.

Type	Description
trDesc	Pointeur vers un descripteur de transaction
dataObj	Pointeur vers l'en-tête d'un objet
dataType	Type de l'objet manipulé

Table 10.2 – Définitions des types utilisés dans STM-HRT

Macro	Description
STATUS(Tx)	Valeur du statut de la transaction *Tx*
INSTANCE(Tx)	Numéro de l'instance du descripteur associé à *Tx*
CURRENT_OBJECT_POS(Oh)	Position de la valeur de l'objet *Oh* dans la table des copies
UPDATE_FLAG(Oh)	Valeur de l'UPDATE_FLAG pour l'objet *Oh*
READ_VECTOR(Oh)	Valeur du READ_VECTOR de l'objet *Oh*
FAIL_VECTOR(Oh)	Valeur du FAIL_VECTOR de l'objet *Oh*
SET_READVECTOR(Oh, Tx)	La transaction *Tx* lit l'objet *Oh* : mise à jour du READ_VECTOR de l'objet *Oh*
READVECTOR_BIT(Oh, Tx)	Test si la transaction *Tx* a ouvert l'objet *Oh*
SET_FAILVECTOR(Oh, Tx)	La transaction *Tx* a échoué : mise à jour du FAIL_VECTOR de l'objet *Oh*
FAILVECTOR_BIT(Oh, Tx)	Test si la transaction *Tx* a déjà échouée
RESET_FAILVECTOR(Oh, Tx)	La transaction *Tx* n'a plus besoin d'être aidée : mise à jour du FAIL_VECTOR de l'objet *Oh*
SET_UPDATEFLAG(Oh)	Mise à 1 de l'*UPDATE_FLAG* de l'objet *Oh*
UPDATEFLAG(Oh)	Test si l'*UPDATE_FLAG* de l'objet *Oh* vaut 1
SET_ACCESSVECTOR(Tx, Oh)	L'objet *Oh* est ouvert par *Tx* : mise à jour du vecteur d'accès

Table 10.3 – Définitions des macros utilisées dans STM-HRT

10.3.3 Algorithmes pour l'ouverture des objets

Pour des raisons de simplicité et de clarté, nous ne présentons pas les étapes d'initialisation et de nettoyage des structures de données, réalisées respectivement en début et fin de transaction. Dans la suite, on note ω_{Tx} la transaction active utilisant le descripteur Tx.

10.3.3.1 La primitive open_read_stm_object()

open_read_stm_object() (Algorithme 1) est appelée lors de l'ouverture d'un objet en lecture. Trois arguments sont présents : l'adresse du descripteur de transaction au sein duquel sera stocké le contexte de la transaction active, l'adresse de l'en-tête de l'objet que l'on souhaite lire et l'adresse de la donnée utilisateur où la valeur de l'objet lu sera copiée.

Algorithme 1: open_read_stm_object()

```
Require: trDesc Tx ; dataObj Oh ; dataType *dataRead ;
 1: Init : char instance, objectId ;
 2: if STATUS(Tx) ≠ RETRY and UPDATE_FLAG(Oh) == 1 then
 3:       update(writerTable[Oh.object_id], Oh) ;
 4: end if
 5: objectId = Oh.object_id ;
 6: CAS(&Tx.read_set[objectId], NULL, Oh) ;
 7: instance = INSTANCE(Tx) ;
 8: read_obj(Tx, Tx, Oh, instance) ;
 9: SET_ACCESSVECTOR(Tx, Oh) ;
10: *dataRead ⇐ *Tx.read_set[objectId] ;
```

Avant d'ouvrir l'objet, la transaction de lecture qui effectue son premier essai vérifie si l'objet à ouvrir est sur le point d'être mis à jour. Pour cela, elle vérifie la valeur de l'*UPDATE_FLAG* de l'objet (ligne 2). Si le flag vaut 1, nous appelons la fonction *update()* à la ligne 3, uniquement pour l'objet concerné. Ceci permet de résoudre au plus tôt un conflit lecture/écriture sur l'objet.

Dès qu'une transaction de lecture est prête à lire un objet, le champ correspondant de la table des objets ouverts en lecture pointe vers l'en-tête de l'objet à lire (ligne 6). L'index de ce champ correspond à l'identifiant de l'objet récupéré à la ligne 5. Désormais, si ω_{Tx} est aidée, les transactions aidantes sont autorisées à lire l'objet au nom de ω_{Tx}.

L'objet est concrètement lu à la ligne 8 en appelant la fonction *read_obj()*, décrite dans l'Algorithme 2 ci-après. À la fin de la lecture, la copie de l'objet lu correspond à la valeur pointée dans la table des objets ouverts en lecture. Désormais, si ω_{Tx} est aidée, les transactions aidantes ne peuvent plus lire l'objet au nom de ω_{Tx} (c.f. échec du CAS ligne 16 de l'Algorithme 2).

Le vecteur d'accès est rempli à la ligne 9 pour mémoriser le fait que l'objet a été ouvert par ω_{Tx}. L'objet lu est enfin copié à l'adresse de *dataRead*, fourni en paramètre (ligne 10). La valeur courante de l'objet devient disponible dans la tâche au sein de laquelle la transaction est exécutée. Elle sera utilisable après le commit et le succès de ω_{Tx}.

10.3.3.2 La primitive read_obj()

read_ obj() (Algorithme 2) est appelée lorsque l'on souhaite lire un objet. Quatre arguments sont présents : deux adresses de descripteurs de transaction (Tx1 est le descripteur de la transaction qui exécute la fonction au nom de la transaction utilisant le descripteur Tx2, en cas d'aide), l'adresse de l'en-tête de l'objet que l'on souhaite lire et la valeur de l'instance du descripteur de transaction sur lequel s'exécute la transaction qui doit lire l'objet. En cas d'aide, cette dernière information permet de maintenir la cohérence des structures de données de STM-HRT.

Algorithme 2: read_obj()

```
Require: trDesc Tx1, Tx2 ; dataObj Oh ; char instance ;
 1: Init : char concurObjectIndex, readCpyIndex, objectId ;
 2: if STATUS(Tx2) == RETRY and INSTANCE(Tx2) == instance then
 3:     SET_FAILVECTOR(Oh, Tx2) ;
 4: end if
 5: loop
 6:     SET_READVECTOR(Oh, Tx2) ;
 7:     concurObjectIndex = CURRENT_OBJECT_POS(Oh) ;
 8:     readCpyIndex = 2 + Tx1.proc_id ;
 9:     Oh.copy_table[readCpyIndex] ⇐ Oh.copy_table[concurObjectIndex] ;
10:     if READVECTOR_BIT(Oh, Tx2) then
11:         break ;
12:     end if
13: end loop
14: objectId = Oh.data_id ;
15: if INSTANCE(Tx2) == instance and STATUS(Tx2) < INACTIVE then
16:     CAS(&Tx1.read_set[objectId], Oh, &Oh.copy_table[readCpyIndex]) ;
17: end if
```

Avant de lire l'objet, une transaction de lecture qui a déjà échoué lors de sa première tentative positionne le bit correspondant à son *proc_id* dans le *FAIL_ VECTOR* à 1 (ligne 2-4). Par exemple, si ω_0 exécutée sur le cœur π_0 échoue, le bit d'indice π_0 sera mis à 1 (voir Figure 10.6) au moment où ω_0 ouvrira l'objet. Ceci permet de « protéger » l'objet des éventuelles mises à jour qui pourraient à nouveau mettre en échec la transaction ω_{Tx2}. Cette mise à 1 est atomique et n'est effectuée que si le bit valait 0.

La lecture de l'objet est effectuée aux lignes 5 à 13. Tout d'abord, la transaction positionne le bit correspondant à son *proc_id* dans le *READ_ VECTOR* à 1 (ligne 6). Pour lire l'objet, on copie la valeur courante (position de l'objet concurrent obtenue à la ligne 7) dans l'emplacement réservé à la copie quand une transaction de *proc_id* accède à l'objet en lecture (position obtenue à la ligne 8). Notons que l'emplacement où nous allons copier l'objet est calculé à partir de ω_{Tx1}, car quand ω_{Tx2} est aidée par plusieurs transactions, cela permet d'éviter les conflits écriture/écriture sur l'emplacement de la copie. L'objet est concrètement copié à la ligne 9.

Toutes les étapes précédentes sont réalisées dans une boucle. En effet, avec une granularité objet, la lecture n'est pas atomique et nous devons être certains d'avoir récupéré une valeur consistante de l'objet. Pour vérifier la consistance, on vérifie si le bit mis à 1 à la ligne 6 vaut

toujours 1 à la ligne 10. Si oui, la condition de sortie de la boucle est respectée. Si ce n'est pas le cas, nous recommençons la lecture. Notons néanmoins qu'un risque surviendrait si l'objet était constamment mise à jour pendant que nous souhaitons copier la valeur de l'objet. Dans un tel cas, nous ne sortirions jamais de la boucle. Nous présentons un problème similaire dans la Section 10.3.5 (page 142). L'analyse porté pour ce même problème peut être appliqué ici.

Une fois qu'une valeur consistante est obtenue, le pointeur correspondant dans le tableau des objets ouverts en lecture pointe vers la valeur qui a été lue (ligne 16). En cas d'aide, plusieurs transactions peuvent lire l'objet en même temps, en effectuant la copie dans des emplacements qui leur sont propres. Le CAS de la ligne 16 permet de pointer la valeur lue qui est prête en premier.

10.3.3.3 La primitive open_write_stm_object()

open_write_stm_object() (Algorithme 3) est appelée lorsque l'on souhaite ouvrir un objet en écriture. Trois arguments sont présent : l'adresse du descripteur de transaction au sein duquel est stocké le contexte de la transaction active, l'adresse de l'en-tête de l'objet que l'on souhaite écrire et l'adresse de la donnée utilisateur qu'il faut écrire.

Algorithme 3: open_write_stm_object()

Require: *trDesc* Tx ; *dataObj* Oh ; *dataType* *dataToWrite ;

```
1: Init : char objectID ;
2: objectID = Oh.data_id ;
3: *Tx.write_set[objectID] ⇐ *dataToWrite ;
4: SET_UPDATEFLAG(Oh) ;
5: SET_ACCESSVECTOR(Tx) ;
```

Lorsqu'une transaction en écriture ouvre un objet, elle recopie la valeur de $dataToWrite$ à l'emplacement pointé par le champ correspondant dans la table des objets ouverts en écriture (ligne 3). Ce pointeur est initialisé dans la phase de démarrage de la transaction, qui n'est pas représentée ici. Cette copie est obligatoirement consistante, car aucune aide n'est possible à ce niveau-là.

À la ligne 4, l'$UPDATE\_FLAG$ de l'objet lu est positionné à 1. Ceci permet d'informer toutes les transactions qui accèdent à l'objet en lecture qu'il est sur le point d'être mis à jour.

Enfin, le vecteur d'accès de ω_{Tx} est rempli à la ligne 5.

10.3.4 Algorithmes pour le commit des transactions

10.3.4.1 La primitive commit_read_stm_tx()

commit_read_stm_tx() (Algorithme 4) est appelée lors du commit d'une transaction de lecture. Deux arguments sont présents : l'adresse du descripteur de transaction au sein duquel est stocké le contexte de la transaction active et l'instance du descripteur pour lequel le commit est nécessaire. Cette dernière information permet de maintenir la cohérence des structures de données en cas d'aide. La fonction retourne un booléen qui vaut $true$ en cas de succès du commit et $false$ sinon.

Le commit d'une transaction de lecture est décomposé en deux étapes. La première (lignes 1-13) permet de vérifier l'intégrité des objets ouverts dans la transaction. À la fin de cette

Algorithme 4: commit_read_stm_tx()

```
Require: trDesc Tx ; char instance ;
    Return value : bool result ;
 1: Init : char objectId, desiredStatus, oldStatus, finalStatus ; dataObj Oh ⇐ NULL ;
    int TxObjectSet ;
 2: desiredStatus = (instance++ ≪ 2) | INACTIVE ;
 3: TxObjectSet = Tx.access_vector ;
 4: objectId = 0 ;
 5: for objectId in TxObjectSet do
 6:       Oh = dataTable[objectId] ;
 7:       if not READVECTOR_BIT(Oh, Tx) then
 8:           goto Fail ;
 9:       end if
10: end for
11: goto Finish ;
12: Fail :
13: desiredStatus = (instance++ ≪ 2) | FAILED ;
14: Finish :
15: oldStatus = Tx.status ;
16: if STATUS(oldStatus) < FAILED and INSTANCE(oldStatus) == instance then
17:       CAS(&Tx.status, oldStatus, desiredStatus) ;
18: end if
19: finalStatus = Tx.status ;
20: return finalStatus == INACTIVE ;
```

étape, l'issue de la transaction est connue : elle va soit échouer, soit réussir. Au cours des lignes 5 à 10, le vecteur d'accès de la transaction récupérée à la ligne 3 est parcouru. Pour chaque objet ouvert, on vérifie si l'objet a été mis à jour depuis sa lecture (ligne 7). Si c'est le cas, la transaction va échouer (lignes 8 et 13). Nous rappelons que le protocole garantit que la seconde tentative d'une transaction de lecture va systématiquement réussir. L'étape que l'on vient de décrire pourrait alors être omise en cas d'aide.

La seconde étape (lignes 13-20) permet de mettre à jour le statut de la transaction. La mise à jour du statut n'est possible que si l'instance du descripteur de transaction est bien l'instance passée en paramètre, et si le statut de la transaction n'est pas *INACTIVE* (i.e. transaction déjà terminée). La mise à jour du statut résulte en un CAS. Si le CAS échoue, cela signifie que le statut n'a plus besoin d'être mis à jour parce qu'une transaction aidante l'a déjà fait à sa place.

Après le succès de la phase de commit, l'instance du descripteur de transaction est incrémentée et les structures de données associées à la transaction sont nettoyées. En particulier, les bits du *FAIL_ VECTOR* sont tous remis à 0 par l'appel de la macro *RESET_ FAILVECOR(...)* pour tous les objets ouverts. Nous ne représentons pas les algorithmes de la fonction de fermeture.

10.3.4.2 La primitive commit_write_stm_tx()

commit_ write_ stm_ tx() (Algorithme 5) est appelée lors du commit d'une transaction en écriture. Un seul paramètre est fourni en entrée : l'adresse du descripteur de transaction au

sein duquel est stocké le contexte de la transaction active.

Algorithme 5: commit_write_stm_tx()

Require: *trDesc* Tx ;

```
1: Init : char objectId ; dataObj Oh ⇐ NULL ; int TxObjectSet ;
2: TxObjectSet = Tx.access_vector ;
3: objectId = 0 ;
4: for objectId in TxObjectSet do
5:     Oh = dataTable[objectId] ;
6:     update(Tx, Oh) ;
7: end for
```

Comme pour le cas d'une transaction de lecture, une transaction en écriture récupère le vecteur d'accès (ligne 2) et le parcourt (lignes 4-7). Pour chaque objet ouvert dans la transaction, la fonction *update()* est appelée.

Au succès de la transaction, l'instance est incrémentée et le statut passe à *INACTIVE*.

10.3.4.3 La primitive update()

update() (Algorithme 6) permet de mettre à jour un objet. Deux arguments sont présents entrée : l'adresse du descripteur de transaction au sein duquel est stocké le contexte de la transaction active et l'adresse de l'en-tête de l'objet manipulé.

La mise à jour d'un objet passe par l'exécution de l'instruction *ATOMIC_UPDATE* (ligne 3 de l'Algorithme 6). En effet, avant de mettre à jour l'objet, il convient de vérifier de façon atomique que les critères suivants sont respectés sur le *concurrency_vector* : *UPDATE_FLAG* = 1 et *FAIL_VECTOR* = 0. Si ces conditions sont respectées, la mise à jour consiste à mettre le *concurrency_vector* à 0 à l'exception du bit *CURRENT_OBJECT_POS* qui doit être complémenté. Par exemple, si le vecteur vaut $XX...XX/00...00/1/0$ avant sa mise à jour, il vaudra $00...00/00...00/0/1$ après la mise à jour. *ATOMIC_UPDATE* est implémenté à base d'instructions LL/SC.

Si au moins l'une des conditions n'est pas respectée, le *concurrency_vector* n'est pas modifié. Si l'*UPDATE_FLAG* valait 0, cela signifie qu'une autre transaction a déjà mis à jour l'objet dans un contexte d'aide. La mise à jour est donc terminée (lignes 4 à 6) et on sort de la fonction. En revanche, si l'*UPDATE_FLAG* valait 1 mais que le *FAIL_VECTOR* était différent de 0, cela signifie que certaines transactions en lecture ont échoué. Dans ce cas, il faut aider toutes les transactions qui en ont besoin. Pour chacune d'elles, le vecteur d'accès est récupéré (ligne 12), parcouru et tous les objets désignés sont ouverts (lignes 14 à 19). Enfin, la fonction de commit pour la transaction à aider est appelée (ligne 20). Une fois l'aide terminée, le bit correspondant du *FAIL_VECTOR* est réinitialisé de manière atomique (i.e. uniquement s'il valait encore 1) (ligne 24).

10.3.5 Le mécanisme d'aide

La progression wait-free est assurée grâce à un mécanisme d'aide.

– Au moment de son commit, une transaction en écriture peut aider une transaction de lecture (en appelant des fonctions *read_obj()* et *commit_read_stm_tx()*) ;

Algorithme 6: update()

```
Require: trDesc Tx ; dataObj Oh ;
 1: Init : char transId, objectId, instanceToHelp ; trDesc helpedTx ;
    dataObj helpedOh ⇐ NULL ; int helpedTxObjectSet ;
 2: loop
 3:     ATOMIC_UPDATE(Oh) ;
 4:     if ! UPDATEFLAG(Oh) then
 5:         break ;
 6:     end if
 7:     transId = 0 ;
 8:     while (transId < (NB_ACTIVE_TRANS - 1)) do
 9:         helpedTx = transTable[transId] ;
10:         instanceToHelp = INSTANCE(helpedTx) ;
11:         if FAIL_BIT(Oh, helpedTx) then
12:             helpedTxObjectSet = helpedTx.access_vector ;
13:             objectId = 0 ;
14:             for objectId in TxObjectSet do
15:                 helpedOh = objectTable[objectId] ;
16:                 if INSTANCE(helpedTx) == instanceToHelp and
                    STATUS(helpedTx) == RETRY then
17:                     read_obj(Tx, helpedTx, helpedOh, instanceToHelp) ;
18:                 end if
19:             end for
20:             if INSTANCE(helpedTx) == instanceToHelp and STATUS(helpedTx) ==
                RETRY then
21:                 commit_read_stm_tx(Tx, instanceToHelp) ;
22:             end if
23:             if (INSTANCE(helpedTx) == instanceToHelp) then
24:                 RESET_FAILVECTOR(Oh, Tx) ;
25:             end if
26:         end if
27:         transId ++ ;
28:     end while
29: end loop
```

– Avant d'ouvrir un objet, une transaction de lecture peut aider une transaction en écriture qui est sur le point de mettre à jour ce même objet (par l'appel de la fonction *update()* dans l'Algorithme 1). En aidant, une transaction de lecture peut être amené à aider d'autres transactions de lecture.

Le mécanisme d'aide doit être maîtrisé pour éviter des problèmes de corruption des structures de données. En effet, dès que de l'aide est possible, le problème que l'on définit sous le terme de ***problème d'instance*** (problème lié au progrès concurrent des tâches sur les cœurs) peut survenir. Pour illustrer ce problème, considérons une transaction ω_i (transaction de lecture sur le cœur $\pi_1 = \pi(\omega_i)$) et une transaction ω_j (transaction en écriture sur le cœur

$\pi_2 = \pi(\omega_j) \neq \pi(\omega_i)$). Si ω_i a déjà échoué une fois et que ω_j est sur le point de mettre à jour un objet ouvert par ω_i, ω_j aide ω_i.

Supposons maintenant que ω_i termine d'elle-même et que ω_k, une nouvelle transaction, démarre sur π_1 (et utilise le descripteur de transaction associé à π_1). STM-HRT doit être en mesure d'éviter que ω_j ne corrompte les actions effectuées par ω_k pendant qu'elle aidait ω_i.

Pour cela, nous utilisons le compteur d'instance qui est inclus dans le champ *status* du descripteur de transaction. Dans les algorithmes *read_ obj()*, *commit_ read_ stm_ tx()* et *update()*, l'instance du descripteur de transaction contenant le contexte de la transaction à aider est passée en paramètre. Nous vérifions constamment dans ces algorithmes que l'instance en cours est bien celle passée en paramètre et que le statut de la transaction est également correct (c.f. tests des lignes 2 et 15 de *read_ obj()*, ligne 20 de *commit_ read_ stm_ tx()* et lignes 17, 24 et 27 de *updates()*). Dans la Section 10.4.4.1 (page 147), nous présentons les choix d'implémentation pour le traitement de ce problème.

10.3.6 Exemple d'illustration

Des exemples illustrant le fonctionnement de STM-HRT lorsque des transactions sont exécutées en totale isolation puis en cas de conflits sont présentés en Annexe D.

10.4 Intégration de STM-HRT dans le RTOS Trampoline

STM-HRT a été implanté dans le système d'exploitation temps réel (RTOS) *Trampoline*. *Trampoline* est présenté dans l'Annexe A. Le service mis en place est un service de communication qui permet l'envoi et la réception de groupes de messages entre des tâches d'OS-Applications différentes. Le protocole s'intègre dans AUTOSAR comme une alternative possible à l'utilisation de l'IOC. Il a en charge le maintien de la consistance des données.

10.4.1 Architecture du service de communication

L'architecture du service de communication inter-cœur est présentée dans la Figure 10.9. Trois niveaux sont représentés. Le plus haut niveau est celui de l'application. On y distingue deux tâches qui peuvent s'envoyer mutuellement des messages. Le second niveau est celui de la configuration. On y trouve ce qui est généré par le compilateur *goil*. Enfin, le niveau le plus bas est celui du noyau. On y trouve l'implémentation du service de communication, dont le protocole STM-HRT. Ce niveau n'est pas modifié par l'intégrateur.

10.4.2 Configuration statique du protocole

Les algorithmes décrits dans la Section 10.3 (page 135) sont implémentés dans le noyau de *Trampoline* (API interne). Ils ne sont pas directement manipulés par l'utilisateur. Pour pouvoir les utiliser, nous devons construire l'architecture que nous venons de décrire. Plus précisément, nous devons décrire le mode d'obtention de la couche de configuration, c'est-à-dire le code glus permettant le lien entre l'utilisation du service dans l'application (utilisation des transaction) et les appels systèmes qui conduisent à l'utilisation des APIs du protocole. La configuration spécifique à notre service de communication doit être supportée par le langage *OIL*. Il faut donc enrichir le langage. En plus de spécifier le nombre de cœurs, de déclarer les tâches et les

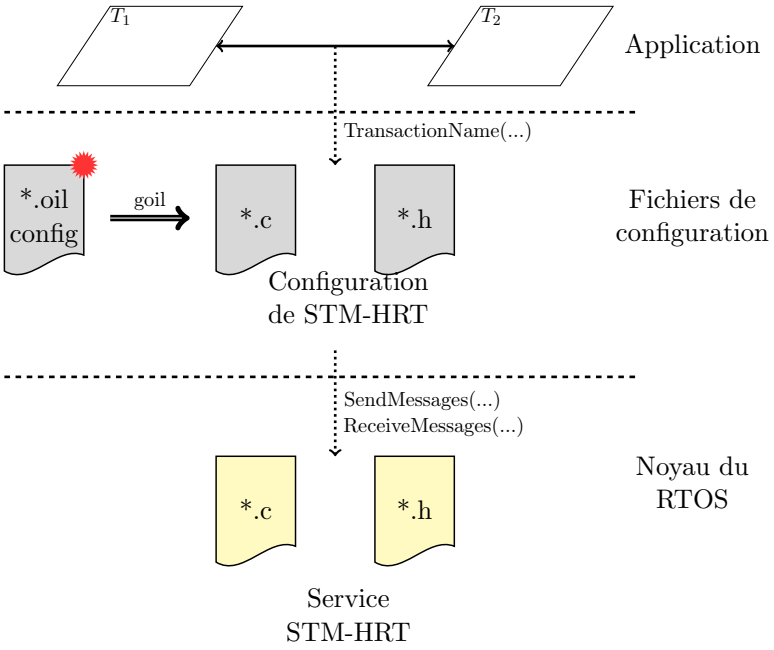


Figure 10.9 – Mise en œuvre du service de communication mis en place avec STM-HRT

OS-Applications, nous devons aussi décrire les transactions et les objets. Cette configuration nous permet d'obtenir les ensembles que nous évoquions sur la Figure 10.1.

Comme décrit par la Figure 10.10, un objet est défini par son type et sa valeur initiale tandis qu'une transaction est définie par le type d'accès (transaction homogène en lecture ou en écriture) et l'ensemble des objets qu'elle manipule. Enfin, une tâche référence les transactions qu'elle souhaite pouvoir exécuter. L'allocation des tâches aux OS-Applications (et donc aux cœurs) n'a pas été modifiée. La configuration analysée hors-ligne permet de générer les structures de données adaptées à la situation et prêtes à l'emploi.

La phase de génération de code permet également de générer des fonctions intermédiaires qui permettent de faire le lien entre l'appel de la transaction par l'utilisateur dans la tâche et l'appel du service dans le noyau. En effet, a priori, nous ne savons pas combien d'objets doivent être ouverts par une transaction. Les fonctions intermédiaires permettent d'encapsuler dans un tableau les objets à ouvrir et de transmettre ce tableau directement au service en un seul appel système (c.f. Figure 10.11).

10.4.3 Comportement en-ligne du service

Pour illustrer le comportement en-ligne, considérons une transaction ω_1 exécutée par τ_1 lisant les objets o_1 et o_2, et une transaction ω_2 exécutée par τ_2 écrivant les objets o_1 et o_2.

En-ligne, lorsque τ_1 exécute ω_1, nous commençons par appeler la fonction *trans*1 générée par *goil* (Figure 10.11 (a)). Cette fonction prend en paramètres les adresses des variables dans lesquelles il faudra écrire les valeurs lues (ici, **trans1(&obj1, &obj2)**). Une table des objets est ensuite construite et passée en paramètre de la fonction d'appel du service

```
OBJECT objectName {
  CDATATYPE = "dataType";
  INITIAL_VALUE = "initialValue";
};
OBJECT ... {
  ...
};

TRANSACTION transactionName {
  ACCESS_TYPE = WRITE;
  OBJECT = objectName;
  OBJECT = ...
};
TRANSACTION ... {
  ...
};

TASK taskName {
  ...
  TRANSACTION = transactionName;
  TRANSACTION = ...
};
```

Figure 10.10 – Extension du langage OIL pour configurer STM-HRT

ReceiveMessages() (on passe également en paramètre la taille de cette table). Cette fonction implémente l'API interne correspondante, présentée dans la Figure 10.8 (lignes 2 à 4).

Lorsque τ_2 exécute ω_2, la fonction *trans*2 générée par *goil* est appelée (Figure 10.11 (b)). Cette fonction prend en paramètres les adresses des variables contenant les valeurs à écrire (ici, **trans2(&obj1, &obj2)**). Une table d'objets est là-aussi construite et passée en paramètre de la fonction d'appel du service *SendMessages*() (on passe également en paramètre la taille de cette table). Comme précédemment, cette fonction implémente l'API interne présentée dans la Figure 10.8 (lignes 6 et 7).

```
function trans1(int *o0, int *o1)
{
  /* ... */
  object_table[0].object_id = o0_id;
  object_table[0].user_data = o0;

  object_table[1].object_id = o1_id;
  object_table[1].user_data = o1;

  ReceiveMessages(object_table, 2);
}
```

(a) Fonction générée pour la transaction ω_1

```
function trans2(int *o0, int *o1)
{
 /* ... */
  object_table[0].object_id = o0_id;
  object_table[0].user_data = o0;

  object_table[1].object_id = o1_id;
  object_table[1].user_data = o1;

  SendMessages(object_table, 2);
}
```

(b) Fonction générée pour la transaction ω_2

Figure 10.11 – Fonctions générées pour l'appel du service

10.4.4 Détails d'implémentation du service

10.4.4.1 Choix d'implémentation lié au *problème d'instance*

Le problème d'instance a été présenté dans la Section 10.3.5 (page 142). Pour être parfaitement robuste, il faudrait que les sections de tests (i.e. tests des lignes 2 et 15 de *read_ obj()*, ligne 20 de *commit_ read_ stm_ tx()* et lignes 17, 24 et 27 de *updates()*) soient atomiques. Pour implémenter cette atomicité, il faudrait utiliser des spinlocks. Pour les raisons de robustesse évoquées dans la Section 2.3.1 (page 33), nous désirons ne pas utiliser de tels mécanismes. C'est pourquoi nous émettons une hypothèse nous permettant de ne pas l'utiliser.

Sur des architectures microcontrôleurs multicœur identiques de puissance de calcul limitée comme le Leopard, **le cadencement des cœurs est du même ordre de grandeur que les accès mémoire**. Nous justifions donc la cohérence de notre choix en argumentant sur la taille de la plage de temps pendant laquelle les sections de code non protégées avec des verrous peuvent être exécutées en parallèle.

- ligne 2-4 de l'Algorithme 2 – la plage de temps est bornée par le temps requis par la transaction aidée pour finir par elle-même ;
- ligne 15-16 de l'Algorithme 2 – la plage de temps est bornée par le temps requis par la transaction aidée pour finir par elle-même et qu'une autre transaction démarre sur le même cœur et ouvre le même objet (protection au niveau du CAS de la ligne 16) ;
- ligne 20-21 de l'Algorithme 4 – aucun problème puisque le CAS de la ligne 21 prend en paramètre le statut sur lequel les tests ont été faits. Au pire-cas, le CAS échoue ;
- lignes 17-18 et 24-25 de l'Algorithme 6 – les tests sont redondants à ceux exprimés dans les Algorithmes 2 et 4. Ils permettent de gagner en performance. Pas de borne temporelle à calculer.
- ligne 27-28 de l'Algorithme 6 – la plage de temps est bornée par le temps requis par la transaction aidée pour finir par elle-même, pour qu'une autre transaction démarre sur le même cœur et que cette dernière échoue.

La situation qui cause le pire-cas (i.e. la plage de temps la plus petite) correspond à celle du premier item. Nous allons analyser le problème de manière quantitative en considérant l'architecture matérielle du Leopard MPC 5643L présentée dans la Section 2.1.2 (page 24). Le problème est illustré sur la Figure 10.12.

Une transaction ω_i s'exécute sur le cœur π_0, échoue et rejoue. ω_i est aidée par ω_j, qui s'exécute sur le cœur π_1. Pendant l'aide ω_i continue d'évoluer normalement. On suppose que l'objet manipulé est o_0.

Pour aider ω_i, ω_j commence par exécuter le test de la ligne 2 de l'Algorithme 2, au nom de ω_i. Il faut alors tester si (1) ω_i est en phase de rejoue (i.e. requiert de l'aide), et si (2) l'instance est bien celle à aider. ω_i respecte ces conditions jusqu'à la ligne 21 de l'Algorithme 4. Une fois le test validé, ω_j doit désormais mettre à 1 le bit *FAIL_ VECTOR* de o_0 à la ligne 3 de l'Algorithme 2. La plage de temps valide pour réaliser cette opération correspond à l'intervalle entre le moment où le test de la ligne 2 ne serait plus vérifié et le moment où ω_i remet le bit *FAIL_ VECTOR* à 0. En effet, au-delà de cette limite, une remise à 1 de ce bit serait une erreur (on ne bornerait plus le nombre d'abandon). C'est cet intervalle qu'il faut dimensionner.

Pour faire le dimensionnement, nous compilons le code pour l'architecture powerPC, avec une optimisation -O2. Nous extrayons et analysons ensuite le code assembleur qui a été pro-

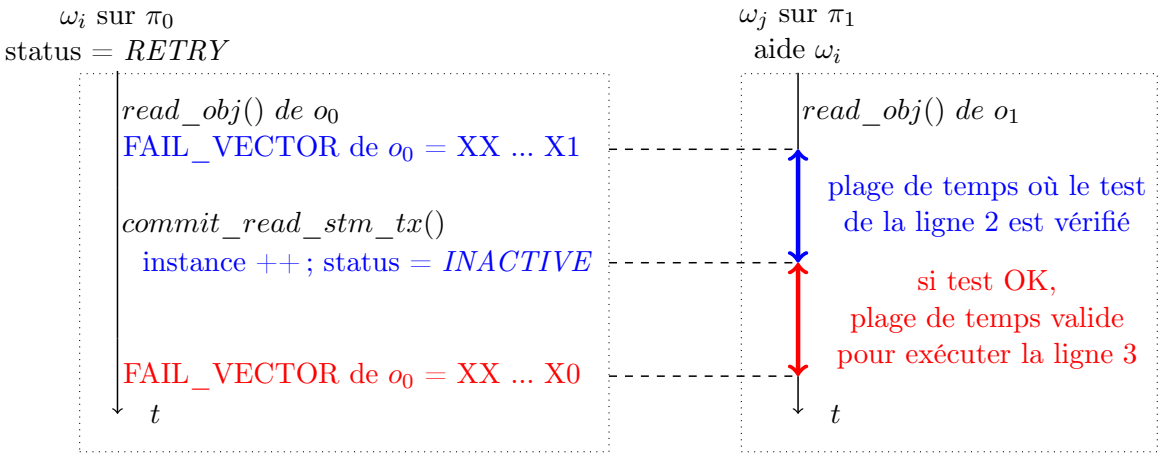


Figure 10.12 – Plage de temps valide pour réaliser les lignes 2 et 3 de l'Algorithme 2 en cas d'aide

duit (powerPC-elf-objdump). En analysant les documentations du cœur e200z4 utilisé dans le Leopard, nous considérons également les informations suivantes (par cœur) :

- Le programme est compilé avec les instructions PowerPC (32 bits) ;
- Le processeur est équipé d'un pipeline à 5 étages (Fetch, Decode, Execute0 or Memory0, Execute1 or Memory1, WriteBack), superscalaire d'ordre 2. Des étages peuvent être gelés en cas de dépendances des instructions. Il y a également un mécanisme de bypass. Au mieux, le débit est de 2 instructions par cycles. Il y a un buffer d'instructions de 8 entrées de 32 bits. 2 instructions 32 bits (ou 4 de 16 bits) y sont insérées à chaque Fetch. Le cache d'instructions vaut 4K et possède 8 instructions de 32 bits par lignes. Il n'y a pas de caches de données. Enfin, l'arbitrage d'accès au bus round robin pour les accès en mémoire (load/store).

Règles considérées pour l'exécution des instructions sur le pipeline (par cœur) :

- Les instructions sont exécutées dans l'ordre du programme : pas de d'exécution dans le désordre ;
- L'étape du fetch permet de récupérer 2 instructions de la mémoire principale ou du cache et de les placer dans le buffer d'instructions. Si le buffer est plein, les instructions lues qui ne passent pas sont perdues ;
- Les étapes de Fetch (succès de cache), Decode, WriteBack sont effectuées en 1 cycle, pour 2 instructions en même temps. Les autres étages peuvent nécessiter plusieurs cycles ;
- Les accès mémoire en lecture (load) prennent 3 cycles. Ceux en écriture (store) prennent 1 cycle. Il y a une seule unité de load/store, c'est-à-dire un seul accès mémoire en même temps ;
- Les accès au cache prennent 1 cycle ;
- L'accès à la mémoire flash prend 4 cycles ;
- Un buffer de ligne ;
- Une instruction ne peut pas changer de pipeline en cours d'exécution ;
- Une instruction est bloquée dans le pipeline si elle n'a pas les ressources suffisantes pour s'exécuter ;

– En cas de gel, tous les étages précédents du pipeline sont gelés ;
– 2 instructions dépendantes ne peuvent pas être exécutées en même temps dans le pipeline superscalaire.

La configuration présentée dans le précédent paragraphe nous conduit à isoler 34 instructions pour ω_i. C'est l'intervalle entre le moment où le test de la ligne 2 de l'Algorithme 2 n'est pus valide et celui où le *FAIL_BIT* est remis à 0. Pour ω_j, nous isolons les 7 instructions nécessaires à la mise à 1 du *FAIL_BIT* (ligne 3 de l'Algorithme 2). Pour calculer le nombre de cycles que cela représente dans chacun des cas, nous calculons le meilleur temps d'exécution (BCET) pour ω_i et le pire temps d'exécution (WCET) pour ω_j.

WCET ω_j : Supposons le pire cas où les 7 instructions sont mal alignées en mémoire. L'état du cache est le suivant : 1 instruction en dernière ligne du cache, 8 sur la ligne suivante et 3 sur la ligne d'après. Le chargement des deux premières instructions va conduire à un défaut du cache. Le chargement de la 9e instruction va également conduire un défaut du cache. On suppose enfin que l'arbitrage sur les accès mémoire conduit toujours à une perte face à l'arbitrage. Une analyse en suivant les règles ci-dessus, conduit à un nombre de cycles égal à 21.

BCET ω_i : Le meilleur temps d'exécution suppose que nous avons toujours des succès de cache, c'est-à-dire que toutes les instructions seront obtenues en 1 cycle. On suppose également qu'au meilleur cas, on gagne toujours l'arbitrage de contention sur le bus. Enfin, on va également supposer que toutes les instructions sont indépendantes, c'est-à-dire que le superscalaire offre les meilleures performances possibles. Nous n'effectuons pas l'analyse fin grain sur les 34 instructions, mais en considérants uniquement les 17 fetchs et les 10 accès mémoire indépendants, nous sommes déjà à 47 cycles.

Le pire cas pour ω_j est vraiment pessimiste, car en réalité, le buffer d'instructions et le cache vont accélérer l'exécution. Le meilleur cas pour ω_i prend en compte des hypothèses qui ne sont pas respectées dans la réalité. L'analyse fin grain du déroulement des instructions n'est pas une chose facile. Il faut en effet disposer de toutes les documentations nécessaires à la compréhension des mécanismes matériels mis en œuvre dans l'architecture. Une autre solution consisterait à modéliser le programme et le matériel. Par exemple, des travaux effectués en monocœur à l'aide de l'outil UPPAAL (UPPAAL, 2013) sont disponibles dans Cassez et Béchennec (2013).

10.4.4.2 Implémentation de l'atomicité

L'implémentation est générique pour toutes les cibles sauf en ce qui concerne l'implémentation des sections atomiques. Une preuve de concept a été faite sous POSIX. Pour fonctionner, *Trampoline* repose alors sur un simulateur appelé *Viper*. L'implémentation sous POSIX n'a pas de visée temps réel donc nous choisissons de simuler les sections atomiques par la prise d'un sémaphore global et en désactivant les signaux entre *Viper* et *Trampoline*.

L'implémentation sur cible n'a pas été réalisée. Nous donnons néanmoins quelques informations liées à son implémentation. Selon nos hypothèses, une transaction sur un cœur ne doit pas être préemptée. Pour ce faire, l'exécution du service est réalisée par le noyau. Dans un contexte AUTOSAR, STM-HRT est un protocole qui peut s'interfacer avec les APIs de l'IOC tel que spécifié par la norme AUTOSAR OS multicœur (AUTOSAR, 2013b). Pour éviter qu'une transaction ne soit préemptée à l'intérieur du noyau pendant les sections atomiques,

nous commençons par désactiver les interruptions. L'atomicité peut être implémentée de plusieurs façons :

- Utilisation d'instructions LL/SC ;
- Si la cible ne supporte pas ce type d'instruction, l'utilisation de verrous peut être envisagé mais une défaillance de celui-ci pourrait se propager.

Chapitre 11

Analyse du protocole STM-HRT

Sommaire

11.1 Modélisation et vérification du protocole avec SPIN

11.1.1 Principe de modélisation

La conception du protocole STM-HRT a été évaluée à l'aide du model checker *SPIN* (Holzmann, 1997). À partir d'un modèle du système écrit en PROMELA (PROtocol MEta LAnguage), *SPIN* (Simple Promela INterpreter) permet de vérifier si les propriétés écrites en logique temporelle LTL (et traduites en automate de Büchi) sont vérifiées. Pour des raisons de performance, SPIN ne fait pas la vérification lui-même, mais génère un code C quei nous avons compilé avec les options suivantes :

- BITSTATE pour compresser les états ;
- DNP pour contrôler la présence de cycles dans le modèle (livelock) ;
- DFS (*Depth-First Search*) pour faire une analyse en profondeur d'abord (par opposition à l'analyse en largeur BFS (*Breadth First Search*)) ;
- Option de compilation gcc O2.

Le binaire obtenu est ensuite exécuté pour faire la vérification avec différents paramètres. Parmi ces paramètres, nous donnons la profondeur maximale en nombre d'états (ce nombre d'états doit être suffisamment important pour pouvoir analyser le modèle dans sa globalité) et la taille de la table de hachage utilisée pour le stockage des états (de la forme -wXX). Ce paramètre est égal à la puissance de 2 du nombre d'état qu'on peut analyser (-w23 pour 8 million d'états). La méthode préconisée dans la documentation de SPIN est la suivante. Nous commençons avec une taille arbitraire et nous recommençons les vérifications en ajoutant 1. Un incrément de 1 permet d'analyser deux fois plus d'états. Nous arrêtons quand le nombre d'états analysés est stable entre deux incréments, ce qui traduit l'exhaustivité de l'analyse.

11.1.2 Protocole de test et résultats

Nous avons implémenté le protocole STM-HRT en PROMELA et défini un ensemble d'exigences à vérifier avec la version 6.2 de *SPIN*. Le modèle est asynchrone et ne fait donc aucune hypothèse sur la progression temporelle de chaque processus. C'est pour cela que la modélisation doit être adaptée à ce comportement. En particulier, tous les blocs atomiques sont modélisés à l'aide de section *atomic{ ... }* du langage PROMELA. Les problèmes étudiées dans la Section 10.3.5 (page 142) sont également modélisées par des sections atomiques. Le modèle de STM-HRT est très fidèle aux algorithmes, mais ne modélise pas les lectures et écritures effectives des objets.

Les exigences testées sont les suivantes :

- Une transaction de lecture peut échouer une seule fois ;
- Une transaction d'écriture ne peut jamais échouer ;
- Tous les objets ouverts en écriture sont mis à jour ;
- Tous les objets ouverts en lecture sont pointés par la table des objets ouverts en lecture ;
- L'échec d'une transaction de lecture conduit bien à la mise à jour des *FAIL_ VECTOR* correspondant aux objets manipulés ;
- Le progrès de toutes les transactions est garanti ;
- La mise à jour et le nettoyage des structures de données sont correctement effectués lors du succès d'une transaction ;

– Aucune transaction ne peut être en famine.

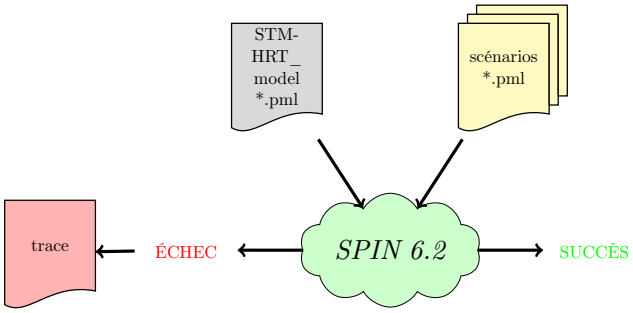


Figure 11.1 – Vérification du protocole STM-HRT avec *SPIN*

Le protocole de test est décrit par la Figure 11.1. Nous faisons abstraction de la chaîne de compilation complète faisant intervenir gcc. L'implémentation du protocole est réalisée dans le fichier *STM-HRT_ model.pml*. Ce fichier comprend le modèle du protocole et les règles à vérifier. Il est figé pendant toute la durée des tests.

Nous avons également établi des scénarios. Chaque scénario est écrit dans un fichier PROMELA dédié et met en compétition des transactions de lecture et d'écriture sur un ensemble d'objets et un ensemble de cœurs. En cas d'échec pour un scénario donné, la trace ayant conduit à la violation de la propriété est générée dans le fichier *Trace*.

Les scénarios utilisés sont donnés dans la Table 11.1. Nous considérons qu'au plus 4 transactions ($\Omega = \{\omega_0,\ \omega_1,\ \omega_2,\ \omega_3\}$) peuvent être actives en même temps (i.e. 4 cœurs dans le système), et au plus 3 données partagées ($R = \{r_0,\ r_1,\ r_2\}$) sont manipulées. Toutes les transactions sont exécutées deux fois. Pour chacune des données r_j, nous associons un objet noté o_j. Notons que les scénarios sont construits pour respecter l'hypothèse mono-écrivain/multi-lecteur.

La Table 11.2 fournit des détails sur les résultats des analyses. La colonne liée au nombre d'états correspond à la taille de l'espace d'état exploré par SPIN. La quantité de mémoire RAM nécessaire pour les vérifications est très importante. Avec la machine utilisées (64 Go de RAM), on ne peut pas conclure sur l'exhaustivité des états explorés pour le test n°7. D'autres analyses seront menées sur une machine possédant plus de mémoire.

11.2 Analyse fonctionnelle de STM-HRT

Dans cette section, nous formalisons les preuves du fonctionnement du protocole STM-HRT.

Avec les deux premiers lemmes, nous démontrons qu'une transaction en écriture ne peut jamais échouer tandis qu'une transaction de lecture peut échouer et redémarrer au plus une fois, peu importe le nombre de conflits.

Lemme 11.1. *Les transactions homogènes d'écriture n'échouent jamais dans STM-HRT.*

Preuve : STM-HRT est un protocole de contrôle de concurrence 1 : m. Par conception, une seule transaction à la fois peut mettre à jour un objet. Cela implique que des conflits

Scénario de test	Transactions mises en jeu	Données r_0	r_1	r_2
Test n° 1 – 1 lecteur / 1 écrivain				
ω_0	Transaction en lecture	X	X	X
ω_1	Transaction en écriture	X	X	X
Test n° 2 – 1 lecteur / 2 écrivains				
ω_0	Transaction en lecture	X	X	–
ω_1	Transaction en écriture	X	–	–
ω_2	Transaction en écriture	–	X	–
Test n° 3 – 1 lecteur / 2 écrivains				
ω_0	Transaction en lecture	X	X	X
ω_1	Transaction en écriture	X	–	–
ω_2	Transaction en écriture	–	X	X
Test n° 4 – 2 lecteurs / 1 écrivain				
ω_0	Transaction en lecture	X	–	–
ω_1	Transaction en écriture	X	–	–
ω_2	Transaction en lecture	X	–	–
Test n° 5 – 2 lecteurs / 1 écrivain				
ω_0	Transaction en lecture	X	X	–
ω_1	Transaction en écriture	X	X	X
ω_2	Transaction en lecture	X	–	X
Test n° 6 – 2 lecteurs / 2 écrivains				
ω_0	Transaction en lecture	X	X	–
ω_1	Transaction en écriture	X	–	–
ω_2	Transaction en écriture	–	X	–
ω_3	Transaction en lecture	X	X	–
Test n° 7 – 2 lecteurs / 2 écrivains				
ω_0	Transaction en lecture	X	–	X
ω_1	Transaction en écriture	X	–	X
ω_2	Transaction en écriture	–	X	–
ω_3	Transaction en lecture	X	X	–

Table 11.1 – Scénarios de test du protocole STM-HRT avec *SPIN*

écrivain/écrivain ne peuvent jamais se produire. Le succès d'une transaction en écriture est alors toujours garanti.

Lemme 11.2. *Les transactions homogènes de lecture peuvent échouer au plus une fois dans STM-HRT.*

Preuve : Des conflits lecture/écriture se produisent dès lors qu'une transaction en écriture met à jour un objet entre l'instant où une transaction de lecture ouvre l'objet et l'instant où elle effectue son commit. Dans le cas d'un échec, la transaction de lecture remplit les *FAIL_ VECTOR* de tous les objets avant de les ouvrir pour la seconde fois. La mise à jour de

Scénario de test	**Nombre d'états analysés**	**Conclusion**
Test n° 1	$5,25 * 10^8$	OK
Test n° 2	$5,25 * 10^8$	OK
Test n° 3	$6,18 * 10^9$	OK
Test n° 4	$5,1 * 10^8$	OK
Test n° 5	$2,32 * 10^{10}$	OK
Test n° 6	$5,1 * 10^{11}$	OK
Test n° 7	out of memory	?

Table 11.2 – Résultats des tests du protocole STM-HRT avec *SPIN*

ce vecteur permet de « verrouiller » l'objet pour éviter des conflits futurs. En effet, dès qu'une transaction en écriture voudra mettre à jour l'objet, l'opération atomique permettant de réaliser cette action échouera tant que le *FAIL_ VECTOR* sera différent de 0, c'est-à-dire, tant qu'il existera au moins une transaction de lecture ayant échoué une fois et n'ayant pas encore réussi. En pratique, une transaction en écriture qui ne parvient pas à mettre à jour l'objet à cause du *FAIL_ VECTOR* aide les transactions de lecture avec lesquelles elle est en conflit. Pour aider, la transaction en écriture exécute au nom de la transaction à aider, les algorithmes *read_ obj()* (pour tous ses objets) et *commit_ read_ stm_ tx()*. Par conséquent, une transaction de lecture ne peut jamais échouer plus d'une fois.

STM-HRT garantit également l'absence de famine des écrivains et prévient les aides cycliques. Ce contrôle est réalisé par la maîtrise des politiques d'aide utilisées dans le protocole.

Lemme 11.3. *Aucune famine n'est possible dans STM-HRT.*

Preuve : Lorsqu'une transaction en écriture aide d'autres transactions avec lesquelles elle est en conflit, une infinité d'échecs successifs de transactions de lecture conduirait à une famine de l'écrivain. Pour éviter ce problème, lorsqu'une transaction de lecture en première tentative ouvre un objet, elle commence par aider celles d'écriture qui sont sur le point de mettre à jour l'objet. Ceci permet de prévenir un conflit qui aurait pu se produire dans le futur. Cette aide est effectuée avant l'ouverture d'un objet. Pour illustrer le mécanisme, considérons la Figure 11.2 sur laquelle nous supposons qu'un seul objet est conflictuel. Une transaction en écriture s'exécutant sur le cœur π_3 aide deux transactions de lecture qui s'exécutent sur les cœurs π_2 et π_1 (les transactions de lecture sur π_1 et π_2 ont échoué pour une cause passée qui n'est pas représentée sur la figure). Après avoir été aidée, une nouvelle transaction de lecture sur le cœur π_2 commence par aider la transaction en écriture du cœur π_3 avant d'ouvrir l'objet. Cette aide assure le succès de la transaction en écriture et une famine n'est donc pas possible.

Lemme 11.4. *Aucune aide cyclique ne peut avoir lieu dans STM-HRT.*

Preuve : Commençons par illustrer l'aide cyclique en considérant deux transactions : ω_i est une transaction de lecture et ω_j est une transaction en écriture. Nous supposons que ω_i échoue

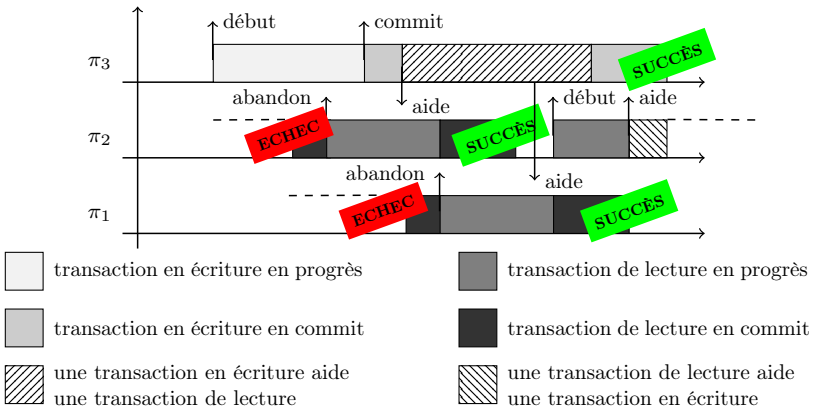


Figure 11.2 – Garantie de progression wait-free et absence de famine

et redémarre. ω_j aide alors ω_i avant de mettre à jour l'objet conflictuel. Si ω_i était autorisée à aider une transaction en écriture pendant son second essai, ω_i aiderait alors ω_j, qui elle-même est déjà en train d'aider ω_i. Pour éviter ce phénomène d'aide cyclique, une transaction de lecture ne peut aider une transaction en écriture que pendant sa première tentative. Ceci est illustré sur le Figure 11.2.

Théorème 11.1. *STM-HRT est un protocole ayant une garantie de progression wait-free.*

Preuve : À partir des lemmes 11.3 et 11.4, ni famine, ni aide cyclique ne peuvent survenir. Dès qu'une transaction commence à s'exécuter, elle va systématiquement se terminer. À partir des lemmes 11.1 et 11.2, nous garantissons également que le nombre de redémarrages des transactions est borné. Par conséquent, toutes les transactions se terminent et réussissent en un nombre fini d'étapes. La garantie de progression est donc wait-free.

Théorème 11.2. *STM-HRT assure la progression même dans le cas d'une défaillance d'une tâche ou d'un cœur.*

Preuve : À partir du théorème 11.1, la garantie de progression de STM-HRT est prouvée wait-free. Néanmoins, pour illustrer le problème pouvant se produire dans le cas de la défaillance d'un cœur ou d'une tâche, considérons deux transactions : ω_i est une transaction de lecture et ω_j est une transaction en écriture. Supposons également que ω_i échoue et redémarre. Avant d'ouvrir à nouveau l'objet, ω_i remplit le *FAIL_ VECTOR* de l'objet pour se prémunir d'une mise à jour intempestive. En parallèle, ω_j aide ω_i au moment de son commit. Si le cœur sur lequel la tâche qui exécute ω_i subit une défaillance, ω_i ne terminera jamais et le *FAIL_ VECTOR* de l'objet requis par ω_j ne sera jamais réinitialisé. Par conséquent, ω_j ne pourra plus progresser à son tour. STM-HRT permet d'éviter ce phénomène en autorisant ω_j à réinitialiser le *FAIL_ VECTOR* à la fin de l'aide. Plus généralement, toutes les opérations réalisées dans le protocole sont indépendantes des actions réalisées sur les cœurs voisins. Une défaillance ayant eu lieu sur un cœur n'est alors pas propagée.

Théorème 11.3. *L'aide est toujours consistante dans STM-HRT.*

Preuve : Le problème d'instance a déjà été présenté dans la Section 10.3.5 (page 142). Ce problème survient lorsqu'une transaction de lecture qui est aidée se termine par elle-même (et qu'une autre démarre sur le même cœur) alors que l'aide a toujours lieu. Le risque est alors de corrompre les structures de données. Pour prévenir les corruptions, on vérifie que (1) l'instance de la transaction active sur le cœur est toujours égale à l'instance de la transaction que l'on souhaite aider et (2) que la valeur du statut est conforme à la valeur attendue avant de mettre à jour un élément critique des structures de données.

11.3 Analyse temporelle de STM-HRT

Dans cette section, nous utilisons les notations introduites dans la Table 10.1. Sans perte de généralités, nous supposons dans cette analyse que toutes les ressources ont la même taille et que le temps d'initialisation des transactions est négligeable devant le temps nécessaire à leur déroulement.

11.3.1 Surcoût d'exécution des transactions en totale isolation

Dans un premier temps, nous introduisons les définitions relatives à l'exécution des transactions en totale isolation. Nous rappelons qu'une transaction est exécutée en totale isolation quand elle évolue sans conflits.

Définition 11.1. *Une transaction homogène d'écriture $\omega_k \in \Omega^W$ exécutée en totale isolation a une durée d'exécution égale à :*

$$e^W_{\omega_k} = |O^{\omega_k}| * (T\_open^W_{o_j} + T\_commit^W_{o_j}) \tag{11.1}$$

$T\_open^W_{o_j}$ et $T\_commit^W_{o_j}$ correspondent respectivement au temps requis pour ouvrir un objet o_j en écriture et le temps nécessaire pour le mettre à jour quand ω_k est exécutée en totale isolation.

Définition 11.2. *Une transaction homogène de lecture $\omega_k \in \Omega^R$ exécutée en totale isolation a une durée d'exécution égale à :*

$$e^R_{\omega_k} = |O^{\omega_k}| * T\_open^R_{o_j} + T\_commit^R_{\omega_k} \tag{11.2}$$

$T\_open^R_{o_j}$ représente le temps nécessaire à l'ouverture d'un objet o_j en lecture quand ω_k est exécutée en totale isolation. $T\_commit^R_{\omega_k}$ représente le temps nécessaire pour le commit de la transaction ω_k. Même si ce paramètre est dépendant du nombre d'objets ouverts ($|O^{\omega_k}|$), la construction $1 : m$ du protocole STM-HRT rend inutile la distinction du commit d'un seul objet o_j.

11.3.2 Surcoût d'exécution des transactions en concurrence

Pour définir une borne supérieure du surcoût d'exécution induit par le protocole de synchronisation STM-HRT sur une tâche τ_i, nous devons trouver une borne supérieure sur le temps nécessaire à l'exécution de chacune des transactions de la tâche. Les Lemmes 11.5 et 11.6 nous donnent au pire-cas la durée maximale d'une transaction en écriture et de lecture respectivement. Pour définir ces bornes supérieures, nous prenons en compte aussi bien l'échec des transactions de lecture que le temps additionnel dû aux différentes aides.

Lemme 11.5. *Toutes les transactions homogènes d'écriture $\omega_k \in \Omega^W$ subissent un surcoût maximal égal à :*

$$T\_overhead^W_{\omega_k} = \sum_{j=1}^{|O^{\omega_k}|} \left(T\_helping^R_{o_j} + T\_attempt\_commit^W_{o_j} \right) \tag{11.3}$$

$T\_helping^R_{o_j}$ est le surcoût subi par ω_k losqu'elle aide des transactions de lecture d'autres cœurs ayant déjà échoué une fois, et manipulant au moins un objet $o_j \in O^{\omega_k}$. La transaction homogène d'écriture aide une transaction de lecture dans sa totalité. $T\_helping^R_{o_j}$ est défini par :

$$T\_helping^R_{o_j} = \sum_{\pi_i \in \Pi(o_j) \setminus \{\pi(\omega_k)\}} \max_{\substack{\omega_m \in \Omega^R_{\pi_i} \\ s.t.\ o_j \in O^{\omega_m}}} e^R_{\omega_m} \tag{11.4}$$

$T\_attempt\_commit^W_{o_j}$ est le temps nécessaire à ω_k pour chaque tentative de mise à jour de l'objet.

$$T\_attempt\_commit^W_{o_j} = \sum_{\pi_i \in \Pi(o_j) \setminus \{\pi(\omega_k)\}} T\_commit^W_{o_j} \tag{11.5}$$

Preuve : Lorsqu'une transaction en écriture ω_k n'est pas autorisée à mettre à jour un objet dans sa phase de commit, le *FAIL_ VECTOR* de cet objet est différent de 0 et de l'aide est requise. ω_k aide alors toutes les transactions de lecture des autres cœurs, qui accèdent à cet objet. Au plus, cette aide sera effectuée pour $(M-1)$ cœurs. Comme illustré par l'equation 11.4, une seule des transactions de lecture est aidée pour un cœur donné. Dans le pire-cas, nous considérons que c'est toujours la plus longue transaction qui est aidée. L'équation 11.5 représente les essais effectués par ω_k pour mettre à jour l'objet. Dans le pire cas, ω_k peut tenter de mettre à jour un objet donné $(M-1)$ fois avant de réussir. Enfin, puisque ω_k peut rentrer en conflit avec des transactions de lecture pour chacun de ses objets accédés (i.e. pour $|O^{\omega_k}|$ objets), l'aide peut être répétée pour les $|O^{\omega_k}|$ objets manipulés par ω_k. Ceci est illustré par la somme présente dans l'Équation 11.3.

Lemme 11.6. *Toutes les transactions homogènes de lecture $\omega_k \in \Omega^R$ subissent un surcoût maximal égale à :*

$$T\_overhead^R_{\omega_k} = \underbrace{\sum_{j=1}^{|O^{\omega_k}|} T\_helping^W_{o_j}}_{\text{première tentative}} + \underbrace{e^R_{\omega_{k,retry}}}_{\text{seconde tentative}} \tag{11.6}$$

$T\_helping^W_{o_j}$ est le surcoût subi lorsque ω_k aide une transaction en écriture s'exécutant sur le cœur $\pi(o_j) \neq \pi(\omega_k)$. La transaction de lecture aide une transaction en écriture uniquement pour l'objet conflictuel. $T\_helping^W_{o_j}$ est défini par :

$$T\_helping^W_{o_j} = \begin{cases} 0 & si\ \pi(o_j) = \pi(\omega_k), \\ \left(T\_helping^R_{o_j} + T\_attempt\_commit^W_{o_j}\right) & sinon. \end{cases} \tag{11.7}$$

$e^R_{\omega_{k,retry}}$ est le temps nécessaire pour rejouer la transaction de lecture.

$$e^R_{\omega_{k,retry}} \begin{cases} 0 & si\ \forall o_j \in O^{\omega_k},\ \pi(o_j) = \pi(\omega_k) \\ e^R_{\omega_k} & sinon. \end{cases} \tag{11.8}$$

Preuve : Une transaction de lecture ω_k peut subir deux types de surcoût (Équation 11.6). Lors de la première tentative, ω_k peut aider des transactions d'écriture à mettre à jour un objet. Ceci n'est possible que si l'objet manipulé par ω_k peut être mis à jour par une transaction en écriture localisée sur un autre coeur (Équation 11.7). Lorsque ω_k recommence, elle ne peut plus effectuer ce type d'aide. Par conséquent, lors de la seconde tentative, nous devons uniquement considérer le temps nécessaire à l'exécution de ω_k en totale isolation (Équation 11.8).

Lorsque ω_k ouvre un objet, l'aide ne peut pas être effectuée si la transaction qui peut mettre à jour l'objet est sur le même coeur (partie haute de l'équation 11.7). Si cela n'est pas le cas (partie basse de l'équation 11.7), nous devons considérer le fait que la transaction en écriture aidée par ω_k, peut également aider au plus (M-2) autres transactions de lecture (nous rappelons que l'aide cyclique n'est pas autorisée). Ce phénomène est déjà pris en compte dans les Équations 11.4 et 11.5.

11.3.3 Borne supérieure sur le temps d'exécution des tâches

Il nous est à présent possible de quantifier le surcoût d'exécution de chaque tâche exécutant des transactions de lecture et des transactions d'écriture.

Lemme 11.7. *Chaque tâche τ_i dans STM-HRT exécutant $|\Omega^R_{\tau_i}|$ transactions homogènes de lecture et $|\Omega^W_{\tau_i}|$ transactions homogènes d'écriture subit le surcoût suivant :*

$$overhead_i = \sum_{j=1}^{|\Omega^R_{\tau_i}|} T\_overhead^R_{\omega_j} + \sum_{j=1}^{|\Omega^W_{\tau_i}|} T\_overhead^W_{\omega_j} \tag{11.9}$$

Preuve : Une tâche exécute $|\Omega^R_{\tau_i}|$ transactions de lecture et $|\Omega^W_{\tau_i}|$ transactions d'écriture. Le surcoût d'exécution total pour toutes les transactions correspond donc à la somme des surcoûts individuels de chacune des transactions.

Ce lemme conduit au résultat que nous établissons dans le Théorème 11.4.

Théorème 11.4. *Une borne supérieure du pire temps d'exécution d'une tâche τ_i dans le contexte de STM-HRT est donnée par :*

$$C_i^{STM-HRT} = C_i + overhead_i \tag{11.10}$$

Preuve : Le pire temps d'exécution d'une tâche dépend du coût induit par le protocole STM-HRT. Le C_i initial obtenu lorsque les transactions sont exécutées en totale isolation doit être augmenté du pire temps d'exécution des transactions de lecture et d'écriture (i.e. scénario de conflits pire-cas décrit dans le Lemme 11.7).

L'analyse temporelle nous permet bien de déduire que le coût du protocole est indépendant de la politique d'ordonnancement choisie et peut être directement inclus dans le pire temps d'exécution des tâches. Ce résultat est intéressant dans le sens où il permet de simplifier l'analyse d'ordonnançabilité du système.

11.4 Analyse de l'empreinte mémoire de STM-HRT

Dans cette section, nous évaluons l'empreinte mémoire de STM-HRT. Nous utilisons les notations définies dans la Table 10.1. Nous considérons les hypothèses suivantes :

- Un pointeur a une taille de 4 octets ;
- Tous les cœurs du système partagent toutes les données ;
- Au plus 32 objets dans le système ;
- Au plus 15 cœurs dans le système ;
- La taille du code des algorithmes de contrôle de concurrence n'est pas prise en compte dans l'analyse. Seules les structures de données nécessaires au fonctionnement de STM-HRT sont considérées.

11.4.1 Espace mémoire dédié à la gestion des objets

La gestion des objets nécessite de disposer de plusieurs tables. Dans les Définitions 11.3 et 11.4, nous présentons respectivement la taille de la table des objets et la taille de la table des écrivains. La taille de ces tables est directement liée au nombre d'objets manipulés (ρ).

Définition 11.3. *L'espace mémoire nécessaire pour stocker la table des adresses de tous les en-têtes des objets manipulés dans le système est donné par :*

$$Size_{objectTable} = \rho * 4 \; octets \tag{11.11}$$

Définition 11.4. *L'espace mémoire nécessaire pour stocker la table des identifiants des cœurs sur lesquels sont allouées les tâches pouvant modifier les données est donnée par :*

$$Size_{writerTable} = \rho \; octets \tag{11.12}$$

Les Lemmes 11.8 et 11.9 permettent de déterminer respectivement l'espace mémoire nécessaire au stockage des en-têtes d'objets et celui associé à la table des copies. Nous noterons *sizeOf(o_j)*, la taille en octets de l'objet o_j.

Lemme 11.8. *L'espace mémoire nécessaire pour stocker les ρ en-têtes d'objets, sans les tables de copies est donné par :*

$$Size_{dataObject} = 9 * \rho \ octets \tag{11.13}$$

Preuve : *L'identifiant de l'objet* prend 1 octet, le *pointeur vers la table de copies* en prend 4 et le *concurrency_ vector* également.

Lemme 11.9. *L'espace mémoire nécessaire pour stocker les ρ tables de copies est donné par :*

$$Size_{copyTable} = \sum_{r_i \in R} sizeOf(o_i) * \left(2 + \sum_{\pi_j \in \Pi} \begin{cases} 1 & si \ \exists \ \omega_k \in \Omega_{\pi_j} \mid o_i \in O^{\omega_k} \\ 0 & sinon \end{cases} \right) \ octets \tag{11.14}$$

Preuve : Le tableau de copies comprend systématiquement un emplacement pour l'objet concurrent et un autre pour la copie lors de la mise à jour. Les autres copies sont réservées aux transactions qui viennent lire l'objet. Tous les cœurs sur lesquels au moins une transaction accède à l'objet doivent posséder leur emplacement.

Théorème 11.5. *L'espace mémoire nécessaire pour stocker les structures de données relatives aux objets et aux copies est donné par :*

$$Size_{objects} = Size_{objectTable} + Size_{writerTable} + Size_{dataObject} + Size_{copyTable} \tag{11.15}$$

Preuve : À partir des lemmes 11.8 et 11.9, nous obtenons l'espace mémoire nécessaire à la gestion des objets en appliquant une simple somme.

11.4.2 Espace mémoire dédié à la gestion des transactions

Dans la Définition 11.5, nous donnons la taille de la table des transactions. La taille de cette table dépend du nombre de cœurs présents dans le système.

Définition 11.5. *L'espace mémoire nécessaire pour stocker la table des adresses de tous les descripteurs de transaction du système est donnée par :*

$$Size_{transTable} = M * 4 \ octets \tag{11.16}$$

Le lemme 11.10 permet de déterminer l'espace mémoire requis pour stocker les M descripteurs de transaction.

Lemme 11.10. *L'espace mémoire nécessaire pour stocker les M descripteurs de transaction, sans les tables d'accès en lecture et écriture est donné par :*

$$Size_{transDesc} = (10 + \left\lceil \frac{\rho}{8} \right\rceil) * M \ octets \tag{11.17}$$

Preuve : *l'identifiant du cœur associé au descripteur* prend 1 octet, le *statut et l'instance* sont stockés sur 1 octet, les *pointeurs vers les tables d'accès en lecture et écriture* prennent 4 octets chacun et la taille du *vecteur d'accès* prend 1, 2, 3 ou 4 octets en fonction du nombre d'objets dans le système.

Les lemmes 11.11 et 11.12 permettent de déterminer les tailles du tableau des accès en lecture et du tableau des accès en écriture.

Lemme 11.11. *L'espace mémoire nécessaire pour stocker la table des accès en lecture pour les M descripteurs de transaction est donné par :*

$$Size_{readsetTable} = \rho * 4 * M \ octets \tag{11.18}$$

Lemme 11.12. *L'espace mémoire nécessaire pour stocker la table des accès en écriture pour les M descripteurs de transaction est donné par :*

$$Size_{writesetTable} = \rho * 4 * M \ octets \tag{11.19}$$

Preuve : Pour des raisons de simplicité, les tables d'accès en lecture et écriture possèdent des entrées pour tous les objets du système.

Théorème 11.6. *L'espace mémoire nécessaire pour stocker les structures de données relatives aux transactions est donné par :*

$$Size_{trans} = Size_{transTable} + Size_{transDesc} + Size_{readsetTable} + Size_{writesetTable} \tag{11.20}$$

Preuve : À partir des lemmes 11.11, 11.12, nous obtenons l'espace mémoire nécessaire à la gestion des transactions en appliquant une simple somme.

11.4.3 Bilan de l'empreinte mémoire de STM-HRT

Le Théorème 11.7 est le bilan de l'empreinte de STM-HRT.

Théorème 11.7. *L'empreinte mémoire de STM-HRT est donnée par :*

$$Size_{STM-HRT} = Size_{objects} + Size_{trans} \tag{11.21}$$

Preuve : L'empreinte mémoire totale de STM-HRT est la somme des résultats obtenus dans les théorèmes 11.5 et11.6.

11.4.4 Exemple

Considérons les hypothèses suivantes :
- $M = 4$ (4 descripteurs de transaction) ;
- $\rho = 32$ (32 en-têtes d'objets) ;
- Toutes les données ont une taille de 4 octets ;
- Les données sont accessibles par tous les cœurs.

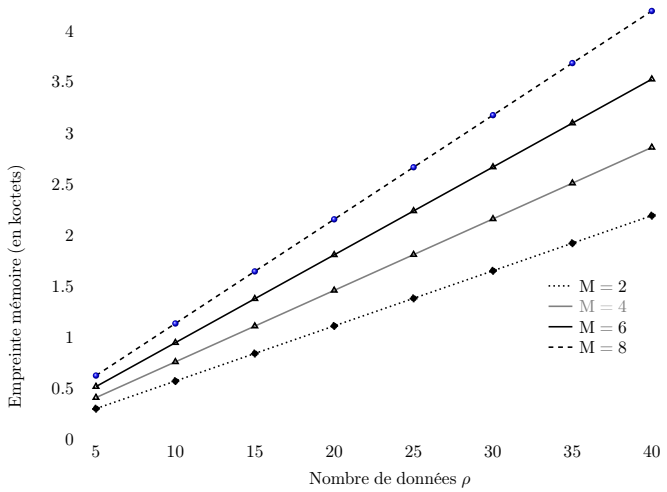


Figure 11.3 – Évolution de l'empreinte mémoire en fonction de ρ et M

Nous déduisons alors que $Size_{objects} = 38 * \rho = 1216$ *octets* et que $Size_{trans} = 14 * M + 8 * M * \rho = 1080$ *octets*. La taille des structures de données est alors de $2,296$ *ko*.

La Figure 11.3 montre l'évolution de l'empreinte mémoire lorsque l'on fait varier M et ρ. Nous constatons qu'à nombre de cœurs constant, l'évolution de l'empreinte mémoire est proportionnelle au nombre de données. On constate également que plus le nombre de cœurs augmente, plus la pente des droites augmente.

L'empreinte mémoire est acceptable et comparable à celle que l'on obtiendrait avec une implémentation de l'IOC d'AUTOSAR. En effet, le facteur dimensionnant de l'empreinte mémoire est le nombre de copies. Dans AUTOSAR, chaque destinataire d'un message possède également sa propre copie.

Il est possible d'optimiser la taille des tables d'accès en lecture et écriture en analysant quels cœurs (au travers des tâches et transactions) peuvent avoir accès à chaque données. En particulier, sachant qu'un objet ne peut être mis à jour que sur un cœur, nous pourrions déjà gagner 512 octets lorsque $\rho = 32$ et $M = 4$.

Chapitre 12

Discussion

L'utilisation d'une architecture multicoeur impose de contrôler les accès parallèles aux données partagées. Si le système est temps réel, le protocole de contrôle doit tenir compte de cette dimension. Des protocoles basés sur l'utilisation de verrous ont été proposés sur la base de ceux déjà existants en monocœur. Bien que ces protocoles permettent d'éliminer l'occurrence d'interblocage ou de famine, leur utilisation pour protéger des sections critiques imbriquées est complexe et constitue une source possible de fautes lors des phases de codage et d'intégration.

Les verrous de type spinlock sont également un canal de propagation d'erreurs. Précisément, si une erreur conduit un coeur à ne pas relâcher un spinlock, cette erreur se propage vers les autres coeurs qui vont se bloquer en attente de la libération du spinlock.

Pour faciliter la mise en œuvre du partage des données et réduire la dépendance des cœurs à ce niveau, nous proposons une solution alternative. Dans ce but, nous avons présenté STM-HRT, un protocole non bloquant qui trouve son origine d'une part dans les travaux sur les mécanismes wait-free, et d'autre par dans les travaux sur les mémoires transactionnelles. Les accès aux données partagées sont encapsulés dans des transactions et le protocole (vu comme un service du système d'exploitation) gère seul les accès et les conflits.

STM-HRT est un mécanisme wait-free à part entière (progression de toutes les transactions). De plus, puisque le nombre d'abandons est borné, il permet de respecter des contraintes temps réel dur.

Afin de doter STM-HRT de garanties de progression wait-free, nous avons dû faire des hypothèses simplificatrices, voire restrictives vis-à-vis des possibilités offertes habituellement par les mémoires transactionnelles. En effet, nous considérons des transactions homogènes, dans un contexte mono-écrivain. En ce sens, STM-HRT offre un sous-ensemble des fonctionnalités usuellement fournies par les mémoires transactionnelles. Nos hypothèses sont en revanche cohérentes avec les stratégies de développement des systèmes embarqués automobiles. Parmi les solutions reposant sur les mêmes hypothèses, nous pouvons nous comparer à l'algorithme de Chen. STM-HRT est plus complet puisqu'il bénéficie du contexte transactionnel pour protéger une section critique au sein de laquelle on accède à un groupe de données, là ou l'algorithme de Chen ne maintient la cohérence que d'une donnée à la fois.

Le protocole mis en place a été modélisé et vérifié partiellement avec l'outil de vérification de modèles SPIN. Au vu de l'importance de l'espace d'états, certains plans de tests n'ont pas pu être vérifiés, mais l'utilisation d'une machine avec plus de mémoire permettra de poursuivre cette vérification.

SPIN ne permettant pas de modéliser la progression des coeurs, deux points difficiles identifiés lors de la conception du protocole ne sont pas couverts par ces tests. Il s'agit d'une part de la famine possible en cas de coallition d'écrivains contre un lecteur (c.f. Section 10.3.3.2, page 139) et d'autre part du problème de détection de la transaction en cours par son numéro d'instance dans le cas de l'aide d'une transaction de lecture (c.f. Section 10.3.5, page 142). Nous avons développé un argument *manuel* pour ces problèmes, basé sur l'étude de l'architecture matérielle du microcontrôleur Leopard - MPC5643L. Une meilleure approche serait d'utiliser des outils de vérification offrant le pouvoir de modélisation nécessaire, tels que UPPAAL Cassez et Béchennec (2013).

Une analyse du coût temporel et de l'empreinte mémoire associés au protocole à été établie.

L'analyse temporelle nous a permis de montrer son impact pire-cas sur le temps d'exécution des tâches comportant des transactions. Nous avons mis en évidence que le coût temporel est directement intégré au pire temps d'exécution de la tâche, ce qui facilite l'analyse d'ordonnançabilité.

L'empreinte mémoire a également été évaluée. Nous remarquons qu'elle est principalement dimensionnée par le nombre de copies nécessaires à chaque donnée partagée. Dans AUTOSAR, une copie est également disponible pour tous les lecteurs. Notre solution est donc comparable aux solutions existantes. De plus, des stratégies d'optimisation basées sur l'analyse statique de l'application sont envisageables pour réduire l'empreinte mémoire.

Pour valider notre approche, une étude de cas sur une application multicœur et une approche par injection de faute nous permettra d'évaluer la robustesse du protocole sur une cible.

Conclusions et perspectives

Le paysage des systèmes embarqués automobiles est en perpétuel changement. La tendance actuelle est à l'utilisation de microcontrôleurs multicœur, comme un moyen pour augmenter la quantité de ressources CPU disponibles par puce, et comme un outil d'aide à la sûreté de fonctionnement. L'introduction de ces architectures nouvelles bouscule les processus classiques de développement et apporte un lot de problèmes qu'il faut connaître et maîtriser. En particulier, on souhaite que l'application développée utilise au mieux les ressources CPU disponibles et que toutes les exigences en termes de sûreté de fonctionnement soient respectées. Le domaine de l'automobile est largement standardisé. Nos travaux s'inscrivent dans la contexte du standard AUTOSAR (élargi aux architectures multicœur depuis la révision 4.0) et de la norme ISO 26262.

Dans ce contexte, nos travaux visent à contribuer à la robustesse des applications s'exécutant sur ces architectures. Après l'élaboration du modèle de fautes induit par la coopération des cœurs parallèles, nous avons ciblé deux axes d'étude :

- Permettre la vérification en ligne de propriétés inter-tâche ;
- Proposer une solution sûre et robuste au problème du partage de données dans les systèmes temps réel durs implantés sur une architecture multicœur à mémoire commune.

Pour répondre à la première problématique, nous avons proposé un service de vérification en ligne intégré dans le noyau d'un système d'exploitation compatible avec le standard AUTOSAR. Les principes théoriques ont été posés et nous ont permis de construire l'outil *Enforcer*. Il permet de générer hors-ligne des moniteurs à partir des propriétés à vérifier (dérivées des exigences fonctionnelles) écrites en logique temporelle LTL et d'un modèle de l'application à surveiller. Il permet également de générer les sondes qui permettent d'intercepter et analyser les évènements du système nécessaire à la vérification.Les moniteurs et les sondes sont utilisés pour vérifier en ligne le comportement du système.

Pour valider le concept, le service de vérification en ligne a été intégré à la chaîne de développement du système d'exploitation *Trampoline*, conforme AUTOSAR. Des tests sur cible nous ont permis de déterminer le coût temporel de la surveillance par rapport aux services de base du noyau. Une étude de cas sur un projet automobile prototype nous a également permis d'intégrer notre solution dans le processus de développement industriel et de mesurer le coût total à l'échelle d'un calculateur, en termes d'empreinte mémoire. Les résultats obtenus démontrent que le ratio impact/gain est tout à fait acceptable et raisonnable vis-à-vis des contraintes dans les systèmes embarqués temps réel automobile et encourage l'utilisation d'un

tel service.

Pour répondre à la seconde problématique, nous avons étudié une approche alternative au protocole de synchronisation bloquant spécifié dans AUTOSAR multicœur pour le partage des données inter-cœurs. Cette approche vise à simplifier la mise en place de la synchronisation et à réduire les fautes de développement ayant lieu pendant le codage. L'alternative proposée s'inspire d'un état de l'art sur les mécanismes à mémoire transactionnelle et sur les mécanismes wait-free, permettant la progression de toutes les tâches de l'application. Des propriétés sur le protocole mis au point ont été partiellement vérifiées à l'aide de l'outil de vérification de modèles SPIN et des éléments de preuves ont également été apportés pour justifier le bon fonctionnement du mécanisme.

La solution a été évaluée en termes de coût temporel par une analyse du protocole. Cette analyse montre que la prise en compte de STM-HRT dans le cadre d'une vérification d'ordonnançabilité revient à intégrer un surcoût dans les pire temps d'exécution des tâches. Cela permet alors l'utilisation de techniques de vérification d'ordonnançabilité simples et bien maîtrisées, ce qui contribue à l'amélioration de la sûreté de fonctionnement.

Une étude de l'empreinte mémoire a également permis de vérifier son adéquation avec le secteur industriel automobile visé. Une implémentation est proposée pour *Trampoline*. Elle permet de s'interfacer avec les APIs de la couche de communication IOC proposée par AUTOSAR.

Perspectives

Enforcer permet actuellement de générer du code pour *Trampoline* qui ne supporte pas à ce jour d'architecture multicoeur. Dès que ce sera le cas, il sera possible d'étudier le portage du service de vérification en ligne. Un problème particulier que nous avons identifié est celui de l'adéquation entre les modèles utilisés pour la vérification et la réalité. Ainsi, des événements produits simultanément par l'application seront pris en compte séquentiellement par les moniteurs, dans un ordre indéterminé. Des événements produits à des instants très proches pourraient être pris en compte dans l'ordre inverse de leurs occurrence. Il faut donc veiller à ce que le système ne réagisse pas à des faux positifs. Ce problème doit être traité conjointement hors ligne et en ligne, en particulier au niveau de la prise en compte des informations produites par le service.

Le service de vérification en ligne utilise des moniteurs qui réagissent sur des évènements discrets. Étant donné notre contexte applicatif, une extension naturelle serait d'offrir la possibilité de vérifier de propriétés temps réel. Cette possibilité viendrait compléter le service de protection temporelle intégré à AUTOSAR. Les travaux existant dans ce domaine, par exemple ceux de Robert *et al.* (2010), montrent les difficultés de la vérification en ligne de propriétés temps réel. En effet, la surveillance de telles propriétés nécessite de stocker des moniteurs modélisant plusieurs futurs possibles, et c'est l'écoulement du temps qui permet d'éliminer les chemins non pris. Des solutions sont encore à trouver pour limiter la taille de ces moniteurs et permettre leur utilisation dans un contexte embarqué..

Au niveau du protocole STM-HRT, les preuves de concept et les analyses nous permettent déjà de disposer d'informations validant son fonctionnement. Pour valider nos hypothèses d'implémentation liées aux problèmes exprimés dans les Sections 10.3.3.2 (page 139) et 10.3.5 (page 142), nous avons analysé l'architecture matérielle du Leopard (MPC 5643L). Il convient désormais de faire une vérification avec un outil de type UPPAAL, prenant en compte les aspects temporels pour modéliser la progression des cœurs. Une étude pratique de STM-HRT est éga-

lement à prévoir pour évaluer le coût temporel sur un cas d'étude concret. Cela permettra dans un premier temps de comparer le protocole avec les autres protocoles existants pour les systèmes multicœur temps réel sur le plan de l'ordonnançabilité des systèmes. Une étude via injection de fautes permettra également d'en évaluer la robustesse.

Annexe A

Le RTOS Trampoline

Trampoline (Béchennec *et al.*, 2006) est un système d'exploitation temps réel (RTOS) conçu conformément aux spécifications de la norme OSEK VDX (OSEK/VDX, 2005b) puis étendu aux spécifications AUTOSAR OS (AUTOSAR, 2013a). *Trampoline* est développé par l'équipe STR (*Système Temps Réel* de l'IRCCyN) sous licence GNU LGPL 2.1 (*Lesser General Public Licence*).

Le package *Trampoline* 2.x, disponible à l'adresse `http://trampoline.rts-software.org`, est composé d'un noyau conforme OSEK VDX et AUTOSAR OS et des BSP (*Board Support Package*) pour différentes cibles. Un compilateur *OIL* nommé *goil* est également fourni pour spécifier et générer la configuration du système d'exploitation.

A.1 Configuration et chaîne de compilation de Trampoline

La chaîne de développement est décrite par la Figure A.1.

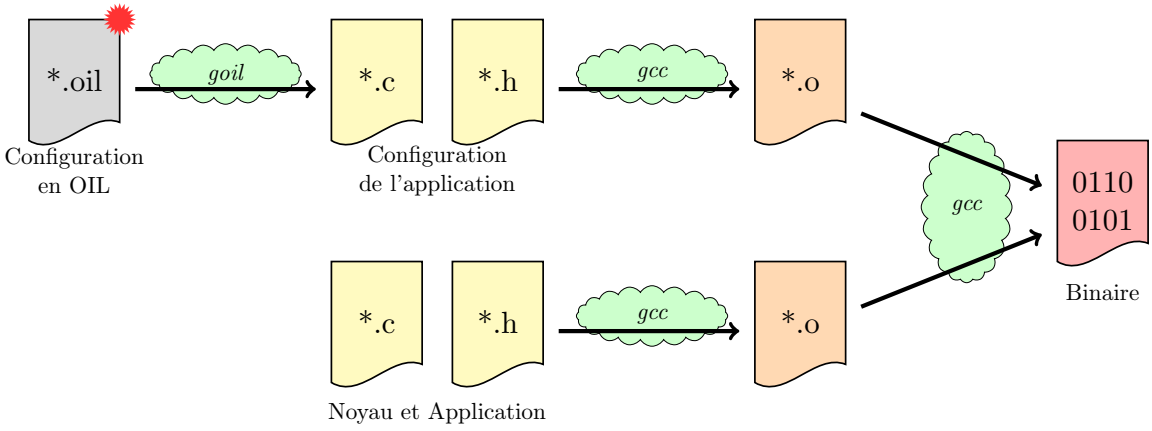


Figure A.1 – Chaîne de compilation de *Trampoline*

La configuration statique du RTOS sous AUTOSAR requiert qu'elle soit faite en *ARXML*.

Cependant, *Trampoline* est configuré en *OIL*, comme spécifié par OSEK dans OSEK OIL (OSEK/VDX, 2005c) et étendu pour prendre en compte les spécificités d'AUTOSAR.

Le RTOS est configuré dans le fichier OIL. Parmi les configurations, nous retrouvons les tâches, les ISRs ou encore les alarmes. Un exemple d'une configuration de base est illustrée par la Figure A.2. Sur l'exemple, la tâche *mytask* est une tâche périodique de période 100 ms. La section de code applicatif définie dans *Trampoline* par *TASK(mytask) { ... }* est exécutée toutes les 100 ms. Une fois la configuration effectuée, le compilateur *goil* l'analyse et génère les sources qui en découlent (e.g. des structures pour les tâches, les alarmes, etc ...). Ces sources sont finalement compilées par *gcc* avec les sources du noyau et celles de l'application pour obtenir le fichier binaire.

```
TASK mytask {
  PRIORITY = 2;                 // Priorité de la tâche
  AUTOSTART = FALSE;
  SCHEDULE = FULL;              // Tâche préemptive
  ACTIVATION = 1;               // 1 activation à la fois
};

ALARM activate_mytask {
  COUNTER = SystemCounter;      // Compteur lié à l'alarme
  ACTION = ACTIVATETASK {
    TASK = mytask;              // Tâche activée par l'alarme
  };
  AUTOSTART = TRUE {
    ALARMTIME = 50;
    CYCLETIME = 100;            // Période de la tâche
    APPMODE = std;
  };
};

COUNTER SystemCounter {         // Compteur de période 1ms
  SOURCE = TIMER_IT;            // Source du timer
  MAXALLOWEDVALUE = 2000;
  TICKPERBASE = 10;
  MINCYCLE = 1;
};
```

Figure A.2 – Exemple de description d'une application en OIL

A.2 La communication interne dans Trampoline monocœur

La communication interne gère les messages produits et consommés par des tâches s'exécutant sur une même puce. La communication interne est spécifiée par OSEK VDX dans OSEK/VDX (2005a). L'architecture de la communication est présentée sur la Figure A.3. Notons qu'il existe deux moyens pour transmettre des messages : la méthode *client/serveur* et la méthode *sender/receiver*. Seule la seconde méthode est illustrée dans la suite.

Les tâches communiquent au travers de *message objects (mo)*, qu'il faut déclarer dans la configuration OIL. On distingue les *sending message objects (smo)*, liés à des tâches émettrices des *receiving message objects (rmo)*, lié à des tâches réceptrices. Un exemple de description

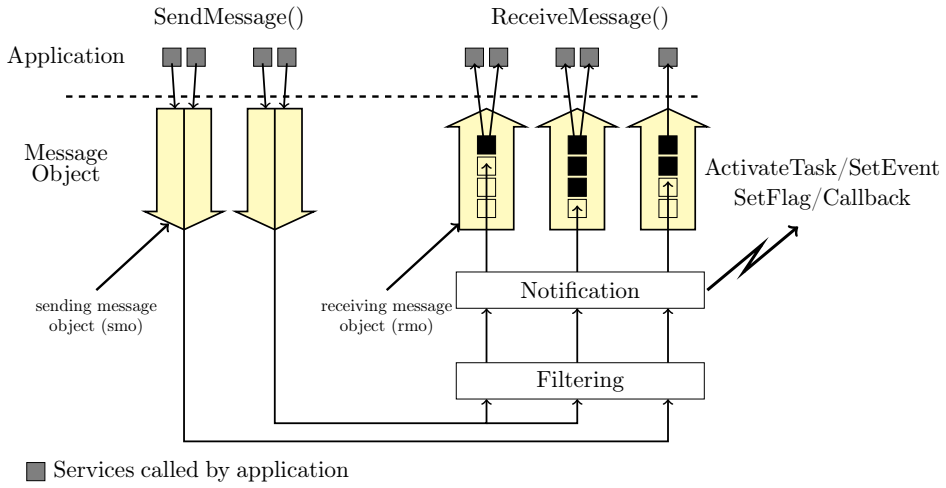


Figure A.3 – Communication interne définie dans OSEK COM (OSEK/VDX, 2005a)

des messages est donné dans la Figure A.4. Chaque *message objet* spécifiant un message en émission est connecté à un ou plusieurs *message objet* lié à un message de réception (i.e. communication de type 1 : M).

```
MESSAGE send_to_b0 {
  MESSAGEPROPERTY =
      SEND_STATIC_INTERNAL {
    CDATATYPE = "u8" ;
  };
  NOTIFICATION = NONE;
};
```

(a) Configuration d'un smo

```
MESSAGE receive_from_b0 {
  MESSAGEPROPERTY =
      RECEIVE_UNQUEUED_INTERNAL {
    SENDINGMESSAGE = send_to_b0;
    INITIALVALUE = 0;
    FILTER = ALWAYS;
  };
  NOTIFICATION = NONE;
};
```

(b) Configuration d'un rmo

Figure A.4 – Différentes approches pour le partage de ressources

Dans la pratique, l'envoi d'une donnée dans un *smo* va avoir pour effet de copier le message dans les buffers associés aux *rmo* correspondants. Deux services permettent d'envoyer et de recevoir des messages :

– Le service $SendMessage(smo, \&data)$, que nous noterons plus simplement $SendMessage()$ permet d'envoyer un message. La procédure d'écriture transmet le message à tous les *rmo* qui sont associés. La zone de filtrage permet d'interdire la diffusion du message quand elle ne respecte pas un certain nombre de critères (e.g. supérieur à .., 1 parmi n, etc.). Il est ensuite physiquement copié dans les buffers, en attendant d'être lu. Deux types de buffers existent : les buffers *queued* sont caractérisés par une taille n. Les messages sont alors insérés les uns après les autres. Seule une lecture peut faire avancer la file. Dans le cas d'un buffer *unqueued*, la valeur du message reste présente tant qu'elle n'a pas été réécrite. Cela signifie aussi qu'elle peut avoir été écrasée sans jamais avoir été lue. Enfin, la mise à disposition d'une nouvelle valeur d'un message peut donner lieu à l'activation

d'une notification. Quatre types de notification existent : activation d'une tâche (ActivateTask), initiation d'un évènement (SetEvent), mise à 1 d'un drapeau (SetFlag) ou création d'un callback (Callback).
- Le service $ReceiveMessage(rmo, \&data)$, que nous noterons $ReceiveMessage()$, permet de lire le message présent dans le buffer associé au *rmo*, et de renvoyer sa valeur vers l'application afin qu'elle puisse être utilisé.

Pour conclure, nous pouvons noter l'existence d'autres modes de transmission de donnée qui ne sont pas détaillés ici, par exemple $SendDynamicMessage()$, $SendZeroMessage$ ou encore $ReceiveZeroMessage()$.

A.3 La communication interne dans AUTOSAR multicœur

La communication interne dans la version multicœur d'*AUTOSAR* passe par l'implantation de l'IOC (*Inter OS-Application Communication*). Le principe de fonctionnement de l'IOC est spécifié par AUTOSAR et est similaire à celui de la communication interne monocœur OSEK.

Une OS-Application est une entité qui regroupe un ensemble d'objets du système d'exploitation (tâches, ISR, alarmes). Des OS-Applications sont situées dans des zones de protection mémoire différentes. Ce découpage permet de regrouper les entités selon différents critères (e.g. paramètres temporels ou fonctions).

Dans AUTOSAR, il existe quatre formes de communication :
- La communication entre des runnables d'une même tâche passe par des variables locales appelées IRV (*InterRunnable Variables*). Ces variables sont stockées dans une zone de protection mémoire commune aux runnables qui y accèdent ;
- Dans AUTOSAR, il est autorisé que la communication entre des tâches qui sont dans une même OS-Application ne passe que par le RTE (*Run-Time Environnement*), c'est-à-dire effectué en dehors du noyau. Dans *Trampoline*, toutes les tâches ont leur propre zone de protection mémoire. Dans un objectif de tolérance aux fautes, il est recommandé de passer par l'OS pour toutes les communications intertâches et donc d'utiliser l'IOC ;
- La communication entre des tâches qui sont dans des OS-Applications différentes est systématiquement géré par l'IOC ;
- Les tâches qui sont sur des cœurs différents sont systématiquement dans des OS-Applications différentes. La communication inter-cœur passe donc systématiquement par l'OS, via l'IOC.

L'IOC permet, comme la communication interne monocœur, d'assurer l'intégrité des messages transmis. La granularité des messages est l'objet, c'est-à-dire qu'il peut s'agir de mots ou de structure de données plus complexes. Les notifications possibles sont les mêmes que celles évoquées dans la section précédente.

Une illustration est proposée dans la Figure A.5.

Quand un runnable d'une tâche doit envoyer un message, elle appelle la fonction *RTE_ Write()* associée à ce message. En cas de communication Inter OS-Application, elle appelle à son tour la fonction *IocSend()* du message, qui est en charge de copier la valeur de la donnée à transmettre dans le buffer correspondant. Lors de la réception, la tâche nécessitant le message appelle la fonction *RTE_ Read()* du message qui elle-même appelle *IocWrite()*, en charge de récupérer la

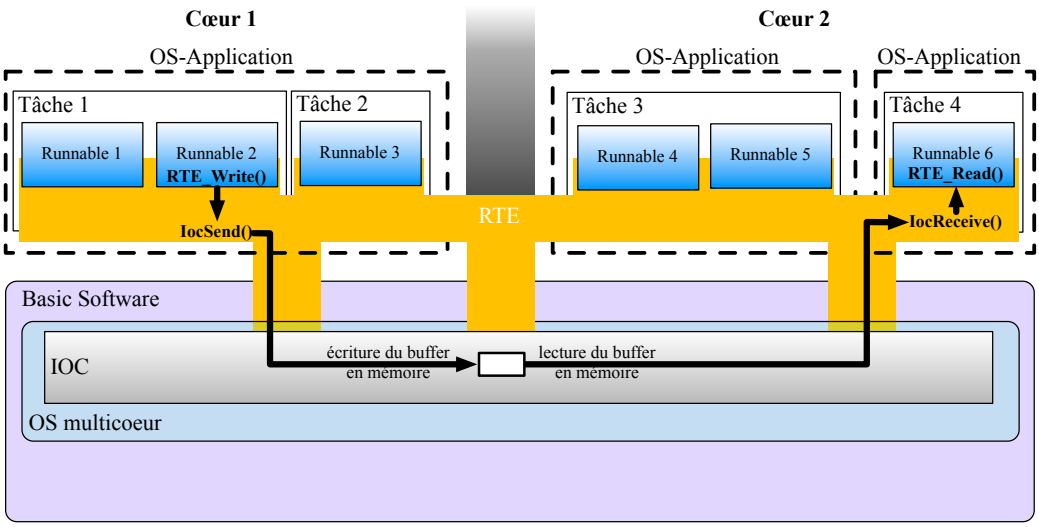


Figure A.5 – L'IOC dans AUTOSAR (AUTOSAR, 2013)

valeur de la donnée dans le buffer.

Notons que dans *Trampoline*, nous n'e générons pas de RTE, donc les services de l'IOC sont directement appelés à partir du code applicatif.

Annexe B

Étude de cas *PROTOSAR*

B.1 Bilan des signaux transitant entre chaque bloc d'un composant

Pour chaque composant (le contrôle de la trajectoire, le contrôle de la vitesse, le frein de parking, le levier de vitesse, le contrôle du volant, la gestion de l'accélération et le contrôle du freinage), nous présentons dans les Tables B.1, B.2, B.3, B.4, B.5, B.6, B.7 les signaux utilisés entre chaque bloc, conformément à l'architecture présentée dans la Section 7.2 (page 97).

En colonne, les blocs désignent le point de départ des signaux tandis que ceux en ligne désignent les points d'arrivée.

de \ vers	pré Monitoring	compute feedback	control	algorithms	post Monitoring
pre Monitoring	∅	∅	1	2	2
compute feedback	∅	∅	∅	∅	∅
control	∅	∅	∅	1	1
algorithms	∅	∅	∅	∅	2
post Monitoring	∅	∅	1	∅	∅

Table B.1 – Signaux pour le composant de contrôle de la trajectoire

de \ vers	pré Monitoring	compute feedback	control	algorithms	post Monitoring
pre Monitoring	∅	3	1	7	7
compute feedback	∅	∅	2	∅	∅
control	∅	∅	∅	1	1
algorithms	∅	∅	∅	∅	2
post Monitoring	1	∅	1	∅	∅

Table B.2 – Signaux pour le composant de contrôle de la vitesse

de \ vers	pré Monitoring	compute feedback	control	algorithms	post Monitoring
pre Monitoring	∅	3	3	2	1
compute feedback	∅	∅	1	∅	∅
control	∅	∅	∅	1	1
algorithms	∅	∅	∅	∅	2
post Monitoring	2	∅	1	∅	∅

Table B.3 – Signaux pour le composant de gestion du frein de parking

de \ vers	pré Monitoring	compute feedback	control	algorithms	post Monitoring
pre Monitoring	∅	3	3	3	3
compute feedback	∅	∅	1	∅	∅
control	∅	∅	∅	1	1
algorithms	∅	∅	∅	∅	2
post Monitoring	2	∅	2	∅	∅

Table B.4 – Signaux pour le composant de gestion du levier de vitesse

de \ vers	pré Monitoring	compute feedback	control	algorithms	post Monitoring
pre Monitoring	∅	3	3	3	3
compute feedback	∅	∅	1	∅	∅
control	∅	∅	∅	1	1
algorithms	∅	∅	∅	∅	3
post Monitoring	1	∅	1	∅	∅

Table B.5 – Signaux pour le composant de contrôle du volant

de \ vers	pré Monitoring	compute feedback	control	algorithms	post Monitoring
pre Monitoring	∅	3	2	6	6
compute feedback	∅	∅	2	∅	∅
control	∅	∅	∅	1	1
algorithms	∅	∅	∅	∅	2
post Monitoring	1	∅	1	∅	∅

Table B.6 – Signaux pour le composant de contrôle de l'accélération

de \ vers	pré Monitoring	compute feedback	control	algorithms	post Monitoring
pre Monitoring	∅	3	3	29	29
compute feedback	∅	∅	1	∅	∅
control	∅	∅	∅	1	1
algorithms	∅	∅	∅	∅	4
post Monitoring	2	∅	1	∅	∅

Table B.7 – Signaux pour le composant de contrôle du freinage

B.2 Traduction des exigences en LTL

Nous traduisons les exigences définies dans la Section 7.3 (page 99) en termes de propriétés inter-tâches écrites en logique LTL. Pour cela, nous commençons par présenter le modèle du système à considérer dans chacun des cas. Nous émettons les hypothèses suivantes :

- Les communications inter-tâches sont réalisées grâce à des buffers *unqueued* (i.e. la donnée est disponible tant qu'elle n'a pas été écrasée) ;
- Le schéma de communication est 1 : m, c'est-à-dire que seule une tâche peut mettre à jour un buffer tandis que m tâches peuvent le lire.

B.2.1 Exigences sur les productions et consommations de données

Pour surveiller les exigences associées à cette catégorie, nous devons modéliser l'envoi et la réception d'un message dans un buffer. Un tel système est illustré sur la Figure B.1. On considère une tâche source T_0 et une autre cible T_1. La communication est réalisée au travers du buffer b_0.

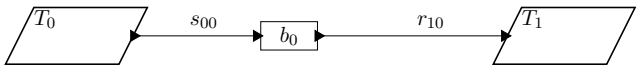


Figure B.1 – Envoi et réception d'un message dans un buffer

B.2.1.1 Exigences 7.1, 7.2 et 7.3

Le comportement du système étudié est modélisé par l'automate m_buffer de la Figure B.2. Seuls trois états sont nécessaires. L'envoi d'une donnée dans le buffer permet les passages de l'état 0 à 1 puis p. Si la donnée est surproduite, c'est-à-dire si elle a été écrasée avant d'être lue, l'automate reste dans l'état p. La consommation de la donnée conduit quant à elle au retour à l'état initial 0.

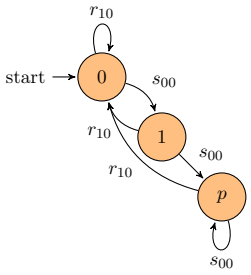


Figure B.2 – m_buffer : Modèle de l'envoi et la réception d'un message dans un buffer

Au vue des exigences à traiter, la formule LTL doit vérifier que nous ne perdons jamais de données. Concrètement, il s'agit de vérifier que l'état p de l'automate m_buffer n'est jamais atteint. Ceci est écrit sous la forme :

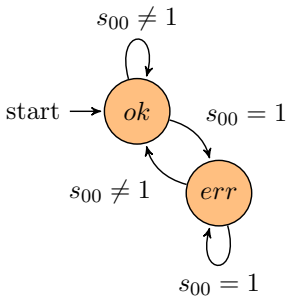
```
always ( not m_buffer.p )
```

B.2.1.2 Exigence 7.4

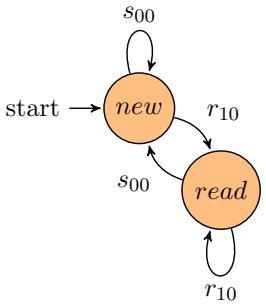
L'Exigence 7.4 peut être traitée de manière similaire sauf que nous autorisons désormais la perte de 2 données sur 3. Il est donc nécessaire de prendre en compte cet aspect en enrichissant le modèle. Pour cela, il suffit d'ajouter à l'automate m_buffer de la Figure B.2, un état nommé 2, entre les états 1 et p. La formule LTL qui en découle ne change pas.

B.2.1.3 Exigences 7.5 et 7.6

Ces exigences requièrent de connaître la valeur de la donnée. En particulier, l'occurrence d'une erreur est mise en évidence par un flag égal à 1. Ceci est représenté sur la Figure B.4(a). Le modèle des lecture/écriture est donné par la Figure B.3(b).



(a) m_error : Modélisation de la détection d'erreurs



(b) m_read_write : Modélisation des accès lecture/écriture pour b_0

Sur l'automate m_error, l'erreur est mise en évidence quand les signaux d'erreurs valent 1. Dans ce cas, on passe à l'état err. Un retour à 0 permet de revenir dans l'état initial nommé ok. Les accès au buffer b_0 sont modélisés par l'automate m_read_write. Dès qu'une donnée est écrite dans le buffer, elle est considérée comme nouvelle, ce qui conduit à l'état new. Dès que la donnée est lue, nous allons dans l'état $read$. Notons que les deux automates sont synchronisés par l'évènement s_{00}. En effet, dès qu'une erreur est détectée (i.e. $s_{00} = 1$), l'automate m_read_write est systématiquement dans l'état new.

L'écriture de la formule LTL permet de vérifier que lorsqu'une erreur est signalée, le flag reste en position jusqu'à ce qu'il soit consommé. Elle est définie comme suit :

```
always ( m_error.error implies (
  (m_error.err and m_read_write.new) until (m_error.err and m_read_write.read)
) )
```

B.2.2 Exigences sur la propagation des données

Pour surveiller les exigences associées à cette catégorie (i.e. Exigences 7.7 et 7.8), nous devons modéliser la partie du système qui traite les erreurs, depuis l'occurrence de l'erreur jusqu'à son traitement. La propagation des messages est illustrée sur la Figure B.1. Désormais, nous avons un buffer b_1 qui permet la communication de T_1 avec T_2.

Le buffer b_0 contient le flag d'erreur (il vaut 1 en cas d'erreur). Le buffer b_1 est la sortie (il vaut 0 quand une erreur est détectée).

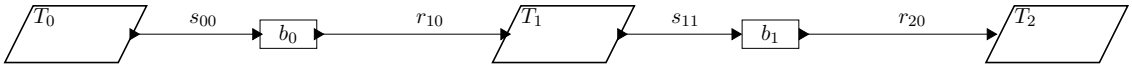
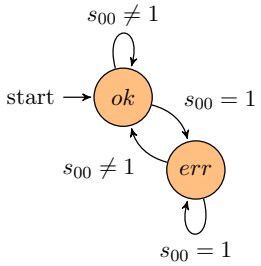
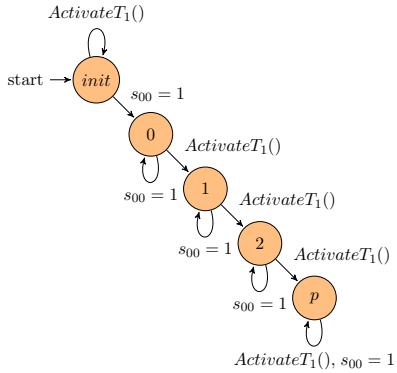


Figure B.3 – Propagation d'un message dans un système

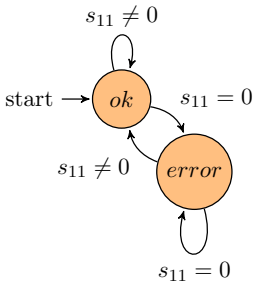
L'automate *m_error* de la Figure B.4(a) a déjà été expliquée dans la précédente règle. *m_actTask* sur la Figure B.4(b) permet de compter le nombre d'activations subit par T_1 depuis l'occurrence d'une erreur. À chaque activation de la tâche, l'automate passe des états 0 à 1 puis 2 puis *p*. On ne revient jamais dans l'état initial puisque les exigences concernées stipulent qu'en cas d'erreur, le système est arrêté. Enfin, l'automate *m_output* sur la Figure B.4(c) représente la sortie de T_1. La sortie vaut 0 quand l'erreur est prise en compte.



(a) *m_error* : Modélisation de la détection d'erreurs



(b) *m_actTask_count* : Modélisation du comptage du nombre d'activations d'une tâche depuis l'occurrence d'une erreur



(c) *m_output* : Modélisation de la sortie de T_1

L'exigence à traiter spécifie un nombre d'activations de T_1 maximal entre l'occurrence de l'erreur et sa prise en compte. Cette borne est définie par l'état *p* de l'automate *m_output*. La formule LTL est donc de la forme :

```
always ( m_error.error implies (
  always not (m_output.ok and m_actTask_count.p)
) )
```

Notons que l'occurrence d'une erreur conduit à l'arrêt complet du système.

B.2.3 Exigences sur la cohérence des données

B.2.3.1 Exigences 7.9 et 7.10

La surveillance de ce type d'exigences nécessite de modéliser la réception des données pour tous les signaux entre deux blocs. Nous surveillons ainsi la consistance d'un groupe de données. Pour ces règles, la situation se modélise par la Figure B.4. Ici, nous considérons un groupe de trois données, mais l'étude nécessite aussi de surveiller des groupes de 2 données, ce qui est moins complexe.

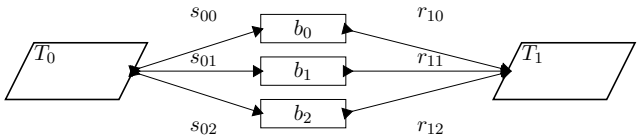
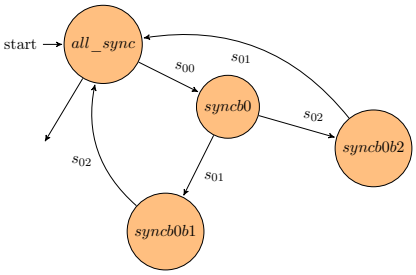
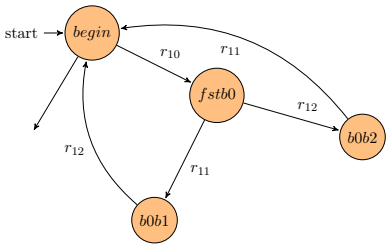


Figure B.4 – Envoi et réception de messages dans des buffers

Les automates modélisant la production des données (m_prod) et la consommation des données (m_cons) dans les buffers sont représentés respectivement sur les Figures B.5(a) et B.5(b). Pour plus de lisibilité, les automates ne sont pas représentés en entier. En effet, puisque nous ne faisons pas d'hypothèses sur l'ordre de production ou de consommation des buffers, on considère qu'ils peuvent être accédés de manière aléatoire. On considère néanmoins que l'activation de T_0 conduit à l'écriture d'un seul message par buffer. Sur la Figure B.5(a), l'automate m_prod permet de représenter la synchronisation des buffers. Les buffers sont synchronisés quand toutes les données ont la même instance. Dès qu'un des buffers est rafraîchi, ils sont désynchronisés. La synchronisation passe par l'écriture successive des deux autres buffers. Sur la Figure B.5(b), l'automate m_cons permet de modéliser la lecture des buffers. Il se lit de la même manière que m_prod.



(a) m_prod : Modélisation correspondant à la production des données



(b) m_cons : Modélisation correspondant à la consommation des données

Pour vérifier que les buffers sont synchronisés pendant leur lecture, nous écrivons la formule LTL sous la forme :

```
always ( (m_cons.fstb0 or m_cons.fstb1 or m_cons.fstb2) implies (
  m_prod.all_sync until m_cons.begin
) )
```

B.2.3.2 Exigence 7.11

Cette exigence est similaire à l'Exigence 3.2 traitée dans nos exemples. Le système considéré est rappelé dans la Figure B.5.

Pour surveiller cette règle, nous considérons les automates illustrés sur les Figures B.6(a), B.6(b) et B.6(c). m_sync permet de modéliser la synchronisation des deux buffers. m_t2 permet de modéliser la lecture des buffers par T_2. Comme précédemment, nous ne faisons aucune hypothèse sur l'ordre de lecture des buffers. Enfin, m_b0 modélise le comportement de b_0. Lorsqu'il est lu par T_1, on passe dans l'état b_0read1. S'il est lu par T_2, on passe dans l'état b_0read2. Enfin, quand il est mis à jour, nous passons à l'état $writeb_0$.

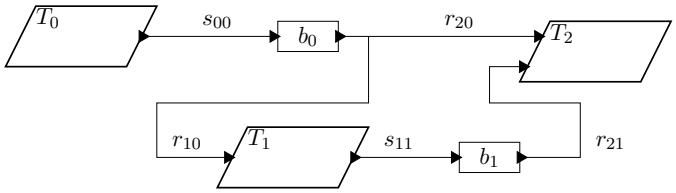
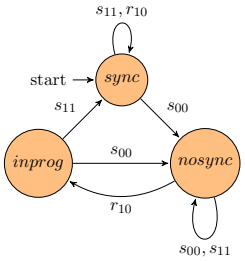
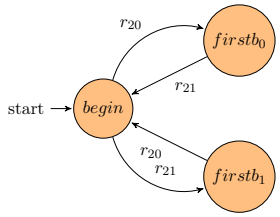


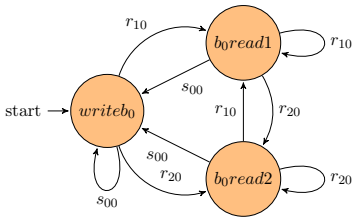
Figure B.5 – Synchonisation des données entre 2 buffers



(a) m_sync : Modélisation de la synchronisation de b_0 et b_1



(b) m_t2 : Modèle du comportement de T_2



(c) m_b0 : Modèle du comportement de b_0

Deux cas doivent être distingués : celui où b_0 est lu en premier par T_2 et celui où c'est b_1.

Dans la première situation, quand T_2 lit b_0 en premier, nous devons interdire l'écrasement de b_0 et attendre la synchronisation avant d'avoir l'autorisation de lire b_1. Dans le second cas, les buffers doivent déjà être synchronisés pour pouvoir poursuivre la lecture. Ces deux conditions sont représentées par les formules LTL suivantes :

```
always ( m_t2.firstb0 implies (
  not (m_b0.writeb0 or m_t2.begin) until (m_sync.sync and m_t2.begin)
) )
```

```
always ( m_t2.firstb1 implies (
  m_sync.sync until m_t2.begin
) )
```

Dans le cas d'étude, il faut considérer tous les signaux qui transitent entre deux tâches. En effet, l'architecture que l'on souhaite surveiller est plutôt telle que présentée dans la Figure B.6. Cette architecture possède $n+m$ buffers. Par exemple, dans le cadre du composant de contrôle de la vitesse, il y a 7 messages d'entrée et 2 messages de sortie. Nous choisissons d'effectuer la vérification de toutes les combinaisons possibles. Dans le cas du contrôle de la vitesse, il faut alors $7 * 2 = 14$ moniteurs de ce type.

Une autre solution consiste à faire des hypothèses sur les ordres de productions et consommations des différents messages. Pour cela, il faut considérer uniquement les buffers 1, n et m.

Un exemple de modèle du système est représenté par trois automates. Sur la Figure B.7(a), nous représentons m_sync : l'automate modélisant la synchronisation de tous les buffers. Pour

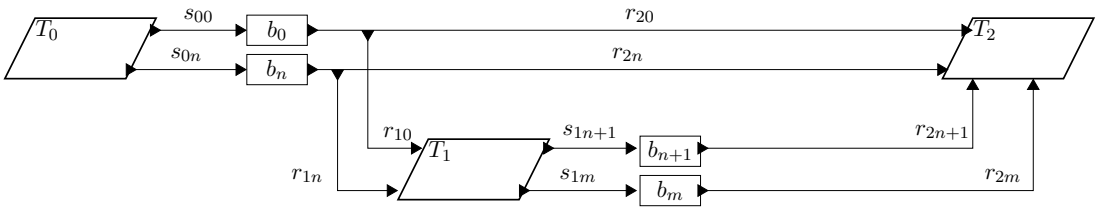
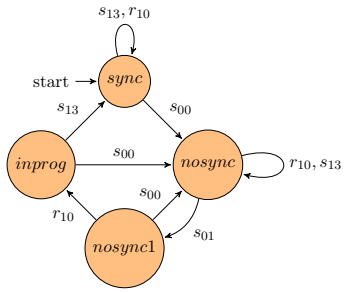
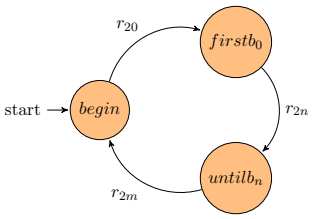


Figure B.6 – Synchonisation des données entre $n + m$ buffers

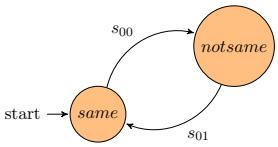
synchroniser les buffers, il faut que T_0 écrive toutes ses données dans b_0 ... b_n avant que T_1 ne fasse ses lectures (commençant b_0). Pour finir la synchronisation, il faut attendre que T_1 écrive dans b_{n+1}, etc. jusqu'à b_m. Sur la Figure B.7(b), m_t2 représente le comportement de T_2. T_2 commence par lire b_0 et tous les buffers jusqu'à b_n puis de b_{n+1} à b_m. Enfin, $m\_consistant$ sur la Figure B.7(c) représente la synchronisation des n buffers d'entrée. Quand toutes les données ont la même instance, on reste dans l'état *same*. Pendant la phase d'écriture, on est dans l'état *notsame*.



(a) m_sync : Modélisation de la synchronisation des $n + m$ buffers



(b) m_t2 : Modèle du comportement de T_2



(c) $m\_consistant$: Consistance des n buffers d'entrées

Pour exprimer la propriété il faut vérifier que lorsque T_2 commence à lire b_0, toutes les entrées ont la même instance. Nous devons ensuite attendre la synchronisation avant de lire les données provenant de b_{n+1} à b_m. La formule LTL correspondant à la propriété devient :

```
always ( m_t2.firstb0 implies (
   m_consistant.same and not(m_t2.begin or m_t2.untilbn)
     until (m_sync.sync and m_t2.begin)
) )
```

B.2.3.3 Exigence 7.12

La surveillance de cette exigence nécessite de considérer l'architecture présentée dans la Figure B.7. Il s'agit du cas où il faut gérer l'accès en lecture à un buffer par deux tâches distinctes.

Pour modéliser le comportement des productions et des consommations dans le buffer, considérons l'automate représenté sur la Figure B.8. À partir de l'état initial, la lecture du buffer par T_1 ou T_2 conduit respectivement aux états b_0read1 et b_0read2. Quand b_0 est écrasé, c'est l'état b_0write qui est atteint.

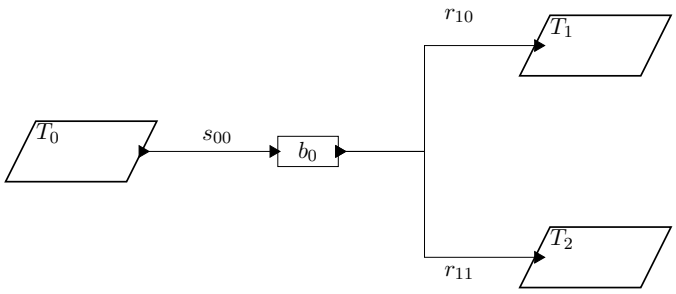


Figure B.7 – Architecture du mécanisme multiconsommateur

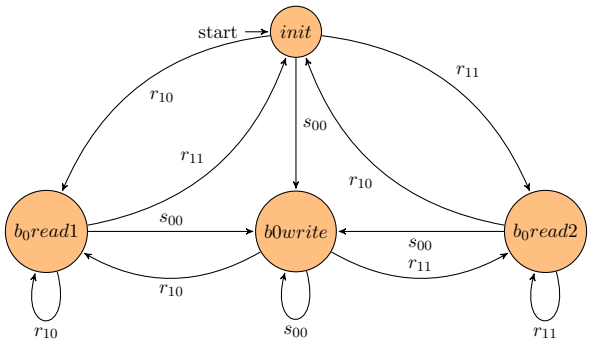


Figure B.8 – m_buffer : Modélisation de l'envoi et la réception d'un message dans un buffer multiconsommateur

Nous devons vérifier que la donnée n'est pas écrasée en cours de lecture. La modélisation choisie permet naturellement de faire cette vérification. Nous proposons donc d'écrire la formule LTL de la forme suivante :

```
always ( (m_buffer.b0read1 or m_buffer.b0read2) implies (
  next m_buffer.init
) )
```

Annexe C

Détails des macros nécessaires pour STM-HRT

Toute les macros sont détaillées dans la Table C.2. Les constantes utilisées sont définies dans la Table C.1.

Constante	Description
NB_ACT_TRANS	Nombre de transactions actives dans le système $\Longleftrightarrow NB\_ACT\_TRANS \leq M$
MASK	Masque pour isoler le READ_VECTOR ou le FAIL_VECTOR $\Longleftrightarrow MASK = (1 \ll NB\_ACT\_TRANS) - 1$
READ_VECTOR_LSB	Bit de poids faible du READ_VECTOR $\Longleftrightarrow READ\_VECTOR\_LSB = 2$
FAIL_VECTOR_LSB	Bit de poids faible du FAIL_VECTOR $\Longleftrightarrow FAIL\_VECTOR\_LSB = (2 + NB\_ACT\_TRANS)$

Table C.1 – Définitions des constantes utilisées dans STM-HRT

Macro	Description
STATUS(Tx)	Valeur du statut de la transaction Tx
	$\Longleftrightarrow Tx.status$ & 3
INSTANCE(Tx)	Numéro de l'instance du descripteur associé à Tx
	$\Longleftrightarrow (Tx.status \ll 2)$ & 63
CURRENT_OBJECT_POS(Oh)	Position de la valeur de l'objet Oh dans la table des copies
	$\Longleftrightarrow Oh.concurrency\_vector$ & 1
UPDATE_FLAG(Oh)	Valeur de l'UPDATE_FLAG pour l'objet Oh
	$\Longleftrightarrow (Oh.concurrency\_vector \gg 1)$ & 1
READ_VECTOR(Oh)	Valeur du READ_VECTOR de l'objet Oh
	$\Longleftrightarrow (Oh.concurrency\_vector \gg READ\_VECTOR\_LSB)$ & $MASK$
FAIL_VECTOR(Oh)	Valeur du FAIL_VECTOR de l'objet Oh
	$\Longleftrightarrow (Oh.concurrency\_vector \gg FAIL\_VECTOR\_LSB)$ & $MASK$
SET_READVECTOR(Oh, Tx)	La transaction Tx lit l'objet Oh : mise à jour du READ_VECTOR de l'objet Oh
	$\Longleftrightarrow Oh.concurrency\_vector\| = 1 \ll (READ\_VECTOR\_LSB + Tx.proc\_id)$
READVECTOR_BIT(Oh, Tx)	Test si la transaction Tx a ouvert l'objet Oh
	$\Longleftrightarrow (READ\_VECTOR(Oh) \gg Tx.proc\_id)$ & 1
SET_FAILVECTOR(Oh, Tx)	La transaction Tx a échoué : mise à jour du FAIL_VECTOR de l'objet Oh
	$\Longleftrightarrow Oh.concurrency\_vector\| = 1 \ll (FAIL\_VECTOR\_LSB + Tx.proc\_id)$
FAILVECTOR_BIT(Oh, Tx)	Test si la transaction Tx a déjà échouée
	$\Longleftrightarrow (FAIL\_VECTOR(Oh) \gg Tx.proc\_id)$ & 1
RESET_FAILVECTOR(Oh, Tx)	La transaction Tx n'a plus besoin d'être aidée : mise à jour du FAIL_VECTOR de l'objet Oh
	$\Longleftrightarrow Oh.concurrency\_vector^{\wedge} = (1 \ll (2 + NB\_ACT\_TRANS + Tx.proc\_id))$
SET_UPDATEFLAG(Oh)	Mise à 1 de l'$UPDATE\_FLAG$ de l'objet Oh
	$\Longleftrightarrow Oh.concurrency\_vector\| = (1 \ll 1)$
UPDATEFLAG(Oh)	Test si l'$UPDATE\_FLAG$ de l'objet Oh vaut 1
	$\Longleftrightarrow (Oh.concurrency\_vector \gg UPDATE\_FLAG)$&1
SET_ACCESSVECTOR(Tx, Oh)	L'objet Oh est ouvert par Tx : mise à jour du vecteur d'accès
	$\Longleftrightarrow Tx.access\_vector\| = (1 \ll Oh.object\_id)$

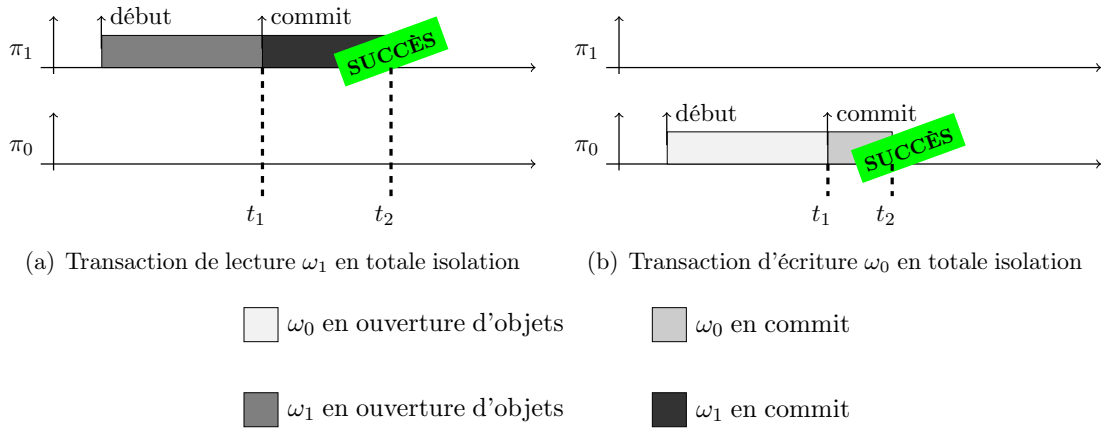
Table C.2 – Définitions des macros utilisées dans STM-HRT

Annexe D

Illustration du fonctionnement de STM-HRT

Illustrons le fonctionnement du protocole sur quelques exemples. Nous reprenons la configuration utilisée dans la Section 10.2.4 (page 135) et nous considérons différentes situations. Tout d'abord, nous présentons le fonctionnement de STM-HRT lorsque des transactions de lecture et d'écriture s'exécutent en totale isolation. Dans un second temps, nous présentons des cas de conflits et nous montrons comment le mécanisme d'aide est mis en jeu.

D.1 Transactions en totale isolation



(a) Transaction de lecture ω_1 en totale isolation

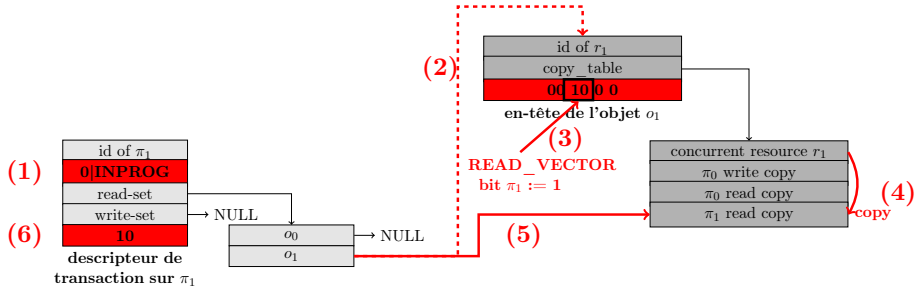
(b) Transaction d'écriture ω_0 en totale isolation

Figure D.1 – 2 transactions ω_0 et ω_1 en totale isolation

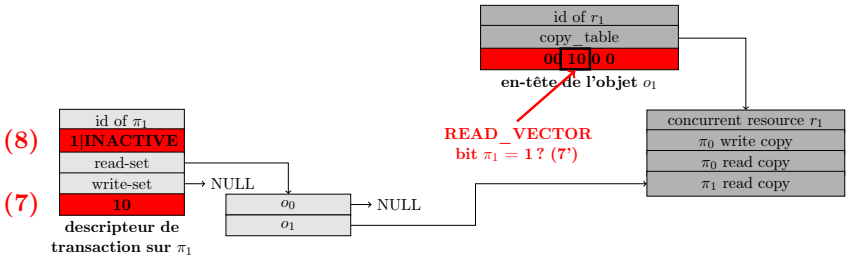
La Figure D.1(a) représente une transaction de lecture ω_1, allouée sur π_1, et exécutée en totale isolation. La Figure D.1(b) représente une transaction d'écriture ω_0, allouée sur π_0,et exécutée en totale isolation. Un seul objet est accédé dans chacune des transactions. ω_0 ouvre

l'objet o_0 en écriture tandis que ω_1 ouvre l'objet o_1 en lecture. Dans ces circonstances, nous pouvons supposer que ω_0 et ω_1 s'exécutent en parallèle sans interférer puisque les ensembles d'objets manipulés sont disjoints. Pour illustrer le fonctionnement du protocole, nous représentons l'état des structures de données aux instants désignés par les traits pointillés.

Considérons dans un premier temps la transaction de lecture ω_1 en totale isolation. Puisque ω_1 s'exécute sur π_1, nous ne représentons que le descripteur de transaction de π_1. De plus, puisque ω_1 n'ouvre que o_1, nous ne représentons que l'en-tête de l'objet de o_1.



(a) ω_1 ouvre o_1 (instant t_1)



(b) ω_1 en commit (instant t_2)

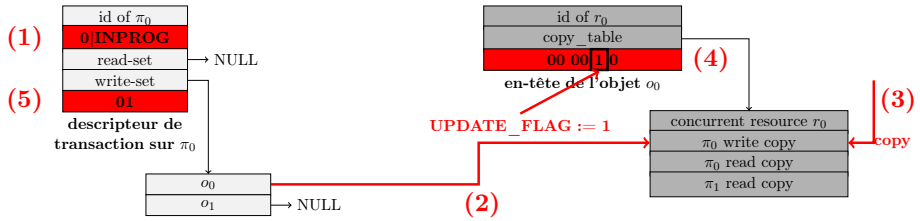
Figure D.2 – Structures de données associées à ω_1 sur l'exemple de la Figure D.1(a)

La Figure D.2(a) représente l'état des structures de données à l'instant t_1 quand ω_1 ouvre o_1. Au démarrage de la transaction, le statut passe de *INACTIVE* à *INPROG* (étape 1). Lorsque la transaction est prête à lire o_1, le pointeur de la table des accès en lecture correspondant à o_1 pointe vers l'en-tête de o_1 (étape 2). En cas de conflits puis d'aide, une transaction aidante pourrait désormais lire o_1 au nom de ω_1. Le *READ_ VECTOR* de o_1 est ensuite mis à jour à l'étape (3) en mettant à 1 le bit correspondant à la transaction active du cœur π_1. L'objet est ensuite copié à partir de la version courante, vers l'emplacement réservé à la copie quand π_1 lit o_1 (étape 4). Pour vérifier que l'objet copié est consistant, on vérifie que le bit mis à 1 à l'étape (3) l'est toujours. Si ce n'est pas le cas, l'objet a été écrasé et on répète les opérations (3) et (4) tant que la condition n'est pas respectée. Une fois que la valeur de l'objet récupéré est consistante, le pointeur de la table des accès en lecture correspondant à o_1 pointe vers cette valeur (étape 5). Pour finir, le vecteur d'accès du descripteur de transaction de π_1 est mis à jour à l'étape (6). Puisque c'est l'objet o_1 qui a été ouvert, c'est le second bit du vecteur d'accès qui est mis à 1.

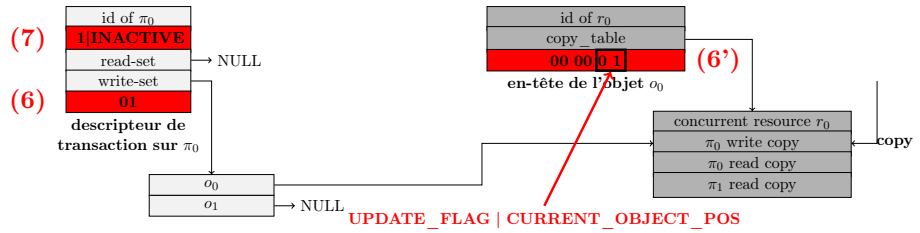
À l'instant t_2 (Figure D.2(b)), ω_1 a réussi son commit. Le vecteur d'accès du descripteur de transaction de π_1 est d'abord parcouru à l'étape (7). Pour chaque objet marqué par un

1 dans ce vecteur, la consistance de l'objet est vérifié. Ici, seul o_1 a été ouvert. Nous regardons donc pour o_1 si le bit mis à 1 lors de l'étape (3) l'est toujours à l'étape (7'). Puisque la transaction est exécutée en totale isolation, aucun conflit ne peut survenir et la condition précédente est vérifiée. La transaction termine en remettant son statut à *INACTIVE* et son instance est incrémentée à l'étape (8). Le nettoyage des structures de données n'est pas illustré.

Considérons à présent la transaction d'écriture ω_0 en totale isolation. Comme précédemment, puisque ω_0 s'exécute sur π_0, nous ne représentons que le descripteur de transaction de π_0. De plus, puisque ω_0 n'ouvre que o_0, nous ne représentons que l'en-tête de o_0.



(a) ω_0 ouvre o_0 (instant t_1)



(b) ω_0 en commit (instant t_2)

Figure D.3 – Structures de données associées à ω_0 sur l'exemple de la Figure D.1(b)

La Figure D.3(a) représente l'état des structures de données à l'instant t_1 lorsque ω_0 ouvre o_0. Au démarrage, le statut passe de *INACTIVE* à *INPROG* (étape 1). Avant de récupérer la nouvelle valeur de l'objet, le pointeur de la table des accès en écriture correspondant à o_0 pointe vers l'emplacement réservé à la copie locale (étape 2). La nouvelle valeur provient de la tâche au sein de laquelle s'exécute la transaction. L'adresse de cette valeur est passée en paramètre de l'API de STM-HRT et son contenu est copié à l'emplacement de la copie locale à l'étape (3). La copie est obligatoirement consistante, car il est impossible que nous observions un conflit dans cette phase. Une fois copié, l'*UPDATE_FLAG* de o_0 est mis à 1 à l'étape (4). Cela permet d'informer les transactions de lecture que l'objet est sur le point d'être mis à jour. Enfin, le vecteur d'accès du descripteur de transaction de π_0 est mis à jour. Puisque c'est o_0 qui est ouvert, c'est le bit de poids faible de ce vecteur qui est mis à 1.

À l'instant t_2 (Figure D.3(b)), ω_0 effectue son commit. Le vecteur d'accès du descripteur de transaction de π_0 est tout d'abord parcouru à l'étape (6). Pour chaque objet marqué par un 1, la mise à jour de l'objet est rendu publique à l'étape (6). Sa mise à jour respecte les conditions présentées dans les algorithmes. Puisque ω_0 est exécutée en totale isolation, le *FAIL_VECTOR* de o_0 vaut 0 et nous pouvons mettre à jour l'objet en réinitialisant l'*UPDATE_FLAG* et en complémentant *CURRENT_OBJECT_POS*. Le vecteur initial valant 00 00 1 0, après la mise à jour il vaut 00 00 0 1. Pour finir, le statut de la transaction repasse à *INACTIVE* et le statut

est incrémenté à l'étape (7). Le nettoyage des structures de données n'est pas illustré dans les figures.

D.2 Transaction de lecture aidant une transaction d'écriture

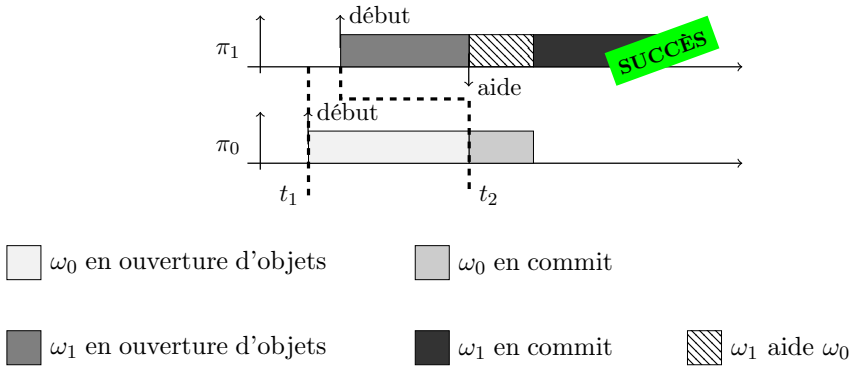


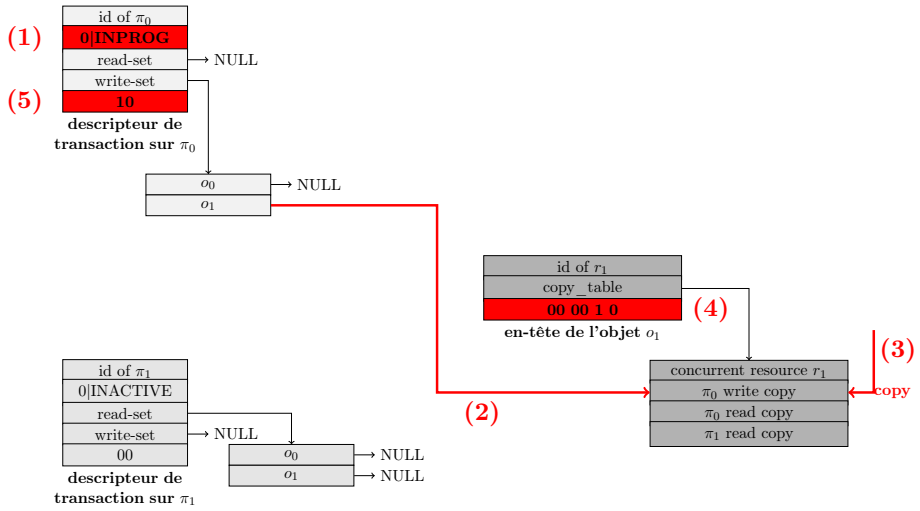
Figure D.4 – Transaction de lecture aidant une transaction d'écriture

L'exemple de la Figure D.4 représente la situation où ω_1 aide ω_0, c'est-à-dire une transaction de lecture aide une transaction d'écriture (en phase d'ouverture des objets). Dans cet exemple, ω_0 et ω_1 ouvrent l'objet o_1. Pour faciliter la présentation, nous ne faisons évoluer qu'une seule transaction à la fois, ce qui permet d'avoir une meilleure visibilité sur l'évolution des transactions tout en restant cohérent avec le comportement parallèle du système et tout en tenant compte des entrelacements. Sur la Figure D.5(a), seule ω_0 évolue tandis que la Figure D.5(b), seule ω_1 évolue. Dans la pratique, ω_0 et ω_1 évoluent en parallèle. Pour des raisons de lisibilité, les cases rouges représentent un changement effectué par l'évolution de ω_0 tandis que les cases bleues représentent un changement effectué par l'évolution de ω_1.

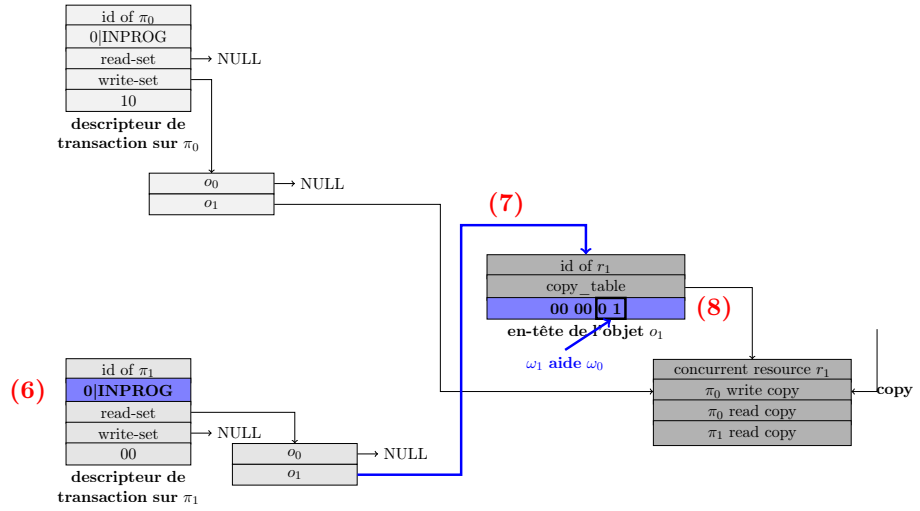
À l'instant t_1 (Figure D.5(a)), ω_0 commence par ouvrir o_1 en écriture. Les étapes (1) à (4) sont identiques à celles détaillées par la Figure D.3(a), en adaptant à l'objet o_1.

À l'instant t_2 (Figure D.5(b)), ω_1 démarre tandis que ω_0 a déjà terminé de lire o_1. On suppose la progression de ω_0 temporairement suspendue avant qu'elle ne commence le commit. Le statut de ω_1 est tout d'abord initialisé à *INPROG* à l'étape (6). Avant d'ouvrir ω_1 en lecture, la transaction analyse l'*UPDATE_FLAG* de o_1 (notons que si le statut de ω_1 valait *RETRY*, cette vérification n'aurait pas été effectuée). Puisque l'*UPDATE_FLAG* de o_1 vaut 1, ω_1 aide ω_0 avant de lire o_1 (étapes 7 et 8). L'aide consiste à réaliser l'opération atomique permettant de mettre à jour o_1. Concrètement, ω_0 remplace le *concurrency_vector* de *00 00 1 0* par *00 00 0 1*.

La suite des opérations est similaire à celle présentée dans les exemples précédents. Le commit de ω_0 sera fait de la même manière que présenté dans l'exemple de la Figure D.3(b) (notons tout de même que la mise à jour est déjà faite donc ce sera plus rapide). La suite de ω_1 sera faite de la même manière que présentée dans l'exemple des Figures D.2(a) et D.2(b), à partir de l'étape (3).



(a) ω_0 ouvre o_0 (instant t_1)



(b) ω_1 aide ω_0 (instant t_2)

Figure D.5 – Structures de données associées à l'exemple de la Figure D.4

D.3 Transaction d'écriture aidant une transaction de lecture

L'exemple de la Figure D.6 représente la situation où ω_0 aide ω_1, c'est-à-dire une transaction d'écriture aide une transaction de lecture (en phase de commit). Dans cet exemple, ω_0 ouvre l'objet o_1 tandis que ω_1 ouvre les objets o_0 et o_1. Comme précédemment, nous ne faisons évoluer qu'une seule transaction à la fois. Nous ne détaillons pas le premier échec de ω_1, qui serait dû à l'écrasement de o_1 par ω_0 pendant la première tentative de ω_1. À l'instant t_1, représenté sur la Figure D.7(a), ω_1 commence son second essai. Nous montrons ensuite aux instants t_2 et t_3, respectivement présentés sur les Figures D.7(b) et D.7(c), comment le mécanisme d'aide est utilisé. Comme précédemment, les cases rouges représentent un changement effectué par l'évolution de ω_0 tandis que les cases bleus représentent un changement effectué

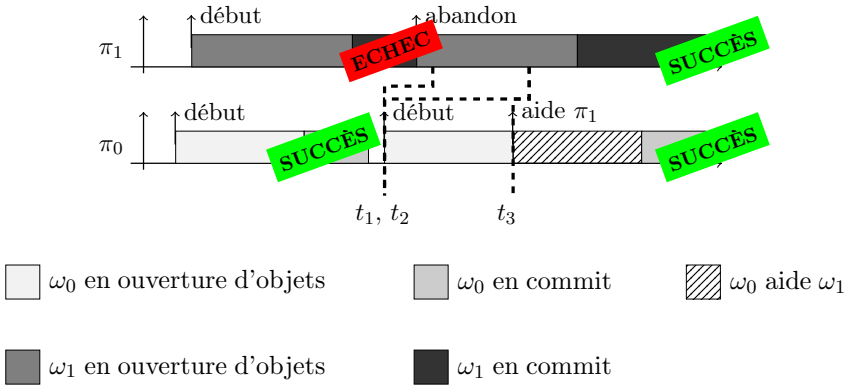


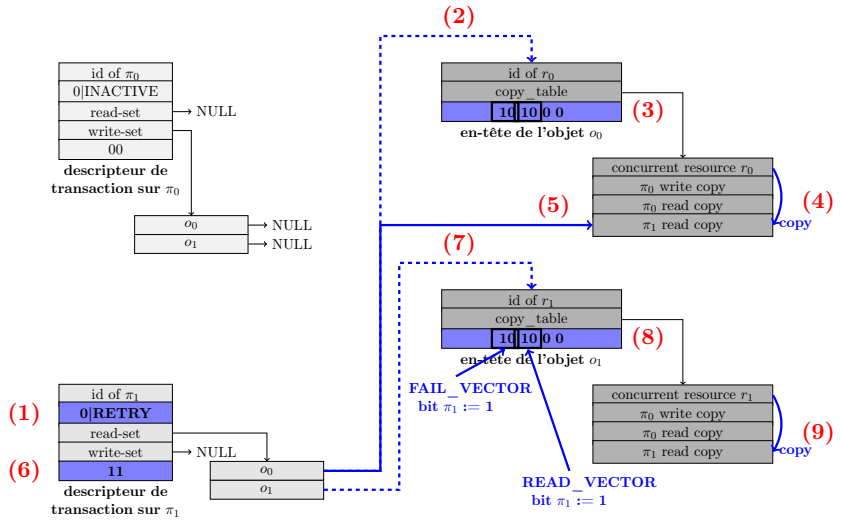
Figure D.6 – Transaction d'écriture aidant une transaction de lecture

par l'évolution de ω_1.

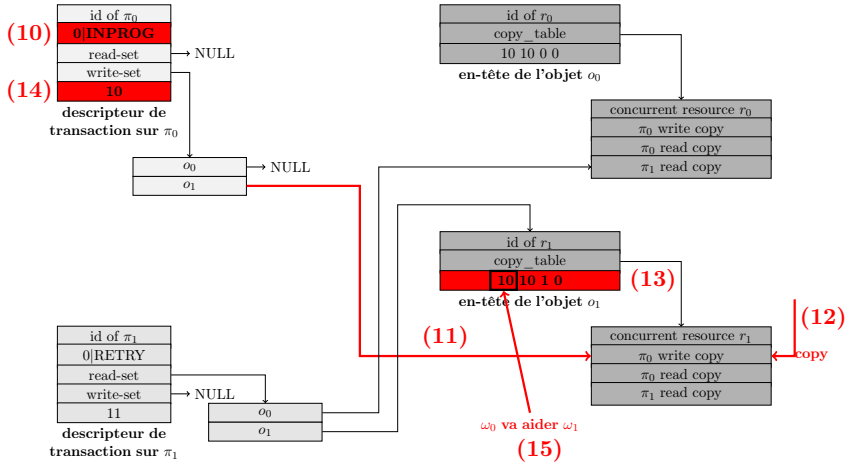
À l'instant t_1 (Figure D.7(a)), ω_1 recommence à s'exécuter après un premier abandon tandis que ω_0 n'a pas encore commencé sa seconde instance. Notons que le vecteur d'accès de ω_1 est déjà égal à 11 puisqu'il avait été rempli lors du premier essai et qu'il n'est pas réinitialisé après un échec. Tout d'abord, le statut de ω_1 est mis en *RETRY* (étape 1). ω_1 ouvre ensuite les objets o_0 puis o_1 de la même façon qu'illustré dans l'exemple de la Figure D.1(a). L'ouverture de o_0 est complète et correspond aux étapes (2), (3), (4), (5) et (6) tandis que l'ouverture de o_1 est représentée jusqu'à l'étape (9).

À l'instant t_2 (Figure D.7(b)), ω_0 démarre. On suppose que ω_1 reste suspendue temporairement. Au démarrage de ω_0, son statut passe à *INPROG* (étape 10). L'emplacement de la copie locale de o_1 est ensuite pointé par le pointeur de la table des accès en écriture correspondant à o_1 à l'étape (11). Le nouvel objet est ensuite copié dans cet emplacement à l'étape (12), l'*UPDATE_FLAG* de o_1 passe à 1 à l'étape (13) et le vecteur d'accès de ω_0 est mis à jour à l'étape (14). Au moment où ω_0 fait son commit, la mise à jour de l'objet est impossible car le *FAIL_VECTOR* de o_1 est différent de 0. ω_0 doit donc aider ω_1 à l'étape (15).

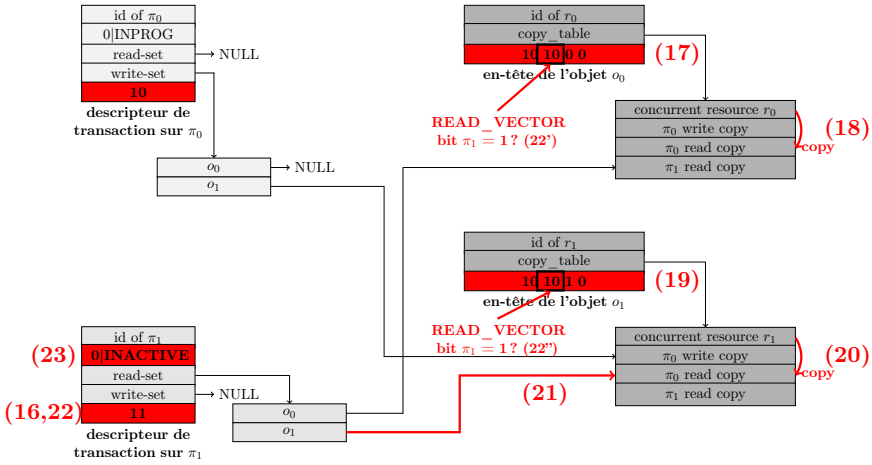
À partir de l'instant t_3 (Figure D.7(c)), ω_0 va effectuer toutes les opérations de ω_1 en son nom. On suppose toujours que ω_1 est suspendue temporairement. ω_0 commence par lire le vecteur d'accès de ω_1. Pour tous les objets marqués dans ce vecteur, ω_0 lit les objets. Elle commence donc par lire o_0 dans les étapes (17) et (18). Nous devons noter qu'à l'étape (18), la valeur courante de l'objet est copiée dans le champ réservé à la copie de ω_0. Puisque le champ correspondant à o_0 dans la table des accès en lecture de ω_1 pointe déjà vers une copie obtenue à l'étape (5), la valeur copiée par ω_0 ne sera pas prise en compte. Lors de l'ouverture de o_1, ω_0 réalise les étapes (19), (20) et (21). Notons également que l'objet courant est copié dans le champ réservé à la lecture par ω_0. Cependant, puisqu'à l'étape (21) le pointeur de la table des accès en lecture pour l'objet o_1 est encore dirigé sur l'en-tête de o_1, le pointeur est dirigé vers la copie lue par ω_0. Une fois les objets lus, ω_0 effectue le commit au nom de ω_1 en vérifiant la cohérence des objets (étapes 22, 22', 22") puis en mettant à jour le statut de ω_0 à l'étape (23). Notons enfin que l'instance de ω_0 n'est pas incrémentée par ω_1, car seule la transaction originale peut effectuer cette opération.



(a) ω_1 rejoue et ouvre o_0 et o_1 (instant t_1)



(b) ω_0 ouvre o_1 (instant t_2)



(c) ω_0 aide ω_1 (instant t_3)

Figure D.7 – Structures de données associées à l'exemple de la Figure D.6

Publications

Conférence internationale avec comité de lecture

S. Cotard, S. Faucou, J.L. Béchennec, A. Queudet and Y. Trinquet, *A Dataflow Monitoring Service Based on Runtime Verification for AUTOSAR*, In 9th International Conference on Embedded Software and Systems (ICESS), Liverpool, 2012.

Workshop international avec comité de lecture

S. Cotard, S. Faucou and J.L. Béchennec, *A Dataflow Monitoring Service Based on Runtime Verification for AUTOSAR OS : Implementation and Performances*, In 8th annual workshop Operating Systems Platforms for Embedded Real-Time applications (OSPERT), Pise, 2012.

Junior Workshop international

S. Cotard, *Runtime Verification for Real-Time Automotive Embedded Software*, In 10th school of Modelling and Verifying Parallel processes (MoVeP), Marseille, 2012.

S. Cotard, *Resource Management in Multicore Automotive Embedded Systems*, In 5th Junior Researcher Workshop on Real-Time Computing (JRWRTC), Nantes, 2011.

Workshop national

S. Cotard, *La Tolérance aux Fautes et les Systèmes Temps Réel Embarqués Multicoeur*, In école d'été Temps Réel (ETR), Brest, 2011.

Bibliographie

AFIS : http ://www.afis.fr. Association Française d'Ingénierie Système, Accès le, 11/10/2013.

Alexandersson, R. et Ohman, P. : On hardware resource consumption for aspect-oriented implementation of fault tolerance. *In Proceedings of the European dependable Computing Conference*, p. 61–66, 2010.

Amdahl, G. M. : Validity of the single processor approach to achieving large scale computing capabilities. *In Proceeding of the American Federation of Information Processing Societies, Spring Joint Computer Conference*, p. 483–485, 1967.

Anderson, J. H., Jain, R. et Ramamurthy, S. : *Implementing Hard Real-Time Transactions on Multiprocessors*, chap. 14, p. 247–260. Kluwer Academic Publishers, 1997a.

Anderson, J. H. et Moir, M. : Universal construction for multi-object operations. *In Proceedings of the 14th annual ACM symposium on Principles of distributed computing*, p. 184–193, 1995.

Anderson, J. H. et Moir, M. : Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, 1999.

Anderson, J. H., Ramamurthy, S. et Jain, R. : Implementing wait-free objects on priority-based systems. *In Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, p. 229–238, 1997b.

Andersson, B., Baruah, S. et Jonsson, J. : Static-priority scheduling on multiprocessors. *In Proceedings of the 22nd IEEE Real-Time Systems Symposium*, p. 193–202, 2001.

Anping, H., Jinzhao, W. et Lian, L. : An efficient algorithm for transforming ltl formula to büchi automaton. *In Proceedings of the International Conference on Intelligent Computation Technology and Automation*, p. 1215–1219, 2008.

ARINC : 651-1 design guidance for integrated modular avionics. Rapport technique, ARINC, 1997.

Arlat, J., Crouzet, Y., Deswarte, Y., Fabre, J.-C., Laprie, J.-C. et Powell, D. : *Encycloplédie de l'informatique et des systèmes d'information*, chap. Tolérance aux fautes, p. 240–270. Vuibert, 2006.

Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., Pasareanu, C., Rosu, G., Sen, K., Visser, W. et Washington, R. : Combining test case generation and runtime verification. *Theoretical Computer Science - Abstract state machines and high-level system design and analysis*, 336(2-3):209–234, 2005.

AUTOSAR : http ://www.autosar.org. AUTOSAR GbR, Accès le, 11/10/2013.

AUTOSAR : AUTOSAR - Specification of operating system. Rapport technique v4.1, AUTOSAR GbR, 2013a.

AUTOSAR : AUTOSAR - Specification of operating system for multicore. Rapport technique v4.1, AUTOSAR GbR, 2013b.

AUTOSAR : AUTOSAR - Specification of RunTime Environment. Rapport technique v4.1, AUTOSAR GbR, 2013c.

Baker, T. P. : A stack-based resource allocation policy for realtime processes. *In Proceedings of the 11th IEEE Real-Time Systems Symposium*, p. 191–200, 1990.

Balakrishnan, S., Rajwar, R., Upton, M. et Lai, K. : The impact of performance asymmetry in emerging multicore architectures. *In Proceedings of the 32nd annual international symposium on Computer Architecture*, p. 506–517, 2005.

Baruah, S. K., Cohen, N. K., Plaxton, C. G. et Varvel, D. A. : Proportionate progress : A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

Bauer, A., Leucker, M. et Schallhart, C. : Monitoring of real-time properties. *In Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science*, p. 260–272, 2006a.

Bauer, A., Leucker, M. et Schallhart, C. : Runtime reflection : Dynamic model-based analyis of component-based distributed embedded systems. *In Modellierung von Automotive Systems*, 2006b.

Bauer, A., Leucker, M. et Schallhart, C. : The good, the bad, and the ugly, but how ugly is ugly ? *In Proceedings of the 7th international conference on Runtime verification*, p. 126–138, 2007.

Bauer, A., Leucker, M. et Schallhart, C. : Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4), 2011.

Béchennec, J.-L., Briday, M., Faucou, S. et Trinquet, Y. : Trampoline - an open source implementation of the osek/vdx rtos specification. *In Proceedings of the International Conference on Emerging Technologies and Factory Automation*, p. 62–69, 2006.

Beizer, B. : *Software Testing Techniques*. Intl Thomson Computer Pr (T), 2nd edition édition, 1990.

Belwal, C. et Cheng, A. : Lazy versus eager conflict detection in software transactional memory : A real-time schedulability perspective. *IEEE Embedded Systems Letters*, 3(1):37– 41, 2011.

Bertrand, D. : *Contribution à la Robustesse des Systèmes Temps Réel Embarqués*. Thèse de doctorat, Université de Nantes, 2011.

Bertrand, D., Faucou, S. et Trinquet, Y. : An analysis of the autosar os timing protection mechanism. *In IEEE Conference on Emerging Technologies and Factory Automation*, p. 1–8, 2009.

Blake, G., Dreslinski, R. G. et Mudge, T. : A survey of muticore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, 2009.

Block, A., Leontyev, H., Brandenburg, B. B. et Anderson, J. H. : A flexible real-time locking protocol for multiprocessors. *In Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, p. 47–56, 2007.

Brandenburg, B. B., Calandrino, J., Block, A., Leontyev, H. et Anderson, J. : Real-time synchronization on multiprocessors : To block or not to block, to suspend or spin ? *In Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, p. 342–353, 2008.

Brandenburg, B. B. et Anderson, J. H. : An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS RT. *In Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, p. 185–194, 2008.

Büchi, J. : On a decision method in restricted second order arithmetic. *In Proceeding of the International Congress on Logic, Method, and Philosophy of Science*, p. 1–12, 1962.

Bucur, D. : Temporal monitors for TinyOS. *In Runtime Verification*, p. 96–109, 2012.

Calandrino, J. M., Leontyev, H., Block, A., Devi, U. C. et Anderson, J. H. : LITMUS RT : A testbed for empirically comparing real-time multiprocessor schedulers. *In Proceedings of the 27th IEEE International Real-Time Systems Symposium*, p. 111–126, 2006.

Candea, G. : The basics of dependability. *In Proceedings of the Symposium on Principles of Database Systems*, 2007.

Cassez, F. et Béchennec, J.-L. : Timing analysis of binary programs with UPPAAL. *In Proceedings of the 13th International Conference on Application of Concurrency to System Design*, p. 41–50, 2013.

CEI 61508 : International standard IEC 61508 : Functional safety of electrical/electronic/programmable electronic safety-related systems (E/E/PES). Rapport technique, International Electrotechnical Commission, 1999.

Chen, J. et Burns, A. : A fully asynchronous reader/writer mechanism for multiprocessor real-time systems. Rapport technique, Department of Computer Science, University of York, 1997.

Cho, H., Ravindran, B. et Jensen, E. : Space-optimal, wait-free real-time synchronization. *IEEE Transactions on Computers*, 56(3):373– 384, 2007.

Clarke, E. M. et Emerson, E. A. : Design and synthesis of synchronization skeletons using branching-time temporal logic. *In Proceeding of the Logic of Programs*, p. 52–71, 1982.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. et Quesada, J. F. : The maude system. *In Proceeding of the 10th International Conference Rewriting Techniques and Applications*, p. 240–243, 1999.

D'Angelo, B., Sankaranarayanan, S., Sanchez, C., Robinson, W., Finkbeiner, B., Sipma, H., Mehrotra, S. et Manna, Z. : Lola : runtime monitoring of synchronous systems. *In Proceedings of the 12th International Symposium on Temporal Representation and Reasoning*, p. 166–174, 2005.

Dechev, D., Pirkelbauer, P. et Stroustrup, B. : Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. *In Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, p. 185–192, 2010.

Déplanche, A.-M. et Cottet, F. : *Encyclopédie de l'Informatique*, chap. Ordonnancement temps réel et ordonnançabilité d'une application. Vuibert Informatique, 2006.

Deursen, A. v., Klint, P. et Visser, J. : Domain-specific languages : an annotated bibliography. *Newsletter ACM SIGPLAN Notices*, 35(6):26–36, 2000.

Dhall, S. et Liu, C. : On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

Dice, D., Shalev, O. et Shavit, N. : Transactional locking II. *In Proceedings of the 20th international conference on Distributed Computing*, p. 194–208, 2006.

DO-178B : Software considerations in airborne systems and equipment certification (rtca do-178b). Rapport technique, RTCA Inc., 1992.

Dwyer, M. B., Avrunin, G. S. et Corbett, J. C. : Patterns in property specifications for finite-state verification. *In Proceedings of the International Conference on Software Engineering*, p. 411–420, 1999.

EASIS : Deliverable d0.1.2 - electronic architecture and system engineering for integrated safety systems. Rapport technique, EASIS, 2006.

Easwaran, A. et Andersson, B. : Resource sharing in global fixed-priority preemptive multiprocessor scheduling. *In Proceedings of the 30th IEEE Real-Time Systems Symposium*, p. 377–386, 2009.

ED-12B : ED-12B - software considerations in airborne systems and equipment certification. Rapport technique, EUROCAE, 1999.

EGAS : Standardized e-gas monitoring concept for engine management systems of gasoline and diesel engines. Rapport technique, BMW, DaimlerChrysler, Volkswagen, Porche, Audi, 2004.

Emerson, E. A. et Clarke, E. M. : Characterizing correctness properties of parallel programs using fixpoints. *Automata, Languages and Programming*, 85:169–181, 1980.

Eswaran, K., Gray, J., R.A., L. et I.L., T. : The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

Falcone, Y., Fernandez, J.-C. et Mounier, L. : Runtime verification of safety-progress properties. *In Proceedings of the International Workshop on Runtime Verification*, p. 40–59, 2009.

Falcone, Y., Fernandez, J.-C. et Mounier, L. : What can you verify and enforce at runtime ? *International Journal on Software Tools for Technology Transfer - Runtime Verification*, 14 (3):349–382, 2010.

Fetzer, C. et Felber, P. : Transactional memory for dependable embedded systems. *In Proceedings of the International Conference on Dependable Systems and Networks Workshops*, p. 223–227, 2011.

Fraser, K. : Practical lock-freedom. Rapport technique, University of Cambridge, 2004.

Freescale : Embedded multicore : An introduction - white paper, 2009.

Freescale : *MPC5643L Microcontroller - Reference Manual*, 2010.

Fujita, M., Tanaka, H. et Moto-oka, T. : Logic design assistance with temporal logic. *In Proceedings of the 7th International Symposium on Computer Hardware Description Languages and Their Applications*, p. 129–138, 1985.

Gai, P., Lipari, G. et Di Natale, M. : Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. *In Proceedings of the 22nd IEEE Real-Time Systems Symposium*, p. 73–83, 2001.

Gastin, P. et Oddoux, D. : Fast LTL to Büchi automata translation. *In Proceedings of the International Conference on Computer Aided Verification*, p. 53–65, 2001.

Gerth, R., Peled, D., Vardi, M. Y. et Wolper, P. : Simple on-the-fly automatic verification of linear temporal logic. *In Proceedings of the 15th IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, p. 3–18, 1996.

Goodenough, J. et Sha, L. : The priority ceiling protocol : A method for minimizing the blocking of high priority ada tasks. *In Proceedings of the 2nd international workshop on Real-time Ada issues*, p. 20–31, 1998.

Guerraoui, R., Herlihy, M. et Pochon, B. : Polymorphic contention management. *In Proceedings of the 19th international conference on Distributed Computing*, p. 303–323, 2005a.

Guerraoui, R., Herlihy, M. et Pochon, B. : Toward a theory of transactional contention managers. *In Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, p. 258–264, 2005b.

Ha, R. et Liu, J. : Validating timing constraints in multiprocessor and distributed real-time systems. *In Proceedings of the 14th International Conference on Distributed Computing Systems*, p. 162–171, 1994.

Haerder, T. et Reuter, A. : Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

Hari, S. K. S. : Low-cost hardware fault detection and diagnosis for multicore systems running multithreaded workloads. Mémoire de D.E.A., University of Illinois, 2009.

Hari, S., Li, M.-L., Ramachandran, P., Choi, B. et Adve, S. : mSWAT : Low-cost hardware fault detection and diagnosis for multicore systems. *In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, p. 122–132, 2009.

Harris, T. et Fraser, K. : Language support for lightweight transactions. *Special Issue : Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, 38(11):388–402, 2003.

Harris, T., Larus, J. R. et Rajwar, R. : *Transactional Memory*. Morgan and Claypool, 2nd édition, 2010.

Havelund, K. et Goldberg, A. : Verify your runs. *Verified Software : Theories, Tools, Experiments - Lecture Notes in Computer Science*, 4171:374–383, 2008.

Havelund, K. et Rosu, G. : Java pathexplorer - a runtime verification tool. *In Proceedings of the 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2001a.

Havelund, K. et Rosu, G. : Monitoring java programs with java pathexplorer. Rapport technique, Research Institute for Advanced Computer science, 2001b.

Havelund, K. et Rosu, G. : Synthesizing monitors for safety properties. *In Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, p. 342–356, 2002.

Herlihy, M. : Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

Herlihy, M. : A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

Herlihy, M., Luchangco, V., Moir, M. et Scherer III, W. N. : Software transactional memory for dynamic-sized data structures. *In Proceedings of the ACM Symposium on Principles of Distributed Computing*, p. 92–101, 2003.

Herlihy, M., Eliot, J. et Moss, B. : Transactional memory : Architectural support for lock-free data structures. *In Proceedings of the 20th Annual International Symposium on Computer Architecture*, p. 289–300, 1993.

Herlihy, M. et Moss, J. E. B. : Transactional memory : architectural support for lock-free data structures. *In Proceedings of the 20th annual international symposium on computer architecture*, p. 289–300, 1993.

Hladik, P.-E., Déplanche, A.-M., Faucou, S. et Trinquet, Y. : Adequacy between AUTOSAR OS specification and real-time scheduling theory. *In Proceedings of the IEEE International Symposium on Industrial Embedded Systems*, p. 225–233, 2007.

Holman, P. et Anderson, J. H. : Supporting lock-free synchronization in pfair-scheduled real-time systems. *Journal of Parallel and Distributed Computing*, 66(1):47–67, 2006.

Holzmann, G. J. : The model checker spin. *IEEE Transactions on Software Engineering - Special issue on formal methods in software practice*, p. 279–295, 1997.

Hursch, W. L. et Lopes, C. V. : Separation of concerns. Rapport technique, College of Computer Science, Northeastern University, Boston, 1995.

ISO 26262 : Automative standard. Rapport technique, Technical Comity ISO, 2011.

ISO 9000 : Systèmes de management de la qualité - principes essentiels et vocabulaire. Rapport technique, Technical Comity ISO, 2005.

Joseph, M. et Pandya, P. : Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

Kantz, H. et Koza, C. : The elektra railway signalling system : field experience with an actively replicated system with diversity. *In 25th International Symposium on Fault-Tolerant Computing*, p. 453–458, 1995.

Kim, M., Lee, I., Sammapun, U., Shin, J. et Sokolsky, O. : Monitoring, checking, and steering of real-time systems. *In Proceedings of the International Workshop on Runtime Verification*, p. 95–111, 2002.

Kripke, S. : Semantical considerations on modal logic. *In Proceedings of the Acta Philosophica Fennica*, p. 83–94, 1963.

Kumar, R., Zyuban, V. et Tullsen, D. M. : Interconnections in multi-core architectures : Understanding mechanisms, overheards and scaling. *In Proceedings of the 32nd annual international symposium on Computer Architecture*, p. 408–419, 2005.

Laprie, J.-C. : *Software Fault Tolerance*, chap. 3 - Architectural Issues in Software Fault Tolerance, p. 47–80. John Wiley & Sons Ltd, 1995.

Laprie, J.-C., Arlat, J., Blanquart, J.-P., Costes, A., Crouzet, Y., Deswarte, Y., Fabre, J.-C., Guillermain, H., Kaâniche, M., Kanoun, K., Mazet, C., Powell, D., Rabéjac, C. et Thévenod, P. : *Guide de la sureté de fonctionnement*. Cépaduès Editions, 1995.

Leucker, M. et Schallhart, C. : A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

Leung, J. et Whitehead, J. : On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237—250, 1982.

Li, M.-L., Ramachandran, P., Sahoo, S. K., Adve, S. V., Adve, V. S. et Zhou, Y. : Understanding the propagation of hard errors to software and implications for resilient system design. *In Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, p. 265–276, 2008.

Liu, C. : Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary*, 2:28–31, 1969.

Liu, C. et Layland, J. : Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 46-61, 1973.

Liu, J. W. S. : *Real-Time Systems*. Prentice Hall, 2000.

Lu, C. : *Robustesse du logiciel embarqué multicouche par une approche réflexive : Application à l'automobile*. Thèse de doctorat, Université de Toulouse, France, 2009.

Lu, C., Fabre, J.-C. et Killijian, M.-O. : An approach for improving fault-tolerance in automotive modular embedded software. *In Proceedings of the International Conference on Real-Time Networked Systems*, p. 132–147, 2009.

Maes, P. : Concepts and experiments in computational reflection. *In Proceedings of the Conference on Object-oriented programming systems, languages and applications*, p. 147–155, 1987.

Marathe, V. J., Scherer III, W. N. et Scott, M. L. : Adaptive software transactional memory. *In Proceedings of the 19th International Symposium on Distributed Computing*, p. 354–368, 2005.

Marshall, C. : Software transactional memory. Rapport technique, University of California, December 2005.

McGregor, J. D. : Dependability. *Journal of Object Technology*, 6:7–12, 2007.

McMillan, K. L. : *Symbolic Model Checking*. Kluwer Academic, 1993.

Meawad, F., Iyer, K. et Vitek, J. : Real-time wait-free queues using micro-transactions. *In Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, p. 1–10, 2011.

Moir, M. : Transparent support for wait-free transactions. *In Proceedings of the 11th International Workshop on Distributed Algorithms*, p. 305–319, 1997.

Moore, G. E. : Cramming more components onto integrated circuits. *Electronics*, 1965.

Myers, G. J. : *The Art of Software Testing*. Wiley, 2nd edition édition, 1979.

NF E 01-010 : Nf e 01-010 mécatronique – vocabulaire. Rapport technique, AFNOR, 2008.

Oddoux, d. : *Utilisation des automates alternants pour un model-checking efficace des logiques temporelles linéraire*. Thèse de doctorat, Université de Paris 7, 2003.

OSEK/VDX : OSEK/VDX - Communication. Rapport technique v3.0.3, OSEK Group, 2005a.

OSEK/VDX : OSEK/VDX - Operating system. Rapport technique v2.2.3, OSEK Group, 2005b.

OSEK/VDX : OSEK/VDX - Osek implementation language. Rapport technique v2.5, OSEK Group, 2005c.

Pattabiraman, K., Kalbarczyk, Z. et Iyer, R. : Application-based metrics for strategic placement of detectors. *In Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing*, p. 75–82, 2005.

Patzina, L., Patzina, S., Piper, T. et Manns, P. : Model-based generation of run-time monitors for autosar. *In Proceedings of the 9th European conference on Modelling Foundations and Applications*, p. 70–85, 2013.

Peters, D. K. : Automated testing of real-time systems. Rapport technique, Memorial University of Newfoundland, 1999.

Pike, L., Goodloe, A., Morisset, R. et Niller, S. : Copilot : A hard real-time runtime monitor. *In Proceedings of the International Conference on Runtime Verification*, p. 345–359, 2010.

Pnueli, A. : The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science*, p. 46–57, 1977.

Queille, J.-P. et Sifakis, J. : Specification and verification of concurrent systems in cesar. *In Proceedings of the 5th Colloquium on International Symposium on Programming*, p. 337–351, 1982.

Rabin, M. O. et Scott, D. : Finite automata and their decision problems. *IBM J. Res. Develop.*, 3:114–125, 1959.

Rajkumar, R. : *Synchronization in real-time systems : a priority inheritance approach.* Kluwer Academic Publishers, 1991.

Randell, B. et Xu, J. : *Software Fault Tolerance*, chap. 1 - The Evolution of the Recovery Block concept, p. 1–21. John Wiley & Sons Ltd, 1995.

Rashid, L., Pattabiraman, K. et Gopalakrishnan, S. : Modeling the propagation of intermittent hardware faults in programs. *In Proceedings of the 16th IEEE Pacific Rim International Symposium on Dependable Computing*, p. 19–26, 2010a.

Rashid, L., Pattabiraman, K. et Gopalakrishnan, S. : Towards understanding the effects of intermittent hardware faults on programs. *In Proceedings of the International Conference on dependable Systems and Networks Workshops*, p. 101–106, 2010b.

Reis, G. A., Chang, J., Vachharajani, N., Rangan, R. et August, D. I. : SWIFT : software implemented fault tolerance. *Proceedings of the International Symposium on Code Generation and Optimization 2005*, p. 243–254, 2005a.

Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., August, D. I. et Mukherjee, S. S. : Software-controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization*, p. 1–28, 2005b.

Robert, T., Roy, M. et Fabre, J.-C. : Early error detection for fault tolerance strategies. *In Proceedings of the International Conference on Real-Time and Network Systems*, p. 159–168, 2010.

Rosenblum, D. S. : A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

Sangiovanni-Vincentelli, Alberto et Di Natale, M. : Embedded system design for automotive applications. *IEEE Computer Society*, 40:42–51, 2007.

Sarni, T., Queudet, A. et Valduriez, P. : Real-time support for software transactional memory. *In Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, p. 477–485, 2009a.

Sarni, T., Queudet, A. et Valduriez, P. : Transactional memory : Worst case execution time analysis. *In Proceedings of the 17th IEEE International Conference on Real-Time and Network Systems*, 2009b.

SCARLET : Systèmes critiques pour l 'automobile : Robustesse des logiciels embarqués temps-réels - l1.1, l2. Rapport technique, CEA, LIST, LAAS, LORIA, IRCCyN, Renault, Trialog, Valeo, 2007.

Scherer III, W. N. et Scott, M. : Contention management in dynamic software transactional memory. *In Proceedings of the Workshop on Concurrency and Synchronization in Java programs*, 2004.

Scherer III, W. N. et Scott, M. : Advanced contention management for dynamic software transactional memory. *In Proceedings of the 24th annual ACM symposium on Principles of distributed computing*, p. 240–248, 2005.

Schoeberl, M., Br, F. et Vitek, J. : RTTM : Real-time transactional memory. *In Proceedings of the 25th ACM Symposium on Applied Computing*, p. 326–333, 2010.

Scott, R., Gault, J. W. et McAllister, D. F. : Fault-tolerant software reliability modeling. *IEEE Transactions on Software Engineering*, SE-13(5):582–592, 1987.

Sha, L., Abdelzaher, T., Arzén, K., Cervin, A., Baker, T., Burns, A., Buttazo, G., Caccamo, M., Lehoczky, J. et Mok, A. : Real time scheduling theory : A historical perspectives. *Journal of Real-Time Systems*, 28(2-3):101–155, 2004.

Sha, L., Rajkumar, R. et Lehoczky, J. P. : Priority inheritance protocols : An approach to real-time synchronization. *In Proceedings of the IEEE Transactions on Computers*, p. 1175–1185, 1990.

Shavit, N. et Touitou, D. : Software transactional memory. *In Proceedings of the 14th annual ACM symposium on Principles of distributed computing*, p. 204–213, 1995.

Shye, A., Blomstedt, J., Moseley, T., Reddi, V. J. et Connors, D. A. : PLR : A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing*, 6(2):135–148, 2009.

Song, O. et Choi, C. : Wait-free data sharing between periodic tasks in multiprocessor control systems. *Control Engineering Practice*, 11(6):601–611, 2003.

Srinivasan, A. et Baruah, S. : Deadline-based scheduling of periodic task systems on multiprocessors. *Journal Information Processing Letters*, 84(2):93–98, 2002.

Taiani, F., Killijian, M.-O. et Fabre, J.-C. : Intergiciels pour la tolérance aux fautes. *Revue des Sciences et Technologies de l'Information, série Techniques et Sciences Informatiques*, 25:599–630, 2006.

Tarjan, R. : Depth-first search and linear graph algorithms. *In Proceedings of the 12th Annual Symposium on Switching and Automata Theory*, p. 114–121, 1971.

Torres-Pomales, W. : Software fault tolerance : A tutorial. Rapport technique, NASA, 2000.

Toshinori, S. et Funaki, T. : Dependability, power, and performance trade-off on a multicore processor. *In Proceedings of the Asia and South Pacific Design Automation Conference*, p. 714–719, 2008.

Tsai, J. J. P., Fang, K.-Y., Chen, H.-Y. et Bi, Y.-D. : A noninterference monitoring and replay mechanism for real-time software testing and debugging. *Journal IEEE Transactions on Software Engineering*, 16(8):897–916, 1990.

Tsigas, P. et Zhang, Y. : Non-blocking data sharing in multiprocessor real-time systems. *In Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, p. 247–254, 1999.

UPPAAL : http ://www.uppaal.org/. Department of Information Technology, Uppsala University, Sweden and Department of Computer Science, Aalborg University, Denmark, Accès le, 11/10/2013.

Wamhoff, J.-T. et Fetzer, C. : The universal transactional memory construction. *In Proceedings of the 6th Workshop on Transactional Computing*, 2011.

Wamhoff, J.-T., Riegel, T., Fetzer, C. et Felber, P. : RobuSTM : a robust software transactional memory. *In Proceedings of the 12th international conference on Stabilization, safety, and security of distributed systems*, p. 388–404, 2010.

Wang, X., Ji, Z., Fu, C. et Hu, M. : A review of transactional memory in multicore processors. *Information Technology*, 9(1):192–197, 2010.

Watterson, C. et Heffernan, D. : Runtime verification and monitoring of embedded systems. *Software, Institution of Engineering and Technology*, 1(4):172–179, 2007.

Weyuker, E. J. : The oracle assumption of program testing. *In Proceedings of the 13th International Conference on System Sciences*, p. 44–49, 1980.

Wolf, W., Jerraya, A. A. et Martin, G. : Multiprocessor system-on-chip (mpsoc) technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27:1701–1713, 2008.

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Sylvain COTARD

Contribution à la robustesse des systèmes temps réel embarqués multicœur automobile

Robustness in Multicore Automotive Embedded Real-Time Systems

Résumé

Les besoins en ressources CPU dans l'automobile sont en constante augmentation. Le standard de développement logiciel AUTOSAR (*AUTomotive Open System ARchitecture*) – développé au sein d'un consortium regroupant des fabricants de véhicules et des sous-traitants – offre désormais la possibilité de s'orienter vers de nouvelles architectures : les microcontrôleurs multicœur. Leur introduction au sein des systèmes embarqués critiques apporte un lot de problèmes allant à l'encontre des objectifs de sûreté de fonctionnement ISO 26262. Par exemple, le parallélisme des cœurs impose de maîtriser l'ordonnancement pour respecter les contraintes de dépendance entre les tâches, et le partage des données intercœur doit être effectué en assurant leur cohérence.

Notre approche s'articule en deux volets. Pour vérifier les contraintes de dépendance entre les tâches, les exigences sur les flots de données sont utilisées pour synthétiser des moniteurs à l'aide de l'outil *Enforcer*. Un service de vérification en ligne utilise ces moniteurs (injectés dans le noyau du système d'exploitation) pour vérifier le comportement du système. Enfin, pour maîtriser le partage des données intercœur, nous proposons une alternative aux protocoles bloquants. Le protocole wait-free STM-HRT (*Software Transactional Memory for Hard Real-Time systems*), est conçu sur les principes des mémoires transactionnelles afin d'améliorer la robustesse des systèmes.

Mots clés

systèmes temps réel, sûreté de fonctionnement, vérification en ligne, détection d'erreurs, flots de données, mémoire transactionnelle, wait-free, multicœur.

Abstract

The feature content of new cars is increasing dramatically, which involves more demanding requirements. AUTOSAR (*AUTomotive Open System ARchitecture*) – an open and standardized automotive software architecture jointly developed by vehicle makers, suppliers and tool developers – enables to benefit from new hardware technology by considering multicore architectures. Despite their undeniable advantages in terms of performance, multicore architectures bring a set of problems that may jeopardize the dependability of embedded applications (ISO 26262 is the dependability standard for automotive). For example, multicore scheduling requires to cope with task dependencies constraints, and data consistency must be ensured for inter core communication.

First, to meet the task dependencies, safety requirements on data flows are used to generate monitors. The tool *Enforcer* has been built in order to do that. A service based on runtime verification uses these monitors (injected in the kernel of the operating system) checks if the system behaves as expected. Finally, we propose an alternative locking mechanism for inter core data sharing. The wait-free protocol STM-HRT (*Software Transactional Memory for Hard Real-Time systems*), based on transactional memory, aims at increasing the robustness of multicore applications.

Key Words

real-time systems, safety, runtime verification, error detection, data flow, transactional memory, wait-free, multicore.