Configuration et Reconfiguration des Systèmes Temps-Reél Répartis Embarqués Critiques et Adaptatifs

Etienne Borde

TELECOM ParisTech

École Doctorale
d'Informatique,
Télécommunications
et Électronique de Paris

Thèse

Configuration et Reconfiguration des Systèmes Temps-Réel Répartis Embarqués Critiques

présentée et soutenue publiquement le 01 Décembre 2009

pour l'obtention du

Doctorat de l'École Nationale Supérieure des Télécommunications

spécialité : Informatique et Réseaux

par

Étienne Borde

Composition du jury

Président :	Dr. Bertrand Braunschweig	ANR - Agence Nationale de la Recherche
Rapporteurs :	Pr. Peter H. Feiler	SEI - Software Engineering Institute
	Pr. Lionel Seinturier	LIFL - Laboratoire d'Informatique de Lille
Examinateurs :	Pr. Jean-Charles Fabre	INPT - Institut National Polytechnique de Toulouse
	Pr. Xavier Blanc	LIP6 - Laboratoire d'Informatique de Paris 6
Directeurs de thèse :	Pr. Laurent Pautet	TelecomParisTech
	Dr. Grégory Haïk	Thales

Á ma famille,

Á mes amis.

Remerciements

4h45. A la barre, Michel vérifie les données de ses instruments de navigation. Il le sait, la ligne d'arrivée est toute proche. Peu importe sa position. Cette course en solitaire lui aura donné les frissons qu'il espérait. Il pense alors à ceux qui l'ont soutenu, ceux qui lui ont permis d'accomplir ce défit.

Le moment tant espéré est enfin arrivé. Je vais à mon tour mettre un point final au travail de recherche sur lequel mon esprit navigue depuis trois ans. A mon tour, je tiens à remercier ceux sans qui cela n'aurait pas été possible.

Laurent Pautet et Grégory Haïk, qui ont su à la fois orienter et propulser mes travaux. Je vous remercie pour votre franchise, votre soutien et votre confiance. J'espère avoir souvent l'occasion de collaborer avec des personnes de votre qualité.

Je tiens également à remercier Peter Feiler et Lionel Seinturier pour avoir accepté la responsabilité de rapporteurs de ce mémoire. Les échanges que nous avons eu ont permis de le consolider. Mes remerciements vont également à Bertrand Braunschweig, Jean-Charles Fabre, et Xavier Blanc pour l'intérêt qu'ils ont porté à mes travaux et pour avoir accepté de faire partie du jury de ma soutenance de thèse.

Le résultat de ces trois ans de travail est le fruit de nombreuses discussions et collaborations avec des ingénieurs de la société Thales. Fédéric Gilliers, Jérôme Chauvin, Frédéric Legusquet, Hugues Balp, Thomas Vergnaud, Pascal Llorens, Olivier Hachet, Virginie Watine, Vincent Seignole, Jean-Louis Gilbert et Thomas Derive. Vous avez tous, plus ou moins directement, contribué à cette réalisation. Mes amis et collègues de TelecomParisTech, vous m'avez permis d'apprécier à sa juste valeur le travail de recherche : Julien Delange, Gilles Lasnier, Xavier Gréhant, Olivier gilles, Hoa Ha Duong, Sébastien Gardoll, sans oublier Béchir Zalila, Irfan Hamid, et Jérôme Hugues.

Enfin, je voudrais remercier les membres de ma famille, ainsi que mes amis, pour leurs encouragements chaleureux et leur soutien indéfectible.

A vous tous, merci de m'avoir permis de mener cette aventure à bon port.

Résumé

Aujourd'hui, de plus en plus de systèmes industriels s'appuient sur des applications logicielles temps-réel réparties embarquées (TR2E). La réalisation de ces applications demande de répondre à un ensemble important de contraintes très hétérogènes, voire contradictoires. Pour satisfaire ces contraintes, il est presque toujours nécessaire de fournir à ces systèmes des capacités d'adaptation.

Par ailleurs, certaines de ces applications pilotent des systèmes dont la défection peut avoir des conséquences financières - voire humaines - dramatiques. Pour concevoir de telles applications, appelées applications critiques, il faut s'appuyer sur des processus de développement rigoureux capables de repérer et d'éliminer les erreurs de conception potentielles.

Malheureusement, il n'existe pas à notre connaissance de processus de développement capable de traiter ce problème dans le cas où l'adaptation du système à son environnement conduit à modifier sa configuration logicielle.

Ce travail de thèse présente une nouvelle méthodologie qui répond à cette problématique en s'appuyant sur la notion de mode de fonctionnement : chacun des comportements possibles du système est représenté par le biais d'un mode de fonctionnement auquel est associé une configuration logicielle. La spécification des règles de transition entre ces modes de fonctionnement permet alors de générer l'implantation des mécanismes de changement de mode, ainsi que des reconfigurations logicielles associées. Le code ainsi produit respecte les contraintes de réalisation des systèmes critiques et implante des mécanismes de reconfiguration sûrs et analysables.

Pour ce faire, nous avons défini un nouveau langage de description d'architecture (COAL : Component Oriented Architecture Language) qui permet de bénéficier à la fois des avantages du génie logiciel à base de composants (de type Lightweight CCM), et des techniques d'analyse, de déploiement et de configuration statique, qu'apporte l'utilisation des langages de description d'architecture (et en particulier AADL : Architecture Analysis and Description Language). Nous avons alors réalisé un nouveau framework à composant, MyCCM-HI (Make your Component Container Model - High Integrity), qui exploite les constructions de COAL pour (i) générer le modèle AADL permettant de réaliser le déploiement et la configuration statique de l'application TR2E, (ii) générer le code de déploiement et de configuration des composants logiciels de type Lightweight CCM, (iii) générer le code correspondant aux mécanismes d'adaptation du système, et (iv) analyser formellement le comportement du système, y compris en cours d'adaptation.

Ce framework à composant est disponible au téléchargement à l'adresse http ://myccm-hi.sourceforge.net.

Mots-clés: systèmes embarqués, systèmes temps-réels, systèmes adaptatifs, systèmes critiques, langages de description d'architecture, *framework* à composant, génération de code, COAL, MyCCM-HI, AADL, Lightweight CCM

Abstract

Nowadays, more and more industrial systems rely on distributed real-time embedded software (DRES) applications. Implementing such applications requires answering to an important set of heterogeneous, or even conflicting, constraints. To satisfy these constraints, it is sometimes necessary to equip DRES with adaptation capabilities.

Moreover, real-time applications often control systems of which failures can have dramatic economical – or worst human – consequences. In order to design such application, named critical applications, it is necessary to rely on rigorous methodologies, of which certain have already been used in industry. However, growth complexity of critical DRES applications requires proposing always new methodologies in order to answer to all of these stakes.

Yet, as far as we know, existing design processes do not tackle the issue of adaptation mechanisms that require to modify deeply the software configuration.

This PhD thesis work presents a new methodology that answers this problem by relying on the notion of operational mode : each possible behaviour of the system is represented by an operational mode, and a software configuration is associated to this mode. Modeling transition rules betwen these modes, it becomes possible to generate and analyze the reconfigurations of the software architecture that implement the system adaptations. The generated code respect the implementation requirements of critical systems, and relies on safe and analyzable adaptation mechanisms.

To achieve this objective, we define a new architecture description language (COAL : Component Oriented Architecture Language), specific to this domain, that enables to profit from advantages of component-based software engineering (based on Lightweight CCM), and analysis, static deployment and configuration techniques that provides architecture description languages (and in particular AADL : Architecture Analysis and Design Language). This methodology also relies on a new component framework, MyCCM-HI (Make your Component Container Model - High Integrity), that exploits COAL constructs so as to (i) generate AADL models enabling static deployment and configuration of DRES applications, (ii) generate code to deploy and configure Lightweight CCM components, (iii) generate code implementing the system adaptation mechanisms, and (iv) formally analyse the behaviour of the system, including during adaptation.

The adopted approach thus reduces complexity of development of adaptative and critical DRES by automating production of adaptation mechanisms while easing their analysis. These two steps, analysis and production, are then part of the automatic production tool chain provided by MyCC-HI.

This component framework is availabe under (L)GPL license at address http ://myccm-hi.sourceforge.net.

Keywords: embedded systems, real-time systems, adaptative systems, critical systems, architecture description lnaguages, component framework, code generation, COAL, MyCCM-HI, AADL, Lightweight CCM

Table des matières

Liste des illustrations

Liste des exemples de code

Liste des tableaux

Première partie

Introduction Générale

Chapitre 1

Introduction

Demain est moins à découvrir qu'à inventer.
Gaston Berger

SOMMAIRE

Dans ce chapitre introductif, nous présentons à la fois le contexte général de notre étude, et une synthèse des problématiques et contributions que nous détaillerons dans la suite de ce mémoire. [1]

1.1 Présentation du contexte général

L'informatique industrielle est en pleine expansion. Avec l'accroissement de la puissance de calcul et la miniaturisation des puces électroniques, c'est toute l'industrie qui est aujourd'hui concernée par cette révolution technologique.

Une caractéristique importante des applications informatiques industrielles est qu'elles doivent se conformer à des exigences hétérogènes (exigences de fiabilité, d'occupation physique, de performance, de consommation énergétique, etc...). Ces exigences sont souvent contradictoires et dépendent du contexte environnemental dans lequel évolue le système :

1. Le travail que nous présentons dans ce mémoire a été réalisé dans le cadre du projet Flex-eWare, financé par L'Ágence Nationale de la Recherche.

baisser la consommation énergétique diminue les performances du système, mais un système devra baisser sa consommation énergétique si son niveau de batterie devient insuffisant.

Nous appelons cela un comportement adaptatif : le système modifie son comportement en cours d'utilisation pour répondre aux variations des caractéristiques de son utilisation. Une panne matérielle (ou logicielle), une étape dans le cycle de vie du système, une modification des ressources disponibles (bande passante ou niveau de charge de la batterie par exemple), un besoin de maintenance, une requête utilisateur, sont autant d'exemples de situations qui nécessitent d'adapter le comportement d'un système.

Ainsi, la spécification des modes de fonctionnement d'un système constitue le plus souvent une des premières étapes du processus de développement d'un système. Faute de méthodologie outillée pour réutiliser cette spécification lors de la conception logicielle, l'implantation des mécanismes d'adaptation est le plus souvent faite manuellement. Le code ainsi développé est difficile à maintenir, et le comportement associé à ces mécanismes d'adaptation est d'autant plus difficile à analyser qu'il est enfoui dans le code de l'ensemble de l'application.

1.1.1 Proposition

Le travail de thèse que nous présentons dans ce mémoire propose une méthode de conception qui améliore la production des systèmes industriels critiques et adaptatifs, que nous appellerons systèmes Temps-Réel Repartis Embarqués (TR²E) critiques et adaptatifs.

Reprenons rapidement chacun de ces adjectifs afin de clarifier le périmètre de notre étude :

- Temps-réel : les fonctionnalités du systèmes doivent être rendus dans un délais fixe. Au delà de ce délai, le résultat de cette fonctionnalité est inexploitable, ce qui peut correspondre à un cas d'erreur du système.
- Réparti : les fonctionnalités du systèmes sont réparties sur différents nœuds de calcul qui communiquent entre eux par envoie de messages sur un bus de communication. Ceci a pour conséquence que ces différents nœuds ne sont pas, à priori, synchronisés.
- Embarqué : le système doit répondre à des contraintes fortes d'occupation mémoire et d'autonomie énergétique. Plus l'occupation mémoire est importante, plus le système est volumineux, lourd, et consommateur d'énergie.
- Critique : un système critique est un système dont les défaillances peuvent avoir des conséquences dramatiques, financières voire humaines.
- Adaptatif : un système adaptatif est un système qui réagit aux variations de son environnement en modifiant son comportement.

De nombreuses méthodes de conception ont été proposées pour faciliter la réalisation des systèmes TR²E critiques. L'utilisation de langages synchrones [Berry, 2000] a ainsi fait ses preuves, non seulement auprès de notre communauté scientifique, mais également auprès de grands industriels dans des domaines particulièrement critiques tels que l'aérospatial, le ferroviaire, ou encore l'énergie nucléaire. Cependant, l'accroissement de la complexité des logiciels, dû à l'augmentation du nombre de fonctionnalités que doivent fournir ces systèmes, nécessite sans cesse de proposer et de mettre en œuvre de nouvelles méthodes de conception [Zalila, 2008].

Pour autant que ces méthodes proposent des solutions techniques à la réalisation de systèmes critiques, elles ne traitent pas spécifiquement des problèmes que pose la réalisation des systèmes adaptatifs.

1.2 Problématiques

La réalisation des systèmes critiques et adaptatifs pose une question difficile, dont la réponse consiste à placer le curseur entre ces deux caractéristiques opposées que sont l'autonomie et le déterminisme. Doter un système de capacités d'adaptation augmente son autonomie, ce qui, en poussant le raisonnement à l'extrême, le rend imprévisible.

Nous devrons donc d'une part contraindre les capacités d'adaptation du système de tel sorte à garantir son déterminisme, et d'autre part proposer des solutions à la mise en œuvre et à l'analyse de ces mécanismes d'adaptation.

Nous regroupons ici les problèmes que posent l'adaptation des systèmes TR²E critiques selon trois axes d'étude, à savoir : la détection des besoins d'adaptation, la mise en œuvre des mécanismes d'adaptation, et enfin l'analyse formelle du comportement d'un système adaptatif.

1.2.1 Détecter et conditionner l'adaptation

Un premier problème concernant l'adaptation dynamique des système TR²E consiste à détecter les conditions environnementales qui nécessitent une adaptation du système. Cela concerne d'une part le déclenchement de l'adaptation, et d'autre part de s'assurer que celui-ci est dans un état qui permet de mettre en œuvre l'adaptation. Par exemple, lorsque la quantité de charge de la batterie est insuffisante, on doit détecter un besoin d'adaptation, mais on ne peut effectuer cette adaptation que si le système est dans un état qui permet de réduire ces fonctionnalités : il ne faut pas que l'adaptation empêche le système de rendre un service prioritaire.

1.2.2 Fiabiliser le processus d'adaptation

Adapter un système nécessite de modifier son comportement, et donc de perturber les fonctionnalités que ce système. En effet, il est nécessaire de suspendre certains services du système pour lui permettre de s'adapter. si ces services sont modifiés par l'adaptation, on ne peut les suspendre qu'une fois qu'ils ont fourni leurs résultats ; sans quoi il est probable que le comportement du système devienne incohérent. Nous appelons protocole d'adaptation (ou protocole de reconfiguration) le protocole de synchronisation des fonctionnalités modifiée pour l'adaptation. Nous devrons proposer un ou plusieurs protocoles qui permettent de réaliser l'adaptation de façon sûre et/ou rapide.

Nous devrons également nous intéresser aux problèmes que posent la présence de pannes dans les systèmes TR²E critiques.

1.2.3 Analyser le processus d'adaptation

Au delà de la mise en œuvre pratique de mécanismes de reconfiguration fiables, nous devrons également proposer une méthode d'analyse du comportement du système adaptatif, afin de vérifier que les mécanismes d'adaptation vérifient un certain nombre de propriétés de sûreté de fonctionnement.

Ces différents problèmes seront fortement raffinés lorsque nous présenterons l'étude détaillée du besoin auquel nous allons répondre (chapitre 3). Présentons maintenant quelques contraintes spécifiques à notre étude.

1.3 Contraintes

1.3.1 Mise en œuvre des systèmes critiques

La mise en œuvre de systèmes critiques nécessite de respecter un certain nombre de contraintes d'implémentation que nous présenterons au chapitre 3. Nous devrons bien sûr respecter ces contraintes lorsque nous réaliserons le prototype qui permettra d'expérimenter notre approche.

1.3.2 Mise en œuvre de la séparation des préoccupations

La mise en œuvre de la séparation des préoccupations dans un contexte industriel nécessite de prendre en considération la variabilité des exigences non-fonctionnelles. La séparation des préoccupation vise à séparer la mise en œuvre des réponses aux exigences fonctionnelles (ce que fait le système) et non-fonctionnelles (comment le système est fait). Dans le contexte des systèmes TR²E, les exigences non-fonctionnelles sont très variées (fiabilité, caractéristiques de la plateforme d'exécution, contraintes d'activation et de temps de réponse, ressources de calcul disponibles, ressources énergétiques disponibles, etc...). Notre solution devra donc permettre de mettre en œuvre la séparation des préoccupations dans un contexte industriel multi-domaines (avionique, spatial, télécommunication, systèmes autonomes, etc...).

1.4 Objectif et approche

1.4.1 Objectif

L'objectif général de notre travail de recherche consiste à améliorer la productivité des équipes logicielles qui développent des systèmes TR²E critiques et adaptatif, en termes de qualité et de productivité. Nous devrons donc automatiser l'analyse et la production des systèmes TR²E critiques et adaptatifs. Pour améliorer la qualité de la production, nous nous appuierons sur des techniques d'analyse automatique des systèmes, ainsi que sur l'utilisation de règles d'implémentation propres à la mise en œuvre des systèmes critiques. Pour améliorer la productivité, nous utiliserons de techniques de génération automatique de code. Pour améliorer la qualité, nous utiliserons des techniques de vérification formelle.

1.4.2 Approche

Pour satisfaire ces objectifs, nous proposons une méthode de conception (voir chapitre 4) divisée en plusieurs étapes. Deux premières étapes favorisent l'analyse formelle du comportement du système en amont du processus de conception, alors qu'une troisième étape automatise la production du logiciel en aval de ce processus.

Notre approche s'appuie sur la notion de mode de fonctionnement, qui représente un sous-ensemble de fonctionnalités du système : lorsque le système est dans un mode donné, il fournit ce sous-ensemble de fonctionnalités. Nous proposons alors de représenter les modes de fonctionnement du système, mais aussi les conditions qui déclenchent et restreignent les changements de modes. Ceci constitue alors un premier niveau de spécification, la spécification système. Enfin, nous proposons de raffiner cette spécification système dans une phase de conception logicielle qui contient (entre autre) la configuration associée à chacun des modes.

Analyse. Les étapes d'analyse s'appuient sur la production automatique et l'utilisation de modèles formels qui représentent le comportement du système et de l'application. Nous proposons de diviser cette analyse en deux étapes distinctes. Une première étape consiste à analyser la spécification système et la logique des mécanismes de changement de mode. Une seconde étape consiste alors à vérifier que les applications logicielles respectent un certain nombre de propriétés de sûreté de fonctionnement.

Production. La dernière étape de la méthode de conception que nous proposons automatise la production (grâce à des techniques de génération de code) de l'ensemble des applications logicielles du système. Ceci contribue à améliorer non seulement la qualité du code produit, qui respecte les règles de programmation des systèmes critiques, mais également la quantité de code produit.

1.5 Plan du mémoire

Dans la suite de ce mémoire, nous détaillons les motivations, les contributions, les expérimentations, et les résultats de notre étude. Cette présentation s'articule autour de quatre grandes parties.

La partie II est consacrée à l'étude théorique des besoins et des solutions existantes dédiés à la réalisation des systèmes TR²E critiques et adaptatifs. Cette partie se divise en deux chapitres. Au cours du chapitre 2, nous étudions les solutions existantes qui visent à améliorer la productivité des équipes de développement logiciel dans le contexte des systèmes TR²E critiques. Au cours du chapitre 3, nous allons nous intéresser aux problèmes spécifiques que posent la réalisation des systèmes adaptatifs, ce qui nous permettra de présenter les travaux qui se sont plus particulièrement intéressés à cette question. Ce chapitre nous permettra de développer les objectifs auxquels nos contributions répondent.

La partie III, composée d'un unique chapitre (chapitre 4) présente l'approche que nous proposons pour satisfaire ces objectifs. Ce chapitre présente la méthode de conception basée sur l'automatisation de l'analyse et de la production des systèmes TR²E critiques et adaptatifs. Ce chapitre présente également les différents protocoles d'adaptation sur lesquels s'appuie notre générateur de code pour effectuer l'adaptation dynamique de façon sûre et/ou rapide.

Au cours de la partie IV, nous présenterons en détail les contributions qui permettent de mettre en œuvre cette approche. Le chapitre 5 présente le langage de description d'architecture sur lequel s'appuient toutes nos contributions. La grammaire de ce langage est fournie en annexe (voir annexe A). Le chapitre suivant (chapitre 6) présente le générateur de code qui automatise de la production logicielle des systèmes TR²E critiques et adaptatifs. Enfin, nous présentons au chapitre 6 les règles de transformation de modèle qui permettent d'automatiser l'analyse formelle du comportement d'un système TR²E critique et adaptatif.

La partie V présente les expérimentations que nous avons effectué grâce à ces contributions, ainsi que les résultats obtenus.

Enfin, nous conclurons l'ensemble de cette étude au chapitre 9 (partie VI).

Deuxième partie

Enjeux industriels et techniques

Chapitre 2

État de l'art

Je n'ai jamais rencontré d'homme si ignorant qu'il n'eut quelque chose à m'apprendre.
Galilée

SOMMAIRE

2.1 Introduction

L'utilisation massive des systèmes informatiques industriels nécessite de développer des méthodes de conception qui augmentent la productivité des équipes de développement logiciel. Depuis le début des années 80, ce besoin a été traité dans de nombreux travaux de recherche qui se sont intéressés à l'automatisation de la production des systèmes TR²E.

Le principal objectif de ces travaux est de répondre aux problèmes que pose la variabilité des exigences non-fonctionnelles de ce domaine. Pour répondre à ce problème, les intergiciels ont d'abord fait leur apparition. Couche d'adaptation entre le code applicatif d'une part et le code dépendant de la plate-forme d'exécution d'autre part, les intergiciels facilitent la mise

en œuvre de la séparation des préoccupations : le code applicatif peut être développé indépendamment de la plate-forme sur laquelle il s'exécute. Pour améliorer encore cette solution, les modèles à composant répondent au besoin de la décomposition fonctionnelle : une application peut ainsi être développée par parties qui réalisent chacune un sous-ensemble des fonctionnalités du système final. L'ensemble de ces fonctionnalités peut alors être obtenu par assemblage, déploiement et configuration de ces composants.

Enfin, l'utilisation de techniques de modélisation a permis d'automatiser le déploiement et la configuration des applications TR²E. En effet, l'information contenue dans une description d'architecture permet soit (i) de déployer dynamiquement une application logicielle, soit (ii) de générer le code permettant un déploiement statique de cette application.

Par ailleurs, le développement de systèmes critiques nécessite de respecter certaines contraintes de réalisation, dont le but est d'augmenter le déterminisme de l'application. Ceci inclus la mise en œuvre d'un processus de conception rigoureux imposant le respect de règles de conception et de développement, et faisant une utilisation exhaustive des procédures de test. Au delà de ces techniques traditionnellement utilisés dans le développement des systèmes critiques, l'utilisation de méthodes formelles s'impose de plus en plus comme une étape nécessaire à la validation des systèmes critiques.

Dans ce chapitre, nous présentons les différents travaux qui ont apporté une réponse à chacun des problèmes que nous venons de présenter. Nous évaluerons alors leur adéquation avec la problématique générale que nous souhaitons traiter : automatiser la production des systèmes TR²E critiques et adaptatifs.

Nous nous intéressons ici plus particulièrement aux solutions qui automatisent la réalisation des système TR²E critiques. Nous nous intéresserons plus particulièrement aux travaux relatifs aux systèmes adaptatifs au cours du chapitre 3, qui s'applique à détailler la problématique que nous traitons dans ce mémoire. La raison de ce découpage est simple : l'analyse des travaux relatifs aux systèmes adaptatifs sera déterminante dans la définition du périmètre de notre problématique.

Organisation du chapitre. Ce chapitre est organisé comme suit. La section 2.2 présente les travaux relatifs à la mise en œuvre de la séparation des préoccupations (depuis le RPC [2] jusqu'aux modèles à base de composants). Nous décrivons ensuite (section 2.3) les méthodes de modélisation des architectures logicielles basées sur les langages de description d'architecture et sur les techniques de méta-modélisation. Ces méthodes servent en effet de point d'entrée aux techniques de configuration automatique des applications, que nous présentons à la section 2.4. Enfin, nous présentons en section 2.5 les spécificités de la réalisation des systèmes critiques.

2.2 Systèmes TR²E, enjeux et solutions

Commençons par définir précisément ce qu'est un système réparti :

Définition 2.2.1 *Système réparti [Coulouris et al., 2005]*
Un système réparti est un ensemble de machines autonomes connectées par un réseau, et équipées d'un logiciel dédié à la coordination des activités du système ainsi qu'au partage de ses ressources.

2. Remote Procedure Call

L'utilisation des systèmes répartis présente l'avantage de pouvoir démultiplier le nombre d'unités de calcul, ce qui permet d'augmenter les performances globales du système. La répartition pose cependant un certain nombre de problèmes, comme par exemple la portabilité et l'interopérabilité des différentes applications qui composent un système réparti. Revenons rapidement sur la définition de chacun de ces termes. La portabilité est la capacité d'une application à être utilisable sur différents types de plate-formes au prix d'un effort de portage minimal de son implémentation. L'interopérabilité est la capacité de deux applications différentes à interagir entre elles (même si leur conception a suivi des stratégies différentes).

Venons-en maintenant à la définition des systèmes temps-réel embarqués.

Définition 2.2.2 *Système temps-réel [Stankovic, 1988]*
La correction du système ne dépend pas seulement des résultats logiques des traitements, mais dépend en plus de la date à laquelle ces résultats sont produits.

Définition 2.2.3 *Système embarqué*
Un système embarqué est un système qui possède une quantité de ressources (ressources de calcul, ressources énergétiques) réduite.

L'interopérabilité et la portabilité caractéristiques font partie de ce que nous appelons plus globalement – c'est à dire pour l'ensemble des systèmes TR2E – les exigences non fonctionnelles d'un système. Outre ces caractéristiques, le choix d'une plate-forme matérielle, d'un protocole réseaux, d'un langage de programmation, de règles d'implémentation, de l'utilisation d'un standard particulier, ou encore de patrons d'interaction entre les applications réparties, sont des exemples courants des exigences non fonctionnelles de la réalisation d'un système.

Dans le domaines des systèmes TR²E, chacune de ces caractéristiques non fonctionnelles peut être traitée de façon très différente d'un domaine d'activité à un autre.

Dans cette section, nous allons nous intéresser aux différentes solutions qui ont permis d'apporter une réponse à cette question : comment minimiser les surcoûts de développement et de maintenance inhérents à la variabilité des exigences non fonctionnelles ?

2.2.1 Gestion des communications

Le point de départ du développement des systèmes répartis concerne la gestion des communications entre les différentes applications réparties. Initialement, les mécanismes de communication entre les différents nœuds du système étaient directement implantés en utilisant l'interface de programmation fournie par le système d'exploitation ou par une bibliothèque dédiée. L'utilisation de code généré a alors fait son apparition pour simplifier encore ces développement en fournissant une interface de programmation unifiée pour différentes plate-formes d'exécution. En fonction des besoins sur lesquels ces intergiciels mettent l'accent, le code généré porte sur différentes parties de l'architecture logicielle.

Nous présentons dans cette sous-section les différentes caractéristiques des intergiciels existants, ainsi que les besoins auxquels ils répondent plus spécifiquement.

RPC, Remote Procedure Call

Le RPC (Remote Procedure Call) est un des premiers mécanismes de communication à avoir eu pour objectif de faciliter la conception des systèmes distribués. La première implémentation de RPC date de 1984 [Birrell and Nelson, 1984]. L'implémentation de référence

aujourd'hui est celle de Sun Microsystems, qui permet d'utiliser les appels distants de fonctions en C et C++. RPC repose sur le modèle client/serveur, et a pour but de faciliter l'appel de procédures distantes lors du développement d'un système distribué en s'efforçant de conserver le plus possible la sémantique habituelle des appels de fonctions. Le schéma ci-dessous illustre l'architecture liée à l'utilisation d'un tel intergiciel. La couche *Stub* (resp. *Skeleton*) a pour rôle coder (resp. décoder) les informations de l'appel sous la forme d'une suite d'octets (resp. d'une structure de données) que *Skeleton* (resp. le serveur) pourra décoder (resp. utiliser).

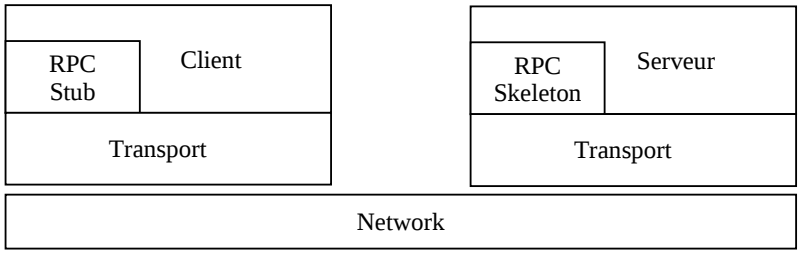

FIGURE 2.1 – Architecture du RPC

Le RPC met l'accent sur la structuration de l'application : le client d'une communication interagit avec un autre composant logiciel en utilisant la même interface de programmation et la même sémantique d'exécution, que le serveur soit déployé sur le même nœud que le client ou sur un nœud différent. Ceci est particulièrement structurant dans la mesure où l'appelant ne continue son exécution que lorsque le résultat de l'appelé lui est parvenu. En revanche, comme nous pourrons le constater dans la suite de ce chapitre, l'utilisation du RPC est en contradiction avec les règles de programmation qu'imposent de respecter la réalisation des systèmes critiques. En effet, l'utilisation du RPC réduit les possibilités d'analyse temporelles de l'application.

Parmi les caractéristiques non-fonctionnelles des systèmes TR²E, le RPC a mis l'accent sur la structuration du code applicatif, au détriment de la capacité d'analyse temporelle.

2.2.2 Au cœur des systèmes répartis, l'intergiciel

L'intergiciel est habituellement considéré comme une couche logicielle ayant pour but de faciliter la prise en charge de communications entre différentes applications réparties sur une architecture matérielle. L'intergiciel sert donc d'interface entre le code utilisateur qui implante les algorithmes de ces applications et la plate-forme d'exécution sur laquelle s'exécute chacune de ces applications. Le terme plate-forme d'exécution désigne le plus souvent l'ensemble constitué par l'architecture matérielle et un système d'exploitation (c'est le cas dans l'ensemble des solutions que nous détaillons dans la suite de cette section), mais certaines fonctionnalités du systèmes d'exploitation peuvent également être implantées par l'intergiciel [Armstrong, 1996].

L'intergiciel a donc pour objectif principale de répondre à l'exigence de "séparation des préoccupations" [Reade, 1989]. La séparation des préoccupations vise à séparer la réponse aux exigences fonctionnelles – *i.e.* ce que fait le système – des réponses aux exigences non fonctionnelles – *i.e.* comment est fait le système –. Dans l'idéal, si l'intergiciel répond à l'ensemble des variations possibles des exigences non fonctionnelles, il devient possible d'adapter

l'application à ces variations en adaptant la configuration de l'intergiciel à un nouveau besoin. Se pose alors la question de savoir si un intergiciel doit être (i) "générique" dans le but de répondre à l'ensemble de ces besoins, (ii) "configurable" dans le but de répondre à un besoin donné, ou s'il doit être (ii) "schizophrène", c'est-à-dire capable de répondre simultanément à plusieurs besoins différents.

La suite de cette section est structurée par cette partition des intergiciels.

Intergiciels génériques

Définition 2.2.4 *Intergiciel générique [Quinot, 2003]*
Un intergiciel générique est un intergiciel dans lequel certains modules sont rendus indépendants du choix d'une personnalité de répartition particulière, et réalisés sous une forme réutilisable. Une personnalité donnée est alors obtenue en instanciant ces modules génériques avec des composants de personnalisation.

Jonathan. Jonathan [Dumant *et al.*, 1999] est un environnement de répartition en langage Java composé d'un noyau réduit, et de plusieurs composants de liaison. Le choix des composants de liaison constitue alors une personnalité de l'intergiciel. David et Jérémie en sont deux exemples, la première s'appuie sur CORBA [(OMG), 2004], la seconde sur Java RMI [Sun Microsystems, 2002].

Discussion. La personnalisation de Jonathan consiste à implanter les interfaces définies dans le noyau réduit de l'intergiciel : l'adaptation de l'intergiciel à de nouvelles exigences non fonctionnelles consiste à implanter les fonctionnalités correspondantes de l'intergiciel. En effet, le noyau de l'intergiciel ne fournit que peu de services (assemblage des composants de personnalités et gestion des ressources matérielles). Ainsi, Jonathan utilise la généricité pour faciliter l'adaptation de l'intergicielaux variation des exigences non-fonctionnels des systèmes répartis. Par conséquent, Jonathan ne fixe pas la sémantique des communications utilisée, qui sera adaptée en fonction de la personnalité implantée. Jonathan privilégie donc l'adaptabilité de l'intergiciel, au détriment de l'analyse et de l'interopérabilité des applications développées avec cet intergiciel.

Intergiciels configurables

Définition 2.2.5 *Intergiciel configurable [Quinot, 2003]*
Un intergiciel est dit configurable lorsque les composants qu'il intègre peuvent être choisis et leur comportement adapté en fonction des besoins de l'application et des fonctionnalités offertes par l'environnement.

TAO, The ACE ORB. TAO (The ACE ORB) est un intergiciel développé par le laboratoire d'informatique de l'université de Vanderbilt [Schmidt *et al.*, 1997]. Il a été conçu dans le but de garantir les caractéristiques de qualité de service des applications temps-réel distribuées s'appuyant sur le standard CORBA [(OMG), 2004]. Outre les différentes caractéristiques des intergiciels CORBA, TAO offre des garanties de qualité de service de bout en bout d'une application répartie. Pour ce faire, TAO permet de configurer :

– la priorité des requêtes de l'application ;

- les paramètres (priorité, periodicité, temps d'exécution, etc...) des tâches qui exécutent le code fonctionnel (code de l'application qui répond aux exigences fonctionnelles) ;
- la politique d'ordonnancement de l'application répartie ;

De plus, TAO permet de contrôler l'utilisation des ressources nécessaires à l'exécution d'une opération sur le serveur. TAO a ainsi largement contribué à la définition du standard RT-CORBA [(OMG), 2005b], évolution de CORBA pour les systèmes temps-réel.

Concernant la portabilité de TAO, celle-ci repose principalement sur l'utilisation de la bibliothèque ACE, un canevas de communication orienté-objet développé en C++ [Schmidt and Cleeland, 2000]. Pour garantir l'interopérabilité entre différentes applications, développées selon des stratégies différentes (langages de programmation différentes par exemple) sur différents *media* de communication, l'intergiciel TAO s'appuie sur les mécanismes de communication spécifiés dans le standard CORBA. Suivant les principes de ce standard, TAO peut par construction interagir avec l'ensemble des applications qui s'appuient sur ce standard.

Les différents travaux effectués autour de l'intergiciel TAO ont ainsi permis de mettre en évidence l'intérêt des approches de génie logiciel reposant sur des intergiciels dont les propriétés de qualité de service sont configurables [Schmidt *et al.*, 1997]. Nous reviendrons plus en détail sur ces aspects au cours de la sous-section 2.4.

Discussion. Des études plus récentes ont montré certaines incompatibilités entre l'utilisation de RT-CORBA et des spécificités de la réalisation des systèmes critiques (spécificités explicitées à la sous-section 2.5). En particulier, la configuration et le déploiement d'une application basée sur TAO est dynamique : elle est faite à l'initialisation (*i.e.* au début de l'exécution) du système. De plus, son architecture orientée objet réduit le déterminisme de l'application (nombreuses indirections). Enfin, si l'utilisation du standard CORBA assure l'interopérabilité, il limite les capacités de TAO à répondre à certaines exigences non fonctionnelles, comme par exemple l'utilisation de mécanismes de communication spécifiques à une application.

Intergiciels schizophrènes

Définition 2.2.6 *Intergiciel schizophrène [Quinot, 2003]*
Un intergiciel est dit configurable lorsque les composants qu'il intègre peuvent être choisis et leur comportement adapté en fonction des besoins de l'application et des fonctionnalités offertes par l'environnement.

PolyORB-HI. PolyORB [Quinot, 2003] est un intergiciel développé par l'ENST (Ecole Nationale Supérieure des Télécommunications) qui implémente le concept d'intergiciel schizophrène [Pautet, 2001]. Cet intergiciel reprend et étend les objectifs de l'intergiciel générique Jonathan. Pour cela, PolyORB s'appuie sur une architecture en trois couches permettant à plusieurs personnalités différentes d'être co-localisées et de coopérer dans une même instance d'intergiciel (d'où le concept de "schizophrénie") : les personnalités applicatives constituent l'intermédiaire entre les composants applicatifs et l'intergiciel grâce à une API dédiée ou un générateur de code. Elles interagissent avec la couche du noyau dans le but de permettre le transfert des requêtes entre les entités. Les personnalités protocolaires s'occupent du transfert des requêtes de la couche neutre en échange de messages sur le réseau de communication choisi. Le cœur neutre se comporte comme une couche d'adaptation entre les personnalités applicatives et protocolaires. Il gère l'utilisation des ressources et est indépendant des deux autres couches.

La figure 2.2 illustre l'architecture de base d'un intergiciel développé avec PolyORB.

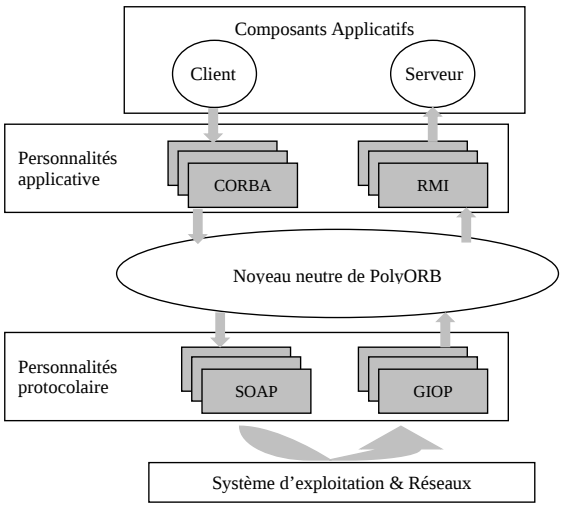

FIGURE 2.2 – Architecture basé sur PolyORB

Discussion. Bien que PolyORB soit statiquement configurable, cette configuration est déterminée à l'initialisation du système en s'appuyant sur des mécanismes d'aiguillage dynamique (par héritage). Pour générer automatiquement la configuration de PolyORB, des travaux ultérieurs ont permis d'utiliser un langage de description d'architecture [Vergnaud, 2006]. Ces travaux n'ont cependant pas permis d'adapter PolyORB aux spécificités de la réalisation des systèmes critiques. Finalement, la définition d'un intergiciel (PolyORB-HI) dédié aux systèmes TR²E critiques associé à un générateur de code (Ocarina) ont permis de définir une méthode de configuration automatique des systèmes TR2E critiques [Zalila, 2008]. Nous reviendrons plus en détail sur ces travaux au cours de la sous-section 2.4.

Synthèse

Les intergiciels que nous venons de présenter permettent, selon des stratégies différentes, de prendre en charge la variabilité des exigences non fonctionnelles d'une application TR²E. Nous avons ici insisté sur l'exemple de la distribution, mais ces considérations s'étendent facilement aux autres propriétés non-fonctionnelles. Les travaux les plus récents que nous avons présentés [Zalila, 2008] répondent aux spécificités de la réalisation des systèmes TR²E critiques. Pour ce faire, la méthodologie associée s'appuie sur une modélisation de l'intégralité du système, ce qui permet d'automatiser la configuration statique du logiciel. Les techniques de modélisation, ainsi que la chaîne de génération associée, seront présenté dans la suite de cet état de l'art.

Avant cela, nous devons prolonger notre étude concernant la séparation des préoccupations. Étudions les solutions qui facilitent le développement du code applicatif. Nous nous limiterons ici aux modèles à base de composants, dans la mesure où ils font partie intégrante de la proposition scientifique que nous faisons dans ce mémoire.

2.2.3 Modèles à base de composants

L'objectif principale des approches à base de composant est de décomposer la réponse aux exigences fonctionnelles d'un système en un ensemble de modules indépendantes les unes des autres que l'on pourra finalement assembler pour couvrir l'ensemble des besoins fonctionnelles du système. Un composant constitue alors une brique logicielle configurable qui va permettre la construction d'une application par composition. Outre cet aspect de décomposition, qui permet de clarifier l'architecture fonctionnelle et ainsi de faciliter la mise en œuvre d'un processus de développement donné, les approches à base de composants favorisent la réutilisation du logiciel ainsi que son adaptation :

- la réutilisation est favorisée par le fait que le composant est considéré comme une entité indépendante, dont les interfaces ont été clairement spécifiées :
- l'adaptation (statique ou dynamique) du logiciel est facilitée dans la mesure ou l'adaptation d'une fonctionnalité donnée consiste à remplacer un composant de l'architecture.

Fractal

Fractal [Leclercq *et al.*, 2007b] est un modèle à composants développé par l'INRIA dans le but de permettre la construction, le déploiement, et l'administration de systèmes complexes tels que des intergiciels ou des systèmes d'exploitation. Pour cela, le modèle Fractal s'appuie sur les quatre principes suivants :

- la capacité d'introspection dont le but est de permettre l'observation du système.
- la capacité de reconfiguration qui permet de déployer et de configurer dynamiquement le système.
- les composants composites sont les composants qui peuvent contenir d'autres sous-composants. Ce principe offre la possibilité de représenter l'architecture du système à différents niveaux d'abstraction.
- les composants partagés sont ceux qui apparaissent en tant que sous-composants de différents composants. Ceci permet de représenter le partage des ressources.

La figure 2.3 illustre le concept de composant Fractal.

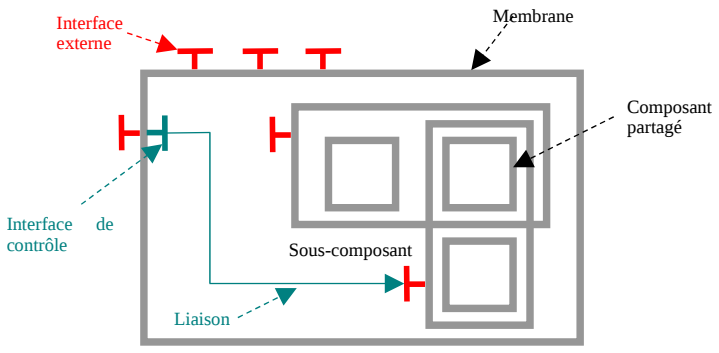

FIGURE 2.3 – Spécification d'architecture Fractal

Un composant Fractal est une entité d'exécution qui possède un ou plusieurs point(s) d'accès appelés interfaces. Les interfaces des composants Fractal peuvent être des interfaces

client ou serveur suivant que le l'accès correspondant soit un accès requis ou fourni. Un composant Fractal est composé d'une membrane, qui permet l'introspection et la configuration dynamique du composant, et d'un contenu, constitué d'un ensemble de sous composants ou d'une partie du code de l'application. Enfin, la liaison constitue l'élément du modèle Fractal permettant aux composants de communiquer.

Depuis sa création, le modèle Fractal a vu naître de nombreuses implémentations, dans différents langages de programmation : Julia en AOKell et ProActive en Java, Think en C, Plasma en C++, FracTalk en SmallTalk, FracNet et .NET.

Discussion. Le modèle de composant Fractal, dans la lignée de l'intergiciel Jonathan, privilégie fortement la généricité de l'approche. Ainsi, Fractal propose d'utiliser la notion de composant non seulement pour représenter les réponses aux besoins fonctionnelles, mais également pour implanter les réponses aux besoins non fonctionnelles. De la même façon que pour les personnalités de Jonathan, l'implantation de ces réponses consiste à développer des composants prédéfinis qui seront finalement instanciés, assemblés et configurés selon les mêmes mécanismes que les composants fonctionnels. Ainsi, les problématiques liées à la distribution ne sont pas intrinsèquement présentes dans Fractal : la sémantique de communication entre deux applications dépend de l'implémentation du composant de communication choisi.

Fractal ne définit pas non plus la sémantique d'exécution de l'application. Les caractéristiques temps-réel de l'application sont donc difficiles à capturer. Dans notre approche, nous n'avons pas choisi ce modèle à composant car il ne permet pas de capturer, de façon univoque la sémantique des communications entre composants.

CCM

CCM (CORBA Component Model) [(OMG), 2006] est un modèle à composant standardisé par l'OMG. Le but de ce type de modèle est de permettre l'encapsulation du code fonctionnel d'un système dans une entité logicielle prise en charge par un environnement d'exécution qui répond aux exigences non fonctionnelles de l'application. L'intérêt d'une telle démarche est de diminuer la complexité du développement d'une application répartie, de faciliter la réutilisation de code fonctionnel, d'augmenter la productivité lors du développement de ces applications, ainsi que d'améliorer la qualité du logiciel produit. Comme son nom l'indique, le CCM s'appuie sur un environnement d'exécution CORBA pour répondre aux exigences non fonctionnelles du système. Ceci permet de fixer la sémantique des communications, mais ne résout pas les problèmes propres au temps-réel.

La figure 2.4 est l'illustration d'un composant CCM.

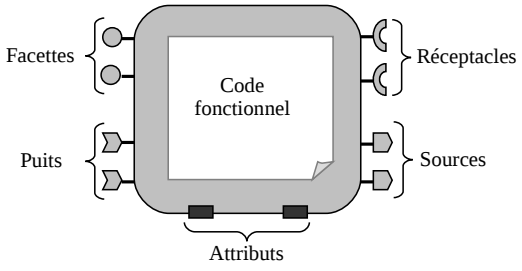

FIGURE 2.4 – Composant CCM

Comme l'illustre cette figure, un composant CCM est constitué :

- d'une référence, qui permet d'identifier le composant au sein de l'architecture ;
- d'attributs, qui sont des paramètres de configuration du composant ;
- de ports, pour connecter les composants entre eux. Ils sont de quatre types : facette/-réceptacle (pour les appels de fonctions synchrones), et sources/puits d'évènements (pour les envoies de message).

Discussion. Contrairement au modèle Fractal, CCM implante des mécanismes de communication entre les composants en s'appuyant sur un intergiciel de type CORBA. Si cela améliore l'interopérabilité et la portabilité du logiciel produit, une telle architecture (même en s'appuyant sur RT-CORBA) ne permet pas de répondre aux spécificités des systèmes temps-réel critiques (que nous verrons par la suite). De plus, le modèle de composant CCM n'offre pas la possibilité de décrire des composants composites, c'est-à-dire composé de plusieurs sous-composants. Cela présente pourtant l'avantage de permettre la modélisation de l'architecture suivant différents niveau d'abstraction.

2.2.4 Conclusion

Dans cette section, nous avons présenté différentes solutions dédiées d'une part au problème de la variabilité des exigences non fonctionnelles, et d'autre part à la décomposition des réponses aux exigences fonctionnelles.

Bien que les approches génériques facilite l'adaptation de l'architecture logicielle aux variations des exigences fonctionnelles, ce gain est réalisé au détriment de l'interoperabilité des outils, et de la capacité des modèles à représenter les caractéristiques spécifiques aux domaine des systèmes TR²E critiques.

La catégorie des intergiciels schizophrènes améliore ces caractéristiques en fournissant la possibilité de configurer statiquement l'ensemble de l'application logicielle tout en prenant en compte ses caractéristiques temporelles.

En ce qui concerne les approches à base de composants, l'utilisation de composants de types CCM présente l'avantage de s'appuyer sur un standard qui définit la sémantique des communications. Ceci améliore l'interopérabilité des outils qui s'appuient sur ce modèle à composants.

De plus, dans le cadre de la conception d'applications TR²E, les approches à base de composants de type CCM apportent un certain nombre d'avantages, comme la décomposition fonctionnelle, la portabilité et la ré-utilisabilité du code fonctionnel. Cependant, cette approche ne résout pas les problèmes liés à la réalisation des systèmes critiques. En outre, l'automatisation de la configuration d'un système TR²E nécessite de disposer d'un modèle complet de l'architecture, qui permette de décrire :

- le déploiement de l'application ;
- la configuration du logiciel (y compris les caractéristiques propres aux systèmes temps-réel) ;
- les connections entre les composants de l'application.

Voyons maintenant les solutions qui répondent à ce besoin de modélisation.

2.3 Modélisation des architectures logicielles

Nous allons ici nous intéresser aux technique de modélisation qui servent de point d'entrée à l'automatisation de la configuration et du déploiement des logiciels TR²E critiques, dans la perspective de modéliser le système, de générer ses applications logicielles, et d'analyser le comportement de ces applications.

2.3.1 Langages de description d'architecture

Principes de base

Les Langages de Description d'Architecture (ADLs) ont pour objectif de fournir une représentation structurée d'un système logiciel. Ils s'appuient généralement sur les concepts de :

- composant, qui représente l'entité de base de la description d'architecture ;
- connecteur, qui définit la sémantique de l'interaction entre deux composants ;
- configuration, qui définit les connexions entre les composant et les connecteurs, ainsi que la valeur initial de leurs attributs de configuration.

Il existe différents types de langages de description d'architecture, en fonction des objectifs qu'ils poursuivent. Nous les avons classés comme suit :

- les ADLs génériques, qui visent à faciliter la conception de l'ensemble des systèmes logiciels ;
- les ADLs formels, dédiés à l'analyse des systèmes critiques ;
- les ADLs spécifiques à un domaine, qui visent à faciliter la conception d'un certain type de systèmes, comme par exemple les systèmes TR²E critiques et adaptatifs.

ADLs génériques

Les ADLs génériques visent à faciliter la conception des systèmes logiciels dans leur ensemble. Ils s'appuient donc sur des concepts génériques (*composant*, *connecteur* et *configuration*) présentés ci-dessus, pour décrire la structure des applications logicielles. La sémantique d'exécution et d'interaction entre ces composants dépend alors le plus souvent de l'implémentation des composants et connecteurs spécifiés. Les ADLs génériques considèrent donc les composants et les connecteurs comme des briques logicielles opaques qui constituent des ensembles de fonctionnalités à configurer.

Fractal ADL. Fractal ADL [Leclercq *et al.*, 2007a] est le langage de description d'architecture associé au modèle de composant Fractal. Ce langage de description d'architecture s'appuie sur XML, et dont la principale caractéristique est l'extensibilité : Fractal ADL permet de définir de nouveaux types d'interactions entre composants, mais permet également d'adapter facilement l'outillage de déploiement et de configuration associé à ce nouvel ADL. Fractal ADL est donc à la fois un langage de description d'architecture, et un "méta-langage" de description d'architecture, c'est-à-dire un langage de description de langage de description d'architecture.

Discussion. L'extensibilité de Fractal ADL, ajoutée à l'extensibilité du modèle Fractal font que cette solution est particulièrement bien adaptée pour répondre à la problématique de l'hétérogénéité des systèmes logiciels. Cependant, la solution choisie pour répondre à ce problème ne permet pas d'imposer et de contraindre la sémantique d'exécution du système. En effet, un utilisateur de Fractal ADL peut facilement définir son propre langage de description

d'architecture, son propre modèle d'interaction, etc... Ce qui rend difficile la mise en œuvre de méthodes d'analyse existantes et basées sur des concepts similaires à ceux définis dans ce langage. La réalisation des systèmes TR²E critiques, par exemple, nécessite de mettre en œuvre différentes méthodes d'analyse qui devront permettre de garantir certaines propriétés concernant le comportement du système. La mise en œuvre de telles méthodes est difficile, si bien qu'il semble nécessaire de limiter le périmètre des définitions du langage de description d'architecture auxquelles l'utilisateur a accès.

D&C. D&C [(OMG), 2005a], est un langage de description d'architecture – standardisé par l'OMG – associé au modèle de composants CCM. Il impose donc la sémantique de communication des composants (les connecteurs sont en fait des connexions de type facette/réceptacle et/ou source/puits d'évènements), mais aussi la sémantique de configuration des composants : les séquences d'appels permettant la configuration des composants est imposée par le standard, et fait appel au code d'implémentation des composants (opérations *configuration_complete* par exemple). La création d'un composant se fait, en suivant ce standard, par allocation dynamique de la mémoire à l'exécution de l'application.

Discussion. De la même façon que fractal, D&C ne restreint pas la sémantique d'exécution et de communication : l'utilisateur est libre de définir l'utilisation des ressources d'exécution au sein du code des composants. De plus, la méthode de déploiement définie dans D&C s'appuie sur des mécanismes (l'allocation dynamique de mémoire par exemple) incompatibles avec les spécificités de la réalisation des systèmes critiques (voir section 2.5).

D'autres ADL ont été définis pour traiter spécifiquement les systèmes critiques : ce sont les ADL formels.

ADL formels

Rapide. Rapide [Luckham *et al.*, 1995] est un langage de description d'architecture qui permet de modéliser les exécutions de tâches concurrentes d'une architecture par le biais de contraintes d'ordre d'exécution, exprimées par le biais d'ensembles partiellement ordonnés. Cette modélisation s'appuie sur des séquences de type *réception d'evènements* $\rightarrow$ *conditions* $\rightarrow$ *actions*. A partir de ces informations, Rapide permet de vérifier des contraintes d'ordonnancement des évènements sur le système.

Wright. Wright [Allen, 1997] s'appuie sur l'algèbre de processus CSP (*Concurrent Sequential Processes*) pour préciser, entre autre, la sémantique d'exécution des connecteurs entre tâches. A partir de ces informations, Wright permet de vérifier l'absence de famine, c'est-à-dire qu'aucun composant n'attendra d'actions d'autres composants qui ne se produiront jamais.

Discussion. Wright et Rapide modélisent la concurrence des tâche d'une architecture. Une tentative a été faite pour transformer des spécification du langage Rapide en spécification Wright. En raison des différences de sémantique entre ces deux langage, seule la partie structurelle de l'architecture a pu être traduite. La principale limitation de ces langages de descriptions d'architecture est qu'ils ne permettent pas de déduire, à partir d'une description d'architecture, à quel type de composant physique correspond un composant logiciel décrit selon ce langage. Prenons l'exemple d'un fil d'exécution (*Thread*). Un tel composant, ainsi que

ses interactions avec d'autres composants, peut vraisemblablement être modélisé en utilisant le langage Wright ou Rapide. Cependant, il sera difficile de déduire de cette description que le composant logiciel en question correspond en réalité ("physiquement") à un fil d'exécution. Ceci limite fortement la possibilité de configurer l'application réelle à partir de cette description.

ADLs spécifiques à un domaine

Comme nous venons de le suggérer, un langage de description d'architecture peut permettre de décrire "physiquement" l'application, en utilisant des éléments concrets propres au type d'application que ce langage doit décrire. C'est le cas du langage AADL (Architecture Analysis and Design Language) [SAE, 2004], qui a été développé pour décrire l'arhitecture système, l'architecture logicielle du système, et l'architecture matériel du système dans un unique modèle, dans le but de représenter le déploiement de l'architecture logicielle sur la plateforme matérielle et de permettre l'analyse du système déployé. Sans ces informations de déploiement, l'analyse des caractéristiques du logiciel, telles que le temps de réponse, reflète l'impacte de la vitesse du processeur, de la politique d'ordonnancement, et des protocoles de communication. Ni Rapide, ni Wright ne permettent de décrire ce type d'information.

AADL. AADL est un langage de description d'architecture particulièrement bien adapté à la conception des systèmes TR²E critiques, dans la mesure où il permet leur analyse. La notion de composant AADL est très différente de la notion de composant telle que nous l'avons utilisée jusqu'ici. En effet, AADL ne définit que des composants concrets, c'est-à-dire des composants tels qu'ils apparaîtront physiquement dans le système final. Ainsi, une architecture AADL s'appuie sur des composants de type processus, fils d'exécution, ou *subprogram*, plutôt que sur des composants logiciels "génériques" représentant des ensembles de fonctionnalité offertes et/ou requises.

Discussion. Le langage AADL s'appuie sur des composants "concrets", ce qui permet d'utiliser ce langage pour configurer et analyser les systèmes TR²E critiques [Zalila, 2008]. Cependant, ce langage ne s'appuie pas sur le concept de composant "générique", ce qui limite la réutilisation des composants AADL dans des domaines d'activités variés, comme cela peut être la cas dans le contexte d'une société comme Thales [Borde *et al.*, 2008]. Nous avons donc choisi d'utiliser des composants génériques de type CCM pour la décomposition fonctionnelle, composants qui seront encapsulés par des composants AADL.

2.3.2 DSL vs UML

La méta-modélisation est une approche beaucoup plus récente que l'utilisation des langages de description d'architecture. Elle consiste à définir l'ensemble des règles et concepts qui permettent de modéliser un ensemble de problèmes. Dans le domaine du logiciel, l'OMG a adopté un approche de modélisation appelé MDA (*Model Driven Architecture*) qui s'appuie sur le langage de modélisation graphique UML [(OMG), 2007]. Dans le domaine des systèmes temps-réel, un profil UML a été standardisé par l'OMG : MARTE (*Modeling and Analysis of Real-Time and Embedded systems*). Définissons rapidement la notion de profil :

Définition 2.3.1 *Profile UML [(OMG), 2007]*
Un profil est un moyen d'extension d'un méta-modèle permettant de spécialiser une méta-classe donnée pour répondre à des besoins de modélisation spécifiques.

UML/MARTE. Le profil MARTE a ainsi été défini dans le but de permettre de modéliser et d'analyser les systèmes TR²E. Il s'agit donc d'un profil spécifique à ce domaine d'activité. Il permet de disposer d'un moyen de modélisation des systèmes TR²E qui favorise l'interoperabilité entre différents outils (outils d'analyse, de conception, etc...). En effet, MARTE permet de modéliser les propriétés non fonctionnelles propres aux systèmes TR²E. Il est intéressant de noter ici que le profile MARTE et AADL sont fortement liés : le métamodèle et la sémantique d'AADL ont été utilisé comme point de départ de la standardisation du profil MARTE. Ce standard définit également un ensemble de règles de transformation entre les stéréotypes MARTE et les concepts AADL, dans le but de modéliser des architectures AADL en utilisant le profile UML/MARTE

Discussion. L'intérêt de l'utilisation des techniques de modélisation dépend principalement de la qualité des outils d'édition associés. Ceci est d'autant plus vrai pour les systèmes complexes, pour lesquels les modèles associés sont souvent "volumineux".

La gestion de configuration des modèles graphiques est également un problème technique lié à ce genre d'outils. En outre, la principale différence entre une telle approche et celles présentées précédemment est que UML/MARTE permet de modéliser graphiquement une application TR²E. UML/MARTE permet ainsi de bénéficier d'une syntaxe graphique, mais n'apporte que peu de bénéfices, par rapport aux langages de description d'architecture, en ce qui concerne la sémantique des éléments modélisés.

Nous venons de faire un rapide tour d'horizon des différentes techniques de modélisation qui permettent de représenter la structure d'une application TR²E. Cette étude nous a permis de montrer qu'aucune des méthodes existantes dans ce domaine ne satisfaisaient nos exigences.

En revanche, nous pourrons ré-utiliser certaines caractéristiques de chacune d'elle pour élaborer notre approche.

Nous allons poursuivre notre état de l'art en étudiant les méthodes et techniques qui permettent d'exploiter ce type de modélisation dans le but de configurer et/ou d'analyser les systèmes TR²E critiques.

2.4 Configuration des systèmes TR²E assistée par ordinateur

Dans cette section, nous allons présenter quelques méthodes de configuration assistée par ordinateur des systèmes TR²E. Cette présentation s'intéressera dans un premier temps aux méthodes qui automatisent la production d'une configuration logicielle à partir de spécifications systèmes et/ou logicielles, produites en amont du processus de développement. Nous verrons ensuite les méthodes qui s'intéressent à la production du système final, à partir du résultat de la phase de conception logicielle.

2.4.1 Configuration automatique des systèmes TR²E

Le but de telles approches consiste à déduire la configuration logicielle qui satisfait au mieux les exigences fonctionnelles et/ou non fonctionnelles des applications TR²E. Nous allons donner un exemple de méthodologie visant à satisfaire les exigences non fonctionnelles d'un système, puis nous nous intéresserons à un exemple d'approche répondant aux exigences fonctionnelles.

Configuration non fonctionnelle

L'exemple que nous avons choisi afin d'illustrer la configuration automatique d'une application TR²E en fonction d'exigences non fonctionnelles consiste à exprimer des relations de dépendance entre les fonctionnalités du système, les composants logiciels qui réalisent ces fonctionnalités, et les ressources physiques nécessaires à la mise en œuvre de ces composants. Dans [Cui and Nahrstedt, 2001], les auteurs présentent un algorithme de sélection de configuration dont le critère est la minimisation de l'utilisation des ressources (CPU et bande passante). La figure 2.5 illustre le type de spécification sur laquelle s'appuie cet algorithme pour définir une configuration qui satisfait cette exigence. Sur cette figure, les arcs représentent les dépendances des blocs fonctionnels sur des composants logiciels et des ressources physiques. Les nombres associés à ces arcs représentent la quantité relative d'utilisation de la ressource par le bloc fonctionnel ou composant logiciel.

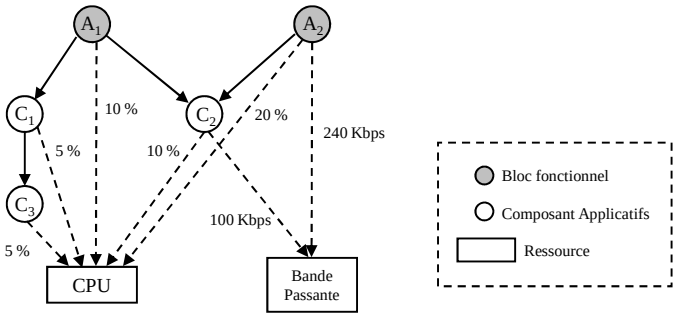

FIGURE 2.5 – Desription des dépendances fonctionnelles → non fonctionnelles

Discussion. L'utilisation d'une telle technique dans le cadre des spécificités des systèmes critiques nécessite de mettre en œuvre des techniques de vérification formelle qui assurerons que la configuration choisie vérifiera certaines contraintes spécifiques de ce type de système. L'utilisation de ces méthodes formelles est déjà une tâche difficile en aval du processus de développement, qui se heurte souvent au problème de l'explosion combinatoire du nombre d'états à parcourir.

Configuration fonctionnelle

Automatiser la configuration fonctionnelle d'un système TR2E consiste à automatiser la sélection d'un ensemble de composants logiciels permettant de rendre un service correspondant à une fonctionnalité du système. Nous allons illustrer cette méthodologie à travers la présentation de CoSMoS [Fujii and Suda, 2004] (Component Service Model with Semantics), qui automatise la configuration fonctionnelle. Ce processus s'appuie sur la notion de graphe sémantique et d'ontologie, dans le but de décrire les entités manipulées par les composants logiciels, ainsi que leurs relations. A titre d'exemple, la figure 2.6 représente le graphe sémantique correspondant à l'assertion : "La distance à parcourir pour aller de Paris à Brest est de 581,71 Km".

Dans CoSMoS, les graphes sémantiques sont utilisés dans le but (i) définir les concepts du domaine d'application du système, de (ii) représenter les liens entre ces notions et les composants logiciels, et (iii) de définir les besoins fonctionnels auxquels répond un composant

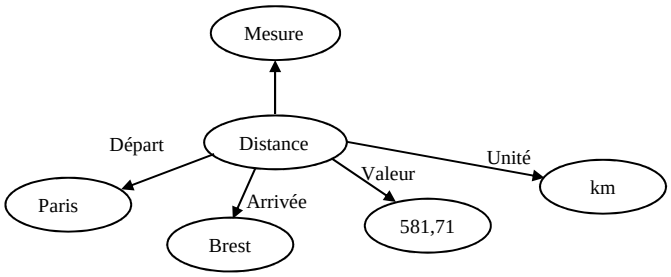

FIGURE 2.6 – Exemple de graphe sémantique

logiciel. La configuration automatique implantée dans CoSMoS permet alors d'automatiser le processus de composition de service dans le but de couvrir un ensemble de besoins fonctionnels du système.

Discussion. La principale limitation d'une telle approche vient de l'utilisation, trop générale, des graphes sémantiques. Le processus de configuration automatique présuppose en effet que les opérations à composer existent, et sont associées à une description sémantique cohérente avec la description sémantique du domaine.

Ces hypothèses fortes qui limitent considérablement les cas d'usage de cette approche. Ce processus suppose également que la description sémantique du domaine d'application est suffisamment complète.

2.4.2 Production du système final

L'utilisation des ressources informatiques dans le but d'assister la conception d'un système TR²E concerne également la production du système final, c'est-à-dire le déploiement et la configuration de l'ensemble des applications de ce système. Le standard D&C [(OMG), 2005a] définit ces notions de la façon suivante :

Définition 2.4.1 *Déploiement [(OMG), 2005a]*
Le déploiement d'une application répartie est la sélection des composants logiciels requis pour réaliser un ensemble de fonctionnalités. Le déploiement requiert aussi le placement de ces composants sur les nœuds de la plate-forme d'exécution.

Définition 2.4.2 *Configuration [(OMG), 2005a]*
La configuration d'une application consiste à initialiser les paramètres de configuration de chacun des composants sélectionnés lors de la phase de déploiement et de les assembler.

Dans cette section, nous allons nous intéresser à différentes méthodologies qui ont été proposées pour faciliter le déploiement et la configuration de système TR²E. Ces méthodologies s'appuient le plus souvent sur l'un des modèles à composants que nous avons présentés précédemment.

MyCCM

MyCCM [**?**] est un outil de conception, de déploiement et de configuration d'applications TR²E qui s'appuie sur les standards D&C et Lightweight CCM. Il utilise la notion de "compo-

sant logiciels prédéfinis" pour réaliser l'activation des composants fonctionnels. Ces composants sont implantés par les développeurs du framework, et peuvent être utilisés lors de la description d'architecture basée sur D&C. Pour répondre aux spécificités du domaine du logiciel temps-réel, MyCCM définit également la notion de service d'interception, pour insérer des services techniques entre deux composants. Un exemple typique de service technique utilisé dans le domaine des systèmes temps-réel est un service d'exclusion mutuelle.

Discussion. MyCCM s'appuie sur le standard RT-CORBA et TAO pour implanter la sémantique d'interaction entre les composants. De plus, le déploiement des composants passe par des mécanismes d'allocation dynamique de mémoire. Pour ces raisons, MyCCM ne répond pas aux spécificités des systèmes TR²E (voir 2.5).

CosMIC

CoSMIC [Gokhale *et al.*, 2003] est une suite d'outils dédiée à la configuration et au déploiement d'applications TR²E. CoSMIC s'appuie sur différents outils et formalismes pour le déploiement d'applications à base de composants respectant les recommandations spécifiées dans le standard D&C. Les descriptions de l'assemblage des composants sont spécifié à partir de la syntaxe graphique CADML (Component Assembly & Deployment Modeling Language), à partir duquel sont générés les plans de déploiement dans un format (XML) respectant le standard D&C. Pour configurer les applications réparties, COSMIC utilise un langage spécifique : OCML (Options Configuration Modeling Language) qui est aussi un langage graphique. Il permet de définir les dépendances entre différentes options qui sont à sélectionner parmi une multitude d'options de configuration de l'application. Dans [Gokhale *et al.*, 2003], les auteurs montrent comment l'utilisation de COSMIC permet d'assembler des composants logiciels et matériels préfabriqués tout en assurant une compatibilité entre leurs différents paramètres de qualité de service et un déploiement correct de l'application. Si cette tâche est effectuée manuellement, les risques de commettre des erreurs deviennent élevés, le passage à l'échelle devient très difficile à gérer (surtout avec des applications TR²E complexes), et cela compromet la possibilité de validation et vérification de l'application.

Discussion. Bien que ces différents aspects soient facilités par l'utilisation de l'approche définie autour de CoSMIC, il n'en reste pas moins que CoSMIC présente un certain nombre de limitations relatives au contexte de notre étude :

1. le fait que CoSMIC repose sur plusieurs langages différents pour spécifier une application (IDL, CADML, OCML. . .) oblige à consolider chacun des modèles lors des modifications de l'application et à assurer leur cohérence ;
2. de même que MyCCM, CoSMIC repose sur l'intergiciel TAO, dont les caractéristiques ne satisfont pas les spécificités des systèmes critiques ;
3. l'utilisation de UML et de XML pose des problèmes de passage à l'échelle quand il s'agit de modéliser des applications avec un très grand nombre de composants [Sriplakich *et al.*, 2008].

FractalADL

FractalADL est à la fois un langage de description d'architecture et un langage de spécification de l'outil de déploiement et de configuration : FractalADL permet en effet de décrire

l'architecture logicielle d'un système logiciel, mais aussi l'ensemble des étapes qui permettront de déployer et de configurer ce système. Dans la lignée de la philosophie de Fractal, la méthode de déploiement et de configuration associée est générique et extensible.

Discussion. Encore une fois, l'extensibilité de Fractal, de FractalADL et de la machine de déploiement associée permet de mieux répondre à la variabilité des exigences liées au domaine des systèmes TR2E. Cependant, cette variabilité implique d'associer une sémantique faible à chacun des composants de l'application, ce qui réduit fortement les possibilités d'analyse des ces systèmes, et les capacités d'interopérabilité entre ces outils d'analyse : pour chaque système, il sera nécessaire de concevoir un outil d'analyse et de déploiement spécifique.

Ocarina

Ocarina [Zalila *et al.*, 2008] est un outil de génération de code qui permet de générer le déploiement statique d'une application à partir de sa description d'architecture en AADL [SAE, 2004]. Le code généré par Ocarina permet de respecter les spécificités de conception des systèmes TR²E critiques. De plus, Ocarina s'appuie sur l'outil Cheddar [Singhoff *et al.*, 2004] pour réaliser l'analyse d'ordonnançabilité de l'application modélisée en AADL [Singhoff *et al.*, 2005]. Enfin, Ocarina s'appuie sur l'intergiciel PolyORB-HI pour interfacer le code généré avec la plate-forme d'exécution sous-jacente.

Discussion. La principale limitation d'Ocarina vient du fait que l'utilisation d'AADL 1.0 limite fortement les possibilités de modéliser et de décomposer les réponses aux exigences non fonctionnelles. En effet, les fonctionnalités d'une application déployée et configurée avec Ocarina doivent être représentés sous la forme de séquences d'appels de sous-programmes (*subprogram*) en AADL 1.0. Cette séquence doit être modélisée à l'intérieur d'une tâche (*thread*) de la description d'architecture. Malheureusement, dans la plupart des systèmes TR2E, le flux d'exécution d'une tâche ne peut pas être représenté sous la forme d'une séquence d'appels.

Au cours de l'état de l'art que nous venons de présenter, nous avons décrit différentes solutions visant à assister par ordinateur la conception des systèmes TR²E. Certaines de ces solutions visaient d'ailleurs plus spécifiquement les systèmes TR²E critiques. Ces solutions couvrent en partie les spécificités liées à la réalisation des systèmes critiques. Dans la section suivante, nous présentons ces spécificités, y compris en ce qui concerne le processus de développement associé.

2.5 Spécificités des systèmes critiques

La perte de certains systèmes peut avoir des conséquences dramatiques, provoquant des pertes financières très importantes, parfois même des pertes humaines. Les exemples de défaillance de tels systèmes ne manquent pas. Parfois mécaniques, parfois électroniques, et parfois logicielles, les causes de ces pertes sont inacceptables. Bien qu'aucun mécanisme de conception ne puisse garantir totalement le bon fonctionnement d'un système dans toutes ses conditions possibles d'utilisation, un certain nombre d'outils, de techniques, et de règles ont

été mises en œuvre afin de prévenir au mieux la défaillance de ce type de systèmes. Parmi les systèmes critiques, ceux de l'industrie aéronautique fournissent un exemple significatif en terme d'exigences de qualification. En Europe, comme aux Etats-Unis, l'obtention d'une certification est une condition *sine qua non* pour qu'un avion puisse voler au dessus du territoire concerné. Cette certification concerne la structure mécanique, la section motrice, et l'électronique embarquée. La partie logicielle du système doit être certifiée en suivant le document de référence du RTCA (*Radio Technical Commission for Aeronautics*), connue sous le nom de "DO-178B' [RTCA and EUROCAE, 1992]. Dans la suite de cette section, nous allons présenter les méthodes de conception préconisées par ce document.

2.5.1 Niveaux de criticité et certification

A travers cette sous-section, nous allons nous intéresser aux exigences de qualification des logiciels embarqués dans l'avionique. Nous allons pour cela présenter les principales recommandations contenus dans le document DO-178B.

Niveaux de criticité

Pour commencer, il est important de noter que la plupart des standards utilisés aujourd'hui s'accordent sur une description et une catégorisation des pannes possibles des éléments d'un système. A chaque niveau de cette catégorisation, qui n'est d'ailleurs pas spécifique aux éléments logiciels du système, correspond un niveau d'assurance requis :

- Panne catastrophique : panne qui empêcherait de poursuivre le vol et de se poser de façon sûre. Niveau de certification associé : A.
- Panne sévère : panne qui réduirait le capacités de l'avion ou de l'équipage à faire face à des conditions de vol défavorables. Niveau de certification associé : B.
- Panne majeure : panne qui réduit significativement les marges de sûreté de l'appareil, provoque une surcharge de travail de l'équipage, ou l'inconfort de passagers. Niveau de certification associé : C.
- Panne mineur : panne qui réduirait peu les marges de sécurité, et qui provoquerait l'intervention de l'équipage dans le cadre de leurs attributions. Niveau de certification associé : D.
- Panne sans effet : panne qui n'a pas d'effet sur les capacités fonctionnelles de l'appareil. Niveau de certification associé : E.

DO178-B : Vérification du logiciel

Le but de cette vérification est de détecter, d'identifier et de supprimer les erreurs introduites pendant la phase de développement du logiciel. Cela est mis en œuvre par le biais de revues, d'analyses, et de tests. Explicitons le rôle de chacune de ces techniques :

- Une revue a pour objectif de contrôler les sorties d'une étape de conception du système. Dans la norme IEEE 729, une revue est définie comme un "examen détaillé d'une spécification, d'une conception, ou d'une implémentation par une personne ou un groupe de personnes, afin de déceler des fautes, des violations de norme ou de développement ou d'autres problèmes".
- Une analyse permet d'examiner en détail les performances, et l'implication d'un composant logiciel en terme de sûreté.

– Les tests constituent la part la plus importante du processus de vérification logicielle. Ces tests doivent être mis en œuvre pour permettre de valider les exigences énoncées dans les phases de spécifications amont du système.

Discussion. Comme nous pouvons le constater à travers cette rapide présentation des préconisations visant à certifier le logiciel de bord des systèmes avioniques civils, la sûreté de fonctionnement d'un logiciel est principalement évalué par le biais de :

– la traçabilité des exigences ;
– les revues de code ;
– les tests exhaustifs.

Compte tenu de la complexité croissante des applications logiciels industriels, il devient de plus en plus difficile de tester exhaustivement l'ensemble des flux d'exécution d'un programme. De plus, le test est un processus qui intervient très tard dans le processus de développement, si bien que la détection d'erreurs de conception à ce moment du processus de développement est particulièrement difficile à résoudre. Ceci justifie pleinement l'utilisation de méthodes formelles, qui permettent de valider le comportement du système bien plus tôt dans le processus de développement.

En particulier, les langages synchrones [Benveniste *et al.*, 2003] ont été particulièrement utilisés dans le domaine de l'avionique pour leur capacité à générer du code fonctionnel à partir de spécification formellement vérifiables.

Les autorités de sûreté ont ainsi compris que les techniques de génération de code peuvent être avantageusement complétées par l'utilisation des méthodes formelles, d'où leur introduction dans la révision C du standard : la DO-178C.

Présentons maintenant les modèles et outils de vérifications sur lesquels s'appuiera la contribution que nous présentons dans ce mémoire.

2.5.2 Vérification et validation, modèles et outils

Il existe différents modèles permettant de vérifier des propriétés comportementales sur un système donné. Certains outils ont également été développés dans le but d'utiliser ces modèles à des fins de vérification.

Dans la suite de cette section, nous allons décrire certains de ces modèles, ainsi que certains des outils qui les utilisent. Nous nous limiterons aux techniques d'analyse d'ordonançabilité, ainsi qu'aux techniques de *model checking*, dans la mesure où ces méthodes ont plus particulièrement été utilisées dans le cadre de l'analyse des systèmes TR²E critiques et adaptatifs [Real and Crespo, 2004; Pedro and analysis for, 1998; Apvrille *et al.*, 2004a; Bertrand *et al.*, 2008; Rolland, 2008].

Chacune de ces techniques présente des avantages et des inconvénients qui en font des outils complémentaires permettant d'augmenter la confiance que l'on peut avoir dans un système. Leur objectif n'est pas de certifier n'importe quel système ou programme, mais de permettre *d'augmenter la confiance que l'on peut avoir dans ce système.*

En effet, il est important de noter ici que la vérification porte toujours sur le modèle du système, et non sur le système lui-même. La vérification permet donc de compléter les résultats du test classique, qui reste indispensable. En effet, le test est le seul processus de vérification s'attachant à vérifier le respect de propriétés sur le système final. Cependant, le test ne permet pas de vérifier le comportement du système de façon exhaustive, et ne permet pas non

plus de vérifier ce comportement en amont du processus de développement. L'utilisation des méthodes que nous allons présenter vise à répondre à ce besoin.

Algorithmes d'analyse d'ordonnançabilité

L'étude de l'ordonnançabilité d'un système vise à s'assurer que l'ensemble des tâches finiront toujours leur exécution dans un délais donné. Cette étude repose sur la modélisation des fonctionnalités de ce système sous la forme différentes tâches s'exécutant sur un processeur. Certaines propriétés de ces tâches sont fondamentales pour évaluer l'ordonançabilité du système.

Une tâche sera alors caractérisée par une propriété d'ordonnancement, établissant son comportement périodique, sporadique, ou apériodique. Une tâche périodique (ou sporadique) sera alors caractérisée par sa période, son temps d'exécution, et son échéance. Une tâche apériodique sera quant à elle caractérisée par son temps d'exécution et son échéance. Enfin, une politique d'ordonnancement (RMS, EDF, etc. . .) doit être associée à ce modèle de tâches. Les travaux présentés dans [Liu and Layland, 1973] décrivent le type d'algorithmes qui permettent d'utiliser ces données afin de déterminer l'ordonnançabilité d'un système. A partir de cette théorie, des outils ont été réalisé dans le but d'automatiser la vérification de faisabilité de l'ordonnancement, c'est-à-dire la capacité de chaque tâche d'un ensemble donné à toujours être exécutée dans un délai prédéterminé.

Cheddar [Singhoff *et al.*, 2004] et MAST [Medina *et al.*, 2002] sont des outils d'analyse d'ordonnançabilité des systèmes temps réel qui permettent de vérifier la faisabilité de l'ordonnancement de ces systèmes, mais également de simuler cet ordonnancement. Dans le cas d'ensembles de tâches périodiques, les tests de faisabilité sont suffisants pour déterminer l'ordonnançabilité d'un système.

Discussion. Ce type d'analyse permet d'avoir une meilleure connaissance du système dans un contexte donné. Cependant, dans certains cas d'applications complexes, l'utilisation de ce type d'outils est insuffisant pour prouver l'ensemble des propriétés de sûreté que doit vérifier le système. En effet, les algorithmes d'ordonnancement utilisés aujourd'hui dans les systèmes temps réel ignorent la présence de synchronisation des processus, excepté dans des cas simples de précédence de tâches ou d'accès à des ressources partagées et protégées. Dans des cas de synchronisation plus complexes, qui mettent en jeu des mécanismes de communication, nous utiliserons des outils de modélisation formelle tels que ceux que nous présentons dans la suite de cette section. S'ils sont plus difficiles à mettre en oeuvre, il fournissent davantage de garanties, et de façon plus précise que les techniques que nous venons de présenter. Ainsi, nous pourrons les utiliser pour analyser la logique des changements de mode, ainsi que les propriétés de sûreté spécifique à ces mécanismes d'adaptation.

Model checking

Avant de nous intéresser aux différents modèles qui permettent d'utiliser cette technique, présentons les principes de base du *model checking*. Le but est ici de vérifier que le système conçu vérifie une propriété donnée. Le *model checking* consiste à modéliser le plus fidèlement possible le comportement d'un système afin de vérifier cette propriété. Pour ce faire, nous devons donc posséder un langage de modélisation formel des systèmes, un langage formel dans lequel exprimer la propriété à vérifier, et des algorithmes de traitement qui soumettent la propriété aux modèles.

Les propriétés standard que l'on souhaite exprimer sont l'accessibilité, la sûreté, la vivacité, l'équité, et l'absence de blocage [Bouyer, 2002] :

- les propriétés d'accessibilité indiquent qu'un état du système peut être atteint ;
- les propriétés de sûreté expriment que, sous certaines conditions, une situation (combinaison d'états du système et de ses sous-systèmes) ne peut jamais arriver ;
- les propriétés de vivacité expriment que sous certaines conditions, une situation (combinaison d'états du système et de ses sous-systèmes) finira par arriver ;
- les propriétés d'équité indiquent que, sous certaines conditions, une situation (combinaison d'états du système et de ses sous-systèmes) se produira un nombre infini de fois ;
- l'absence de blocage indique que le système ne se trouve jamais dans un état qu'il n'a plus la possibilité de quitter.

Il est également possible d'exprimer des propriétés spécifiques au système. Ces propriétés devront être exprimées dans une logique compatible avec les modèles et les algorithmes de preuve utilisés.

Parmi les formalismes les plus utilisés pour représenter un système TR²E, nous pouvons citer les réseaux de Petri temporisés [Gorrieri and Siliprandi, 1994] et les automates temporisés [Alur and Dill, 1994], qui présentent chacun l'avantage de pouvoir modéliser les caractéristiques temporelles de l'application.

UPPAAL [Bengtsson *et al.*, 1996] et Roméo [Gardey *et al.*, 2005] sont deux outils de *model checking* qui permettent d'analyser respectivement le comportement de systèmes modélisés par le biais d'automates temporisés et de réseaux de Petri temporisés. En ce qui concerne l'expression des propriétés formelles, tout deux s'appuient sur la logique TCTL (Timed Compotational Tree Logic) [Alur *et al.*, 1992].

Discussion. Les techniques que nous venons de présenter permettent d'analyser le comportement logique et/ou temporel d'un système TR²E. Nous devrons donc utiliser ces techniques, ou proposer des extensions de leur théorie dans le but de permettre l'analyse d'un système TR²E **adaptatif**. Par ailleurs, si ces techniques permettent de représenter le comportement du système de façon formelle, cette modélisation ne constitue en réalité qu'une abstraction du système physique. Intéressons nous maintenant aux spécificités de la réalisation concrète de tels systèmes.

2.5.3 Réalisation des systèmes critiques

Dans cette section, nous allons présenter les principales limitations liées à la réalisation des systèmes critiques : l'absence d'allocation dynamique de mémoire, et le respect du profile Ada Ravenscar [Burns, 1999].

Initialisation de l'application

L'initialisation d'un système TR2E critique nécessite de respecter un certain nombre d'exigences qui imposent de réaliser statiquement (*i.e.* dès la compilation du code) l'initialisation des structure de données de l'application. Il est ainsi interdit de faire appel à des instructions d'allocation dynamique de mémoire dans le code des applications critiques. Si on l'autorise, alors on s'expose par exemple aux erreurs d'incohérence des pointeurs. L'éviter facilite ainsi la vérification : on peut vérifier par analyse statique (interprétation abstraite) les opérations

mémoire, la cohérence des pointeurs, le respect de la taille de la pile,etc... [Xavier Allamigeon and Charles Hymans, 2007].

Profile Ravenscar

Outre ces limitations, le profil Ada Ravenscar a également été défini pour garantir le déterminisme et l'analysabilité de l'ordonnancement d'un système TR²E. Ce profil impose ainsi :

- de définir statiquement l'ensemble des tâches de l'application ;
- d'empêcher que plus de deux tâches aient accès simultanément à une ressource partagée (*i.e.* la taille de la file d'attente ne peut être supérieure à 1) ;
- d'utiliser le protocole PCP [Sha *et al.*, 1990] pour protéger l'accès aux données partagées ;
- d'interdire les communications distantes synchrones (dans le cas de systèmes répartis bien sûr).

Discussion

Bien sûr, le fait qu'un système soit qualifié de critique n'impose pas systématiquement d'utiliser les restrictions du profile Ravenscar ou d'interdire l'allocation dynamique de mémoire. Comme nous l'avons vu précédemment, il existe différents niveau de criticité, et chacun d'entre eux impose un niveau de fiabilité plus ou moins exigent. Toutefois, ces restrictions sont des "bonnes pratiques" que nous respecterons dans notre approche.

Au cour de l'état de l'art que nous venons de présenter, nous avons décrit les différentes approches qui visent à faciliter la réalisation des systèmes TR²E, ainsi que les spécificités liées à la réalisation des systèmes critiques.

2.6 Synthèse

Dans ce chapitre, nous avons présenté les solutions dédiées aux problèmes que pose la réalisation des systèmes TR²E critiques. Cependant, aucune des solutions présentées dans ce chapitre ne propose de méthode d'analyse et de production des systèmes TR²E critiques et adaptatifs. Et pour cause, nous avons choisi de présenter conjointement ces travaux et notre problématique. En effet, l'étude des travaux autour des systèmes adaptatifs sera d'une importance capitale dans la définition précise du périmètre de notre problématique. Cette étude, ainsi que la définition précise de la problématique que nous avons traité, fera l'objet du chapitre suivant. Comme nous pourrons le constater dans ce chapitre, la réalisation de systèmes TR²E critiques et adaptatifs pose un certain nombre de problèmes difficiles à résoudre.

Le langage de description AADL est celui qui s'approche le plus d'une solution de représentation du comportement adaptatif d'un système tout en permettant l'analyse temporelle de ce comportement. Cependant, il n'existe pas de méthode de conception qui permette d'automatiser le production du code correspondant.

Si les travaux que nous avons présentés ne traitent pas directement de ces problèmes, nous avons toutefois pu identifier grâce à cette étude un certain nombre de concepts, méthodes et techniques que nous pourrons ré-utiliser.

Par exemple, nous nous appuierons sur le modèle à composant Lightweight CCM pour faciliter la décomposition des réponses aux besoins fonctionnels du système. Ce modèle à composant fixe la sémantique de communication entre les composants, et s'appuie sur un standard, ce qui facilite l'interopérabilité des outils basés sur ce standard. Certaines adaptations seront cependant nécessaires pour répondre aux exigences spécifiques de la réalisation des systèmes critiques. Enfin, ce choix a également été motivé par le fait que ce standard a déjà été déployé sur différents programmes de la société Thales.

D'un autre coté, nous avons choisi d'utiliser un langage de description d'architecture spécifique au domaine des systèmes TR²E critiques pour faciliter la production et l'analyse des applications logicielles correspondantes. Nous ré-utiliserons ainsi les résultats existants autour des intergiciels schizophrènes, ainsi que l'approche basée sur AADL et Ocarina pour la réalisation des systèmes TR²E critiques.

Au cours du chapitre suivant, nous détaillons la problématique du travail présenté dans ce mémoire : comment automatiser la réalisation des systèmes TR²E critiques et **adaptatifs** dans une approche dirigé par les modèles et qui facilite la mise en œuvre de la séparation des préoccupations. La mise en évidence de cette problématique fera également l'objet d'une étude des travaux qui y ont, en parti, répondu.

Chapitre 3

Problématique industrielle

Il n'existe pas de catégorie de science que l'on puisse appeler science appliquée. Il y a la science, et les applications de la science, liées ensemble comme le fruit à l'arbre qui le porte.

Louis Pasteur

SOMMAIRE

3.1 Introduction

Dans ce chapitre, nous allons montrer que la réalisation des systèmes critiques et adaptatifs constitue à la fois un besoin croissant dans le domaine des systèmes TR²E, et un problème difficile à résoudre.

Systèmes adaptatifs, un besoin croissant. L'état de l'art que nous venons de présenter montre l'évolution de l'ensemble des travaux de recherche qui visent à faciliter la réalisation des systèmes TR²E : depuis l'apparition des intergiciels, jusqu'à l'utilisation intensive de techniques de génération de code, l'objectif poursuivi est d'améliorer l'efficacité du développement de ces applications en termes de qualité et de productivité.

Cette dynamique de recherche répond au problème suivant : les applications logicielles TR²E sont de plus en plus complexes, et donc difficiles à réaliser. De plus, cette complexité s'explique principalement par le nombre de contraintes hétérogènes et parfois contradictoires auxquelles ce type d'applications doit répondre.

Pour répondre à ce problème, adapter le système aux évolutions de son environnement opérationnel permet de sélectionner, à un instant donné de son cycle de vie, l'ensemble des fonctionnalités qui répondent le mieux aux contraintes de ce système. Il s'agit alors de sélectionner, en fonction de contraintes fonctionnelles et non-fonctionnelles (performance, embarquabilité, tolérance aux pannes) du système, l'ensemble des services que ce dernier doit fournir. Cette sélection de services s'appuyant sur des mécanismes de modification de la configuration de l'application logicielle, nous utiliserons le terme de "reconfiguration logicielle" pour désigner ces mécanismes d'adaptation.

C'est dans cette perspective que la réalisation de systèmes TR²E adaptatifs (ou reconfigurable) prend une importance de plus en plus importante dans le domaine des systèmes TR²E.

Systèmes critiques et adaptatifs, un problème difficile. La réalisation de systèmes adaptatifs et critiques est d'autant plus difficile qu'elle nécessite de prendre en considération deux aspects contradictoires : fournir des capacités d'adaptation à un système TR²E vise à augmenter l'autonomie de ce système, tandis que le développement d'un système critique nécessite de s'assurer, dès la conception de ce système, qu'il vérifie un certain nombre de propriétés de sûreté de fonctionnement.

Présentation de la problématique. Considérant que l'automatisation de la configuration logicielle *et* la reconfiguration logicielle constituent une solution au problème difficile la réalisation des systèmes TR²E critiques *et* adaptatifs, nous souhaitons évaluer la possibilité de combiner les bénéfices de ces deux approches pour augmenter l'efficacité des équipes de développement logiciel (qualité, productivité). Nous pourrions utiliser ici le terme de (re)configuration, qui regroupe en fait deux problématiques différentes :

- d'une part, si la reconfiguration d'un système s'appuie sur des mécanismes automatiques de détermination de la configuration logicielle à employer (comme pour sa configuration initiale), nous parlerons de **reconfiguration automatique** ;
- d'autre part, la configuration d'un système peut contenir les spécifications de ses futures configurations, et donc de ses futures reconfigurations. Nous parlerons alors de reconfiguration **pseudo-dynamique**, dans la mesure où l'ensemble des configurations atteignables sont dans ce cas connues dès la configuration initiale du système.

Organisation du chapitre. Dans la suite de ce chapitre nous allons présenter, à travers un exemple industriel de système TR²E critique et adaptatif, la complexité inhérente à la réalisation de tels systèmes (section 3.2). Ceci permettra également de montrer l'importance croissante que prennent ces systèmes dans le domaine des applications TR²E critiques. Nous nous attellerons ensuite à délimiter le périmètre de notre problématique (section 3.3), puis nous présenterons les contraintes spécifiques de notre étude (section 3.4). Enfin, nous synthétiserons (section 3.5) l'ensemble des objectifs que nous souhaitons atteindre afin de répondre à cette problématique.

3.2 Systèmes adaptatifs, présentation des besoins industriels

Fournir des capacités d'adaptation à des systèmes embarqués critiques est un besoin industriel central. Tous les systèmes embarqués sont dotés de différents modes de fonctionnement, et l'énumération de ces modes et des transitions associées constitue la prémière étape de la spécification de ces systèmes.

Ce besoin de capacités d'adaptation est d'autant plus significatif dans le contexte actuel, qui impose un accroissement constant du nombre de fonctionnalités que doivent remplir ces systèmes tout en respectant des contraintes de performance et d'embarquabilité de plus en plus fortes. Pour mieux comprendre ce besoin, nous nous sommes intéressés au domaine des systèmes autonomes : comme nous pourrons le constater au cours de ce chapitre et tout au long de ce mémoire, ce type de système illustre particulièrement bien la problématique que nous allons traiter. Cependant, bien que nous nous soyons concentrés sur un type de système en particulier, tous les secteurs industriels concernés par l'utilisation de systèmes TR²E critiques sont confrontés à la nécessité d'équiper ces systèmes de capacités d'adaptation ; ne serait-ce que parce que l'adaptation d'un système est une solution au problème de la tolérance aux pannes.

Les systèmes autonomes trouvent de nombreuses applications, dans des domaines d'activités variés : qu'il s'agisse de systèmes de détection de mines (terrestres ou sous-marines), de systèmes d'exploration spatiale ou sous-marine, ou de systèmes de surveillance (observation de feux de forêts, de zones sinistrées par un tremblement de terre, de mouvements de foules, etc...), l'utilisation de systèmes autonomes semble particulièrement indiquée pour palier les difficultés d'établir un lien de communication fiable et/ou suffisamment rapide entre le système et un éventuel centre de contrôle. Dans ce mémoire, nous prendrons comme exemple applicatif de nos travaux le cas d'un robot mobile terrestre d'observation (UGV - *Unmanned Ground Vehicle*). Ce type de système nous semble particulièrement pertinent dans le cadre de notre étude puisqu'il regroupe l'ensemble des caractéristiques des systèmes TR²E critiques et adaptatifs :

- en tant que système composé de sous-systèmes eux-mêmes composés récursivement de sous-systèmes, l'UGV constitue un **système complexe** ;
- la nature des exigences allouées à un UGV, qui peut être amené à se diriger de façon automatique, en font un **système temps-réel**. En effet, le non respect des échéances temporelles du sous-système d'asservissement des commandes du véhicule constitue un cas d'erreur du système.
- en tant que système autonome doté de ses propres ressources de calcul, de ses propres ressources énergétiques, et en tant que système contraint en termes d'espace physique occupé, et de poids de sa charge utile, l'UGV constitue un **système embarqué**.
- en tant que système déployé sur le terrain et doté d'une certaine autonomie opérationnelle, l'UGV est un système "**mission-critical**", voire "**safety-critical**" s'il est amené à opérer en support d'une intervention humaine.
- en tant que système autonome devant s'adapter aux modifications de son environnement aussi bien qu'à son propre état d'intégrité, l'UGV constitue un **système adaptatif**.

Dans la suite de cette section, nous allons insister sur deux points importants parmi ceux que nous venons de présenter : la composition de ce système complexe, et les besoins d'adaptation de ce système. Nous insistons sur ces deux points dans la mesure où ils sont particulièrement dimensionnant pour l'élaboration de la problématique que nous allons traiter.

3.2.1 L'UGV, un système complexe

Le véhicule autonome que nous allons prendre comme exemple dans l'ensemble de ce mémoire est un système complexe dans la mesure où il est composé de plusieurs sous-systèmes, pouvant eux-mêmes être composés de sous-systèmes.

La figure 3.1 illustre l'architecture système de l'UGV, c'est-à-dire l'ensemble des sous-systèmes qui le composent, ainsi que leurs liens de communication. Sur cette figure, les ports de communication représentent des sources ou des puits d'évènements : les différents sous-systèmes qui composent l'UGV communiquent par le biais d'envois de messages asynchrones afin de concevoir chaque sous-système indépendamment de son déploiement : en effet, nous avons montré au chapitre précédent que dans le cas des systèmes critiques, les communications distantes doivent être asynchrones (voir section 2.5). Pour pouvoir déployer chaque sous-système indifféremment sur un même nœud ou sur des nœuds différents, il faut donc que ces systèmes communiquent de façon asynchrone.

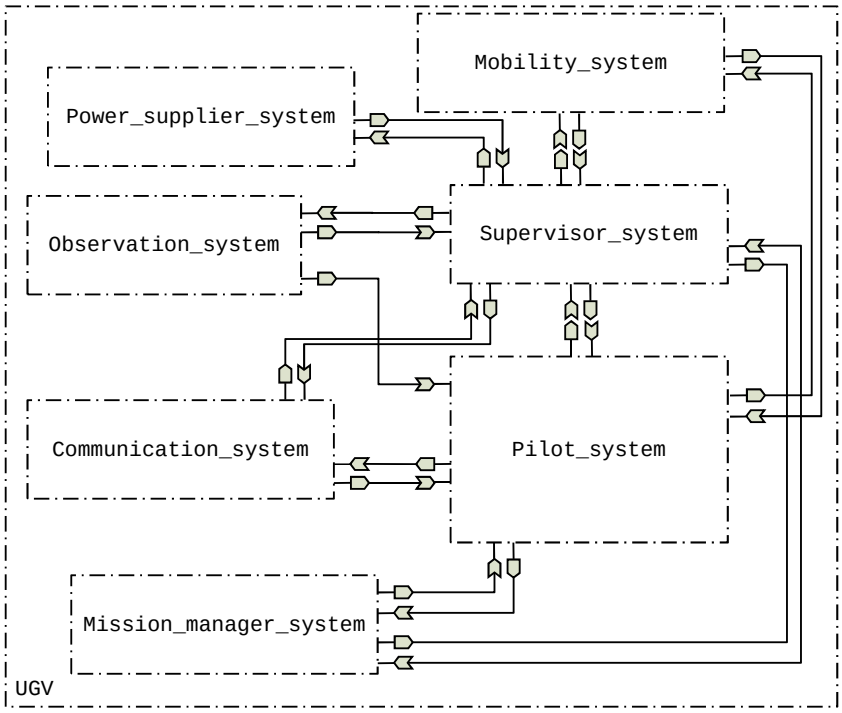

FIGURE 3.1 – Architecture système d'un UGV

Comme le montre la figure 3.1, notre UGV est constitué des sous-systèmes suivants :

- un sous-système de pilotage (`Pilot_system`) qui détermine les commandes de direction du robot ;
- un sous-système de gestion d'énergie (`Power_supplier_system`) qui supervise le niveau de charge de la batterie ;
- un sous-système d'observation, (`Observation_system`) qui fournit une image de la région observée et détecte les obstacles à proximité du véhicule ;
- un sous-système de communication (`Communication_system`) qui sert d'interface avec l'utilisateur final du robot (réception des données transmises par le robot, envoie de commande) ;

– un sous-système de supervision de mobilité (`Mobility_system`), qui contrôle le fonctionnement des attributs de mobilité du robot (moteur et roues) ;
– un sous-système de gestion de mission (`Mission_manager_system`), qui permet de préprogrammer le comportement du robot, tout au long de la mission qu'il doit accomplir (routage, modes de fonctionnement, etc...) ;
– un sous-système de supervision (`Supervisor_system`), qui gère les modes de fonctionnement du système global.

En outre, chacun de ces sous-systèmes possède un certain nombre de modes de fonctionnement, qui peuvent dépendre les uns des autres. Nous décrivons cela dans la sous-section suivante.

3.2.2 L'UGV, un système TR²E adaptatif

En général, un UGV possède plusieurs modes de fonctionnement. Un mode de fonctionnement représente de façon abstraite l'ensemble des fonctionnalités que le système (ou le sous-système) doit fournir lorsqu'il est dans ce mode de fonctionnement. Prenons un exemple simple, que nous utiliserons au cours de ce chapitre dans le but d'illustrer notre problématique, et dans la suite de ce mémoire afin de mettre en œuvre et d'évaluer nos contributions.

Spécification des modes de fonctionnement

Considérons, au sein de l'UGV, le cas d'un système de pilotage qui possède deux modes de fonctionnement. Un mode automatique, dans lequel le système détermine les commandes de direction en fonction de sa position courante, d'une route qui a été préprogrammée, et de son environnement tel que le perçoivent ses senseurs ; et un mode manuel dans lequel le système de pilotage détermine les commandes de direction en fonction d'instructions que l'opérateur lui envoie.

Spécification des changements de mode

Le système de pilotage passe du mode manuel au mode automatique sur commande de l'utilisateur, ou en cas de déconnexion du système de communication. De la même façon, le système passe du mode automatique au mode manuel sur demande de l'utilisateur, ou en cas de panne du sous-système de localisation, et à condition que le système de communication ne soit pas déconnecté. Il faudra alors s'assurer que le sous-système de guidage n'est jamais dans le mode automatique (respectivement manuel) quand le sous-système de localisation est dans le mode manuel (resp. automatique).

Le changement de mode *"automatique* $\rightarrow$ *manuel"* devra également respecter la séquence suivante : on commencera par arrêter le sous-système de localisation, ce qui permettra lorsque celui-ci aura confirmé son changement de mode, de basculer le sous-système de guidage dans le mode manuel. De la même façon, on démarrera le sous-système de localisation avant de faire basculer le sous-système de guidage dans le mode automatique. De plus, pour des raisons de tolérance aux pannes, il faudra pouvoir basculer dans un mode donné si le sous-système de localisation ne confirme pas suffisamment rapidement qu'il a bien atteint le mode cible de sa reconfiguration. Enfin, il faudra être en mesure d'assurer que le mécanisme d'adaptation qui correspond à la transition du mode manuel au mode

automatique se fait en moins de 200 millisecondes.

Cette spécification des modes et changements de mode du système de navigation fait apparaître un certain nombre de problèmes techniques que nous devrons traiter au cours de notre étude. Généralisons ces problèmes techniques.

3.2.3 Généralisation des problèmes techniques

Problème 1 ***: provoquer le changement de mode.***

Comme nous pouvons le constater sur l'exemple simple du système de pilotage, les conditions qui nécessitent un changement de mode peuvent être de nature très différentes, et répondre aussi bien à des exigences fonctionnelles (demande de l'utilisateur) que non-fonctionnelles (tolérance aux pannes). Cette variété impose de représenter les motifs de changement de mode de façon générique : n'importe quel composant de l'architecture peut provoquer un changement de mode.

Problème 2 ***: conditionner le changement de mode.***

Outre le problème de la détection des conditions qui nécessitent le changement de mode, se pose celui des conditions qui permettent le changement de mode. Le fait que le système ne puisse passer du mode automatique au mode manuel que si le système de communication est accessible est une contrainte importante ; de même que la contrainte de précédence entre l'établissement du mode de fonctionnement du sous-système de localisation et du sous-système de guidage.

Problème 3 ***: modifier le comportement du système.***

Un changement de mode s'accompagne souvent de modifications du comportement du système. Le comportement du système étant défini par la configuration de ses applications logicielles, un changement de mode modifie alors les caractéristiques de cette configuration. Ainsi, la configuration de l'application de navigation dans le mode manuel et dans le mode automatique seront différentes.

Problème 4 ***: analyser le changement de mode.***

La logique de changement de mode doit être analysable, de telle sorte (i) à garantir la cohérence des modes de fonctionnement d'un système, vis-à-vis des modes de ses sous-systèmes, et (ii) à garantir qu'un mécanisme d'adaptation (pouvant nécessiter plusieurs changements de mode) respecte une échéance temporelle prédéterminée. Nous devrons ainsi nous assurer que le sous-système de guidage n'est jamais dans le mode automatique (respectivement manuel) alors que le sous-système de localisation est dans le mode manuel (resp. automatique). Nous devrons également nous assurer que la transition du mode automatique au mode manuel ne prend jamais plus de 200 millisecondes.

Problème 5 ***: Tolérer les pannes externes.***

La logique des changements de mode doit permettre d'assurer que le système continuera de fonctionner en cas d'indisponibilité d'un sous-système donné. Il faudra ainsi s'assurer qu'une panne du système de localisation n'empêche pas de continuer d'utiliser le système en mode manuel.

Comme nous l'avons expliqué en introduction de ce chapitre, nous souhaitons bénéficier des techniques de configuration et d'analyse des systèmes TR²E pour faciliter la gestion de leurs reconfigurations dynamiques. Nous allons maintenant préciser le périmètre de notre problématique vis-à-vis de la (re)configuration dynamique en répondant à cette question : veut-on traiter les problèmes que posent la reconfiguration automatique, ou ceux soulevés par la reconfiguration pseudo-dynamique. Nous placerons bien sûr cette étude dans le contexte des systèmes critiques.

3.3 (Re)configuration automatique des applications TR²E critiques

Commençons par étudier l'idée selon laquelle la reconfiguration d'un système s'appuie sur des mécanismes de sélection automatique de la configuration à utiliser.

3.3.1 Reconfiguration automatique

La notion de configuration automatique des applications TR²E recouvre un large ensemble de travaux dont l'objectif est de déterminer automatiquement la configuration logicielle qui correspond le mieux aux besoins fonctionnels et non-fonctionnels d'un système donné. Ces exigences sont exprimées en amont du processus de développement, et une méthode de sélection associée détermine l'ensemble des configurations logicielles qui y répondent. Différentes techniques ont été utilisées dans le but de réaliser la configuration automatique de systèmes TR²E, comme par exemple la résolution de contraintes [Cui *et al.*, 2001], l'expression et l'utilisation des contrats [Chang and Collet, 2007; Grondin *et al.*, 2006], ou encore la définition et l'utilisation des ontologies [Fujii and Suda, 2004].

La mise en œuvre d'une telle approche nécessite alors de modéliser les contraintes fonctionnelles et/ou non-fonctionnelles d'une application, et d'automatiser la production de la configuration logicielle qui répond le mieux à ces contraintes.

Dans le cadre des mécanismes d'adaptation des systèmes TR²E, l'utilisation de ce type de solution augmente fortement l'autonomie des systèmes adaptatifs [Grondin *et al.*, 2006]. En effet, la capacité de déterminer automatiquement la configuration logicielle qui répond le mieux aux caractéristiques de l'environnement d'exécution d'un système – en termes d'exigences fonctionnelles et non-fonctionnelles – lui permettrait de réagir de façon totalement autonome aux variations de ces caractéristiques.

En revanche, la mise en œuvre de telles approches est compromise dans le cadre de la réalisation de systèmes TR²E critiques : la réalisation de tels systèmes nécessite de garantir que son comportement vérifie un certain nombre de propriétés de sûreté de fonctionnement. Par exemple, que le processus d'adaptation se fait en respectant une échéance temporelle fixe. Comme nous pourrons le constater dans la suite de ce mémoire, fournir ce type de garantie pendant les phases de conception du système est une tâche complexe. Aujourd'hui, nous ne disposons d'aucune méthode qui permette de fournir ce type de garantie alors que les modalités de mise en œuvre de la reconfiguration au sein du système réel ne sont pas intégralement connues à l'avance.

Nous allons donc abandonner cette problématique de la reconfiguration automatique, qui ne nous semble pas appropriée à la réalisation des systèmes critiques.

3.3.2 Configuration automatique de la reconfiguration

Intéressons nous maintenant au concept de reconfiguration “pseudo-dynamique”, c’est-à-dire aux mécanismes de reconfiguration ayant lieu alors que le système est en cours de fonctionnement, mais dont on connaît exhaustivement, pendant les différentes phases de conception, l’ensemble de ses configurations possibles, ainsi que leurs conditions de mise en œuvre. Cette idée n’est pas nouvelle. Elle a notamment été introduite dans le langage AADL à travers la notion de “mode opérationnel”. Cette notion a d’ailleurs été utilisé dans le but d’analyser formellement des spécifications AADL qui modélisent des reconfigurations “pseudo-dynamiques” [Bertrand *et al.*, 2008; Rolland, 2008]. Le problème que nous avons traité au cours de nos travaux étend ses travaux, puisqu’il permet non seulement d’analyser, mais surtout de synthétiser des systèmes “pseudo-dynamiquement” reconfigurables.

La notion de configuration automatique des applications TR²E recouvre également, en aval du processus de développement, la capacité à **automatiser la production et l’analyse** d’une telle application. Ce type de problématique est aujourd’hui traité par l’utilisation de techniques de génération de code et d’analyse de modèles formels [Halbwachs *et al.*, 1991; Weil *et al.*, 2000; Zalila *et al.*, 2008]. Le domaine de la génération de code et de la vérification formelle a été largement marqué par l’essor des langages synchrones tels que Lustre [Halbwachs *et al.*, 1991] et Esterel [Weil *et al.*, 2000]. Ces langages, et les compilateurs associés, ont été largement utilisés dans le cadre des systèmes les plus critiques, dans le domaine de l’avionique, du spatial, ou encore du nucléaire. Cependant, ces techniques s’appuient sur une hypothèse forte, qui consiste à considérer que les différentes actions spécifiées dans ces langages se font chacune dans un temps nul. L’analyse formelle de ces langages s’appuie fortement sur cette hypothèse. En ce qui concerne l’implémentation des actions en question, l’hypothèse utilisée est alors moins forte (et plus réaliste) : elle consiste à considérer que les actions sont réalisées dans un temps borné dont on connait les limites.

Les systèmes embarqués modernes sont de plus en plus complexes, et force est de constater que les hypothèses sur lesquelles s’appuient les techniques relatives aux langages synchrones ne sont plus valables dans le cadre de ces systèmes [Garavel and Thivolle, 2009].

Afin de faciliter la conception des systèmes TR²E critiques pour lesquelles les hypothèses synchrones ne s’appliquent pas, les techniques d’Ingénierie Dirigée par les Modèles (IDM) sont aujourd’hui considérées comme une solution prometteuse. Dans ces approches, il est fréquent de séparer la réalisation du code métier (qui vise à répondre aux besoins fonctionnels de l’application) du code technique (qui vise à prendre en charge la réponse aux besoins non-fonctionnels). Cette démarche porte traditionnellement le nom de “séparation des préoccupations” [Reade, 1989]. La mise en œuvre de cette démarche repose le plus souvent sur un intergiciel, qui constitue la couche d’adaptation entre le code fonctionnel d’une part, et le code technique d’autre part. Un certain nombre de services ont été définies comme étant l’ensemble de services minimal que doit fournir un intergiciel [Pautet, 2001]. Dans le but d’optimiser l’utilisation des ressources d’exécution, le code de ces services peut-être constitué par une certaine quantité de code minimal (qui constitue l’intergiciel proprement dit) et par une certaine quantité de code généré, produit en fonction des spécifications issues des processus de conception logicielle (dirigée par les modèles) [Zalila, 2008].

Cependant, si de telles approches permettent de faciliter la réalisation des systèmes critiques [Hamid *et al.*, 2008; Zalila, 2008], elles ne prennent pas en compte les problèmes liés

à la réalisation des systèmes TR²E adaptatifs. En effet, la gestion des mécanismes d'adaptation d'une application TR²E ne fait pas partie des services habituels d'un intergiciel. Pourtant, comme nous avons pu le constater dans ce chapitre (voir section 3.2), la mise en œuvre de protocoles de reconfiguration nécessite de synchroniser le processus de changement de mode avec certains des services de l'integiciel, et en particulier le service d'activation.
Le bilan de cette sous-section est riche d'enseignements. Tout d'abord, les mécanismes de reconfiguration pseudo-dynamique, ainsi que les applications configurées statiquement par des méthodes d'IDM, sont analysables. Cependant, ces approches de déploiement statique ne prennent pas en compte la notion de reconfiguration pseudo-dynamique, dont la mise en œuvre nécessite de compléter et/ou d'utiliser les services fournis par un intergiciel. Dans la sous-section suivante, nous allons détailler davantage les enjeux de la mise en œuvre de la reconfiguration pseudo-dynamique.

3.3.3 Mise en œuvre de la reconfiguration pour les systèmes adaptatifs

Le principal problème, lorsqu'il s'agit de modifier le comportement d'une application, consiste à déterminer "l'état stable" (en anglais *safe state*) du système qu'elle pilote. Cet état correspond à l'ensemble des conditions qui garantissent que la reconfiguration de l'application ne risque pas d'altérer le bon fonctionnement du système [Moazami-Goudarzi, 1999]. Dans [Goudarzi and Kramer, 1996], l'état stable est atteint en isolant du reste de l'application les nœuds à reconfigurer. Ceci nécessite alors d'enregistrer les requêtes qui auraient été émises vers ce nœud, puis de les ré-émettre une fois la reconfiguration du nœud en question terminée. Cette technique est raffiné dans [Polakovic *et al.*, 2007] : la méthodologie présentée consiste à isoler un ensemble de composants logiciels plutôt que l'intégralité du nœud. Le but de ces travaux est de faciliter les évolutions du logiciel tout en garantissant qu'il continue de rendre certains services. La reconfiguration sert alors à la maintenance en ligne (corrective ou évolutive) du logiciel, mais ne constitue pas une solution à la réalisation de systèmes adaptatifs.

Pour illustrer la notion d'état stable, reprenons l'exemple du système de pilotage. Supposons que les fonctionnalités fournies par le sous-système de guidage dans le mode automatique soient réalisées par l'intermédiaire de deux tâches, *auto_pilot_activity* et *guidance_activity*. Nous faisons maintenant l'hypothèse que ces tâches partagent une donnée de telle sorte qu'un verrou exclus l'accès mutuel des tâches à cette donnée. Supposons également que pour modifier le comportement de ce sous-système lors du passage du mode automatique au mode manuel, il faille modifier le flux d'exécution de la tâche *guidance_activity*, et que dans le mode manuel, celle-ci n'utilise pas le verrou. Dans ce cas, si le changement de mode a lieu alors que *guidance_activity* a pris le verrou mais ne l'a pas encore rendu, alors il y a tout lieu de penser que l'application aboutira à une situation d'inter-blocage. [Sha *et al.*, 1988] propose un protocole de reconfiguration particulier pour palier au problème de la continuité du flux d'exécution dans les sections critiques : le changement de mode n'a pas lieu tant qu'une tâche est dans une section critique. Comme nous allons le constater dans la suite de ce chapitre, ce protocole ne répond pas à l'ensemble des enjeux de la mise en œuvre de la reconfiguration pseudo-dynamique.

En effet, cette question de la continuité des flux d'exécution dans le sections critiques est orthogonale à la nécessité de garantir un état cohérent des données du système. Ceci est d'autant plus significatif dans le cas de systèmes temps-réel, puisque la cohérence des données dépend du moment où celles-ci sont produites. Ainsi, le fait de modifier le comportement d'un système TR²E en cours d'exécution peut amener ce dernier à produire des données in-

cohérentes entre elles si la reconfiguration n'est pas effectuée au "bon moment". Illustrons ce problème à travers l'exemple du système de navigation. Nous considérons ici un changement de mode de type "nominal $\rightarrow$ dégradé". Nous supposons que le sous-système de localisation est composé d'une tâche périodique (*positioning_activity*) qui envoie toutes les 10 millisecondes la position courante du robot au sous-système de guidage selon un format de données très précis dans le mode nominal, et un format de données moins précis dans le mode dégradé. Ce message est alors traité par la tâche *auto_pilot_activity* du sous-système de guidage pour calculer, toutes les 10 millisecondes, les commandes de direction du robot. Dans ce sous-système de guidage, la tâche *guidance_activity* calcule toutes les 100 millisecondes la vitesse du robot, à partir des dix dernières valeurs de position publiées. Supposons que le changement de mode *"nominal $\rightarrow$ dégradé"* ait lieu après l'envoi de la valeur des deux dernières positions. Dans ce cas, lorsque le système reviendra dans le mode nominal, il est évident que les 8 valeurs qui viendront compléter les deux valeurs restées en suspens aboutiront à un calcul de vitesse incohérent, et cela risque de constituer un cas de panne du système. Dans la suite de ce rapport, nous considérerons que les solutions à ce type de problème visent à garantir la "disponibilité de flux de données cohérents".

Voici donc la question qui nous intéresse : quel est "l'état stable " d'une application TR²E ? Cette question a principalement été traitée par la communauté scientifique en s'intéressant soit

- aux algorithmes ou services que doit utiliser le développeur de l'application pour permettre de mettre en œuvre la reconfiguration dynamique [Polakovic and Stefani, 2008; Schneider *et al.*, 2004; Real and Wellings, 1999],
- soit par le biais d'une analyse formelle du système en cours de reconfiguration dans le but de garantir un certain nombre de propriétés relatives à la reconfiguration dynamique [Apvrille *et al.*, 2004b; Pedro and analysis for, 1998; Real and Crespo, 2004],
- soit enfin dans le cadre de la synthèse d'automates de mode s'appuyant sur les langages synchrones [Maraninchi and Rémond, 1998; Labbani *et al.*, 2005].

Malheureusement, aucun de ces travaux ne répond à l'ensemble des problèmes que nous avons présenté jusque là : certains ne s'intéressent pas spécifiquement à la reconfiguration des systèmes critiques [Polakovic and Stefani, 2008; Schneider *et al.*, 2004]. Les exigences de réalisation de la plupart des systèmes TR²E modernes invalident les hypothèses fondatrices de l'utilisation des langages synchrones [Berry, 2000]. Les travaux qui s'intéressent à l'analyse de la reconfiguration dynamique ne proposent pas d'algorithmes de mise en œuvre de cette reconfiguration [Pedro and analysis for, 1998; Real and Crespo, 2004]. Enfin, [Real and Wellings, 1999] ne s'intéresse pas à la problématique de la gestion des mécanismes d'adaptation en tant que succession de changements de mode (ce qui nécessite la mise en oeuvre d'analyses particulières), ni aux différents enjeux relatifs à un changement de mode (temps de reconfiguration, intégrité des flux de donnés).

S'ils ne répondent pas directement à notre problématique, ces travaux sont riches d'enseignements : l'hétérogénéité forte des protocoles de reconfiguration montre que la gestion d'un changement de mode est un problème multicritères : la réduction du temps de reconfiguration, la réduction du temps d'interruption de service, la garantie de la disponibilité de données cohérentes, la capacité à analyser l'ordonnançabilité, etc... font partie de ces critères [Real and Crespo, 2004]. Pour illustrer ce besoin de répondre à plusieurs critères, prenons un exemple simple : lorsqu'un système autonome (type robot mobile d'observation) détecte un niveau de batterie insuffisant, il faut qu'il puisse réagir pour se mettre dans une configuration "sûre" (par exemple, interrompre sa progression et signaler sa position). Plusieurs solutions sont alors possibles pour permettre cette modification de comportement : le système peut soit (i) attendre

que les conditions d'adaptation soient réunies, soit (ii) interrompre son travail pour traiter plus rapidement le besoin de changement de comportement. En d'autres termes, la reconfiguration peut être privilégiée par rapport à la réponse aux autres besoins du système, ou inversement. Lorsque la reconfiguration est privilégiée, le but est alors de réduire le temps de reconfiguration, c'est à dire le temps qui sépare l'émission d'un ordre de changement de mode de la réception d'une confirmation que le mode cible a bien été atteint. Dans le cas contraire, le mécanisme de reconfiguration vise à réduire les perturbations que le système subit lors de la reconfiguration en privilégiant l'exécution normale des tâches en cours. De façon orthogonale, il faudra pouvoir traiter le problème de la cohérence des flux de données.

Ainsi, une des difficultés majeures de la reconfiguration pseudo-dynamique des systèmes TR²E critique réside dans la quantité de critères de mise en œuvre d'un changement de mode. En effet, ces critères vont être très différents d'un système à l'autre, et d'une reconfiguration à l'autre. Ceci est d'autant plus vrai que cette mise en œuvre va nécessiter, dans le cas de systèmes répartis, des mécanismes de transactions qui permettront de synchroniser les différents sous-systèmes sur un état global de l'application.

Nous devrons donc, au cours de notre étude, proposer différents protocoles de reconfiguration qui permettront de répondre à ces critères, tout en intégrant leur spécification dans une méthodologie qui permette de spécifier, d'analyser et de produire automatiquement le comportement global du système en cours de reconfiguration.

Résumons les problèmes que nous avons identifié au cours de cette section :

Problème 6 ***: isoler les critères pertinents d'un changement de mode.***

Pour répondre à la problématique générale que nous avons défini dans ce chapitre, nous devrons isoler les cirtères de reconfiguration pertinents dans le cadre de notre étude.

Problème 7 ***: protocoles de reconfiguration pseudo-statique.***

Une fois le problème 6 résolu, nous devrons définir des protocoles de reconfiguration pseudo-dynamique qui permettent d'atteindre l'état stable du système en privilégiant le(s) critère(s) pertinent dans le cadre du changement de mode concerné.

Au cours de cette section, nous avons présenté les enjeux majeurs de la (re)configuration dynamique des systèmes TR²E critiques et adaptatifs. Pour compléter cette présentation, nous allons maintenant détailler les contraintes auxquelles notre approche devra se confronter.

3.4 Contraintes de configuration des systèmes critiques adaptatifs

Commençons par les contraintes qu'impose l'utilisation de notre solution dans le contexte industriel de la société Thales.

3.4.1 Contraintes de mise en œuvre industrielle

La mise en œuvre de la séparation des préoccupations dans un contexte industriel tel que celui de la société Thales est fortement facilitée par l'utilisation de modèles à base de

composants [Borde *et al.*, 2008]. Ces modèles permettent en effet de regrouper et de représenter efficacement les fonctionnalités offertes et/ou requises par les différentes entités qui constituent l'application. En revanche, les modèles à base de composants constituent un héritage des méthodes de développement des systèmes d'information. S'ils sont adaptés pour la spécification des interfaces fonctionnelles et leurs implémentations, ils ne permettent pas de représenter efficacement les éléments architecturaux propres au domaine des systèmes TR²E critiques reconfigurables. La prise en compte de ces caractéristique a été à l'origine du succès du langage AADL [SAE, 2004], puis du profile UML/MARTE [Mallet *et al.*, 2009].

Pour analyser les caractéristiques d'une application logicielle, il faut représenter les propriétés de cette application. Certaines propriétés se retrouvent d'un système à l'autre ; en particulier lorsque ces systèmes appartiennent au même domaine d'activité. Un langage spécifique à un domaine définit les propriétés nécessaires à l'analyse d'une application logicielle de ce domaine. Plus la sémantique d'éxécution définie par ce langage est précise, plus l'analyse sera fiable. Les langages synchrones ([Halbwachs *et al.*, 1991; Berry, 2000]) ont fondé leur succès sur une sémantique formelle qui facilite l'analyse du code produit. Le langage de description d'architecture AADL [SAE, 2004] connait aujourd'hui un essor important du fait de la précision de sa sémantique et de son adéquation avec les spécificités des systèmes TR²E. AADL a ainsi été utilisé dans de nombreux travaux de recherche qui ont montré que sa sémantique était suffisamment précise pour se prêter à certains types de vérification, en particulier de vérification d'ordonnançabilité [Singhoff *et al.*, 2005]. Malheureusement, AADL 1.0 ne définit pas la notion de composant logiciel générique (ensemble de fonctionnalités offertes et/ou requises).

Nous devrons donc réconcilier ces deux approches, pour bénéficier à la fois des avantages de la séparation des préoccupations, mais également des capacités d'analyse des langages spécifiques à un domaine (DSL), dans le but de générer le code spécifique à la mise en œuvre des systèmes TR²E critiques et adaptatifs.

En outre, si AADL définit le concept de mode opérationnel, et UML fournit les diagrammes de type *"state-charts"*, aucune de ces approches d'IDM n'offre de méthodologie pour automatiser la production de systèmes TR²E critiques et **adaptatifs**.

Problème 8 ***: réconcilier DSL et composants logiciels génériques.***

Notre approche devra bénéficier des avantages des DSL et des approches à base de composants logiciels génériques.

Dans la suite de cette section, nous présentons les contraintes spécifiques à la réalisation des systèmes critiques.

3.4.2 Contraintes propres aux systèmes critiques

La réalisation de systèmes critiques nécessite de respecter un certain nombre de critères de conception et d'implémentation dans le but d'augmenter le déterminisme de l'application produite. Nous allons détailler ici ces critères, principalement issus de l'état de l'art que nous avons réalisé (voir chapitre 2).

Contraintes de réalisation

Les contraintes de réalisation des systèmes TR²E critiques visent principalement à augmenter les possibilités d'analyse de telles applications.

Le profil Ravenscar d'Ada [Burns, 1999] a ainsi été défini dans le but de faciliter l'analyse d'ordonnançabilité des applications produites en respectant ce profil. Ce profil limite principalement le modèle de concurrence de l'application en limitant son utilisateur à la définition de groupes de tâches définies statiquement, et dont l'activation est cyclique. Ce profil limite également le modèle de communication en interdisant l'utilisation de mécanismes de communication distantes bloquantes type RPC (hors "one way"). Ce type de contrainte permet d'améliorer les possibilités de vérification **temporelle** des systèmes TR²E.

Les contraintes d'implémentation imposées lors de la réalisation des systèmes critiques visent principalement à augmenter l'analysabilité du code **binaire** produit (espace mémoire nécessaire, absence de références nulles, etc...). Ces limitations concernent principalement : l'absence d'allocation dynamique de mémoire, l'absence d'appels récursifs, l'absence de boucles infinies, et l'utilisation de types de données de taille bornée.

Vérification formelle des architectures logicielles

La vérification formelle des architectures logicielles s'appuie sur différentes techniques qui visent à améliorer la confiance que l'on peut avoir dans une conception logicielle. Ces techniques sont relativement difficiles à utiliser, si bien qu'il est le plus souvent envisagé de masquer cette complexité en automatisant la production des modèles formels correspondant à une spécification logicielle [Haddad *et al.*, 2006]. Dans ce cas, il est nécessaire de garantir que ce qui est analysé formellement correspond à ce qui est ou sera réellement implanté.

L'utilisation d'un standard assure l'interopérabilité de différentes méthodes de conception, d'analyse, et de production du logiciel. Par ailleurs, bénéficier de tels outils permet de renforcer le standard en question en garantissant que ses constructions sont formellement analysables, et que l'implantation de sa sémantique est automatisable. Cependant, rien ne garanti alors que la sémantique analysée correspond bien à la sémantique implantée, surtout si l'analyse ne s'appuie pas directement sur la sémantique d'exécution telle qu'elle est réellement implantée, mais seulement sur la spécification du standard.

Pour répondre à cette question, nous proposons que le modèle formel soit produit parallèlement au code généré dans le but de configurer l'application, ce qui réduit la distance sémantique entre ce qui sera finalement implanté et ce qui sera vérifié par ailleurs. Dans le cadre de notre étude, nous devrons donc fournir les moyens de vérifier qu'un système TR²E adaptatif respecte les propriétés que nous avons exprimées précédemment (voir sous-section 3.2.2).

Nous allons maintenant aborder une contrainte importante de la réalisation des systèmes TR²E critiques : la gestion de la répartition.

3.4.3 Gestion de la Répartition

La gestion de la reconfiguration dynamique des systèmes répartis est un problème complexe dans la mesure où il couvre un large spectre de domaines d'expertise, comme par exemple la tolérance aux pannes et l'algorithmique répartie. En effet, la gestion de la reconfiguration dans un système TR²E critique nécessite de s'assurer que la reconfiguration du

système se fera dans un temps borné, même en cas de panne. Dans la suite de ce mémoire, **nous ferons l'hypothèse simplificatrice d'un réseau synchrone**, ce qui permet d'augmenter le déterminisme temporel des systèmes TR²E.

3.5 Synthèse

Objectifs de Notre Contribution Notre objectif principal consiste à proposer une méthodologie de conception qui s'appuie sur les techniques modernes de conception et de réalisation des systèmes TR²E critiques pour traiter les spécificités des systèmes TR²E critiques et adaptatifs. Une telle méthodologie devra donc :

1. fournir une syntaxe de modélisation des caractéristiques temps-réel de l'architecture, tout en bénéficiant des avantages des techniques industrielles de mise en œuvre de la séparation des préoccupations ;
2. fournir une syntaxe de modélisation des modes de fonctionnement et des mécanismes de changements de mode d'un système TR²E ;
3. automatiser l'analyse des protocoles de synchronisation des modes d'un système et de ses sous-systèmes ;
4. imposer le respect de contraintes de conception et de réalisation de systèmes critiques ;
5. fournir une syntaxe de modélisation des différentes configurations logicielles associées aux modes de fonctionnement du système ;
6. définir et s'appuyer sur les protocoles de synchronisation qui permettent d'atteindre "l'état stable" d'un système TR²E adaptatif ;
7. faciliter la résolution du compromis entre les différentes caractéristiques d'un changement de mode (rapidité, interférence, cohérence des flux de données) ;
8. automatiser l'analyse du comportement du système en cours de reconfiguration (y compris en termes d'exigences temporelles) ;

Dans la suite de ce mémoire, nous allons présenter (chapitre 4) l'approche générale que nous avons adoptée afin de répondre à l'ensemble des problématiques, contraintes et objectifs que nous venons de présenter. Nous illustrerons ensuite (chapitre 5, 6 et 7) cette méthodologie à travers la description d'outils et de résultats qui nous ont permis de la mettre en œuvre. Enfin, nous validerons ces résultats au travers d'expérimentations que nous avons menées autour du système de navigation que nous avons présenté dans ce chapitre.

Troisième partie

Démarche scientifique

Chapitre 4

Mise en œuvre et analyse de la reconfiguration dynamique

Un problème sans solution est un problème mal posé.
Albert Einstein

SOMMAIRE

4.1 Introduction

Rappel des objectifs. Les travaux que nous avons référencés dans l'état de l'art et la problématique que nous avons présenté n'ont pas réussi à répondre à l'ensemble des objectifs que nous nous sommes fixé. Ce constat trouve différentes sources d'explication :

– les langages de description d'architecture spécifiques à un domaine, tel que AADL 1.0, ne définissent pas la notion de composant logiciel générique ;

- les travaux relatifs à la vérification formelle de systèmes TR²E critiques et adaptatifs [Rolland, 2008] s'appuient sur une sémantique d'exécution dont l'implantation n'est pas générée. Se pose alors la question de la distance sémantique entre ce qui est implanté et ce qui est analysé ;
- la définition de protocoles de reconfiguration pseudo-dynamique [Real and Wellings, 1999; Real and Crespo, 2004] ne prend pas en compte l'ensemble des contraintes de mise en œuvre de mécanismes de changement de mode. De plus, ces travaux ne permettent pas de mener l'ensemble des analyses nécessaires à la réalisation de systèmes TR²E critiques et adaptatifs ;
- les travaux basé sur l'IDM et la génération de code ne prennent pas en compte la notion de mode de fonctionnement [Zalila, 2008].

Notre objectif principal consiste donc à définir une méthodologie de conception des systèmes TR²E critiques et adaptatifs, et à prouver sa faisabilité.

Démarche scientifique. L'approche que nous présentons dans ce chapitre s'inspire des différents travaux que nous avons présenté, et définit une méthodologie qui répond à l'ensemble des objectifs et problèmes décrits au chapitre 3. Pour ce faire, voici la démarche que nous avons suivi : nous avons commencé par définir les entités de modélisation propres aux systèmes TR²E critiques et adaptatifs. Nous avons ensuite proposé une méthodologie qui permette d'automatiser au maximum les différentes étapes de conception, d'analyse et de production d'un système TR2E critique et adaptatif [**?**]. Pour prouver la faisabilité de cette méthode, nous avons prototypé chacune des étapes de cette méthodologie, en commençant par l'étape de génération de code dédié aux mécanismes d'adaptation : ceci nous a permis de définir clairement leur sémantique d'exécution. Nous nous sommes ensuite assuré que cette sémantique était analysable.

Organisation du chapitre. Dans ce chapitre, nous présentons (section 4.2) une méthodologie qui améliore la productivité des développements logiciels des systèmes TR²E critiques et adaptatifs. Au cours de la section 4.3, nous décrivons le cas d'étude industriel qui nous permettra de prouver la faisabilité de notre approche. Enfin, la section 4.4 fera l'objet d'une définition de l'ensemble des protocoles de reconfiguration que nous avons sélectionné pour répondre aux enjeux de la mise en œuvre de mécanismes d'adaptations dans le contexte d'applications TR²E critiques.

4.2 Méthodologie

4.2.1 Modes de fonctionnement : spécification système, impact logiciel

Pour commencer, présentons le statut habituel des modes de fonctionnement dans le processus de développement en cycle en V des systèmes TR²E. Comme nous avons pu le constater lors de notre description de l'UGV, la spécification d'un système TR²E commence généralement par une décomposition de ce dernier en un ensemble de sous-systèmes. L'ensemble des comportements attendus du système est ensuite spécifié au cours d'une phase d'énumération des modes de fonctionnement de ce système et de ses sous-systèmes. Les conditions qui permettent et/ou nécessitent un changement de mode sont alors définies. Enfin, bien que cette étape n'ait pas encore été illustrée autour de l'exemple de l'UGV, la phase

de conception logicielle permet de définir les configurations logicielles qui correspondent à ces comportements.

La figure 4.1 détaille le processus de développement en cycle en V d'un système TR²E adaptatif. La partie de gauche de cette figure illustre ainsi, de haut en bas, les phases de spécification, de conception logicielle, et d'implémentation d'une application TR²E. Ces différentes phases sont séparées par des flèches verticales, et sont composées d'étapes intermédiaires afin de mettre en évidence celles qui concernent les modes de fonctionnement.

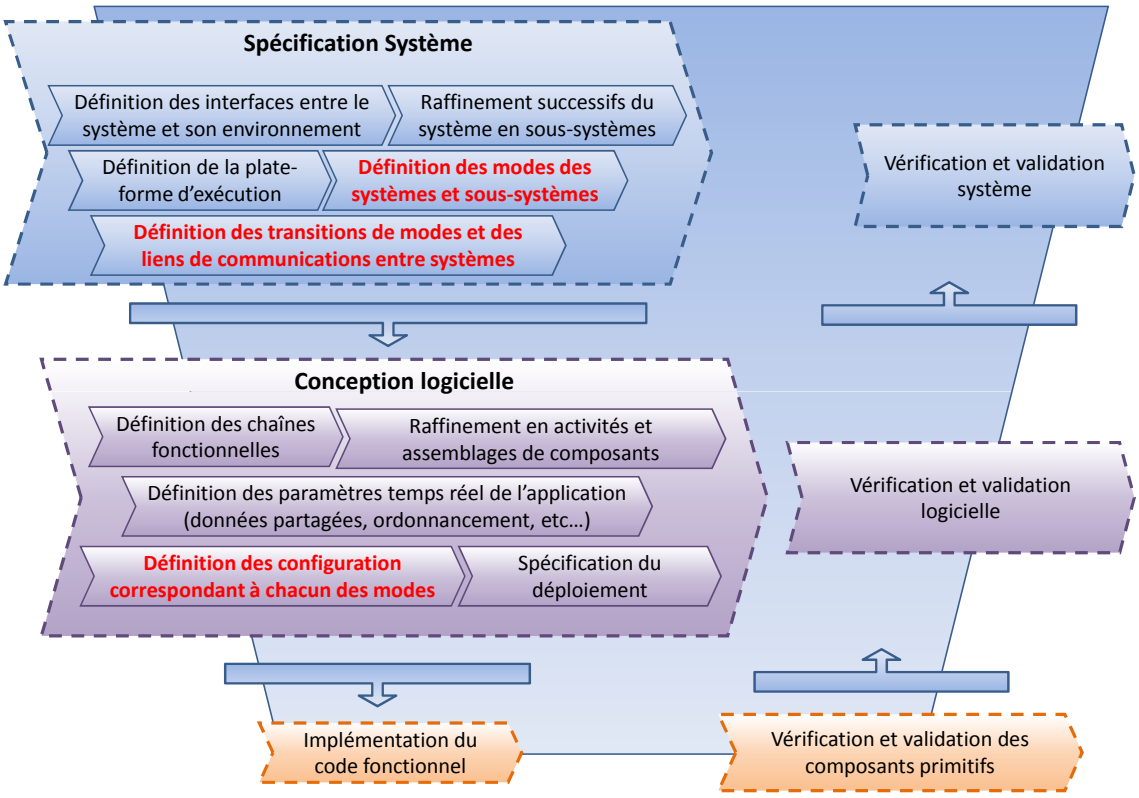

FIGURE 4.1 – Processus de conception d'un système TR²E reconfigurable

Sur cette figure, nous constatons que la spécification des modes de fonctionnement d'un système TR²E fait partie de la phase de spécification système, alors que la définition des configurations correspondantes à ces modes fait partie de la phase de conception logicielle. Faute de méthodologie prenant en compte ces deux niveaux de spécification, les mécanismes de changement de modes sont implantés manuellement au sein du code fonctionnel des applications. Ceci rend ces mécanismes d'autant plus difficiles à maintenir et à analyser que leur implémentation est enfouie dans le code de l'application.

La méthodologie que nous présentons dans ce chapitre automatise les étapes d'analyse et de production de telles applications, depuis leur spécification système jusqu'à leur implantation réelle sur la plate-forme cible. Pour ce faire, elle s'appuie sur trois étapes :

- une étape d'analyse de la spécification système ;
- une étape d'analyse du comportement temporel de l'architecture logicielle ;
- une étape de production du code correspondant aux spécifications systèmes et logicielles.

La mise en œuvre de ces étapes nécessite de modéliser conjointement l'architecture système et logicielle : l'architecture système contient la spécification des modes de fonctionnement, tandis que l'architecture logicielles définit les configurations qui correspondent à ces modes. Commençons par décrire la façon dont nous avons choisi de modéliser le comportement dynmique associé à un système TR²E critique et adaptatif.

4.2.2 Modéliser les modes et changements de mode du système

Dans cette section, nous présentons la méthode que nous avons élaboré pour représenter du comportement dynamique d'une application TR2E. Par comportement dynamique, nous incluons à la fois la possibilité de représenter les modes de fonctionnement de l'application, les mécanismes qui permettent de passer d'un mode à l'autre, mais également la possibilité d'associer à un mode de fonctionnement donné une configuration logicielle représentative de l'ensemble des fonctionnalités que doit fournir le système dans ce mode de fonctionnement.

Mode de fonctionnement

Un mode de fonctionnement représente un état du système dans lequel celui-ci doit fournir un ensemble de fonctionnalités. Lorsqu'un système détecte des variations significatives de son environnement d'exécution, celui-ci doit s'adapter en modifiant ses caractéristiques fonctionnelles, c'est dire en modifiant l'ensemble de fonctionnalités qu'il fournit.

Déclencher un changement de mode

Comme nous l'avons montré au chapitre 3 (Problème 1), un changement de mode peut-être provoqué pour différentes raisons, de nature très variées. La détection d'une panne matériel peut provoquer le passage d'un mode "nominal" vers un mode "dégradé". Une étape quelconque dans le cycle de vie du système qui, entrant dans une zone géographique va ralentir et affiner ses capacités d'observation peut se traduire par un changement de mode. Une requête d'un utilisateur final, qui va demander à prendre le contrôle du système qui passera ainsi du mode "automatique" au mode "manuel". Compte-tenu de cette variabilité, nous n'allons pas ici proposer de méthode de détection automatique des conditions qui nécessitent un changement de modes. Nous considérons en effet que le besoin est trop vaste, et qu'il doit être implanté soit (i) dans le code fonctionnel de l'application dont sont responsables les implémentations de composants gros grain, soit (ii) en étendant notre méthodologie (pour traiter par exemple le cas de déclenchements suite à une panne).

Dans notre approche, les composants de l'architecture logicielle peuvent envoyer une commande de changement de mode à l'infrastructure d'exécution de l'application logicielle afin que celle-ci réalise le changement de mode en question. Pour ce faire, il est nécessaire de modéliser (i) le lien de communication entre le composant responsable du déclenchement du changement de mode, et l'infrastructure d'exécution de l'application TR²E, et (ii) le changement de mode qu'occasionne la réception d'une commande de changement de mode provenant de ce composant. Enfin, nous devrons également modéliser le fait que le changement de mode d'un système donné peut provoquer un changement de mode d'un autre système.

Toutes ces considérations nous ont finalement amené à modéliser les modes de fonctionnement et leurs transitions par le biais d'automates communicants appelés **"automates de modes"**. Au sein de la représentation de l'architecture logicielle, ces automates sont connec-

tés aux composants fonctionnels de l'architecture qui peuvent ainsi envoyer leurs requêtes de changement de mode.

Cette approche permet de répondre au problème 1 (voir chapitre 3).

Conditionner le changement de mode

Nous avons restreint l'expressivité des automates de mode :

- nous considérons qu'un automate de mode est un composant logiciel prédéfini qui ne peut communiquer avec les autres composants et/ou automates de mode que par le biais d'envoi de messages qui ne transporteront que des données de type énuméré ; la valeur transmise correspond alors à une requête de changement de mode particulière, ou à un statut qui renseigne sur l'état courant de l'automate ;
- un automate de mode peut par ailleurs définir un (ou plusieurs) attribut(s) logiciel(s) de type entier afin d'implanter un (des) compteur(s).

Cette restriction n'est pas limitative dans le cadre de la réalisation des systèmes TR2E adaptatifs : la spécification des modes et des changements de mode d'un système est une tâche complexe ; la restriction des types de données associés à ces automates diminue cette complexité, sans pour autant limiter les capacités d'expression du comportement dynamique du système.

Notre démarche de modélisation permet alors de représenter la logique de changement de mode d'un automate : lorsqu'un automate de mode reçoit un ordre de changement de mode, celui-ci évalue une condition booléenne qui conditionne le passage vers un mode cible. Cette condition booléenne peut être composée :

- de la comparaison du contenu d'un (ou d'une conjonction de) message(s) dont la valeur correspond à la valeur énumérée attendue pour que le changement de mode ait lieu ;
- et/ou de la comparaison des attributs de l'automate avec des valeurs constantes.

Enfin, nous proposons de compléter la définition des changements de mode avec un ensemble d'actions, qui se composent :

- des émissions de message (de type énumérés) au travers des sources d'évènements de l'automate qui pourront être soit des ordre de reconfiguration à destination d'autres automates de modes, soit des messages de statut à destination des composants fonctionnels pour confirmer l'atteinte d'un mode cible.
- et/ou de l'initialisation d'un ou de plusieurs attributs de l'automate de mode.

Ces principes de représentation permettent de répondre au problème 2 exprimé au chapitre 3. Par ailleurs, les limitations de modélisation que nous avons décrit ici facilitent l'analyse des changements de mode et contribuent donc à répondre au problème 4.

Résister aux pannes d'un sous-système L'automate de mode peut également spécifier des transitions de mode afin d'empêcher une attente infinie sur un message qui aurait du provenir d'un système en panne. Pour ce faire, nous permettons de définir des transitions temporisées : lorsque la borne temporelle de la transition est atteinte, cette transition doit être franchie et le mode cible atteint.

Cette solution permet de répondre au problème 5 exprimé au chapitre 3.

Présentons l'approche de modélisation que nous avons choisi pour décrire l'architecture logicielle d'un système TR²E critique et adaptatif.

4.2.3 Modéliser l'architecture du logiciel TR²E

Comme nous l'avons présenté au chapitre 3 (voir sous-section 3.4.1), un de nos objectif est d'utiliser à la fois une approche à base de composants logiciels génériques, et un langage de description d'architecture spécifique au domaine des applications TR²E critiques et adaptatifs.

Nous avons pour cela défini un nouveau langage de description d'architecture, que nous avons appelé COAL (Component-Oriented Architecture Language). Ce langage de description d'architecture spécifique aux systèmes TR²E critiques et adaptatifs :

- définit la notion de mode de fonctionnement d'un système, les conditions des changements de mode, et l'impact des changements de mode sur l'architecture logicielle ;
- s'appuie sur un modèle de composants "gros grains" (Lightweight CCM) ;
- définit une sémantique d'exécution analysable (en s'appuyant sur la sémantique d'exécution d'AADL).

`La définition du langage COAL permet de répondre au problème 8 exprimé au chapitre 3.`

Nous avons déjà précisé, au cours de ce chapitre, comment nous représentons les modes et les changements de mode. Nous allons maintenant nous intéresser à la représentation de la reconfiguration pseudo-dynamique. La description du langage COAL et de sa sémantique seront davantage détaillés au cours des chapitre 5 et 6.

Changements de mode et reconfiguration pseudo-dynamique

Définir la configuration cible. Un langage de description d'architecture représente par définition des composants logiciels, des composants matériels, l'allocation du logiciel sur le matériel, ainsi que les connexions entre composants logiciels ou matériels. Par ce biais relativement abstrait – appelé "configuration logicielle" – l'architecte logiciel représente la spécification des fonctionnalités rendues par son application. Ainsi, adapter le comportement du système consiste à modifier dynamiquement cet ensemble de fonctionnalités, et donc leurs caractéristiques.

Dans notre approche, nous avons choisi de décrire la configuration correspondant à un mode donné en précisant quelles caractéristiques de l'architecture restent valides (ou non) dans ce mode. A partir de cette description, nous proposons de générer le code qui va valider ou invalider cette caractéristique en fonction du mode courant du système. Nous proposons également de modéliser formellement le comportement de la reconfiguration ainsi réalisée.

Périmètre de la reconfiguration. Précisons ici le périmètre des variations que nous souhaitons implanter grâce aux mécanismes de reconfiguration :

1. la modification des connexions ;
2. la modification des valeurs des attributs d'un composant ;
3. l'activation et/ou la désactivation de tâches périodiques.

Ainsi, lorsque nous associons une connexion à un ensemble de modes de fonctionnement, cela signifie que la connexion en question n'est valide que si l'un au moins de ces modes de fonctionnement est le mode courant du système. Si ce n'est pas le cas, deux situations peuvent se présenter :

– soit le port “client” (réceptacle ou source d’événements) est connecté dans le mode courant, auquel cas le port “serveur” correspondant (facette ou puits d’évènements) est contacté ;
– soit le port “client” n’est pas connecté dans le mode courant du système, auquel cas une exception particulière est renvoyée à l’appelant.

Lorsqu’une valeur d’attribut est associée à un ensemble de modes de fonctionnement donné, cela signifie qu’à chaque fois que le système entre dans ce mode de fonctionnement, l’attribut est réinitialisé avec la valeur en question.

Enfin, lorsqu’une activité périodique est active dans un ensemble de mode de fonctionnement donné, cela signifie qu’elle appellera son point d’entrée périodiquement tant que le mode courant sera l’un de ces modes, tandis que lorsque le mode courant ne fera pas partie de cet ensemble, alors la tâche continuera d’être ordonnancée, mais elle finira immédiatement son exécution sans entrer dans le code fonctionnel.

Le périmètre de reconfiguration que nous avons présenté ici permet de répondre au problème 3 que nous avions identifié au chapitre 3 : la modification du comportement du système est modélisée en associant une connexion, une valeur d’attribut ou une activité à un mode de fonctionnement donné.

Extensions possibles Nous présentons ici d’autre mécanismes de reconfiguration, ainsi que les raisons pour lesquels nous ne les avons pas sélectionné :

1. Modifier le point d’entrée d’une tâche aurait permis de conditionner le code fonctionnel exécuté par cette tâche à un ensemble de modes de fonctionnement. Nous n’avons pas implanté ce mécanisme parce qu’il peut être réalisé facilement en utilisant un composant intermédiaire dont la connexion dépend du mode courant. L’intégration de la modification du point d’entrée constitue une optimisation de cette solution.
2. Désactiver une tâche sporadique nécessite de déterminer ce qui doit être fait des files de messages en attente de l’activation de ces tâches. Dans ce premier prototype, nous n’avons pas pu traiter ce problème : les tâches sporadiques sont désactivées si leurs ports d’entrée ne sont plus connectées, et elles continuent de traiter l’ensemble des messages que contiennent leurs files d’attente à ce moment là.
3. Désactiver un composant consiste à déconnecter l’ensemble des ports de ce composant. Ce mécanisme de déconnexion est déjà couvert par notre proposition, la désactivation de composant consiste simplement à étendre ce mécanisme à l’ensemble des ports d’un composant ;
4. Suppression/Ajout de composants. Une suppression de composant correspond à une libération d’espace mémoire, alors que l’ajout constitue une allocation. Ce type de mécanisme peut être utile pour améliorer les caractéristiques techniques de l’application (principalement son embarquabilité). Dans le cadre des systèmes critiques, l’utilisation de ce type de mécanismes est interdite.
5. De la même façon que précédemment, le but de la suppression et/ou de l’ajout de tâches est d’optimiser l’utilisation de l’espace mémoire disponible. Encore une fois, l’état de l’art que nous avons présenté stipule que l’utilisation de ce type de mécanisme est interdite dans le cadre de la réalisation des systèmes critiques.

6. Suppression/Ajout de nœuds. Dans le cadre des systèmes TR²E, la suppression de nœuds fait partie des pannes possible d'un système. Nous ne nous intéressons pas ici aux mécanismes de détection, ou de rétablissement en cas de panne matérielle. Nous considérons les changements de modes comme une solution intéressante pour mettre en œuvre une politique de tolérance aux pannes, mais la détection des conditions nécessitant ce changement de mode doit être implanté par d'autres composants logiciels. En ce qui concerne l'ajout de nœuds, cela nécessite de réaliser à la volée de nouvelles connexions avec ces nœuds. Cela nécessite donc de réaliser une nouvelle analyse du comportement du système pour s'assurer que l'ajout de ce nœud et des fonctionnalités qu'il contient ne viole pas les propriétés vérifiées avant son ajout.
7. Migration de composants. Cette problématique rejoint celle que nous avons traitée précédemment : l'ajout et la suppression de composants.

4.2.4 Produire et analyser l'application

Nous présentons ici le processus de conception sur lequel s'appuie notre méthodologie. Ce processus se divise en deux étapes.

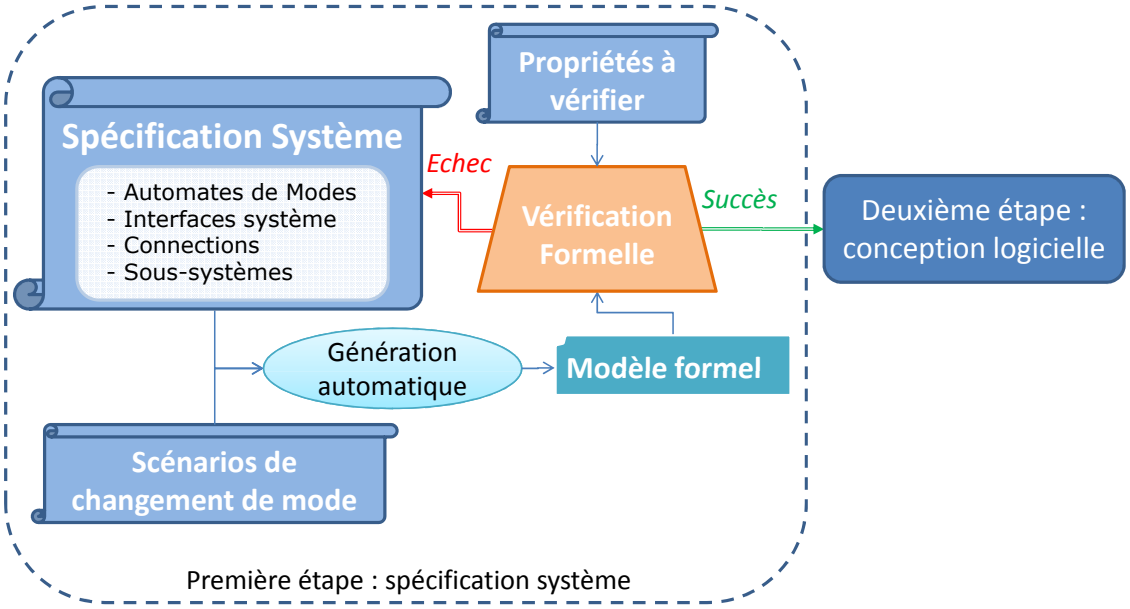

FIGURE 4.2 – Processus de Développement Proposé : spécification système

La figure 4.2 illustre la première étape de ce processus de développement, étape d'analyse de l'architecture système du point de vue des modes de fonctionnement. Pour ce faire, l'architecte système décrit l'architecture du système qu'il doit réaliser : il spécifie les automates de mode de ce système et de ses sous-systèmes, il décrit les connexions entre ces différents automates et systèmes, et définit un scénario qui permet de simuler des requêtes de changement de mode. Enfin, il spécifie les propriétés qu'il souhaite vérifier concernant le comportement des changements de mode. Comme le montre la figure 4.2, la deuxième étape du processus de développement doit être mise en œuvre une fois que ces propriétés ont été validées.

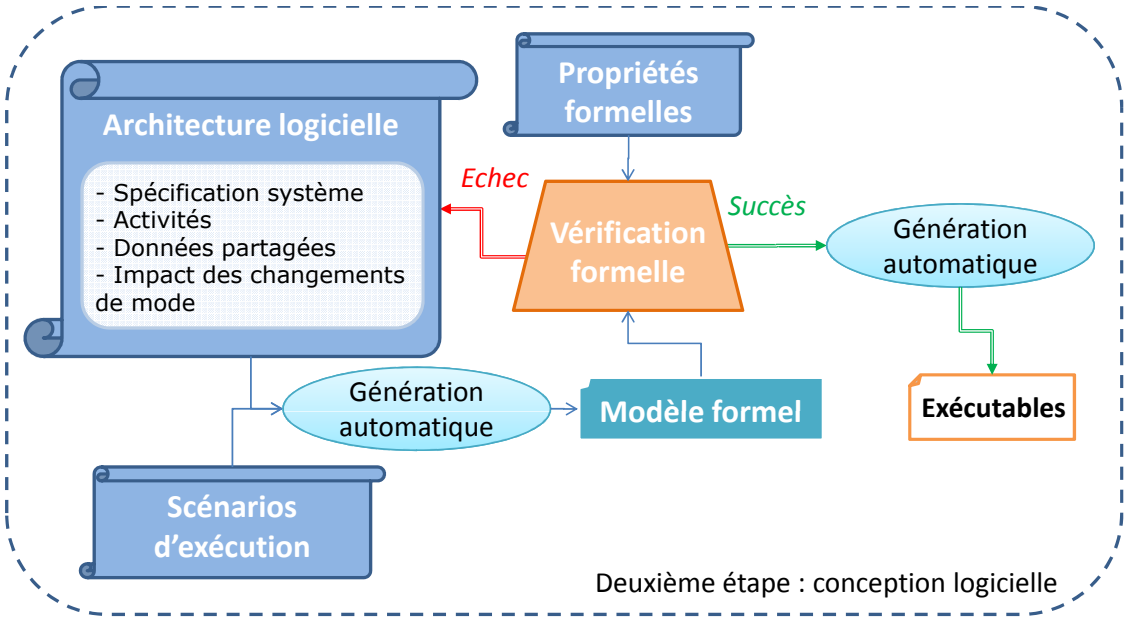

FIGURE 4.3 – Processus de Développement Proposé : conception logicielle

La figure 4.3 illustre la deuxième étape du processus de conception. Cette étape vise à produire l'application TR²E du système après s'être assuré que le comportement associé vérifiait un certain nombre de propriétés. Pour ce faire, l'architecte logiciel doit décrire l'intégralité de l'architecture logicielle du système (y compris les configurations dépendant du mode de fonctionnement courant), fournir un scénario d'exécution de ces différentes tâches, et spécifier les propriétés formelles qu'il souhaite garantir. A partir de ces éléments, un outillage dédié vérifie l'ensemble des propriétés formelles exprimées, et en cas de succès génère le code technique de l'application, et le compile avec le code d'implémentation des composants.

Afin de prouver la faisabilité de cette approche, nous avons réalisé un cas d'étude industriel que nous présentons dans la section suivante.

4.3 Cas d'étude industriel

Le cas d'étude que nous avons réalisé correspond au système de pilotage de l'UGV. Ce système de pilotage peut fonctionner selon deux modes : un mode automatique et un mode manuel. Ce système est composé de deux sous-systèmes. Un sous-système de localisation, qui doit fournir la position courante du système toutes les dix millisecondes, et un sous-système de navigation, dont le comportement varie selon le mode courant du système : dans le mode automatique, le sous-système de navigation calcule les commandes de guidage lorsqu'il reçoit la position courante du système fournie par le sous-système de localisation ; dans le mode manuel, le sous-système de navigation calcule les commandes de guidage à partir des ordres émis par l'utilisateur final du véhicule.

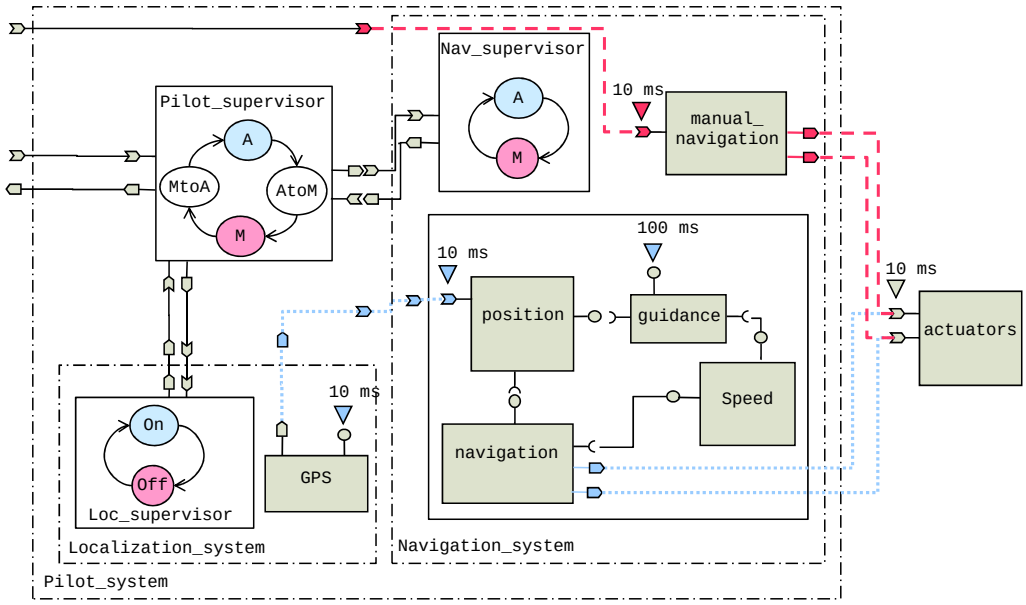

FIGURE 4.4 – Architecture système et logicielle du système de pilotage

4.3.1 Architecture système et logiciel

La figure 4.4 illustre l'architecture globale de cet exemple : elle synthétise la description d'architecture système et logiciel telle qu'exprimée avec COAL. Les limites des différents systèmes sont dessinées avec des contours en trait discontinu. L'automate de mode du système de pilotage est représenté dans le coin supérieur gauche. De fait, il décrit quatre modes : *A* représente le mode automatique, *M* le mode manuel, *MtoA* et *AtoM* étant des modes de transition. Le comportement du système est le suivant. A partir du mode manuel, l'utilisateur peut requérir de passer en mode automatique (M $\rightarrow$ MtoA). Si le sous-système de localisation est hors service, la requête est rejetée et le système revient dans le mode *M*. Autrement, la transition *MtoA* $\rightarrow$ *A* est déclenchée et une commande de mode est envoyée pour placer le sous-système de navigation dans le mode A. Le passage du mode *A* au mode *M* est similaire, en dehors du fait qu'il peut également se produire en cas de panne du sous-système de localisation.

Différentes activités permettent d'activer les chaînes fonctionnelles de ce système. Par exemple, l'activité *Localization_activity*, situé en bas à gauche de la figure (les activités sont représentées par un triangle surplombant une facette ou un puits d'évènements) permet ainsi d'activer le composant GPS toutes les 10 millisecondes.

Sur cette figure, nous avons également schématisé l'impact des modes de fonctionnement sur l'architecture logicielle. Les connexions représentées avec des traits en pointillés sont ainsi valides dans le mode *A* pour le sous-système de navigation, et dans le mode *On* pour le sous-système de navigation. De la même façon, l'activité *guidance_activity* n'est active que lorsque le sous-système de navigation est dans le mode *A*. Enfin, les connexions représentées en tirets ne sont valides que lorsque le sous-système de navigation est dans le mode *M*.

4.3.2 Implémentation des composants

Décrivons ici le comportement des différents composants, en partant de leur point d'activation.

Composant *GPS* Le comportement du composant GPS est assez simple : il envoie la position courante du robot au sous-système de navigation.

Composant *position* Sur réception de la position courante du robot, l'activité *auto_pilot_activity* active le composant *position*, qui stocke la position courante, et appelle l'opération *compute_dir* du composant *navigation*.

Composant *navigation* Lorsque l'opération *compute_dir* est appelée le composant *navigation* appelle le composant *Speed* pour récupérer la dernière valeur de vitesse calculée, puis il calcul les commandes à envoyer aux actionneurs du robot et les envoie.

Composant *guidance* Lorsque l'activité *guidance_activity* active le composant *guidance*, celui-ci récupère les dix dernières positions stoquées dans le composant *position* et calcule la vitesse courante du robot. Il stocke ensuite cette valeur dans le composant *Speed*.

Ce comportement constitue le code fonctionnel de l'application, qui est directement implanté dans le code des composants. Seule une représentation abstraite du comportement de ces composants permettrait d'exhiber ce comportement à des fins d'analyse par exemple.

4.3.3 Mise en œuvre de notre méthodologie

Cet exemple va nous permettre d'illustrer la méthodologie que nous avons présenté au cours de ce chapitre. Nous allons ainsi montrer, grâce à cet exemple, que nous pouvons répondre à l'ensemble des problèmes que nous avons présenté au chapitre précédent (chapitre 3). Nous montrerons ainsi, dans la suite de ce mémoire, que notre méthodologie automatise :

- l'analyse du comportement des changements de mode ;
- l'analyse du comportement logiciel en cours de changement de mode ;
- la production des applications de ce système, y compris leurs mécanismes d'adaptation.

4.4 Protocoles de reconfiguration pseudo-dynamique

4.4.1 Critères de sélection d'un protocole de changement de mode

Une des problématiques majeures des protocoles de changement de mode consiste à garantir un état cohérent de l'ensemble du système temps réel. Pour cela, il suffit d'atteindre un "état stable" de l'application, c'est à dire l'état de l'application qui permet d'effectuer la reconfiguration de telle sorte à garantir que la reconfiguration peut avoir lieu sans altérer le bon fonctionnement de l'application.

Voici la définition de l'état stable que nous utiliserons dans la suite de ce mémoire : une application est dans l'état stable lorsqu'aucune des tâches dont l'exécution dépend de la valeur du mode courant n'est en cours d'exécution.

Nous avons présenté au chapitre 3 (voir sous-section 3.3.3) différents critères de mise en oeuvre de la reconfiguration pseudo-statique. Dans le cadre des systèmes temps réels, ces critères concernent :

- l'analyse de l'ordonnancement du système ;
- la rapidité de la reconfiguration ;
- la disponibilité d'ensemble de données cohérentes entre elles.

A travers la définition de ces critères, nous avons répondu au problème 6

4.4.2 Proposition d'un mécanisme de synchronisation

Afin d'implanter les mécanismes de changement de mode, nous proposons de représenter le mode de l'application via une donnée dont la valeur constitue le mode courant de l'application. Changer de mode consiste alors à mettre à jour cette variable et à réaliser certaines actions de reconfiguration comme par exemple la mise à jour d'attributs de composants, ou encore l'envoi d'un message de synchronisation (commande de changement de mode, envoi d'information de statut sur le mode courant, etc...).

Dès lors, les tâches de l'application responsables de réaliser un changement de mode (que nous qualifierons de **tâches de reconfiguration**), vont écrire sur cette donnée lors d'un changement de mode. Inversement, le flux d'exécution de certaines tâches va dépendre du mode courant de l'application (pour savoir quelle connexion est valide par exemple). Ces tâches, que nous qualifierons de **tâches impactées par la reconfiguration** vont par conséquent lire la valeur du mode courant.

La valeur du mode courant est donc implanté grâce à une donnée partagée, écrite par les tâches responsables d'effectuer la reconfiguration, et lue par les tâches impactées par la reconfiguration.

Nous proposons donc d'utiliser un verrou de type "lecteur/écrivain" pour protéger l'accès en lecture et en écriture à cette variable de mode. De plus, afin de respecter une contrainte de cohérence minimale, qui vise à garantir qu'une tâche ayant commencé son exécution dans un mode de fonctionnement donné poursuit et finit son exécution dans ce mode, nous ajoutons que l'acquisition du verrou en lecture par une tâche impactée par la reconfiguration se fait au début d'exécution de cette tâche, et qu'elle ne libère ce verrou qu'en fin d'exécution.

Le patron d'exécution d'une tâche impactée par un changement de mode est donc le suivant :

1. déclenchement de la tâche
2. acquisition du verrou en lecture
3. exécution du code fonctionnel
4. libération du verrou en lecture

En revanche, l'acquisition en écriture par une tâche de reconfiguration se fait lorsque celle-ci écrit la valeur du mode courant. Ainsi, si un ensemble de tâches impactées par la reconfiguration sont en cours d'exécution lorsque la tâche de reconfiguration essaie d'acquérir le verrou en écriture, celle-ci est bloquée tant que l'ensemble des tâches impactées n'ont pas fini leur exécution dans le mode courant. Cet algorithme permet donc de garantir le niveau de cohérence locale minimal. De plus, cet algorithme correspond à un protocole de reconfiguration synchrone, ce qui facilite l'analyse d'ordonnançabilité de l'application [Real and Crespo, 2004].

Présentons maintenant les différents protocoles que nous avons conçus afin de répondre aux enjeux de la reconfiguration dynamique (réduction des perturbations, réduction du temps de reconfiguration, et garantie de la cohérence des données). Pour illustrer le comportement d'un système configuré selon chacun de ces protocoles, nous utiliserons un sous-ensemble de notre cas d'étude composé des tâches *guidance_activity*, *localization_activity*, *auto_pilot_activity*, des composants *GPS*, *position*, *guidance*, *Speed*, *navigation* et de l'automate de mode *Nav_supervisor*. Nous appellerons T_R la tâche de reconfiguration, c'est à dire celle qui effectue le changement de mode de l'automate *Nav_supervisor*. Le changement de mode considéré correspond à la transition $A \rightarrow M$.

Profitons de cet exemple pour illustrer la notion de tâche impactée par la reconfiguration dynamique. *guidance_activity* et *autopilot_activity* sont impactées par un changement de mode $A \rightarrow M$ où $M \rightarrow A$ de l'automate de mode *Nav_supervisor* dans la mesure ou le flux d'exécution de ces deux tâches dépend de la valeur du mode de cet automate. En revanche, l'activité *localization_activity* n'est pas impactée par un changement de mode de cet automate : quelque soit le mode courant de l'automate *Nav_supervisor*, l'exécution de *localization_activity* reste identique. Enfin, il est évident que l'activité *localization_activity* est impactée par un changement de mode de l'automate de mode *Nav_supervisor*, mais cet automate ne fait pas partie du périmètre simplificateur que nous mettons en exergue dans ce. Nous appellerons par la suite **assemblage de référence** l'assemblage constitué par les éléments du système de pilotage que nous avons sélectionné ici.

4.4.3 Privilégier le déroulement de l'application

Privilégier le déroulement de l'application sur la reconfiguration revient à spécifier que la tâche qui configure la reconfiguration s'exécute avec une priorité plus faible que la priorité d'autres tâches de l'application. Cela permet de réduire la perturbation que subit le système du fait de la reconfiguration, puisque dans ce cas la tâche de reconfiguration ne préempte pas les tâches de l'application, plus prioritaires. Ce protocole de reconfiguration présente aussi l'avantage de faciliter l'analyse d'ordonnançabilité du système [Real and Crespo, 2004] en réduisant le temps de blocage des tâches du fait de la reconfiguration.

La figure 4.5 illustre le comportement de notre assemblage de référence lors d'un changement de mode configuré avec ce protocole (protocole que nous désignerons par le sigle RWLock dans la suite de ce mémoire). Ici, T_R correspond à la tâche sporadique qui configure le port de l'automate *Nav_supervisor* sur lequel ce dernier reçoit l'ordre de changement de mode correspondant : T_R exécute donc le changement de mode $A \rightarrow M$ sur réception de la requête de changement de mode.

Sur cette figure, nous pouvons constater que le déroulement de l'application est bien privilégié par rapport à la reconfiguration : l'exécution de T_R est retardée jusqu'à ce que l'ensemble des tâches impactées aient libéré la resource d'exécution.

Dans la sous-section suivante nous allons nous intéresser au protocole réduisant ce pire temps de reconfiguration.

4.4.4 Privilégier la mise en œuvre de la reconfiguration

Privilégier la reconfiguration peut être nécessaire lorsque le temps disponible pour mettre en œuvre les mécanismes d'adaptation est particulièrement court. Dans ce cas, il est intéressant de disposer d'un protocole accélérant la reconfiguration. Nous proposons pour cela, non seulement d'attribuer à la tâche de reconfiguration une priorité supérieure à la priorité

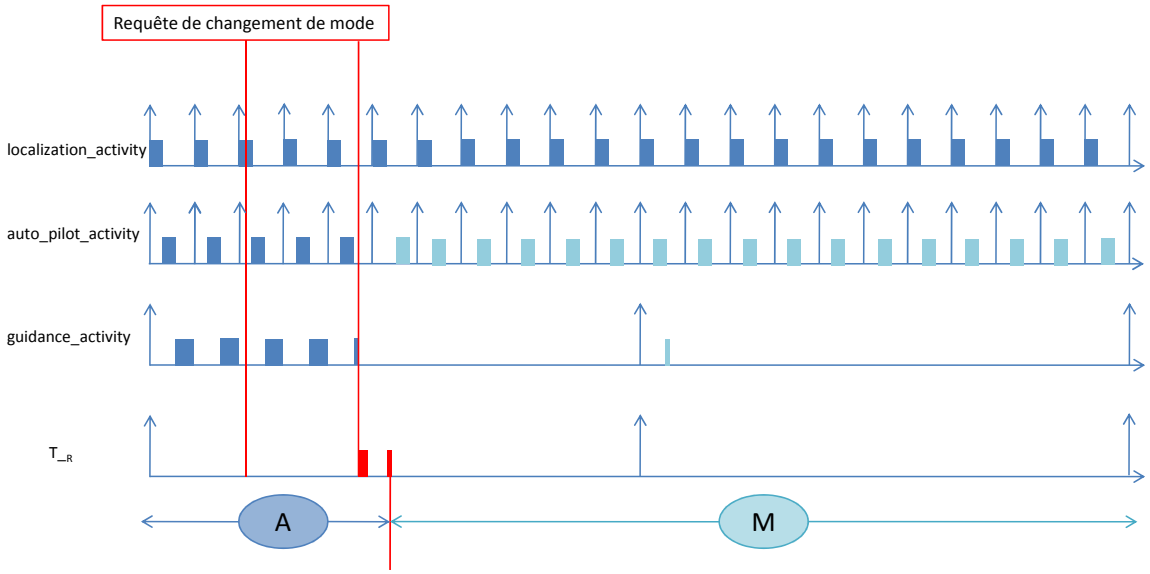

FIGURE 4.5 – Protocole de Reconfiguration Privilégiant l'Application

des tâches impactées par le changement de mode, mais également d'utiliser un mécanisme d'inversion de priorité du type PCP [Sha *et al.*, 1990] (*priority ceiling protocole*). La sémantique d'exécution de ce mécanisme est la suivante : toutes les tâches qui ont acquis un verrou en lecture héritent de la priorité plafond spécifiée par l'architecte logicielle lorsque la tâche de reconfiguration est bloquée sur l'acquisition du verrou en écriture. Une contrainte s'impose alors sur la valeur de cette priorité plafond : elle doit être supérieure à la priorité maximum des tâches impactées par la reconfiguration. Ce protocole garantit que seule l'itération en cours d'exécution des tâches impactées se terminera et qu'aucune autre ne recommencera avant que la reconfiguration n'ait lieu.

Ceci réduit donc le pire temps (voir sous-section 7.3.1) d'un changement de mode. Ceci est également illustré sur la figure 4.6, qui représente le comportement de notre assemblage de référence configuré avec cette politique de reconfiguration. Le principale inconvénient de ce protocole est qu'il rend plus difficile l'analyse d'ordonnançabilité du système [Real and Crespo, 2004] : les temps de blocage sont tellement importants que l'analyse ne peut réussir que si les tâches ont des temps d'exécution très court.

La figure 4.6 illustre le comportement de notre assemblage de composant lors d'un changement de mode configuré avec la politique qui vise à réduire le temps de reconfiguration (protocole que nous désignerons par le sigle PCP dans la suite de ce mémoire). Ici encore, Ici, T_R correspond à la tâche sporadique qui configure le port de l'automate *Nav_supervisor* sur lequel ce dernier reçoit l'ordre de changement de mode correspondant : T_R exécute donc le changement de mode $A \rightarrow M$ sur réception de la requête de changement de mode.

Nous pouvons constater sur cette figure que la reconfiguration est privilégiée par rapport à l'application : la reconfiguration a lieu dès que l'exécution courante des tâches impactées est terminée ; aucune exécution de tâche impactée n'est relancée une fois que la requête de reconfiguration a été émise.

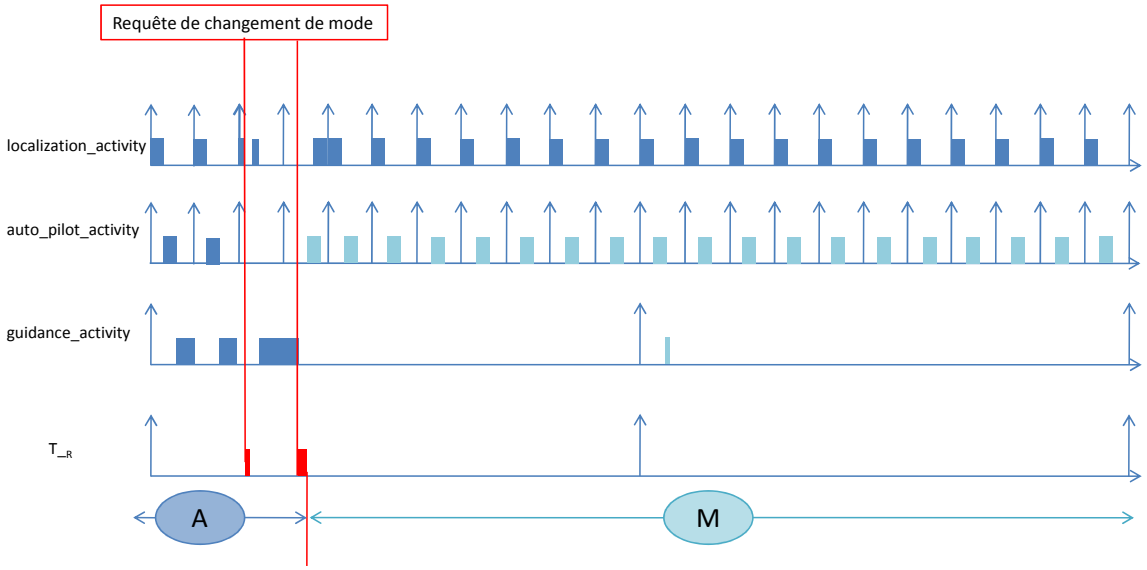

FIGURE 4.6 – Protocole de Reconfiguration Privilégiant la Reconfiguration

4.4.5 Garantir la disponibilité de données cohérentes

Les protocoles que nous venons de présenter sont à mettre en relation avec un autre besoin fort dans les systèmes reconfigurables : la disponibilité de données cohérentes entre elles. Pour répondre à ce besoin, que nous avons décrit au chapitre 3 (voir sous-section 3.3.3), nous proposons un troisième protocole de reconfiguration, qui doit être combiné avec les deux protocoles présentés précédemment (ce qui donne en réalité quatre protocoles). Ce protocole consiste à identifier les sous-ensembles de tâches périodiques qui doivent être synchronisées lors d'un changement de mode. Dans ce cas, lorsqu'une requête de changement de mode est reçue par le système, celui-ci enregistre le contenu de cette requête. Ce n'est que lorsque l'hyper-période de l'ensemble des tâches impactées synchronisées sera atteinte que le mécanisme de reconfiguration sera réellement déclenché. Dès lors, les tâches synchronisées ne doivent pas être déclenchées à nouveau (pour laisser la reconfiguration se dérouler). De plus, les tâches impactées et non-synchronisées doivent être synchronisées avec la tâche de reconfiguration par l'un des deux protocoles présentés précédemment. Pour implanter ce mécanisme de synchronisation à l'hyper-période, nous proposons de définir une tâche périodique dont la période est égale à l'hyper-période de l'ensemble des tâches périodiques synchronisées impactées par le changement de mode. Lorsque le changement de mode utilise également le mécanisme d'inversion de priorité type PCP, alors la priorité de cette tâche est égale à la priorité plafond correspondante. Ceci afin d'accélérer le processus de reconfiguration. Lorsque le changement de mode n'utilise pas ce mécanisme d'héritage de priorité, alors la priorité de la tâche de reconfiguration est égale à la plus haute priorité des tâches synchronisées impactées (plus un). Cette valeur correspond à la plus petite valeur de priorité garantissant qu'à l'hyper-période, la reconfiguration a lieu sans que les tâches impactées synchronisées ne recommencent leur exécution.

La figure 4.7 illustre le comportement de notre assemblage de référence s'il est configuré par cette dernière politique (politique que nous désignerons par le sigle TGRP dans la suite de ce mémoire). Ici, T_R correspond à la tâche périodique de reconfiguration que nous avons défini au paragraphe précédent.

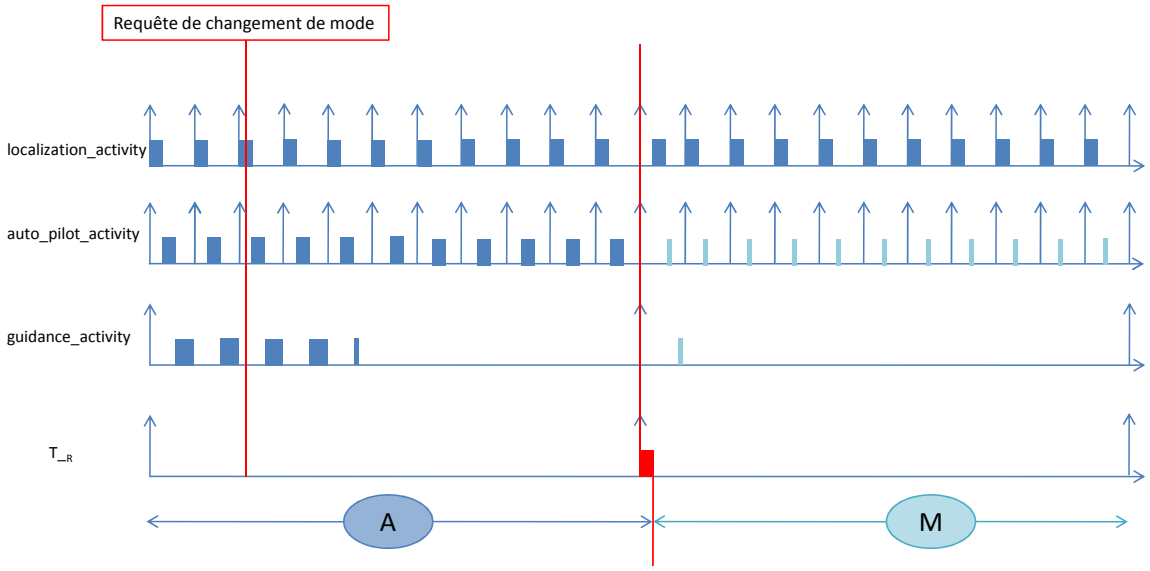

FIGURE 4.7 – Protocole de Reconfiguration Privilégiant la Cohérence des Données

Sur cette figure, nous pouvons constater que la reconfiguration a lieu à l'hyper-période des tâches *localization_activity* et *guidance_activity*. En extrapolant les résultats présentés dans [Real and Wellings, 1999] à cette politique de reconfiguration, nous déduisons que cette politique introduit un surcoût en termes de temps de reconfiguartion (dû à l'attente de l'hyper-période), mais ne rend pas plus difficile l'analyse d'ordonnançabilité.

L'ensemble des protocoles de changement de mode que nous avons présenté dans cette section permet de répondre au problème 7, présenté au chapitre 3.

Le tableau 4.1 permet de résumer les avantages et inconvénients, par rapport aux critères sélectionnés, de chacune des quatre politiques présentées dans ce chapitre.

Protocole de reconfiguration / *Critère*	P1	P2	P3	P4
ordonançabilité	+++	-	+++	+
rapidité	-	+++	-	+
cohérence des données	++	++	+++	+++

P1 : protocole privilégiant l'application

P2 : protocole privilégiant la reconfiguration

P3 : protocole garantissant la cohérence des données (sans PCP)

P4 : protocole garantissant la cohérence des données (avec PCP)

TABLE 4.1 – Avantages et inconvénients des différents protocoles de changement de mode

4.5 Synthèse

Dans ce chapitre, nous avons présenté une approche qui répond aux différents problèmes que nous avons décrit au chapitre 3. Cette approche comprend une méthode de conception,

ainsi qu'un ensemble de protocoles de changement de mode qui répondent aux critères de réalisation des systèmes TR2E critiques et adaptatifs. Dans ce chapitre, nous avons également présenté le cas d'étude industriel que nous utiliserons pour démontrer la faisabilité de notre approche.

Cette approche s'appuie sur plusieurs étapes, qui nécessitent :

1. de modéliser l'architecture système et logiciel d'un système TR²E critique et adaptatif ;
2. de définir la sémantique d'exécution associée à cette modélisation et de générer le code correspondant ;
3. de s'assurer que cette sémantique d'exécution est analysable.

C'est en suivant cette démarche (modélisation $\rightarrow$ définition de la sémantique d'exécution $\rightarrow$ analysabilité) que nous avons réussi à résoudre l'ensemble des problèmes que nous avons présenté jusque là. Ainsi, chacun de ces points fera l'objet d'un chapitre particulier de la partie IV, consacrée à la description détaillée des contributions scientifiques de nos travaux.

Quatrième partie

Détail des contributions

Chapitre 5

COAL, un langage de description d'architectures à base de composants

L'idée sans le mot serait une abstraction.
Victor Hugo

SOMMAIRE

5.1 Introduction

Dans ce chapitre, nous présentons en détail le langage de description d'architectures à base de composants que nous avons spécifié dans le but de réconcilier les approches à composants logiciels génériques et les langages de description d'architecture. Ce langage, appelé *Component Oriented Architecture Language* (COAL), reprend et étend les formalismes déjà existants dans les standards *Deployment and Configuration* (D&C) [(OMG), 2005a] de l'*Object Management Group* (OMG) et le langage *Architecture Analysis and Design language* (AADL) [SAE, 2004]. Revenons rapidement sur les raisons de ces différents choix. Comme nous l'avons expliqué au paragraphe 3.4.1, une de nos principales contraintes dans le but de rendre notre démarche compatible des exigences de mise en œuvre industrielle consiste

à utiliser un modèle à base de composants, s'appuyant plus précisément le standard Lightweight CCM [(OMG), 2003]. Cependant, si ce standard décrit précisément l'ensemble des interfaces de configuration et de déploiement des composants logiciels, il n'offre pas de moyen représenter ce déploiement. Tout au plus suggère-t-il l'utilisation du standard D&C dans le but de spécifier le déploiement d'architectures à base de composants Lightweight CCM. Si certains concepts présents dans D&C nous ont semblé pertinents dans le but de décrire le déploiement d'applications à base de composants, d'autres ne nous ont pas semblé adaptés à la spécification d'applications TR²E **critiques** : D&C fait partie des standards dans lesquels la sémantique définie pour les éléments qui le composent est faible, ou abstraite : tout est composant et la sémantique de ces composants est définie par leur implémentation. Libre alors à l'utilisateur de ce standard de définir sa propre sémantique pour associer à un type de composant donné des caractéristiques prédéfinies propres aux applications temps-réel ou aux applications reconfigurables. Ainsi, D&C ne définit pas de concepts propres aux applications temps-réel ou aux applications reconfigurables. De ce fait, une approche basée sur D&C n'est pas appropriée pour relever les défis auxquels est confronté le génie logiciel des systèmes TR²E **critiques et adaptatifs**. En effet, relever ces défis nécessite selon nous de définir une sémantique précise concernant les concepts spécifiques au domaine des systèmes temps-réel embarqués. AADL est un standard qui tend à répondre à cet objectif. Nous nous sommes donc inspiré de certains de ses concepts, que nous avons également étendus. Dans sa version la plus récente (AADL 2.0), AADL semble être en mesure d'exprimer le même niveau d'information que le langage COAL.

Au moment où nous avons commencé nos travaux, plusieurs raisons nous ont poussé à ne pas utiliser directement AADL afin de répondre à la problématique présentée précédemment. En effet, ce standard, dans sa première version (AADL 1.0), ne permet pas de décrire des composants logiciels génériques tels qu'il peuvent être spécifiés avec Lightweight CCM. Si ces aspects ont été améliorés dans la deuxième version du standard, les outils associés n'ont pas encore évolué vers la prise en compte de ces nouveautés. De plus, la notion de mode de fonctionnement telle qu'elle est décrite dans le standard AADL 1.0 ne permet pas (i) d'associer à un automate de mode une ou plusieurs tâches, (ii) d'identifier des sous-ensembles de tâches qui doivent être synchronisés entre eux lors d'un changement de mode, ou encore (iii) de référencer un mode de niveau système pour y associer une configuration logicielle.

Puisque AADL 2.0 n'existait pas encore au moment où nous avons commencé nos travaux, nous avons choisi de créer un nouveau langage de description d'architecture qui s'appuie sur les concepts présents dans ces deux spécifications. De D&C, nous reprendrons la notion d'implémentation et d'instance de composant logiciels gros grains (ou génériques). Nous reprendrons les notions de système (*system*), de tâche (*thread*), de processus (*process*), de données partagées (*data*) et de mode de fonctionnement (*operational mode*), définies dans le standard AADL 1.0 et qui n'existent pas nativement dans D&C. Enfin, nous réutiliserons des notions communes à ces deux standards, tels que la notion de nœud de calcul, de lien de communication, ou de connexion entre composants.

Bien entendu, COAL ne se contente pas de rassembler *stricto sensu* ces différents concepts, mais permet d'adapter les notions définis dans AADL aux spécificités des approches à base de composants ; de même qu'il permet d'adapter les notions de D&C aux spécificités des systèmes TR²E critiques. Plus généralement, ce langage vise alors à fournir un point d'entrée pour automatiser les différentes transitions du processus de développement que nous avons présenté au chapitre 4, figure 4.1.

Dans la suite de ce chapitre, nous détaillons à travers de nombreux exemples les différentes constructions que fournit COAL pour modéliser une architecture TR²E reconfigurable

et à base de composants. La grammaire de COAL est par ailleurs fourni en annexe de ce mémoire (voir annexe A).

5.2 Spécification système

Dans cette section, nous montrons les différentes constructions définies dans COAL dans le but de représenter les interfaces externes, ainsi que la composition interne d'un système. Ceci correspond au premier ensemble de cinq processus présents dans la partie supérieure à gauche dans la figure 4.1.

5.2.1 Définition des interfaces systèmes

Comme nous l'avons expliqué dans le chapitre précédent, le système de pilotage a pour but d'émettre des commandes aux actionneurs, qu'il calcule à partir des ordres de contrôle qu'il reçoit : ce n'est pas un système autosuffisant. En d'autres termes, il a des interfaces avec d'autres systèmes, principalement des capteurs et des actionneurs. Dans COAL, un système interagit avec son environnement au travers de ports d'évènements qui lui permettent d'émettre (respectivement de recevoir) des données en direction de (resp. à partir de) son environnement. L'exemple 5.1 représente les interfaces de notre système de pilotage. Dans cet exemple, *direction*, *Pilot_modes*, *Steering_cmd* et *Speed_cmd* (lignes 5 à 9) sont des types d'évènements qui ont été définis dans des fichiers IDL (*Interface Description Language*) importés par la spécification COAL à l'aide de la spécification *import comp_ima_Pilot ;* (voir exemple 5.1, ligne 2).

Les ports *dir_cmd*, *mode_cmd* et *mode_sts* permettent d'échanger des données avec l'utilisateur final (commande de direction, requêtes de changement de mode et état du commutateur de mode). Enfin, les ports *Steering_cmd* et *Speed_cmd* permettent d'envoyer des commandes aux actionneurs de façon à guider le véhicule.

```
1
2  import comp_ima_Pilot;
3
4  system Pilot {
5    consumes direction      dir_cmd;       // dir_cmd port
6    consumes Pilot_modes    mode_cmd;      // mode_cmd port
7    publishes Pilot_modes   mode_sts;      // mode_sts port
8    publishes Steering_cmd  Steering_cmd;  // Steering_cmd port
9    publishes Speed_cmd     Speed_cmd;     // Speed_cmd port
10 };
```

Exemple 5.1 – Interfaces du système Pilot

5.2.2 Définition des interfaces

L'import de *comp_ima_Pilot* permet de référencer les types de données définis dans "direction.idl3". Le contenu de ce fichier est retranscrit dans l'exemple 5.2 :

```
1
2  struct DirXY
3  {
4    float speed;
5    float orientation;
6  };
7
8  eventtype direction
9  {
10   public DirXY dir;
11 };
```

Exemple 5.2 – Définition du type d'événement direction

Les différents extraits de spécification que nous avons choisis de montrer ici illustrent la façon dont on définit les interfaces d'un système à l'aide de COAL, tout en référençant des éléments décrits en suivant le standard Lightweight CCM. Outre ses interfaces de communication, un système est également constitué de différents éléments qui réalisent les fonctionnalités pour lesquels il a été spécifié.

5.2.3 Définition de la composition du système

Un système peut contenir des sous-systèmes, un modèle de la plate-forme cible (processeurs, bus, processus), des composants logiciels, des activités et un automate de mode. Ce qui différencie fondamentalement un système d'un composant logiciel est qu'un système peut contenir des éléments de description d'architectures matérielles, et qu'un composant logiciel et ses sous-composants ne peuvent être déployés que sur un seul espace d'adressage, alors que les éléments constitutifs d'un système peuvent être déployés sur plusieurs espaces d'adressage.

Avec COAL, la description de la composition du système consiste à spécifier sa *réalisation* (voir exemple 5.3), par opposition à l'étape précédente qui ne consistait qu'à représenter les interfaces d'un système.

Sous-systèmes

Un système donné peut être composé d'un ou plusieurs sous-systèmes. L'exemple 5.3 permet de représenter les différents sous-systèmes contenus dans un système donné. La réalisation *Pilot_real* du système *Pilot* est composée de deux sous-systèmes : un sous-système de localisation (*Localization_real*) et un sous-système de navigation (*Navigation_real*).

```
1 system Pilot_real realizes Pilot {
2   subsystems {
3     system Localization_real;
4     system Navigation_real;
5   };
6 };
```

Exemple 5.3 – Sous-systèmes du système Pilot

Equipements du système

```
1 system Pilot_real realizes Pilot {
2    ...
3   bus IP_1;
4   target board_1 {
5     uses bus IP_1 at address 10.222.145.42;
6   };
7 };
```

Exemple 5.4 – Déclaration des équipements

COAL s'inspire fortement de AADL en ce qui concerne la description des équipements d'un système, tels que des cartes électroniques et des réseaux de communication : ils peuvent être représentés grâce à la notion de cible (*target*) et de bus. Une cible représente une mémoire, ou un espace d'adressage, sur laquelle un système d'exploitation peut avoir été instancié. Un bus représente un lien de communication déployé entre différentes cibles (voir l'exemple 5.4).

Modes du système

La définition des modes opérationnels fait le plus souvent partie du processus de spécification système, cependant que la modélisation de l'impact d'un changement de mode sur l'architecture logicielle fait partie de la conception logicielle. Par conséquent, les composants systèmes et logiciels peuvent contenir des composants qui mettent en œuvre des automates de mode. Des communications entre ces automates de mode permettent alors de modéliser et de synchroniser les différentes étapes de ces changements de mode dans les différents systèmes et sous-systèmes. Les communications correspondantes sont spécifiées en utilisant les ports d'événements et les connexions des automates de mode. Les types de données échangés entre les automates de mode sont limités aux données énumérées représentant les requêtes de changement de mode. Ces requêtes peuvent provenir d'autres automates de mode ou de composants fonctionnels de l'application. L'exemple 5.5 présente les modes et les changements de mode du système de pilotage. Dans cette spécification, le mode automatique est déclaré comme mode initial du système. Dans la définition *Pilot_supervisor_impl* (lignes 6 à 13), nous définissons les règles de transition entre les modes définis dans *pilot_supervisor* ; cette définition exprime : "dans le mode *automatic*, la réception de la commande de mode *manual* entraîne le passage au mode *automatic_to_manual*. "

```
system Pilot_real realizes Pilot {
  mode automaton Pilot_supervisor {
    mode { automatic (initial), automatic_to_manual, ...};
    consumes Pilot_modes Pilot_mode_cmd; ...
  };
  mode automaton implementation Pilot_supervisor_impl implements Pilot_supervisor {
    in mode automatic: {
      [Pilot_mode_cmd?(request=manual)] --> automatic_to_manual {
                          Localization_mode_cmd!(request:=start);
                          Navigation_mode_cmd!(request:=automatic);
      };
    }; ...
  };
};
```

Exemple 5.5 – Changements de mode du système pilot

Ce qui précède a permis de présenter les activités de modélisation correspondant à l'ingénierie de systèmes. Le chapitre suivant décrit comment effectuer les tâches de génie logiciel en utilisant le langage COAL.

5.3 Conception logicielle

Une fois un système décomposé en sous-systèmes et les sous-systèmes décomposés en sous-systèmes, les contenus des sous-systèmes feuilles constituent l'architecture logicielle du système. Cette architecture comprend :

- des processus qui représentent chacun un espace d'adresses logiques sur lesquels la ou les applications sont instanciées ;
- des composants logiciels qui représentent des sous-ensembles de fonctionnalités qui doivent être assemblées ;
- des activités qui représentent des fils d'exécution (*threads*) indépendants ;
- des données qui représentent les données internes aux composants et qui peuvent être lues et/ou écrites dans le code de ces composants.

Ces différents éléments permettent de mettre en œuvre les cinq processus de conception logicielle tels que définis au centre et à gauche de la figure 4.1.

5.3.1 Processus

Un processus représente un espace d'adresses logiques sur lesquels peut être instanciée une application logicielle.

```
1 system Navigation_real realizes Navigation_subsystem
2 {
3     process Pilot_proc {
4       target : board_1;
5       ...
6     };
7     ...
8 };
```

Exemple 5.6 – Processus du sous-système de navigation

5.3.2 Composants

Dans la première étape de notre processus de conception logicielle (voir figure 4.1), l'architecte logiciel définit les chaînes fonctionnelles de l'application TR²E. Dans un premier temps, cela correspond aux composants, à leurs interfaces et ports qui réaliseront les différentes fonctionnalités de ces chaînes fonctionnelles.

Un composant est une sous partie de l'ensemble des fonctionnalités d'un système. Les fonctions sont ainsi décomposées en parties plus petites, plus faciles à développer et à réutiliser. La définition d'un composant doit permettre de décrire la façon dont ce dernier interagit avec les autres composants au travers de ports. Cette définition spécifie alors non seulement les services que le composant fournit (comme le fait un objet), mais aussi les services qu'il requiert et qui devront lui être fournis par d'autres composants. Ceci permet aux composants d'être connectés de façon externe. De plus, les composants décrivent leurs paramètres de configuration de façon à pouvoir être configurés de façon externe. Notre contribution s'appuie sur le standard Lightweight CCM de l'OMG [(OMG), 2003].

Interfaces de composant

Les interfaces de composants sont spécifiées en IDL, langage dans lequel les interfaces et les types d'évènements peuvent être définis de façon à représenter une sémantique de communication synchrone ou asynchrone.

L'exemple 5.7 représente la spécification de l'interface *getPos* (ligne 6) et du type d'évènement *setPos* (ligne 9).

```
1  module position {
2    struct PosXY {
3      float x;
4      float y;
5    };
6    interface getPos {
7      PosXY getPosition( );
8    };
9    eventtype setPos {
10     public PosXY position;
11   };
12 };
```

Exemple 5.7 – Définition des interfaces du module position

Types de composants

Les types de composants sont définis selon le standard Lightweight CCM de l'OMG [(OMG), 2003]. L'exemple 5.8 présente la définition du composant *positionShM* qui fournit des accès aux données représentant la vitesse courante du système.

```
1  module position {
2    component PositionShM {
3      provides getPos      get_pos;   // get_pos facet
4      consumes setPos      set_pos;   // set_pos event sink
5    };
6  };
```

Exemple 5.8 – Définition des composants du module position

Implémentation des composants

Une fois les types de composants définis, l'architecte logiciel peut affiner la définition des composants grâce aux implémentations de composants. Pour y parvenir, il peut soit définir son composant comme composant composite (c'est-à-dire un composant qui est lui-même composé de sous composants), soit comme composant primitif (c'est-à-dire un composant qui correspond à morceau de code). L'exemple 5.9 correspond à la définition d'un composant composite, alors que l'exemple 5.10 illustre la définition d'un composant primitif.

```
1  system Navigation_real realizes Navigation_subsystem {
2     ...
3    subcomponents {
4      component implementation auto_navigation_impl
5                                  implements Navigation {
6        component instance posShM_inst
7                                  instantiates position_PositionShM_impl{
8                                  };
9          ...
10     };
11   };
12    ...
13 };
```

Exemple 5.9 – Implémentation du composant de navigation automatique

L'exemple 5.10 illustre la spécification d'une implémentation de composant primitif. Dans notre méthodologie actuelle, la lecture ou l'écriture d'une donnée partagée peut être décrite dans ce type de spécification. Dans une version ultérieure, d'autres éléments descriptifs du comportement du composant pourraient venir compléter ce type de spécification.

```
1  component implementation manual_navigation_impl
2                                  implements Navigation {};
```

Exemple 5.10 – Implémentation du composant de navigation manuelle

Instances de composants

Enfin, les instances de composants représentent composants déployées sur la plate-forme cible. Comme illustré dans l'exemple 5.11, ces instances sont donc associées à un processus donné.

```
1 system Navigation_real realizes Navigation_subsystem {
2     ...
3   auto_navigation_inst.process : Pilot_proc;
4 };
```

Exemple 5.11 – Déploiement des instances de composant

5.3.3 Définition des activités

La définition des chaînes fonctionnelles s'appuie sur la notion (i) d'activité périodique, dont la période se déduit le plus souvent de l'analyse des lois physiques qui dirigent le système, (ii) d'assemblages de composants, c'est-à-dire d'ensembles de composants et sous-composants connectés entre eux et qui fournissent la fonctionnalité requise, et (iii) d'activité sporadique, qui permettent de mettre en œuvre les communications asynchrones de l'application.

Le concept d'activité est très similaire à celui de *thread* en AADL. La principale différence vient du fait que l'on se place dans un contexte de composants logiciels dont le comportement est directement implanté par le développeur de composant. Ainsi, il n'est pas possible d'exprimer le contenu de l'activité sous la forme d'une séquence d'appel d'opérations (comme on peut le faire en AADL) puisque le fait qu'un composant en appelle un autre composant est directement implanté dans le code de cette implémentation. Ainsi, on n'exprime que le point d'entré d'une activité, c'est à dire l'opération que cette activité va appeler sporadiquement ou périodiquement.

L'exemple 5.12 illustre la définition des activités *guidance_activity* et *auto_pilot_activity*, qui correspondent respectivement à l'activation de la chaîne fonctionnelle de calcul de la vitesse du système, et à la prise en charge de communications du sous-système de localisation vers le port *set_pos* de l'instance de composant *auto_nav_inst* afin de propager l'activité de calcul de la trajectoire du robot (voir paragraphe 4.3 et figure 4.4).

```
1  periodic activity guidance_activity
2  {
3          configures guidance_inst.activation operation activate;
4          period : 100 ms;
5          priority : 10;
6          stack size : 20 KByte;
7  };
8
9  sporadic activity auto_pilot_activity
10 {
11         configures auto_nav_inst.set_pos;
12         period : 10 ms;
13         priority : 20;
14         stack size : 20 KByte;
15 };
```

Exemple 5.12 – Définition des activités chaînes fonctionnelles

L'exemple 5.13 illustre la connexion qui permet à la chaîne fonctionnelle de se propager depuis le composant englobant (port *NMEA_current_pos*) vers vers le port *set_pos* de l'instance de composant *auto_nav_inst*.

```
1  connections External_Nav_cnx
2    {
3            ...
4      auto_nav_inst.set_pos <−> NMEA_current_pos;
5    };
```

Exemple 5.13 – Définition des connections des chaînes fonctionnelles

5.3.4 Impact des changements de mode

Dans l'exemple 5.14, le mot clé "in modes" est utilisé pour représenter le fait que la connexion entre l'interface graphique de l'utilisateur et le sous-système de navigation ne peut être utilisée que dans le mode manuel (*manual*).

```
1  system Navigation_real realizes Navigation_subsystem {
2     ...
3    connections External_Navigation_cnx {
4      manual_navigation_inst.Steering_cmd <−> Steering_cmd
5                              in modes Navigation_supervisor_inst.manual;
6      auto_navigation_inst.Steering_cmd  <−> Steering_cmd
7                              in modes Navigation_supervisor_inst.automatic;
8       ...
9    };
10 };
```

Exemple 5.14 – Impact des changements de mode

5.4 Spécification de la politique de reconfiguration

Comme nous l'avons vu au chapitre 4 (section 4.4), le comportement d'un système en cours de reconfiguration dépend de la politique de reconfiguration mise en œuvre. En effet, le choix de la politique de reconfiguration va permettre de réaliser un compromis entre la nécessité de réduire le temps de reconfiguration, de réduire la perturbation de l'application du fait de la reconfiguration, ou encore de garantir la disponibilité de données cohérentes entre elles.

5.4.1 Politique par défaut

Lorsque l'utilisateur utilise uniquement les éléments de description que nous avons présentés jusque là, la priorité de la reconfiguration vis-à-vis du reste de l'application n'est déterminée que par la priorité de la tâche qui sera déclenchée pour mettre en œuvre ce changement de

mode. Ainsi, lorsque le changement de mode peut avoir lieu (c'est-à-dire lorsque cette tâche de reconfiguration est la tâche la plus prioritaire du système), le verrou de type lecteur/écrivain est réservé en écriture par la tâche de reconfiguration. Si ce verrou était déjà réservé en lecture par une tâche impactée par la reconfiguration, alors la reconfiguration est suspendue jusqu'à ce que toutes les tâches impactées aient libéré le verrou.

Ainsi, les interférences subies par la tâche de reconfiguration peuvent être dues à l'exécution de tâches moins prioritaires que la tâche de reconfiguration mais plus prioritaires qu'une des tâches impactées par la reconfiguration.

Si la tâche de reconfiguration a une priorité inférieure à la priorité de toutes les tâches impactées par la reconfiguration, alors nous pouvons borner son temps de reconfiguration (voir équation (7.1)), et nous pouvons affirmer que parmi les protocoles de reconfiguration que notre outil permet d'implanter, celui-ci permettra de minimiser les interférences de la reconfiguration sur le reste de l'application.

Dans cette sous-section, nous avons montré que le fait de donner une priorité importante à la tâche de reconfiguration ne permettait pas de s'assurer que la reconfiguration sera effectuée en priorité par rapport aux tâches de l'application dont la priorité est inférieure. Dans la sous-section suivante, nous présentons un moyen de spécifier que l'ensemble des changements de mode d'une instance d'automate de mode doit se faire avec une priorité prédéterminée : la priorité plafond.

5.4.2 Utilisation d'une priorité plafond

Lorsque la reconfiguration est plus prioritaire que certaines tâches de l'application, il faut non seulement paramétrer la tâche de reconfiguration avec une priorité supérieure à la priorité de ces tâches, mais il faut aussi s'assurer que les tâches dont la priorité est comprise entre la priorité d'une tâche impactée et la priorité de la tâche de reconfiguration ne perturbera pas la reconfiguration. Pour cela, nous avons étendu le mécanisme d'héritage de priorité de type PCP aux verrous du type lecteurs/écrivains : lorsqu'une tâche bloque la tâche de reconfiguration parce qu'elle a acquis le verrou, cette tâche hérite de la priorité plafond qui est nécessairement supérieure à la priorité de la tâche de reconfiguration.

Ce mécanisme permet ainsi de réduire le pire temps de reconfiguration, comme l'exprime l'équation (7.2).

L'exemple 5.15 illustre l'utilisation de cette politique, associée à l'instance d'automate de mode *Pilot_supervisor_inst*.

```
1 system Pilot_real realizes Pilot {
2  ...
3    mode automaton instance Pilot_supervisor_inst
4                     instantiates Pilot_supervisor_impl {
5           ceiling priority : 25;
6    };
7  ...
8 };
```

Exemple 5.15 – Utilisation de la priorité plafond

5.4.3 Synchronisation des tâches à l'hyper-période

Afin de garantir la disponibilité de données cohérentes entre elles pendant la mise en œuvre de la reconfiguration, nous avons décidé d'effectuer le changement de mode à l'hyper-période des tâches qui doivent être synchronisées. Pour identifier ces tâches, nous proposons à l'utilisateur d'identifier les tâches qui nécessitent d'être synchronisées en référençant les tâches périodiques dans des sous ensembles de synchronisation.

Ainsi, dans l'exemple 5.16, les activités *guidance_activity* et *positioning_activity* sont identifiées comme appartenant au même groupe de synchronisation.

```
1 system Pilot_real realizes Pilot {
2  ...
3    synchronization (guidance_activity,positioning_activity );
4  ...
5 };
```

Exemple 5.16 – Groupe de tâches synchronisées

Nous souhaitons ici positionner cette propriété vis-à-vis des propriétés que l'on peut exprimer en AADL. En effet, il existe en AADL une propriété qui permet de spécifier le fait qu'une tâche doit être synchronisée (propriété "synchronized"). Cependant cette propriété ne permet pas d'identifier différents sous-ensembles de tâches à synchroniser.

Prenons l'exemple de trois tâches (*T1*, *T2* et *T3*) qui doivent être synchronisées de la façon suivante : *T1* doit être synchronisée avec *T2* et *T2* doit être synchronisée avec *T3*. En COAL, il est possible d'identifier ces deux groupes distincts de synchronisation. En AADL, on dira que *T1*, *T2* et *T3* doivent être synchronisées. Considérons maintenant un changement de mode qui impacte la tâche *T1* seulement. Dans ce cas, il est possible de déterminer à partir de la spécification COAL que la reconfiguration aura lieu à l'hyper-période de *T1* et *T2*. En utilisant AADL il n'est pas possible d'identifier différent groupes de synchronisation, si bien que la reconfiguration aura lieu à l'hyper-période de *T1*, *T2* et *T3*.

5.5 Synthèse

Nous avons présenté dans ce chapitre le langage de description d'architectures à base de composants génériques, COAL. Ce langage permet de modéliser un système TR²E critique et adaptatif tout au long du processus de développement que nous avons présenté au chapitre précédent (voir figure 4.1). En effet, COAL permet de suivre ce processus de développement en proposant de modéliser la spécification système et les modes de fonctionnement associés, puis l'architecture logicielle à base de composants génériques, les configurations associées à chaque mode, et enfin l'implémentation des composants logiciels fonctionnels.

De plus, COAL fourni un point d'entrée à la méthode de conception que nous avons proposée au chapitre précédent (voir figures 4.2 et 4.3). En effet, la spécification système en COAL fournit une représentation des systèmes, sous-systèmes, et modes de fonctionnement associés. COAL fournit donc une solution concrète pour représenter les automates de mode définis au chapitre précédent (voir sous-section 4.2.2). Ceci permet d'automatiser la première étape de notre méthodologie (voir figure 4.2), à condition bien sûr de montrer que la sémantique d'exécution définie par COAL est analysable (ceci fera l'objet du chapitre 7).

Nous devrons donc définir l'implémentation de cette sémantique d'exécution. Or COAL permet de représenter les configurations associées à chaque mode de fonctionnement, et fournit

donc une réponse concrète à la proposition formulée au chapitre précédent pour modéliser la reconfiguration pseudo-dynamique (voir sous-section 4.2.3).

La définition de l'implémentation de la sémantique d'exécution associée nous permettra de mettre en œuvre la deuxième étape de notre méthodologie (voir figures 4.3), à condition encore une fois que la sémantique d'exécution définie soit analysable.

Enfin, COAL résout le problème lié à l'utilisation conjointe de composants logiciels génériques (tels que ceux définis dans Lightweight CCM) et de composants concrets d'un langage de description d'architecture spécifique au domaine des systèmes TR²E (type AADL).

Dans la suite de ce mémoire, nous allons donc définir l'implémentation de la sémantique d'exécution des systèmes pseudo-dynamiquement reconfigurables. Nous montrerons alors que cette sémantique d'exécution peut être générée et analysée.

Chapitre 6

MyCCM-HI, mise en œuvre automatique de la reconfiguration dynamique

La connaissance s'acquiert par l'expérience, tout le reste n'est que de l'information.
Albert Einstein

SOMMAIRE

6.1 Introduction

Dans ce chapitre, nous montrons que l'utilisation d'un langage déclaratif tel que COAL permet d'automatiser la production du code d'implémentation des applications TR²E critiques et adaptatives.

Production et analyse. L'utilisation d'un langage déclaratif tel que COAL facilite l'analyse formelle du comportement de cette application en fournissant un point d'entrée à la génération de modèles formels. Cette modélisation formelle représente le comportement de l'application logicielle qui sera finalement instanciée sur la plateforme cible. Pour s'assurer que le modèle

analysé correspond au comportement du système déployé, nous allons générer conjointement le code technique de l'application, qui réalise ce comportement, et le modèle formelle associé, qui permet d'analyser ce comportement. La génération du code de l'application est donc un préambule à la génération des modèles formels de sons comportement.

Contexte technologique. Dans le présent chapitre, nous allons décrire en détail notre outil de génération de code associé au *framework* MyCCM-HI [3] qui permet de mettre en œuvre la démarche de génie logiciel que nous avons proposée au chapitre 4. MyCCM-HI est disponible en version *open source* à l'adresse : http ://myccm-hi.sourceforge.net.

MyCCM-HI constitue en fait un évolution majeure du framework à composant MyCCM [Borde *et al.*, 2008]. En effet, MyCCM-HI répond aux problématiques spécifiques des systèmes TR2E critiques, adaptatifs, et très contraints en termes d'embarquabilité, alors que MyCCM s'intéresse à la réalisation de systèmes dont les contraintes majeures sont les performances réseaux et la maintenance en ligne. MyCCM s'appuie ainsi sur un processus de déploiement dynamique, alors que MyCCM-HI utilise des techniques de déploiement statique.

Organisation du chapitre. Pour décrire les générateurs de code de MyCCM-HI, nous commencerons par présenter les choix technologiques sur lesquels s'appuie ce prototype. Nous présenterons alors sa structure générale, ce qui nous permettra d'illustrer l'utilisation de chacun des choix technologiques que nous avons fait. Enfin, nous décrirons plus particulièrement le code généré dédié à la mise en œuvre de la reconfiguration pseudo-dynamique. Nous montrerons ainsi que l'utilisation d'un langage déclaratif tel que COAL permet d'automatiser la production du code des systèmes TR²E critiques auto-adaptatif.

6.2 Choix technologiques

La variabilité des exigences non-fonctionnelles est la principale problématique que doit traiter une démarche de génie logiciel dédié à la réalisation des systèmes TR²E. Ainsi, il est utopique de réaliser une unique démarche qui couvre l'ensemble des exigences de ce domaine. Du système de contrôle aérien, au logiciel d'asservissement du pod de reconnaissance installé sur un avion de chasse, en passant par la radio logicielle qui équipe les soldats en opération, il est évident que les habitudes de conception, guidées par des exigences fonctionnelles et non-fonctionnelles très différentes, aboutissent à des démarches très différentes de conception et de réalisation des logiciels TR²E.

Dans le domaine des systèmes TR²E, il est donc nécessaire d'adapter la démarche de génie logiciel au domaine d'activité qui l'utilise. Plusieurs solutions ont été envisagées pour répondre à ce besoin. Une première solution consiste à utiliser des composants logiciels prédéfinis afin d'implanter le comportement répondant aux besoins spécifiques d'un domaine [Borde *et al.*, 2007]. Il suffit alors de connecter ces composants prédéfinis avec le reste de l'application pour obtenir le comportement désiré.

Une telle approche n'est pas suffisante dans le contexte des systèmes critiques. Non seulement elle masque la sémantique du composant derrière une implémentation particulière, ce qui rend l'analyse du comportement de l'application difficile, mais surtout, un certain nombre d'exigences doivent nécessairement être prise en charge par le générateur de code

3. Make your Component Container Model - High Integrity

en lui même. Par exemple, en ce qui concerne les systèmes critiques, il est nécessaire de disposer de générateurs de code qui ne produisent aucune allocation dynamique de mémoire.

Pour répondre à la variabilité des exigences non-fonctionnelles des systèmes TR²E, une autre solution consiste à adapter le générateur de code lui même. L'objectif étant de faciliter cette adaptation, nous nous sommes appuyés sur trois méthodes de génie logiciel complémentaires : la génération de l'analyseur syntaxique, la méta-modélisation, et l'utilisation d'un intergiciel schizophrène.

Présentons les bénéfices de chacune de ces méthodes dans le but d'adapter une méthode de génie logiciel dirigée par les modèles à de nouvelles exigences non-fonctionnelles.

Génération de l'analyseur syntaxique Adapter une démarche de génie logiciel dirigée par les modèles aux variations des exigences non-fonctionnelles d'un domaine nécessite le plus souvent de modifier le format de représentation associé à cette démarche. Cela revient finalement à concevoir un langage spécifique au domaine d'activité pour lequel nous devons proposer une démarche de génie logiciel. Dans ce mémoire, nous nous plaçons dans le domaine des systèmes critiques et adaptatifs. Nous avons donc limité la sémantique d'exécution aux activités périodiques et sporadiques. Si nous nous étions placés dans un autre domaine d'activité, les activités apériodiques auraient été utiles. Plutôt que de ré-inventer un nouveau langage pour pouvoir traiter ce besoin, pourquoi ne pas étendre la spécification de COAL ? Une modification du langage est donc nécessaire pour pouvoir modéliser les entités correspondant à cette extension. L'utilisation d'une méthode de génération d'analyseur syntaxique facilite ce type d'adaptations en proposant des mécanismes d'héritage de grammaire. En outre, au delà de la génération du code de l'analyseur syntaxique, cette méthode fournit une représentation graphique des règles grammaticales, une représentation graphique de l'arbre syntaxique abstrait des exemples de test, ainsi que la vérification de l'absence d'ambiguïtés ou de boucles infinies dans les règles spécifiées (une ambiguïté correspond au fait que deux règles grammaticales identiques aboutissent à deux actions différentes).

Méta-modélisation Une fois que l'analyseur syntaxique a été mis à jour pour intégrer les besoins de représentation spécifiques d'un domaine d'activité, sont mises à jour les structures de données associées au générateur de code, ainsi que les outils de transformation de modèle. La méta-modélisation facilite la résolution de ce problème dans la mesure où elle permet (i) de partager une représentation graphique de la structure de données de chaque modèle (AST et/ou modèles intermédiaires) ; (ii) de générer le code des interfaces associées à cette structure de données ; et (iii) de générer des outils de représentation des données associées à une instance du méta-modèle. Ainsi, si on dispose de trois méta-modèles, par exemple M_a, M_b et M_c, il est alors possible de réaliser et de tester les transformateurs de M_a vers M_b et de M_b vers M_c en parallèle puisqu'on peut instancier indépendamment les structures de données associées à chacun des méta-modèles M_a et M_b. Ceci facilite largement le prototypage d'un outil. De plus, un méta-modèle donné peut référencer un autre méta-modèle. Ainsi, pour ajouter la méta-donné correspondant à une tâche sporadique, il suffit de référencer le méta-modèle de COAL et d'ajouter la méta-classe correspondant à la tâche apériodique pour définir l'AST du nouveau langage.

Intergiciel schizophrène Outre la nécessité d'adapter les générateurs de code, un besoin important lors de l'adaptation d'une méthode de génie logiciel dédié au systèmes TR²E

consiste à utiliser un intergiciel capable de prendre en compte la variabilité des exigences de ces systèmes. Pour ce faire, nous avons choisi de nous appuyer sur la notion d'intergiciels schizophrènes [Pautet, 2001]. Avec ce type d'intergiciel, les services fournis sont adaptés en fonction des besoins applicatifs (personnalités applicatives) et des besoins de communication (personnalités protocolaires). Ainsi, l'utilisation d'un bus de terrain particulier consistera en l'utilisation d'une personnalité protocolaire spécifique de cet intergiciel.

Dans cette section, nous avons présenté les méthodes de génie logiciel sur lesquels nous nous sommes appuyés pour améliorer l'adaptabilité de notre prototype à différents domaines d'activité. Dans la section suivante, nous allons illustrer la mise en œuvre de cette démarche à travers la réalisation des générateurs de code du *framework* à composants MyCCM-HI.

6.3 Structure générale du compilateur

MyCCM-HI est un générateur de code qui interprète une spécification d'architecture logicielle (aujourd'hui au format COAL) pour produire l'ensemble des exécutables d'une application TR²E critique pseudo-dynamiquement reconfigurable. Il s'agit donc d'un compilateur dont le format d'entrée est une spécification COAL, et le format de sortie est un ensemble de fichiers source (code C) qui seront eux même compilés pour produire les exécutables binaires. Comme tout compilateur, MyCCM-HI se décompose en trois parties (frontale, centrale, et dorsale) que nous allons décrire dans les sous-sections qui suivent. La figure 6.1 illustre, à travers ces trois parties, l'ensemble des étapes nécessaires à la génération de code. Sur cette figure, sont également représentées les étapes qui s'appuient sur les méthodes de génie logiciel que nous avons présentées à la section précédente (voir légende, en haut à droite de la figure).

Le modèle AADL produit (représenté en bas à droite de la figure) est ensuite utilisé par Ocarina pour produire le code d'activation et de communication des applications. Ce code s'appuie alors sur l'intergiciel schizophrène PolyORB-HI (C).

6.3.1 Partie frontale

Pour des raisons de simplicité de mise en œuvre et d'adaptation, nous avons choisi de nous appuyer sur une **représentation textuelle** des architectures à base de composants. Nous avons donc conçu un analyseur syntaxique et un méta-modèle qui permettent d'obtenir, à partir d'une spécification COAL donnée, l'arbre syntaxique abstrait (AST[4]) correspondant. Afin de réaliser cette partie frontale de notre outil de génération de code, nous nous sommes appuyés sur un générateur d'analyseur syntaxique et sur une technique de métamodélisation : nous avons utilisé ANTLR pour réaliser l'analyseur syntaxique, et EMF pour modéliser et générer le code correspondant à l'arbre syntaxique abstrait de ce langage d'entrée (voir figure 6.1, du fichier COAL à l'AST).

ANTLR (ANother Tool for Language Recognition) est un outil de génération de code dédié à la réalisation d'interpréteurs de langages : à partir d'une description de la grammaire du langage et d'un ensemble d'actions associées, ANTLR génère le code de reconnaissance du langage et exécute les actions associées dès que la règle de grammaire est rencontrée. Il permet également de représenter l'AST d'un exemple de test, de représenter graphiquement les règles grammaticales du langage, ou encore de détecter les ambiguïtés de la spécification.

4. Abstract Syntax Tree

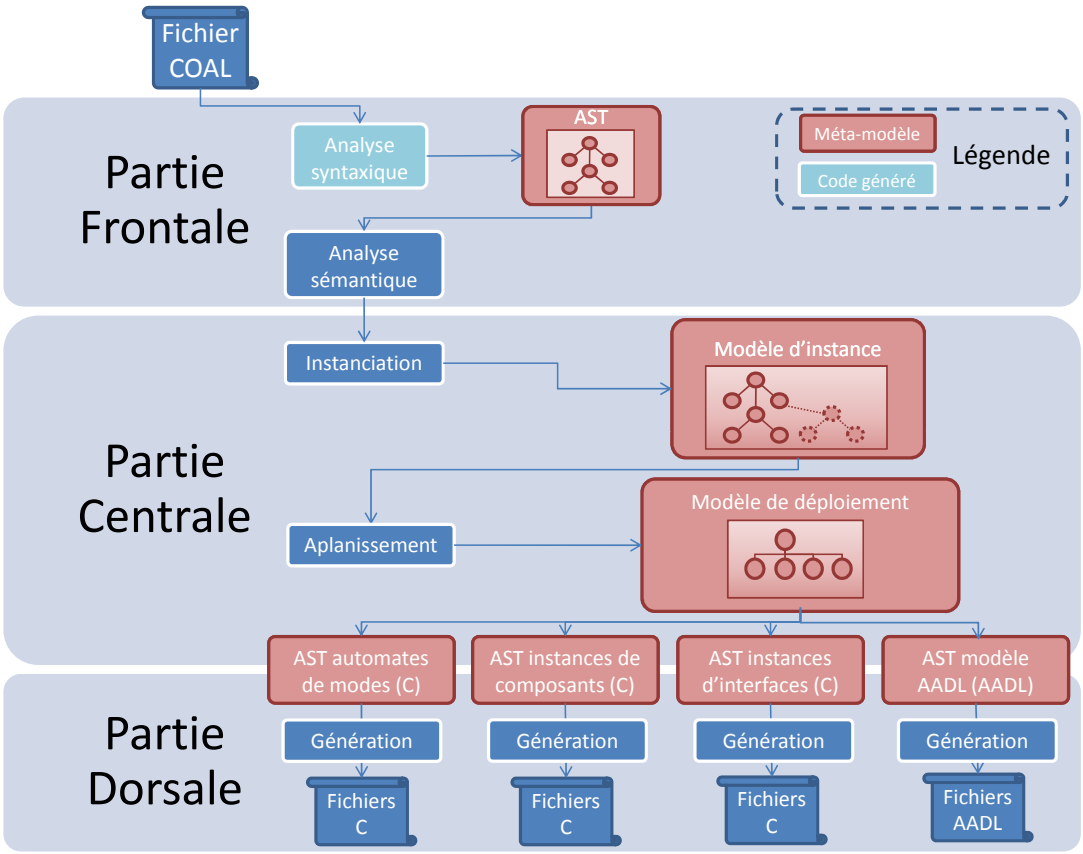

FIGURE 6.1 – Processus de conception d'un système TR²E reconfigurable

EMF (Eclipse Modeling Framework) est un outil de modélisation et de génération de code qui permet (i) de modéliser graphiquement un méta-modèle sous la forme d'un diagramme de classe, (ii) de générer le code des interfaces Java correspondant à ce diagramme de classes, et (iii) de générer le code d'un "plugin eclipse" correspondant à un éditeur arborescent d'instances de ce méta-modèle. Cet outil a été utilisé dans le cadre de la réalisation de MyCCM-HI afin de modéliser l'AST du langage COAL, ainsi que pour modéliser les modèles intermédiaires qui permettent d'aboutir à la génération de code (voir sur la figure 6.1 les éléments qui correspondent à la légende).

Comme le montre la figure 6.1, MyCCM-HI effectue dans la partie frontale certaines vérifications sémantiques sur l'instance de modèle obtenu. MyCCM-HI vérifie par exemple que :

- tout puit d'événement doit être associé par une activité sporadique, qui traitera les évènements arrivant sur ce puit ;
- toute instance de composant primitif est bien associée à un processus d'exécution ;
- toutes les activités sporadiques et périodiques définissent une période strictement positive et une priorité positive ;
- toutes les implémentations de composants primitifs doivent être instanciées au moins une fois ;

– tout port “client” (réceptacle ou source d’évènements) d’une instance de composant primitif doit être connecté à un port “serveur” (facette ou puits d’évènement) d’un composant primitif.
– aucune communication synchrone (via une connexion facette/réceptacle) ne doit être effectuée entre deux processus d’exécution différents ;
– la priorité plafond associée à une instance d’automate doit être supérieure à la plus haute priorité des tâches impactées par cette instance d’automate ;
– etc...

Une fois que MyCCM-HI s’est assuré que les règles syntaxiques et sémantiques du langage COAL sont bien respectées, la partie centrale du traitement (phases de transformations de modèle) peut commencer.

6.3.2 Partie centrale

La partie centrale du générateur de code consiste en une série de transformations de modèle qui produisent une représentation simplifiée de l’architecture logicielle. Comme nous l’avons expliqué au chapitre 6, COAL est un langage de description d’architecture “hiérarchique” dans la mesure où un composant peut être un composite, c’est-à-dire contenir d’autres composants (composites et/ou primitifs). Une première étape de transformation de modèle (voir figure 6.1) vise donc à obtenir une représentation “aplatie” de l’architecture. Dans la suite de ce chapitre, nous appellerons “modèle de déploiement” la structure de donnée associée à cette représentation aplatie.

Pour obtenir le modèle de déploiement équivalent à une spécification COAL donnée, une première étape consiste à construire un modèle d’instance de l’architecture. Il s’agit alors d’enrichir le modèle obtenu en sortie de la partie frontale afin d’obtenir une représentation qui ne contient plus que des instances de composants (primitif et composites) et des implémentation de composants primitifs.

Construction du modèle d’instances Lorsque l’architecte logiciel décrit une implémentation de composant, celle-ci peut être constituée de plusieurs sous-composants (voir exemple de spécification COAL 5.9). Chaque instance de ce composite contient alors les “sous-instances” correspondant à ces sous-composants. Les figures 6.2 et 6.3 illustrent ce propos : sur la figure 6.2, l’implémentation de composant *cpt_impl* contient deux instances de composants. Supposons alors, comme l’illustre la figure 6.2, que cette implémentation de composant soit instanciée deux fois (*cpt_inst1* et *cpt_inst2*). Dans ce cas, les instances de composants contenus dans *cpt_impl* doivent chacune être instanciées deux fois. D’où le résultat de la transformation, décrit sur la figure 6.3.

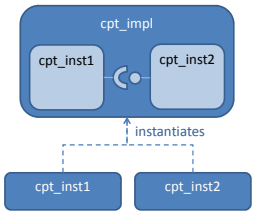

FIGURE 6.2 – Exemple de modèle issu de la partie frontale

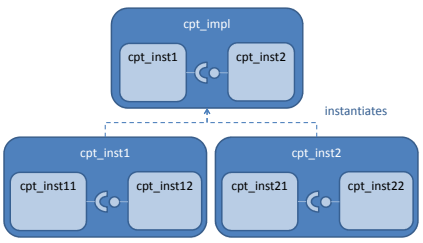

FIGURE 6.3 – Modèle d’instance correspondant

La construction du modèle d'instance nécessite non seulement de recopier les instances de composants qui sont créées plusieurs fois du fait de l'instanciation multiple d'un composant composite, mais également de recopier leurs paramètres de configuration (connections, initialisation d'attributs, déploiement sur un processus donné, etc...). Les figures 6.2 et 6.3 illustrent ce propos à travers la connexion entre les composants *cpt_inst11* et *cpt_inst12* d'une part, et *cpt_inst21* et *cpt_inst22* d'autre part.

Construction du modèle de déploiement Une fois le modèle d'instance obtenu, il s'agit de transformer ce modèle en un modèle "aplati", c'est-à-dire ne contenant que des instances de composants primitifs. Il s'agit alors de contracter l'ensemble des connections entre composites et/ou primitifs à un ensemble de connections entre instances de composants primitifs. Ainsi, pour chaque composant primitif du modèle d'instance, on récupère les primitifs auxquels il est connecté et on recopie ses paramètres de configuration dans une structure de donnée non hiérarchique représentative du modèle de déploiement. Le but de cette étape est d'obtenir une représentation simplifiée de l'architecture, donc plus facile à parcourir pour créer les modèles correspondant aux AST des langages cibles.

Intergiciel schizophrène et construction du modèle AADL Comme nous l'avons expliqué à la section précédente, nous avons fait le choix d'utiliser un intergiciel schizophrène. En outre, nous avons voulu que cet intergiciel soit dédié aux systèmes TR²E critiques. Nous avons donc choisi l'intergiciel PolyORB-HI (en langage C), ainsi que le générateur de code associé (Ocarina) qui s'appuie sur le langage de description d'architecture AADL. Nous avons donc conçu un générateur de code qui, à partir d'une spécification COAL, produit un modèle AADL équivalent (voir figure 6.1 en bas à droite).

La construction du modèle AADL est une étape importante du processus de génération de code puisqu'elle permet de bénéficier des travaux et outils réalisés autour de ce langage. Dans un premier temps, nous limitons l'utilisation de ce langage au périmètre des composants AADL suivants : processus (*process*), tâches (*thread*), canaux de communication (*event data ports, connections, et associations aux bus*), processeurs (*processor*), et bus (*bus*).

L'algorithme de création du modèle AADL s'appuie sur deux hypothèses :

- toute opération de composant est susceptible de solliciter chacun des ports de sortie de ce composant (réceptacles ou source d'évènement) ;
- les communication de types "facettes-receptacles" sont systématiquement réalisées dans la tâche qui exécute le code du composant "client".

L'algorithme de création du modèle AADL consiste alors en un parcours d'arbre dont les éléments sont des composants logiciels (de type CCM) et dont la racine est le point d'entré une activité COAL. A partir de ce composant racine, on crée un processus AADL correspondant au processus sur lequel est instancié le composant, puis une tâche AADL (*thread*) équivalente à l'activité (c'est à dire périodique si l'activité est périodique, ou sporadique si l'activité est sporadique). Cette tâche AADL est alors un sous-composant (au sens AADL) du processus que nous venons de créer. On parcourt ensuite l'ensemble des connections du composant racine. Pour une connexion du type source/puit(s) d'évènements, on ajoute un port de sortie/d'entrée (*event data port*) à la tâche AADL correspondante, et au processus AADL si le destinataire du message n'est pas co-localisé avec l'émetteur. Pour une connexion de type réceptacle→facette, le composant courant devient le composant connecté au réceptacle, et on étudie ses connections.

Ce premier traitement permet de définir l'ensemble des tâches et processus AADL ainsi que leurs ports de communication (entrées/sorties). Une table de correspondance associe un ensemble port AADL à une source ou un puits d'évènements CCM. Il suffit alors de parcourir les connections définies dans le modèle de déploiement pour définir les connections correspondantes dans le modèle AADL.

6.3.3 Partie dorsale

La partie dorsale de MyCCM-HI a pour objectif de générer le code correspondant à l'architecture logicielle spécifié par l'architecte logiciel. Ceci comprend :

- l'instanciation et l'initialisation des composant ;
- la définition du CIF [5] (Component Implementation Framework) [(OMG), 2003] ;
- les connections entre les composants ;
- le code d'activation des composant ;
- l'implémentation des automates de mode ;
- le code de synchronisation entre tâches impactées par la reconfiguration et tâches configurant les automates de mode ;
- le code permettant de réaliser les actions de reconfiguration (mise à jour de valeurs d'attributs, modification de connections) ;

Patron de génération Afin de faciliter l'implantation de l'ensemble du générateur de code, nous l'avons divisé en différentes étapes de génération dont le patron de génération est toujours le même. Il consiste en une phase d'expansion, et une phase de génération.

La phase d'expansion consiste en une transformation de modèle. Le modèle d'entrée est le modèle de déploiement. Le modèle de sortie est un modèle représentatif de l'ensemble des données nécessaires à la génération de code (AST du langage cible sur la figure 6.1). Par exemple, pour la génération du modèle AADL, le modèle d'entrée est le modèle de déploiement, qui sera transformé pendant la phase d'expansion en un modèle AADL équivalent utilisé par Ocarina pour la génération de code C. Dans ce cas, la phase d'expansion consiste à définir le modèle AADL en suivant les principes décrits précédemment.

La phase de génération utilise l'outil *"StringTemplate"*, un langage constitué de balises qui constituent le degré de variabilité de la génération : ces balises seront remplacées pendant la génération de code par les chaînes de caractères contenues dans le modèle résultant de la phase d'expansion.

Dans ce chapitre, nous avons jusque-là présenté la structure générale de MyCCM-HI. Nous allons maintenant illustrer son fonctionnement à travers la production du code dédié à la mise en œuvre de la reconfiguration dynamique.

5. le CIF d'un composant correspond à l'ensemble des opérations que le développeur de composant doit implanter pour que son composant puisse être déployé, configuré, et connecté aux autres composant. Cette spécification contient également les opérations correspondant aux ports du composant, y compris celle que le développeur de composant peut utilise pour envoyer un message via une source d'évènement ou appeler un autre composant via un réceptacle.

6.4 Génération de code pour la reconfiguration dynamique

La génération de code pour la reconfiguration pseudo-dynamique, telle que nous l'avons présentée au chapitre 4, nécessite trois étapes de génération :

1. la génération des automates de modes, depuis leur définition jusqu'à leur instanciation ;
2. la génération des mécanismes de synchronisation entre tâches impactées par la reconfiguration et tâches mettant en œuvre cette reconfiguration ;
3. la génération des actions de reconfiguration (modifications de connections, modifications de valeurs d'attributs, désactivation d'activités périodiques).

Dans la suite de cette section, nous détaillons ces différentes étapes de génération : à partir de l'exemple présenté au chapitre 5 et de la spécification COAL associée, nous présentons le code généré correspondant à la mise en œuvre la reconfiguration pseudo-dynamique.

6.4.1 Génération du code des automates de mode

Commençons par le processus de génération qui vise à définir, implémenter et instancier un automate de mode. Ce processus se divise en trois étapes. Une première étape de génération des interfaces part de la définition en langage COAL des automates pour aboutir à une définition (en C) des types de données associées. Une deuxième étape vise à générer l'implémentation de ces automates. Enfin, une dernière étape permet de générer le code d'instanciation des automates. Les paragraphes suivants présentent chacune de ces étapes.

Génération des interfaces Prenons l'exemple 5.5, qui représente la spécification COAL de l'automate de mode du système de pilotage (cet exemple a été donné au chapitre 5). A partir de cette spécification, une première étape de génération consiste à définir un composant logiciel équivalent, ainsi que ses interfaces. L'exemple 6.1 représente le résultat de cette transformation.

```
1  module Pilot_real {
2
3    enum Pilot_supervisor_modes_t
4    {
5          Pilot_real_Pilot_supervisor_automatic,
6          Pilot_real_Pilot_supervisor_automatic_to_manual,
7           ...
8    };
9
10   component Pilot_supervisor
11   {
12         provides ::MCCM::reconfiguration rcf;
13         consumes Pilot_modes Pilot_mode_cmd;
14          ...
15         attribute  Pilot_supervisor_modes_t current_mode;
16   };
17 };
```

Exemple 6.1 – Définition du composant générée à partir d'un automate de mode

La définition des modes de l'automate *Pilot_supervisor* dans le fichier COAL se traduit par la définition du type énuméré *Pilot_supervisor_modes_t*, qui contient les différents modes possibles de cet automate. L'automate est quant à lui considéré comme un composant logiciel dont les ports correspondent aux ports de communication de l'automate défini dans le fichier COAL, à l'exception de la facette *rcf* (ligne 12) et de l'attribut *current_mode* (ligne 15) :

- la facette *rcf* constitue un point d'entrée pour la reconfiguration dynamique, dans la mesure où elle fourni un accès à l'opération de l'automate qui réalise le changement de mode. L'exemple 6.2 fournit ainsi la définition de cette interface ;
- l'attribut *current_mode* représente quand à lui le mode courrant de l'automate de mode. Puisque le mode initial de cet automate est le mode *automatic*, cet attribut devra être initialisé avec la valeur *Pilot_real_Pilot_supervisor_automatic* (voir exemple 6.6, ligne 7).

```
1 module MCCM
2 {
3   interface reconfiguration
4   {
5     void performTransition ();
6   };
7 };
```

Exemple 6.2 – Définition de l'interface "reconfiguration"

Ces définitions d'interfaces seront ensuite utilisées pour générer le code correspondant à ces définitions. Ce processus de génération de code s'appuie sur le standard Light-weight CCM [(OMG), 2003] et ses règles de transformation vers le langage de programmation C.

Génération des implémentations La génération des implémentations des automates consiste à générer (i) la définition de la structure de données correspondant à cette implémentation, et (ii) le code des opérations de cette implémentation.

L'exemple 6.3 représente la structure de donnée associée à l'implémentation d'automate *Pilot_supervisor_impl*.

```
1  typedef struct Pilot_supervisor_impl
2  {
3          ...
4     MCCM_reconfiguration_Pilot_supervisor_impl * m_rcf;
5          ...
6     struct Pilot_modes * m_Pilot_mode_cmd;
7      ...
8     Pilot_supervisor_modes_t m_current_mode;
9     mccm_rwlock_t * m_lock;
10 } Pilot_supervisor_impl;
```

Exemple 6.3 – Définitoin de l'implémentation de l'automate Pilot_supervisor_impl

Dans cette définition, le terme *m_rcf* correspond à un pointeur vers la facette *rcf* de l'automate ; *m_Pilot_mode_cmd* correspond au puits d'évènements *Pilot_mode_cmd* ; *m_current_mode* représente la valeur courante du mode de fonctionnement de l'automate ; *m_lock* permet de protéger l'accès au mode courant en lecture/écriture.

Voyons maintenant le code généré pour implémenter les opérations de l'automate à àtravers les exemples 6.4 et 6.5 :

```
1
2  void Pilot_supervisor_impl_push_Pilot_mode_cmd
3  (
4    PortableServer_Servant _servant,
5    AP_modes *evt,
6    CORBA_Environment * _env
7  )
8  {
9          Pilot_supervisor_impl * servant = (Pilot_supervisor_impl *) _servant;
10         *servant->m_Pilot_mode_cmd = *evt;
11 }
```

Exemple 6.4 – Pilot_supervisor_impl : Implémentation des opérations de réception d'évènements

L'exemple 6.4 montre le code associé à la réception d'une commande de changement de mode : la valeur de l'évènement correspondant à cette commande de changement de mode est stockée dans la structure de données de l'automate. L'exemple 6.5 représente la fonction de l'implémentation de l'automate *Pilot_supervisor_impl* qui met à jour la valeur du mode courant. Cette fonction correspond à l'opération *performTransition* définit dans l'interface reconfiguration (voir exemple 6.2). La structure de cette opération est la suivante :

1. acquisition du verrou en écriture (ligne 10) ;
2. branchement conditionnel sur la valeur du mode courant (lignes 12 à 14) ;
3. évaluation des conditions de mise en œuvre du changement de mode (ligne 16) ;
4. mise à jour du mode et exécution des actions associées (ligne 18 à 27). Une fois qu'un évènement de changement de mode a été consommé, c'est à dire utilisé dans une transition de mode, le champ du port correspondant est réinitialisée avec la valeur -1 (voir ligne 27). Cette valeur ne peut déclencher de changement de mode puisque le champ en question est de type énuméré ;
5. libération du verrou en écriture (ligne 35) ;

```
1
2  void Pilot_supervisor_impl_performTransition
3  (
4    PortableServer_Servant _servant,
5    CORBA_Environment * _env
6  )
7  {
8    CORBA_Environment env;
9    Pilot_supervisor_impl * component = (Pilot_supervisor_impl *) _servant;
10   mccm_rwl_writelock(component->m_lock);
11   Pilot_supervisor_modes_t current_mode = Pilot_supervisor_impl_get_current_mode(component, &env);
12   switch(current_mode)
13   {
14       case Pilot_supervisor_automatic :
15       {
16         if (component->m_Pilot_mode_cmd->request == c_manual)
17         {
18           Pilot_supervisor_impl_enter_mode_automatic_to_manual(component, &env);
19
20           component->m_Loc_mode_cmd->request = stop ;
```

```
21              Pilot_supervisor_impl_push_Loc_mode_cmd(component,
22                              component->m_Loc_mode_cmd,
23                              &env);
24
25              ...
26
27              component->m_AP_mode_cmd->request = -1;
28            }
29            break;
30          }
31           ...
32          default:
33                break;
34      }
35      mccm_rwl_writeunlock(component->m_lock);
36  }
```

Exemple 6.5 – Pilot_supervisor_impl : Implémentation de l'opérations de changement de mode

Génération des instances Enfin, pour finaliser la génération des automates de modes, il faut générer leur instanciation statique. L'exemple 6.6 représente cette initialisation :

```
1  Pilot_supervisor_impl  Pilot_supervisor_inst  =
2  {
3      ...
4    &Pilot_supervisor_inst_rcf_INSTANCE,
5      ...
6    &Pilot_supervisor_inst_Pilot_mode_cmd_evt_port                    ,
7    Pilot_supervisor_automatic                                        ,
8    &Pilot_supervisor_inst_rwlock,
9      ...
10 };
```

Exemple 6.6 – Instanciation de l'instance d'automate Pilot_supervisor_inst

Chacune des références utilisées dans cet exemple correspond à une variable globale dont la définition a été générée par ailleurs. Insistons ici sur le fait que le mode initial de l'automate est bien initialisé avec la valeur *Pilot_supervisor_automatic*.

Maintenant que nous avons présenté les structures de données et opérations générées à partir de la définition des automates de modes, voyons comment ces éléments s'interfacent avec le reste de l'application pour mettre en œuvre les politiques de synchronisation décrites au chapitre 4.

6.4.2 Génération des mécanismes de synchronisation

Nous avons déjà présenté à la section précédente comment le code de l'automate de mode utilise un verrou de type lecteur/écrivain pour protéger l'accès au mode courant en écriture. Nous allons ici ajouter trois considérations qui permettent de réaliser les mécanismes de synchronisation proposés précédemment (voir chapitre 4). Premièrement, nous allons montrer comment les tâches impactées protègent l'accès en lecture au mode courant. Ensuite, nous nous intéresserons au fonctionnement des verrous de type lecteurs/écrivains qui implantent

une politique d'héritage de priorité de type PCP. Enfin, nous nous intéresserons au code généré pour les tâches qui réalisent la reconfiguration.

En ce qui concerne le processus de génération, nous avons en réalité quatre cas à prendre en compte :

1. l'automate de mode n'est pas configuré avec PCP, et n'impacte pas de tâches périodiques synchronisées ;
2. l'automate de mode est configuré avec PCP, et n'impacte pas de tâches périodiques synchronisées ;
3. l'automate de mode n'est pas configuré avec PCP, et impacte des tâches périodiques synchronisées ;
4. l'automate de mode est configuré avec PCP, et impacte des tâches périodiques synchronisées.

Nous réutiliserons ces différents cas dans la suite de cette section pour illustrer leur incidence sur le code généré.

Protection en lecture du mode courant Nous nous plaçons ici dans les cas 1 et 3. Dans ces cas, une tâche impactée par un changement de mode de l'automate s'exécute de la façon suivante :

1. acquisition du verrou en lecture ;
2. exécution du code fonctionnel ;
3. libération du verrou en lecture.

L'exemple 6.7 illustre le code généré pour la tâche *guidance_activity* définie dans l'exemple 5.12. La fonction définie dans cet exemple sera appelée toutes les 100 millisecondes, conformément à la spécification COAL correspondant à l'activité *guidance_activity*.

```
1  void
2  guidance_activity_guidance_inst_activation(__po_hi_task_id task_id)
3  {
4      mccm_rwl_readlock(&Nav_supervisor_inst_rwlock);
5
6      CORBA_Environment env;
7      MCCM_activable_activate(guidance_inst.m_activation, &env);
8
9      mccm_rwl_readunlock(&Nav_supervisor_inst_rwlock);
10 }
```

Exemple 6.7 – Code généré pour l'activité guidance_activity

Ici, la fonction *mccm_rwl_readlock* correspond en réalité à l'encapsulation d'un verrou lecteur/écrivain tel que défini dans la norme POSIX [IEEE, 2004].

Verrous lecteur/écrivain et héritage de priorité Nous allons maintenant nous intéresser au cas des tâches impactées par un automate de mode configuré avec la politique *PCP* (cas 2 et 4). Ce cas est très similaire au cas précédent : la seule différence vient du fait que l'opération de protection utilisé s'appuie sur un verrou de type lecteur/écrivain dont nous avons ré-implanté le fonctionnement. Il suffit alors de remplacer *mccm_rwl_readlock* par

mccm_pcprwl_readlock et *mccm_rwl_readunlock* par *mccm_pcprwl_readunlock* pour obtenir le code généré adéquat. Une opération similaire est nécessaire au niveau du code généré pour l'automate de mode pour adapter le code d'acquisition du verrou en écriture (voir exemple 6.5). La fonction *mccm_pcprwl_readlock* enregistre l'ensemble des "identifiants systèmes" (*thread_id POSIX*) des tâches ayant pris le verrou en lecture. Lorsque l'automate de mode prend ce verrou en écriture, si le verrou est déjà pris par un ensemble de tâches lectrices, alors ces tâches lectrices héritent de la priorité plafond spécifiée par l'utilisateur (voir exemple 5.15).

Comme nous venons de le voir à travers les deux paragraphes précédents, le fait que l'automate soit configuré avec PCP ou non aura une influence sur le type de verrou utilisé mais ne modifiera pas le patron d'exécution des tâches impactées par la reconfiguration. Le fait que les changements de modes doivent se faire à l'hyper-période d'un ensemble de tâches périodiques (cas 3 et 4) va avoir une incidence sur le patron d'exécution des tâches qui exécutent le changement de mode.

Tâches impactées non-synchronisées Présentons maintenant le patron d'exécution des tâches de reconfiguration dans le cas où l'automate de mode n'impacte pas de tâches synchronisées (cas 1 et 3). Dans ce cas, le code de reconfiguration est exécuté dès que la commande de changement de mode est reçue : l'exemple 6.8 représente le code appelé par l'activité sporadique qui configure le port *Pilot_mode_cmd* de l'instance d'automate *Pilot_supervisor_inst*. Nous voyons ici que la fonction "performTransition" est invoquée (ligne 11) dès que la requête de changement de mode a été enregistrée (lignes 4 à 9).

```
1  void pilot_modes_activity_pilot_supervisor_inst_pilot_mode_cmd(__po_hi_task_id task_id, AP_modes data)
2  {
3    CORBA_Environment env;
4    Pilot_real_CCM_Pilot_supervisor_push_Pilot_mode_cmd
5      (
6       &Pilot_real_Pilot_supervisor_inst,
7       &data,
8       &env
9       );
10
11   Pilot_supervisor_impl_performTransition(&Pilot_supervisor_inst, &env);
12 }
```

Exemple 6.8 – Réalisation immédiate du changement de mode

Synchronisation à l'hyper-période Dans le cas ou l'automate de mode impacte un ensemble de tâches synchronisées, le changement de mode se fait en deux temps : lors de la réception de la requête de changement de mode, l'information correspondante est stockée. Une tâche périodique s'exécutant à l'hyper-période des tâches synchronisées est alors crée pour exécuter le code de changement de mode.

Nous allons maintenant étudier le troisième volet des étapes de génération du code dédié à la reconfiguration pseudo-dynamique.

6.4.3 Génération des actions de reconfiguration

La dernière étape de génération consiste à prendre en considération l'impact des changements de mode sur l'architecture logicielle. Nous allons ici expliquer comment le code généré

par MyCCM-HI permet (i) de modifier des connections, (ii) de mettre à jour des valeurs d'attributs, et/ou (iii) de désactiver des activités périodiques.

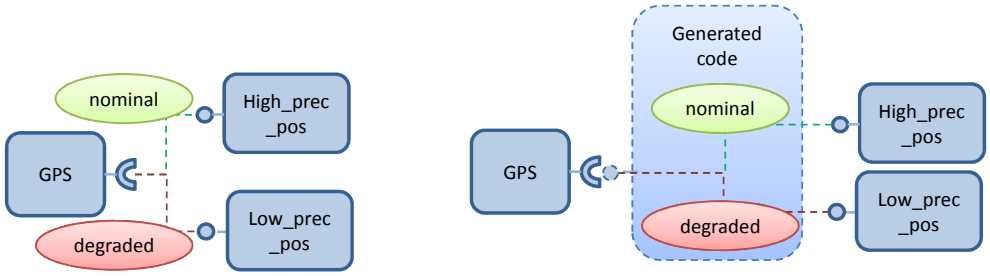

FIGURE 6.4 – Modélisation et représentation du code généré pour l'aiguillage d'une connexion facette/réceptacle

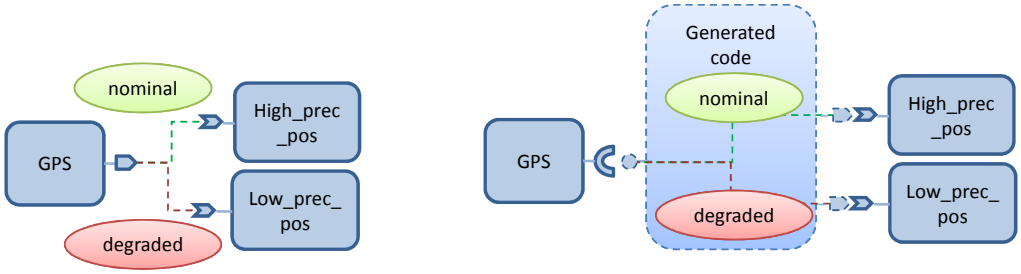

FIGURE 6.5 – Modélisation et représentation du code généré pour l'aiguillage d'une connexion source/puits d'évènements

Aiguillage des connections La façon dont l'aiguillage dynamique des connections est réalisé dépend du type de connexion :

- s'il s'agit d'une connexion de type facette/réceptacle, l'appel est local. Lorsque cette connexion dépend du mode de fonctionnement d'un automate, le code généré qui réalise cette connexion lit la valeur du mode courant de cet automate et appelle la facette connectée dans ce mode. Comme l'illustre la figure 6.4, MyCCM-HI génère dans ce cas une interface dédiée à l'aiguillage de la connexion ;
- s'il s'agit d'une connexion de type source/puits d'évènements, MyCCM-HI ajoute au *thread* AADL qui parcours le composant émetteur autant de ports de sortie qu'il existe de canaux de communication possibles en fonction du mode. Ensuite, MyCCM-HI génère le code responsable d'émettre l'évènement sur le canal correspondant au mode courant de l'automate. Tout se passe en fait comme si MyCCM-HI générait un composant intermédiaire responsable de l'aiguillage de la connexion, ce qu'illustre la figure 6.5. Cette stratégie de génération n'a pas été choisi pour des raisons d'optimisation : l'utilisation de composants a un coût en termes de quantité de code généré (donc embarqué).

Modification des valeurs des attributs Pour ce qui est de la modification d'attributs, nous considérons l'attribut d'un composant comme une variable que seul ce composant peut consulter. Ceci permet de s'assurer que nos protocoles de changement de mode vérifient le niveau de cohérence minimal exprimé précédemment et que nous rappelons ici : "toute tâche ayant commencé dans un mode de fonctionnement donné continue et finit son exécution courante en ayant accès à des données cohérentes vis-à-vis de ce mode".

La modification d'une valeur d'attribut est alors considéré comme une action de reconfiguration : si un attribut de composant définit une valeur dans un mode de fonctionnement donné, alors cet attribut sera réinitialisé avec cette valeur dès que le mode en question sera atteint. La mise à jour d'une valeur d'attribut est donc considéré lors de la génération de code comme une action associée au changement de mode, au même titre que l'émission d'un événement via un port de sortie de l'automate de mode. Le code correspondant est généré dans la fonction *"enter_mode"* de l'automate (voir exemple 6.5, ligne 18). Lorsque l'automate atteint ce mode, le code de mise à jour de l'attribut est exécuté.

Désactivation des activités périodiques La désactivation des activités périodiques consiste à n'appeler le point d'entré de l'activité que si le mode courant est un des modes dans lesquels l'activité est considérée comme active. Dans le cas contraire, le fil d'exécution correspondant à la tâche reste activé périodiquement, mais aucun code fonctionnel n'est exécuté ; ceci permet de garantir que le code de la tâche correspondante reste localement synchronisé avec le code des autres tâches du nœud sur lequel elles s'exécutent.

6.5 Synthèse

Dans ce chapitre, nous avons présenté l'ensemble du prototype que nous avons réalisé pour faciliter la conception et la réalisation des systèmes TR²E critiques et adaptatifs. Nous allons ici synthétiser cette présentation en nous attachant à décrire la chaîne de production associée à MyCCM-HI. Nous discuterons ensuite des améliorations possibles visant à faciliter la maintenance évolutive de ce prototype pour d'autres domaines d'activités.

6.5.1 Chaîne de production automatique

Comme nous l'avons expliqué précédemment, nous avons choisi d'utiliser l'intergiciel PolyORB-HI. Pour ce faire, il nous a fallu générer automatiquement un modèle AADL équivalent au modèle COAL que fournit l'utilisateur de MyCCM-HI en entrée du processus de génération. Par ailleurs, nous venons d'exposer le code généré par MyCCM-HI pour gérer la reconfiguration dynamique. Ainsi, MyCCM-HI génère, à partir de la spécification COAL, un modèle AADL équivalent, ainsi qu'un ensemble de fichier sources (en langage C). Comme le montre la figure 6.6, le modèle AADL est ensuite utilisé par Ocarina pour générer le code qui s'appuie sur PolyORB-HI pour réaliser l'activation des chaines fonctionnelles ainsi que les communications entre les différentes tâches de l'application. Enfin, le code produit par MyCCM-HI, le code produit par Ocarina, ainsi que le code d'implémentation des composants (fournis par l'utilisateur) sont compilés pour produire l'ensemble des exécutables de l'application TR²E.

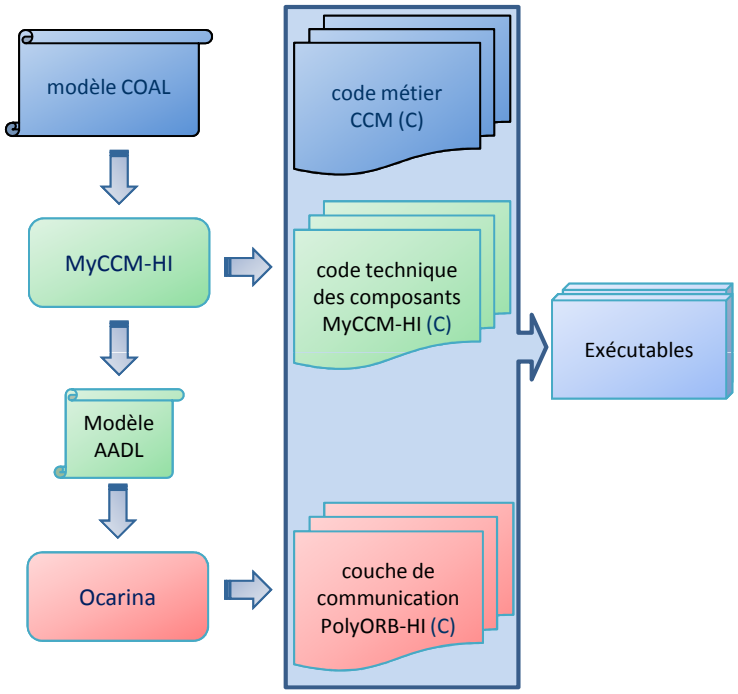

FIGURE 6.6 – MyCCM-HI, une chaîne d'outils

6.5.2 Maintenance évolutive : améliorations possibles

Les méthodes et outils de génie logiciel que nous avons utilisé (ANTLR, EMF, Ocarina) ont largement facilité la réalisation de MyCCM-HI. Nous n'avons pas encore testé les possibilités offertes par ces méthodes en terme de maintenance évolutive pour répondre à de nouvelles exigences fonctionnelles. Présentons ici ces possibilités.

Les générateurs d'analyseurs syntaxiques permettent d'utiliser la notion d'héritage de syntaxe. Ainsi, il suffit de définir un ensemble syntaxique minimal (basé sur D&C par exemple), et de l'enrichir avec certaines extensions (basées sur AADL par exemple) pour obtenir une spécification de grammaire équivalente à la grammaire de COAL. Définir une nouvelle syntaxe consiste alors à définir une syntaxe qui hérite d'un ensemble de spécifications grammaticales prédéfinies.

De la même façon, un méta-modèle peut en référencer un autre et utiliser les classes et relations définies dans le méta-modèle référencé.

La vision que nous souhaitons présenter ici, visant à améliorer l'adaptabilité du prototype MyCCM-HI, consiste alors à utiliser ces deux notions (héritage de grammaire et référencement de méta-modèle) pour adapter le générateur de code aux besoins d'un domaine d'activité.

Dans ce chapitre, nous avons insisté sur la notion de génération de code visant la réalisation des systèmes TR²E critiques. Dans la méthodologie que nous avons présenté au chapitre 4, cette étape correspond à l'étape finale du processus (voir figure 4.3). La réalisation de cette étape suppose que les étapes de vérification système et logicielles ont été franchies avec succès. Bien que nous n'ayons pas pu automatiser ces étapes, nous allons présenter dans le chapitre suivant les modèles et analyses sur lesquels elles s'appuient.

Chapitre 7

Analyse et vérification formelle de la reconfiguration dynamique

Une théorie physique sera dite déterministe si à partir de la connaissance supposée des résultats de certaines des mesures initiales, il est possible de prévoir avec certitude n'importe quelle mesure ultérieure.

Paulette Février

SOMMAIRE

7.1 Introduction

Réaliser un système critique nécessite d'utiliser tous les moyens disponibles pour augmenter la confiance que l'on peut avoir dans le logiciel associé. Les procédés de certification que nous avons décris (voir chapitre 2) s'appuient principalement sur des méthodes de gestion de projet qui visent (i) à documenter l'ensemble du code produit, (ii) à documenter l'ensemble des tests réalisées, et (iii) à mettre en relation les exigences systèmes avec ce code et ces tests.

Les méthodes de vérification formelle devraient faire leur apparition dans les futures versions des standards de certification.

L'objectif de ces techniques est de vérifier mathématiquement que certaines propriétés seront toujours vérifiées par le système conçu. De plus, l'utilisation automatique des techniques

de vérification à partir de spécifications logicielles permet de détecter au plus tôt dans le processus de développement des incohérences qui pourraient se manifester (ou pire, ne pas se manifester) pendant les phases de test.

Les techniques de vérification que nous allons utiliser s'appuient sur des modèles formels qui représentent le comportement du système et de ses applications logicielles.

Grâce à cette modélisation, et à l'utilisation d'algorithmes de vérification, nous pourrons vérifier les propriétés suivantes :

- un mode donné est accessible ;
- la spécification n'induit pas de blocage dans un mode donné ;
- deux modes fortement incohérents entre eux ne sont jamais simultanément actifs ;
- le passage d'un mode à un autre n'excède pas un certain intervalle de temps ;
- deux modes faiblement incohérents ne sont pas simultanément actifs plus d'une certaine période de temps ;
- le système reconfigurable est ordonnançable.

Nous distinguons ici deux types d'incohérence dans les modes de fonctionnement d'un système et de ses sous-systèmes. Des incohérences fortes, ce qui signifie que les modes en question ne doivent jamais être simultanément actifs ; et des incohérences faibles, ce qui signifie que les modes en question peuvent être simultanément actif, mais cette incohérence doit être temporellement bornée.

L'utilisation de techniques de vérification formelle est une tâche difficile dans la mesure où elle nécessite (i) de bien choisir le formalisme à partir duquel sera faite la vérification, (ii) de bien connaître le formalisme en question ainsi que le formalisme d'expression des propriétés à vérifier ; et (iii) de représenter suffisamment finement le comportement du système, ce qui constitue un travail long et fastidieux ; donc source d'erreurs.

Comme nous l'avons expliqué au chapitre 4, notre méthodologie facilite la vérification de ces propriétés en générant les modèles formels correspondant au comportement exprimé dans la spécification système et logicielle. Cette vérification est faite en deux étapes. Dans un premier temps, la spécification des systèmes, sous-systèmes et de leurs automates de mode donne lieu à la génération de modèles formels, puis à la validation des mécanismes de changement de mode. Si les propriétés relatives à cette première phase d'analyse sont vérifiées, la deuxième étape du processus donne lieu à une analyse de la spécification logicielle.

Pour réaliser ces analyses, nous avons choisi de nous appuyer sur deux types de techniques différentes. Pour la plupart des propriétés que nous avons mentionnées, nous nous appuierons sur des techniques de vérification formelle, et nous utiliserons pour cela l'outil UPPAAL [Behrmann *et al.*, 2004]. Cet outil permet de modéliser des automates communicants, ce qui facilite la transformation entre nos automates de mode et les automates UPPAAL. De plus, cet outil permet de modéliser les paramètres temporels d'une application, et de vérifier les propriétés temporelles associées. Pour simplifier l'analyse du temps maximal de reconfiguration, nous nous appuierons également sur une analyse au pire cas qui exploite principalement la notion de pire temps d'exécution d'une tâche.

Nous allons diviser la présentation des modèles et techniques sur lesquels s'appuient cette étude en deux sections : nous nous intéresserons d'abord à l'analyse système, puis à l'analyse logicielle.

7.2 Analyse système

L'analyse système consiste à analyser un sous-ensemble du langage COAL constitué par :

– les systèmes et sous-systèmes de la spécification ;
– les automates de modes associés à ces systèmes et sous-systèmes (types, implémentations, instances) ;
– les connexions entre les instances d'automates.

Commençons par expliciter le type de propriétés que l'on souhaite vérifier.

7.2.1 Propriétés à vérifier

A partir d'une spécification système, nous nous intéressons aux propriétés suivantes :
– un mode donné est-il atteignable ?
– peut-on rester bloqué dans un mode donné ?
– deux automates de modes peuvent-il être simultanément dans deux modes de fonctionnement incohérents ?

Pour vérifier ces propriétés, nous avons besoin (i) de modéliser le comportement des automates de modes, (ii) de modéliser les scénario d'exécution possibles afin de *fermer le système* (il s'agit par exemple de modéliser le comportement de l'utilisateur), et (iii) de spécifier les propriétés à vérifier.

7.2.2 Modélisation formelle des automates de mode

Nous proposons ici une règle de transformation entre l'implémentation d'un automate de mode d'une part, et sa modélisation formelle avec UPPAAL. Prenons le cas générique d'un changement de mode depuis un mode initial *Source* vers un mode *Target*. Supposons que l'automate de mode correspondant possède un puits d'évènement *si* et une source d'évènement *so*. Ces évènements utilisent chacun la même structure de données constituée d'un unique champ *value*, de type énuméré, et qui peut prendre la valeur *target*.

Voici l'expression de la règle de changement de mode (en utilisant une syntaxe COAL) : ***in mode*** *Source : { [si ?(value=target)] –> Target {so !(value :=target)} ;} ;* . Cette spécification signifie que lorsque l'automate de mode reçoit la commande de changement de mode *target* sur le port *si*, alors son nouveau mode courant est *Target* et il envoie un message contenant la valeur *target* via son port *so*.

La figure 7.1 représente le patron de modélisation correspondant à ce comportement de changement de mode.

Les modes *Source* et *Target* sont représentés par deux états portant le même nom que leur modes. Le mode initial de l'automate de mode correspond à l'état initial de l'automate UPPAAL. Lorsque l'automate de mode reçoit un évènement via le port *si* (transition avec synchronisation *si ?*), l'automate UPPAAL atteint un état intermédiaire dans lequel la valeur du message va être évaluée. Si la valeur correspond à la valeur *target* (transition avec garde *automaton.si.value==target*), le mode *Target* est atteint, et le message correspondant est envoyé via le port *so* (transition avec synchronisation *so !* et mise à jour *automate.so.value :=target*. Sur cette figure, *automaton* correspond à la structure de donnée de l'automate de mode. Cette structure, exprimée dans les déclarations de variables UPPAAL, permet de représenter l'implémentation de l'automate. Elle contient un champ *si* et un champ *so* qui stockent les dernières valeurs reçues ou émises sur les ports de communication correspondants. Une fois que la valeur courante a été utilisée dans un changement de mode, elle est ré-initialisée avec la valeur -1 (voir mise à jour *automaton.si.value :=-1*).

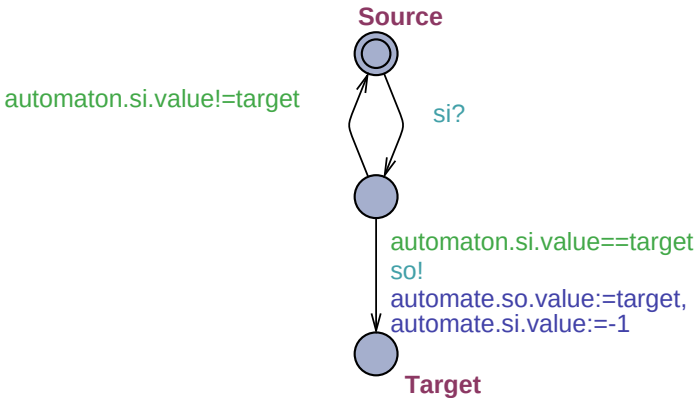

FIGURE 7.1 – Patron de modélisation d'un changement de mode

Le comportement représenté à travers l'automate UPPAAL correspond à celui du code généré tel que nous l'avions illustré à travers l'exemple de code 6.5.

7.2.3 Modélisation des connexions entre automates

Les automates de modes communiquent par le biais de connexions entre leurs sources et puits d'évènements. La figure 7.2 illustre la règle de transformation entre une connexion COAL et son équivalent UPPAAL.

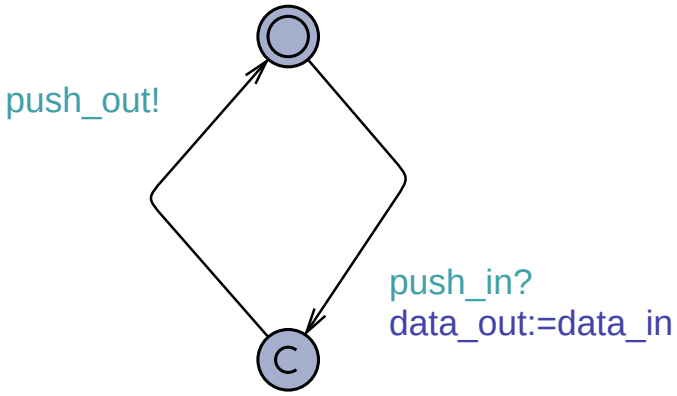

FIGURE 7.2 – Modélisation formelle des connexions entre automates

Sur cette figure, *push_in* est le canal UPPAAL qui correspond à la source d'événement de l'automate émetteur, alors que *push_out* est un canal UPPAAL qui correspond au puit d'évènement de l'automate destinataire. Enfin, *data_out* (respectivement *data_in*) est une donnée UPPAAL qui correspond à un champs de l'automate de mode destinataire (*resp.* émetteur) du message.

Ainsi, lorsque l'automate de mode émetteur envoie la donnée via son port source, l'automate UPPAAL de la connexion passe de l'état initial vers l'état intermédiaire (en bas

de la figure) via une transition qui met à jour la donnée à envoyer (transition avec mise à jour *data_out :=data_in*). Depuis cet état intermédiaire, l'automate UPPAAL de la connexion retourne dans son état initial en activant l'automate destinataire du message (transition avec synchronisation *push_out*.

Le comportement représenté ici modélise la sémantique de communication *push/push* des composant de type Lightweight CCM. Ce même patron de modélisation pourra donc être utilisé pour représenter les communications entre composants fonctionnels.

7.2.4 Modélisation de l'environnement du système

Pour que le modèle obtenu soit analysable, il faudra modéliser le comportement de l'environnement du système (données des capteurs, comportement de l'utilisateur, etc...). Cette opération sera effectuée en modélisant le comportement des composants logiciels correspondant à cet environnement (dans notre cas, l'utilisateur du système).

7.2.5 Types de vérifications possibles

A partir des modèles obtenus en appliquant les règles de transformation décrites précédemment, nous vérifierons trois types de propriété :

1. l'absence d'interblocage ;
2. l'accessibilité d'un mode donné ;
3. l'exclusion mutuelle de modes incohérents entre eux.

Dans la suite de ce chapitre, nous allons présenter les résultats analytiques ainsi que les règles de transformation de modèle qui permettent d'analyser formellement le comportement temporel de l'architecture logicielle.

7.3 Analyse logicielle

L'analyse logicielle vise à permettre de vérifier des propriétés temporelles, telles que l'ordonnançabilité du système, le pire temps de reconfiguration, le temps pendant lequel deux automates de mode sont dans des états incohérents, etc...

Nous nous sommes intéressés dans un premier temps à la notion de pire temps de reconfiguration. Ceci afin de montrer que l'utilisation d'une politique de reconfiguration donnée avait une incidence importante sur la notion de pire temps de reconfiguration, mais aussi pour fournir une méthode capable d'estimer ce pire temps de reconfiguration facilement (c'est-à-dire sans avoir recours à des techniques aussi sophistiquées que la vérification formelle).

Pour plus de précision, et dans le but d'élargir le spectre des vérifications possibles, nous nous sommes intéressés ensuite à la vérification formelle des architectures logicielles pseudo-dynamiquement reconfigurables. La suite de cette section est organisée ainsi : dans une première sous-section, nous présentons les formules mathématiques qui permettent, en fonction de la politique de reconfiguration choisi par l'architecte logiciel, d'estimer le pire temps de reconfiguration. Dans une seconde sous-section, nous décrivons les règles de transformation de modèle permettant d'automatiser l'analyse formelle du comportement temporelle de l'architecture logicielle.

7.3.1 Analyse numérique du pire temps de reconfiguration

L'analyse du pire temps de reconfiguration repose sur une analyse au pire cas considérant non seulement que l'ensemble de tâches de l'application peuvent être actives au moment où la requête de reconfiguration est reçue par l'automate de mode mais encore que toutes ces tâches sont susceptibles de s'exécuter pendant un temps correspondant à leur pire temps d'exécution. Ces hypothèses aboutissent à un pessimisme important dans l'évaluation du pire temps de reconfiguration, mais la facilité et la rapidité d'utilisation de ces formules les rendent très utiles pour fournir une estimation grossière du pire temps de reconfiguration d'un système TR²E.

Nous allons ici considérer quatre cas de configurations possibles d'une application TR²E critique reconfigurable. Ces quatre cas correspondent aux différentes configurations que nous avons présentées au chapitre 4, et qui visent :

1. à réduire les perturbations que subit l'application à cause de la mise en œuvre de la reconfiguration (voir sous-section 4.4.3) ;
2. à réduire le temps de reconfiguration (voir sous-section 4.4.4) ;
3. à garantir la cohérence de flux de données tout en minimisant les perturbations (voir sous-section 4.4.5) ;
4. à garantir la cohérence de flux de données tout en réduisant le temps de reconfiguration (voir sous-section 4.4.5).

Calculons maintenant le pire temps de reconfiguration associé à chacune des ces configurations.

Configuration visant à privilégier l'application

Rappelons la configuration à laquelle nous faisons référence ici (voir sous-section 4.4.3) :

1. la tâche de reconfiguration à une priorité inférieure à l'ensemble des tâches impactées par la reconfiguration ;
2. l'instance d'automate de mode considérée n'est pas configurée avec une priorité plafond.

Calculons dans cette configuration le pire temps de reconfiguration (WCRT) en fonction du pire temps d'exécution des différentes tâches en présence. Soit Im l'ensemble des tâches impactées par la reconfiguration. Ces tâches ont alors une priorité supérieure à celle de la tâche réalisant la reconfiguration. Soit Un l'ensemble des tâches qui ne sont pas impactées par la reconfiguration. L'équation (7.1), simple extension des résultats fournis par [Real and Crespo, 2004], donne le pire temps de reconfiguration associé à l'utilisation de ce protocole. Dans cette équation, une tâche j est caractérisée par sa période T_j et son pire temps d'exécution C_j ; P_R correspond à la priorité de la tâche de reconfiguration ; enfin, $Un\ t.q.\ P_j \geq P_R$ correspond à l'ensemble des tâches non impactés par la reconfiguration telles que leur priorité est supérieure à la priorité de la tâche de reconfiguration. Nous rappelons ici que ce protocole de reconfiguration impose que la tâche de reconfiguration ait une priorité inférieure à la plus petite priorité des tâches impactées. Ceci explique pourquoi l'ensemble de ces tâches impactées peut retarder la mise en œuvre de la reconfiguration.

$$WCRT = \sum_{j \in Im} \lceil \frac{WCRT}{T_j} \rceil C_j + \sum_{j \in Un\ t.q.\ P_j \geq P_R} \lceil \frac{WCRT}{Tj} \rceil C_j \tag{7.1}$$

Dans cette équation, le premier terme correspond aux interférences des tâches impactées par la reconfiguration dynamique, alors que le second terme correspond aux interférences des tâches qui ne sont pas impactées par la reconfiguration et dont la priorité est supérieure à la priorité de la tâche effectuant la reconfiguration.

Configuration visant à privilégier la reconfiguration

Rappelons la configuration à laquelle nous faisons référence ici (voir sous-section 4.4.4) :

1. la tâche de reconfiguration à une priorité supérieure à l'ensemble des tâches impactées par la reconfiguration ;
2. l'instance d'automate de mode considérée est configurée avec une priorité plafond. Pour plus de simplicité, on suppose ici que la priorité plafond est égale à la priorité de la tâche de reconfiguration.

Déterminons le pire temps de reconfiguration associé à cette configuration. Les interférences possibles proviennent (i) des tâches impactées qui peuvent toutes avoir été déclenchées au moment de la réception de la requête de changement de mode, et (ii) des tâches non-impactées et dont la priorité est supérieure à la priorité de la tâche de reconfiguration. Chacune de ces possibilités est présente respectivement dans le terme à gauche et à droite de la somme retranscrite dans l'équation (7.2).

$$WCRT = \sum_{j \in Im} C_j + \sum_{j \in Un \ t.q. \ P_j \geq P_R} \lceil \frac{WCRT}{Tj} \rceil C_j \tag{7.2}$$

Nous nous apercevons ici que cette configuration réduit fortement le pire temps de reconfiguration en imposant que les tâches impactées ne s'exécutent, au pire, qu'une seule fois après réception de la requête de changement de mode : le terme de droite de l'équation (7.1) et de l'équation (7.2) sont identiques ; seul le terme de gauche change et correspond dans l'équation (7.1) à plusieurs exécutions des tâches impactées et dans l'équation (7.2) à une unique exécution de chacune de ces tâches.

Configurations visant à garantir l'intégrité des flux de données

Rappelons les configurations auxquelles nous faisons référence ici (voir sous-section 4.4.5). Dans un premier temps, nous considérons que :

1. la reconfiguration a lieu à l'hyper-période d'un ensemble de tâches impactées ;
2. l'instance d'automate de mode considérée n'est pas configurée avec une priorité plafond.

Dans un deuxième temps, nous considérons que l'automate de mode est configuré avec une priorité plafond.

Calculons dans chacun de ces cas le pire temps de reconfiguration. L'équation (7.3) permet de calculer le pire temps de reconfiguration dans le cas où le mécanisme de reconfiguration n'utilise pas de priorité plafond, alors que l'équation (7.4) correspond au pire temps de reconfiguration combinant la synchronisation et l'héritage de priorité.

$$
\begin{aligned}
WCRT &= Hyper(SG) + C_R \\
&+ \sum_{j \in Un \ st. \ P_j \geq P_R} \lceil \frac{WCRT - Hyper(SG)}{Tj} \rceil C_j \\
&+ \sum_{i \in Im \ t.q. \ T_i \notin SG} \ \sum_{j \in Un \ t.q. P_R > P_j \geq P_i} \lceil \frac{C_i}{T_j} \rceil C_j \\
&+ \sum_{j \in Im \ t.q. \ T_j \notin SG} C_j
\end{aligned} \tag{7.3}
$$

Dans cette équation, $Hyper(SG)$ correspond à l'hyper-période du groupe de synchronisation ; C_R correspond au pire temps d'exécution de la tâche de reconfiguration ; le terme suivant correspond aux interférences des tâches non impactées et dont la priorité est supérieure à la priorité de la tâche de reconfiguration ; le terme suivant correspond aux interférences des tâches non impactées et dont la priorité est comprise entre la priorité de la tâche de reconfiguration et la priorité des tâches impactées et non synchronisées ; le dernier terme correspond au temps d'exécution de l'ensemble des tâches impactées non synchronisées.

Considérons maintenant que l'automate de mode est configuré avec une priorité plafond :

$$
\begin{aligned}
WCRT &= Hyper(SG) + C_R + \\
&\sum_{j \in Un \ st. \ P_j \geq P_R} \lceil \frac{WCRT - Hyper(SG)}{Tj} \rceil C_j \\
&+ \sum_{j \in Im \ t.q. \ T_j \notin SG} C_j
\end{aligned} \tag{7.4}
$$

Dans ce cas, lorsqu'une tâche impactée n'appartenant pas au groupe de synchronisation bloque la tâche de reconfiguration, la tâche impactée hérite de la priorité plafond. L'ensemble des tâches non impactées dont la priorité est strictement inférieure à la priorité de la tâche de reconfiguration et supérieure à la priorité de la tâche impactée est nulle puisque la tâche impactée en question et la tâche de reconfiguration ont la même priorité : la priorité plafond. D'où la différence entre l'équation (7.3) et (7.4), qui induit une réduction du pire temps de reconfiguration dans le cas de l'équation 7.4.

Dans cette sous-section, nous avons établi que l'utilisation d'une politique de reconfiguration donné avait un impact important sur la valeur du pire temps de reconfiguration. Nous avons également proposé une méthode d'estimation du pire temps de reconfiguration associé à un changement de mode. Pour un passage d'un mode à un autre qui passe par plusieurs modes intermédiaires, il faut additionner les résultats correspondant à chaque changement de mode. Nous allons maintenant présenter les résultats que nous avons obtenus en utilisant des techniques de vérification formelle. L'utilisation de ces techniques permet de réduire le pessimisme du calcul en représentant plus finement le comportement du système.

7.3.2 Modélisation formelle des changements de modes

Contexte de l'étude

L'utilisateur de notre méthode dispose de moyens de vérifications formelles qui permettent de garantir un certain nombre de propriétés relatives au comportement temporel d'un système TR²E. En particulier, il peut vérifier :

- l'ordonnançabilité du système ;
- le temps maximal requis pour passer d'un mode de fonctionnement donné à un autre mode de fonctionnement ;
- le temps maximal pendant lequel deux sous-systèmes sont susceptibles de rester dans des modes de fonctionnement incohérents entre eux.

Dans la section 7.2, nous avons montré qu'il est possible de générer des automates UPPAAL correspondant aux automates de mode définis en COAL. Nous devons maintenant étendre ce résultat dans le but de permettre la vérification des propriétés temporelles que nous venons d'énumérer.

Le principal problème, lors de l'utilisation de techniques de vérification formelle est que le nombre d'état à parcourir pour vérifier une propriété dépasse rapidement les capacités de la machine. En supposant que l'on dispose d'une machine fournissant suffisamment de ressources pour traiter ce nombre d'état, encore faut-il s'assurer que la propriété à vérifier est décidable, c'est-à-dire que l'algorithme de vérification donnera toujours un résultat concernant cette propriété.

Intéressons nous à ce problème dans le cas des automates temporisés [Alur and Dill, 1994]. Concernant la propriété de l'ordonnançabilité, qui se ramène à un problème d'accessibilité d'un état, [Krčál and Yi, 2004] montre que cette propriété n'est pas décidable :

1. si les temps d'exécution des tâches sont des intervalles, et
2. si une tâche peut annoncer qu'elle a fini son exécution, et
3. si une tâche peut préempter une autre tâche.

Il se trouve que ces hypothèses représentent la sémantique d'exécution de la plupart des systèmes TR²E. Elles correspondent ainsi aux caractéristiques de la sémantique d'exécution de COAL.

La modélisation de la préemption a été traitée par le biais d'automates temporisés dit "stopwatch" [McManis and Varaiya, 1994]. Ce type de modélisation ne résout pas le problème de l'indécidabilité de l'ordonnançabilité. Il constitue cependant un pré-requis important des travaux sur lesquels nous allons nous appuyer pour modéliser formellement un système TR²E reconfigurable. En effet, des travaux récents ont proposé d'utiliser une approximation des automates "stopwatch" par le biais d'automates temporisés [Madl *et al.*, 2009].

Le principe de base de cette approximation consiste à modéliser le temps pendant lequel une tâche est préemptée par un état dédié dans lequel le temps correspondant est compté via un entier incrémenté. Il s'agit d'une approximation dans la mesure où le temps de préemption est compté par unité de temps (milliseconde, dixième de milliseconde, microseconde, etc...). Il s'agit donc d'une méthode de discrétisation du temps de préemption : une fois préemptée, la tâche qui s'exécute utilise la ressource d'exécution pendant un temps multiple (entier) de l'unité de temps considérée.

Au cours de nos travaux, nous avons prolongé l'étude menée par l'université de Vanderbilt dans le but (i) de modéliser des tâches sporadiques (le modèle d'exécution de CORBA utilise des tâches apériodiques), (ii) de modéliser le comportement des automates de modes et leurs

relations avec les autres tâches du système (préemption, utilisation des verrous), et (iii) de modéliser le comportement des implémentations de composants.

Modélisation formelle des activités et de l'ordonnanceur

Nous présentons ici les patrons de modélisation associés aux activités périodiques 7.4 et sporadiques 7.5. La figure 7.3 représente le patron de modélisation d'un timer qui active une tâche périodique donnée (transition *Task_activation !*) toutes les *Task.Period* millisecondes. Enfin, la figure 7.6 représente le modèle UPPAAL d'un ordonnanceur qui décide, chaque fois qu'une tâche est activée et/ou désactivée laquelle d'entre elles doit avoir accès à la ressource d'exécution.

Ces réprésentations sont très fortement inspirées des modèles décrits dans les travaux de l'université de Vanderbilt [Madl *et al.*, 2009]. Les principales différences sont :

- la possibilité de modéliser des tâches sporadiques (figure 7.5) ;
- la possibilité d'appeler le code exécuté par une tâche (figures 7.4 et 7.5 : transition *operation_call !*) ;
- la possibilité de modéliser l'acquisition d'un verrou (figure 7.6 : la transition *Preempt_Node ?* depuis un état *Sched_<Nom_Tache>* qui correspond au fait que la tâche est bloquée sur l'acquisition d'un verrou).

Activation périodique. La modélisation de l'activation des activités périodiques s'appuie sur un ensemble de modèles de *timer* (une par activité) qui déclenche périodiquement l'activation de la tâche correspondante. Le modèle UPPAAL d'un tel composant est représenté figure 7.3. L'état initial correspond à l'état initial du système d'exploitation : aucune tâche n'est activée mais elle vont toute l'être "simultanément" pour un premier tour d'élection. Le second état de cette automate correspond à l'attente de la prochaine échéance. Lorsque cette échéance est atteinte, une nouvelle activation de la tâche est déclenchée (transition avec synchronisation *Task_activation* et garde *t>=Task.Period && !Enable_Task[Task.Id]*).

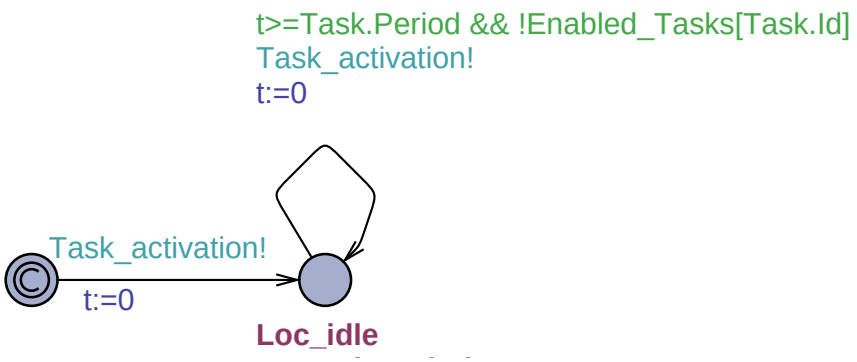

FIGURE 7.3 – Patron de modélisation formelle d'un *timer*

Tâche périodique. Le patron de modélisation d'une tâche périodique se compose de quatre états principaux :

- *Task_idle*, la tâche n'est pas active (entre la fin de sa précédente exécution et la prochaine période de cette tâche).

– *Task_wait*, la tâche est active mais n'est pas ordonnancée (attente de la ressource d'exécution pour cause de préemption, de blocage sur une ressource partagée)
– *Task_run*, la tâche est exécutée ;
– *Task_error*, atteint lorsque la tâche a dépassé son échéance (problème d'ordonnançabilité).

La transition *Task_idle* → *Task_wait* correspond à l'activation périodique de la tâche (transition *Task_activation !* sur la figure 7.3). La transition *Task_wait* → *Task_run* se décompose en deux transitions, une transition (*Task_start ?*) qui correspond à l'élection de la tâche courante par l'ordonnanceur, et une transition ((*operation_call !*) qui correspond à l'exécution du code fonctionnel de cette tâche. La transition *Task_run* → *Task_idle* est franchie lorsque l'exécution de l'opération correspondant au code fonctionnel de la tâche est terminée (voir figure 7.4, transisition *operation_call ?*). Dans ce cas, la tâche signale à l'ordonnanceur la fin de son exécution (transition *Task_start !*).

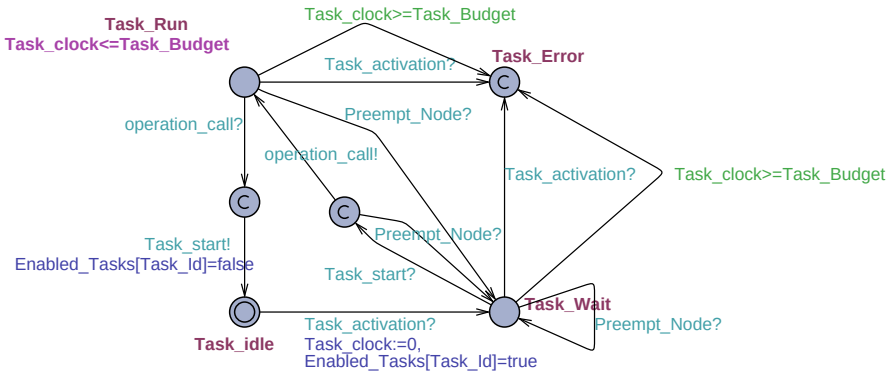

FIGURE 7.4 – Patron de modélisation formelle d'une tâche périodique

Tâche sporadique. Les principales différences entre le patron de modélisation d'une tâche périodique et celui d'une tâche sporadique vient du fait que la tâche sporadique est activée sur réception d'un événement (figure 7.5, transition *push_event ?*), et que la tâche attend la prochaine période avant d'être à nouveau activable (figure 7.5, transition *Task_clock>=Period*).

Ordonnanceur. Le patron de modélisation de l'ordonnanceur est représenté figure 7.6. Nous supposons ici que deux tâches s'exécutent sur le nœud correspondant à cet ordonnanceur : *Task* et *Other_Task*. Le modèle obtenu par transformation de modèle se compose :

– de deux états indépendants des tâches qui s'exécutent sur le nœud correspondant à cet ordonnancement : états *Sched_idle* et *Sched_select* ;
– et d'autant de paires d'états que de tâches s'exécutant sur ce nœud (voir figure 7.6, états *Sched_Task*, *Sched_Other_Task*).

Les transitions entre ces différents états représentent les possibilités suivantes :

– *Sched_idle* → *Sched_select* : cette transition doit être franchie si une des tâches est activée (initialisation puis atteinte de la période de cette tâche via la transition *Task_activate*) ; ou si le verrou est libéré (transition *Preempt_Node*).
– *Sched_select* → *Sched_Task* : cette transition peut être franchie si la tâche *Task* a la plus forte des priorités des tâches activées (précondition *Enable(Task_Id)==true*).

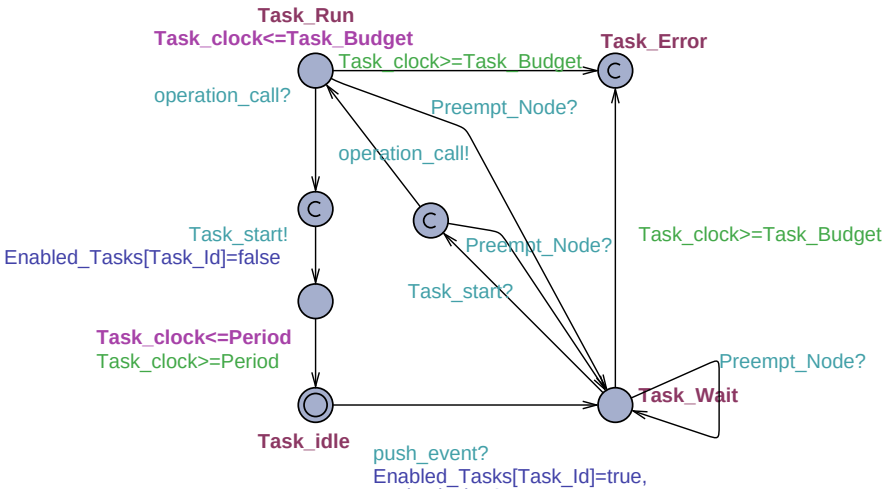

FIGURE 7.5 – Patron de modélisation formelle d'une tâche sporadique

Une deuxième étape consiste alors à commencer l'exécution de cette tâche (transition *Task_Start !*).

- *Sched_Task* → *Sched_select* : cette transition peut être franchie si une autre tâche que la tâche courante est activée, ce qui provoque une nouvelle élection de la tâche à exécuter (transition avec synchronisation *Other_Task_activate ?*), ou si la tâche courante finit son exécution (transition avec synchronisation *Task_Start !*), ou si la tâche courante est bloquée sur l'acquisition d'un verrou (transition avec synchronisation *Preempt_Node ?*).
- *Sched_select* → *Sched_idle* : aucune des tâches n'est active (transition avec précondition *!Enable(Task_Id) && !Enable(Other_Task_Id)*).

Notons ici que le canal de synchronisation UPPAAL *Task_start* est à la fois utilisé pour signaler le début et la fin de la tâche *Task*. Ceci afin de limiter le nombre de canaux utilisés.

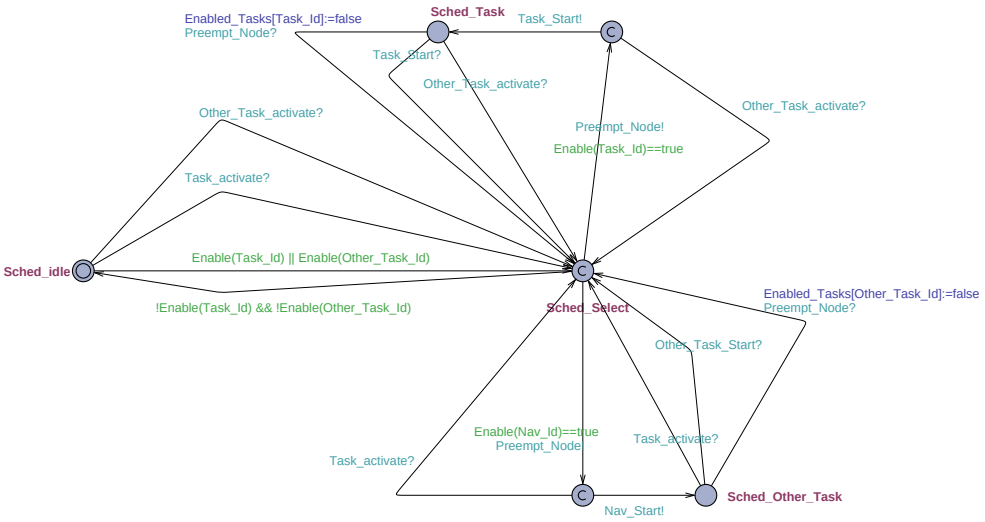

FIGURE 7.6 – Modélisation formelle de l'ordonanceur d'un nœud

Modélisation formelle des automates de modes

Nous allons reprendre l'exemple du changement de mode défini précédemment (voir section 7.2.2) pour illustrer les règles de transformation d'un automate de mode COAL vers un automate UPPAAL dans le cadre de la spécification logicielle.

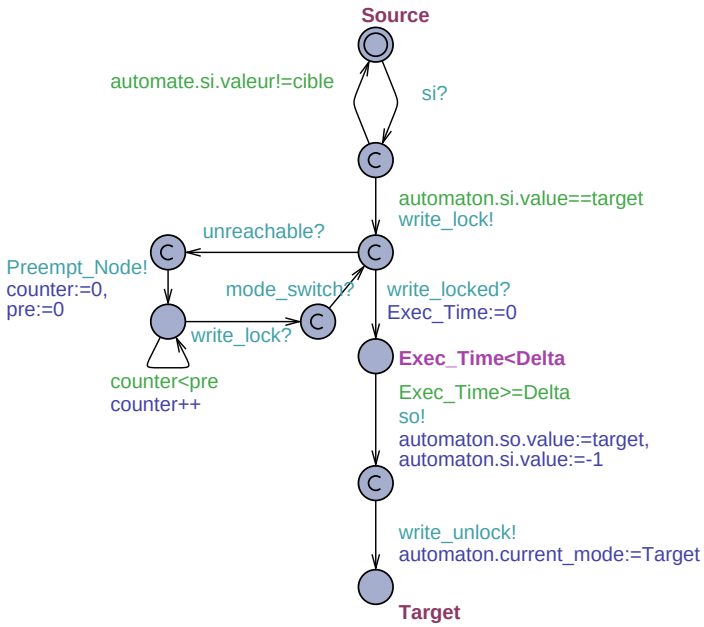

FIGURE 7.7 – Patron de modélisation logicielle d'un changement de mode

La figure 7.7 illustre l'automate de mode associé à cette même spécification, mais pour vérifier le comportement temporel de l'application. Nous pouvons noter trois différences majeures par rapport à l'automate UPPAAL représenté figure 7.1 :

1. le changement de mode n'est plus déclenché directement par la réception de la commande de changement de mode. La réception de cette commande est gérée par une tâche sporadique qui une fois ordonnancée va déclencher le changement de mode (transition *mode_switch ?*) ;
2. un ensemble d'états représente les mécanismes d'acquisition du verrou en écriture : pour que le changement de mode ait lieu, il faut que le verrou correspondant à l'instance d'automate soit libre. Si tel est le cas, la transition *Write_locked ?* est franchi. Dans le cas contraire, la transition *unreachable* est franchie et la tâche exécutant le code de l'automate est désactivée jusqu'à ce que le verrou soit à nouveau libre ;
3. lorsque le mode cible est atteint, la valeur du mode courant de l'automate est mise à jour (transition avec mise à jour *automaton.current_mode :=target*).

Modélisation formelle des vérrous lecteur/écrivain

La figure 7.8 représente le patron de modélisation formelle d'un verrou de type lecteur/écrivain n'utilisant pas la notion de priorité plafond. Ce patron se compose de trois états principaux :

– l'état *free*, qui correspond au fait que le verrou peut-être pris en lecture ou en écriture ;
– l'état *readLocked*, qui correspond au fait que le verrou a été acquis en lecture ;
– l'état *writeLocked*, qui correspond au fait que le verrou a été acquis en écriture.

Les transitions entre ces différents états représentent les possibilités suivantes :

– *free* → *readLocked* : le verrou est libre, une tâche l'acquiert en lecture (transition *read_lock ?*) ;
– *free* → *writeLocked* : le verrou est libre, une tâche l'acquiert en écriture (transition *write_lock ?*) ;
– *readLocked* → *writeLocked* : le verrou est libéré (précondition *read_count==1* et transition *read_unlock ?*), mais certaines tâches sont en attente pour prendre le verrou en écriture (précondition *writersWaiting>0*).
– *readLocked* → *free* : le verrou est libéré (précondition *read_count==1* et transition *read_unlock ?*), et aucune tâche n'est en attente pour prendre le verrou en écriture (précondition *writersWaiting==0*) ;
– *writeLocked* → *free* : le verrou est libéré (transition *write_unlock ?*), et aucune tâche n'est en attente pour prendre le verrou en lecture (précondition *readersWaiting==0*) ;
– *writeLocked* → *readLocked* : le verrou est libéré (transition *write_unlock ?*) mais certaines tâches sont en attente d'écriture (précondition *writersWaiting>0*) ;
– *writeLocked* → *writeLocked* : le verrou est libéré (transition *write_unlock ?*) mais un certain nombre de tâches sont en attente de lecture (précondition *readersWaiting>0*) et aucune tâche n'est en attente d'écriture (précondition *writersWaiting==0*) ;

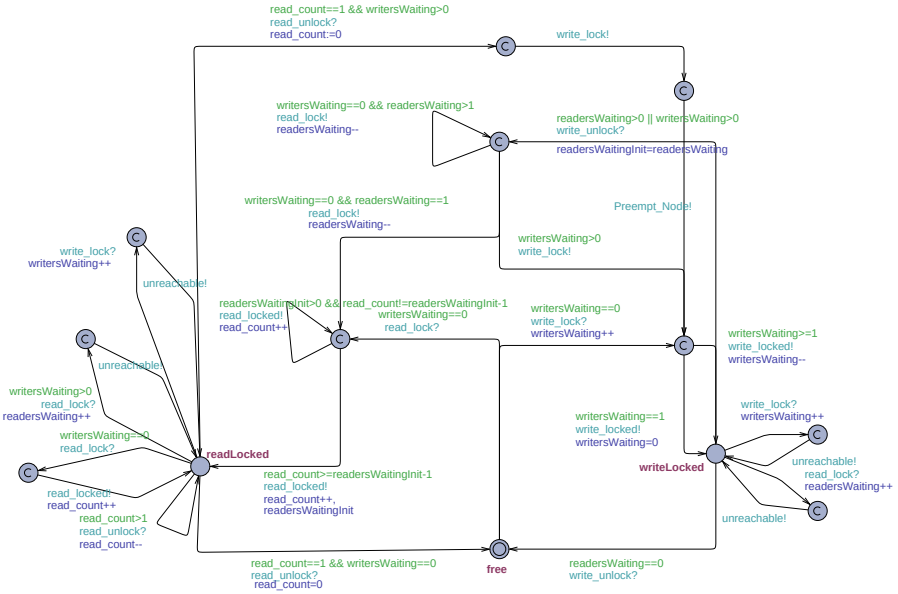

FIGURE 7.8 – Patron de modélisation formelle d'un verrou lecteur/écrivain

Modélisation formelle du comportement de l'application

La modélisation formelle des composants logiciels est générée à partir d'une description du comportement du composant. Cette modélisation décrit les temps de calcul, les boucle, les branchements conditionnels ainsi que les communications de ce composant avec les composants auxquels il est connecté. La compilation de cette information permet de représenter le

comportement du code fonctionnel exécuté par une tâche, en terme de temps d'exécution et d'envoies de messages.

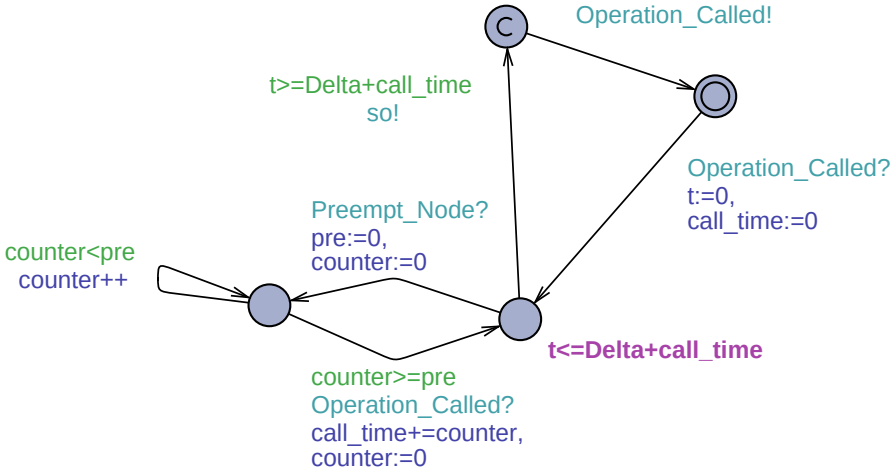

FIGURE 7.9 – Patron de modélisation du code fonctionnel d'une tâche

La figure 7.9 représente le type de modélisation obtenu par ce procédé. Sur cette figure, la transition *operation_call !* correspond à l'exécution du point d'entrée de la tâche ordonnancée. L'état atteint correspond à la modélisation d'un ensemble d'opération dont le pire temps d'exécution de *Delta*. A partir de cet état, la tâche ordonnancée peut être préemptée. La transition avec synchronisation *Preempt_Node ?* représente cette préemption, alors que la variable *counter* et l'horloge *pre* permettent d'estimer le temps correspondant à cette préemption (discrétisation du temps de préemption). Ensuite, une transition vers l'état intermédiaire (transition avec synchronisation *so !* permet de représenter l'envoie d'un message via le port *so* après le calcul de Delta unités de temps. Enfin, une transition *Operaion_Called !* permet de signaler la fin de l'exécution du code fonctionnel de la tâche.

Si la tâche considérée est impactée par la reconfiguration, nous insserérons la première partie (jusqu'à la transition avec synchronisation *read_locked ?* incluse) du patron représenté figure 7.10 entre l'appel à la fonction (transition avec synchronisation *Operation_Called ?* sur la figure 7.9) et le début du code fonctionnel. Nous insserernons la seconde partie (transition avec synchronisation *read_unlock !* plus état final) juste avant la fin de l'exécution du code fonctionnel (transition avec synchronisation *Operation_Called !* sur la figure 7.9).

Sur cette figure, le mécanisme d'accès en lecture est très similaire au patron utilisé pour l'accès en écriture, les synchronisations *write_...* étant remplacés par des synchronisations *read_...*.

Voyons maintenant comment modéliser l'impact d'un changement de mode sur le comportement de l'application.

Modélisation formelle de l'impact d'un changement de mode

Nous montrons ici la modélisation d'une connexion de ports d'évènements qui dépend de la valeur du mode courant du système. Nous considérons le même patron de connexion que

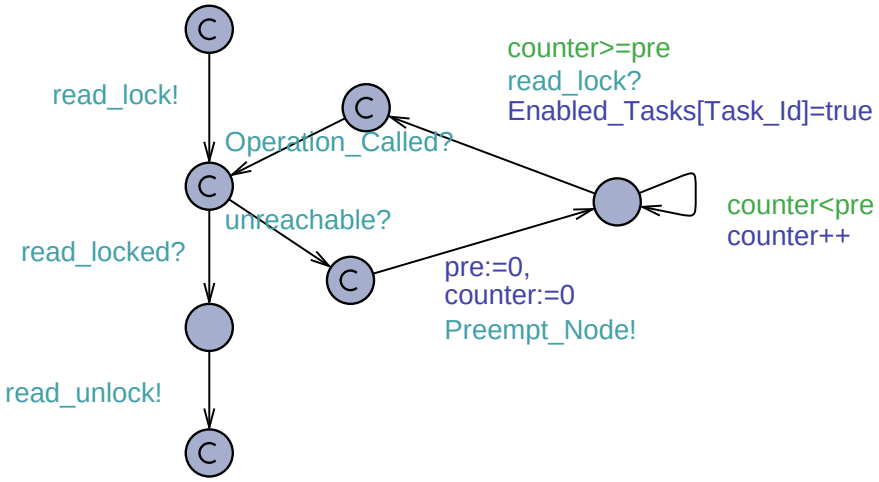

FIGURE 7.10 – Patron de modélisation de l'accès en lecture au verrou lecteur/écrivain

celui illustré figure 7.2, à la différence près que l'émetteur et le destinataire sont des composants fonctionnels, et que le destinataire est un port *push_out_target* dans le mode *Target* et un port *push_out_source* dans le mode *Source*. La figure 7.11 illustre le comportement que nous venons de décrire.

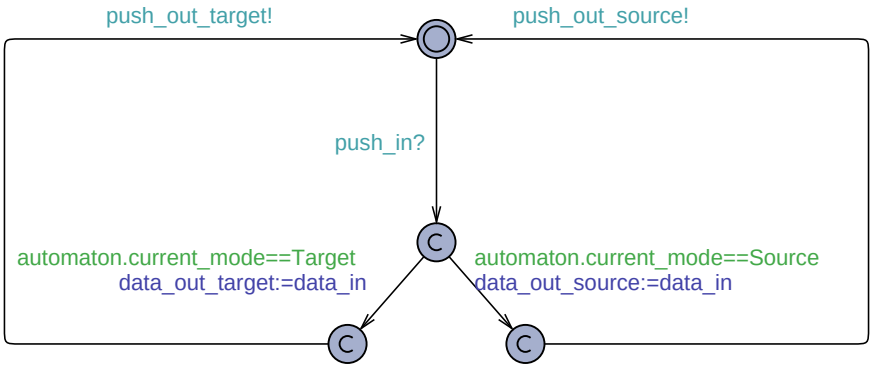

FIGURE 7.11 – Patron de modélisation d'une connexion dont le destinataire dépend du mode courant

Sur cette figure, nous constatons que l'envoie du message via le port *push_out_target* (transition avec synchronisation *push_out_target !*) n'est possible que si le mode courant est le mode Source (transition avec garde *automaton.current_mode==Source*.

Voyons maintenant les propriétés que ces modèles nous ont permis de vérifier.

Types de vérification possibles

Pour effectuer notre modélisation, nous avons dû utiliser la notion de priorité sur les canaux de synchronisation UPPAAL. Ceci afin de garantir que l'élection d'une tâche à exécuter ne se

fera que lorsque toutes les tâches activables auront été activées. Nous ne pouvons donc pas faire d'analyse d'interblocage sur ce modèle.

En revanche, nous pouvons analyser :

1. l'ordonnançabilité du système ;
2. l'accessibilité d'un mode donné ;
3. le fait que deux automates de mode ne restent pas dans des états incohérents pendant un temps supérieur à un certain délai.

Nous illustrerons au chapitre 8 l'analyse de ces différentes propriétés sur le cas d'étude du système de pilotage.

7.4 Synthèse

Dans ce chapitre, nous avons présenté trois possibilités d'analyse des systèmes TR²E critiques et adaptatifs. La première est une méthode basé sur le *"model-cheking"* qui permet d'analyser formellement la logique des changements de mode. La seconde est une méthode analytique qui permet d'estimer le pire temps de reconfiguratin correspondant à une unique transition de mode. Enfin, notre troisième méthode s'appuie sur des techniques de *"model-cheking"* pour analyser le comportement des application logicielles TR²E critiques.

Pour chacune des méthodes qui s'appuient sur le *"model-checnking"*, nous avons proposé des règles de transformation qui permettent d'automatiser la production d'un modèle formel correspondant à une description en COAL. Ces modèles permettent ainsi de vérifier l'ensemble des propriétés que nous avions présenté au chapitre 3.

Ces différentes méthodes d'analyse sont complémentaires. L'analyse de la spécification système permet d'évaluer le comportement dynamique du système au plus tôt dans le processus de conception, c'est-à-dire avant même que l'architecture logicielle n'ait été spécifiée. La méthode d'analyse numérique du pire temps d'exécution permet quant à elle d'obtenir une estimation rapide pire temps de reconfiguration associé à un changement de mode. Cette méthode ne nécessite pas de modéliser le comportement des chaînes fonctionnelles, ni de l'environnement d'exécution du système. Enfin, la dernière méthode que nous avons présentée permet de raffiner les résultats obtenus en terme de pire temps de reconfiguration, et permet également d'analyser beaucoup plus finement le comportement temporel du système.

Pour valider les différentes règles de transformation que nous venons de présenter, nous avons modélisé formellement l'exemple du système de pilotage. Les résultats de cette expérimentation, ainsi que l'ensemble des résultats expérimentaux que nous avons obtenu sont présenté au cours du chapitre suivant.

Cinquième partie

Validation

Chapitre 8

Expérimentations et Résultats

Il faut admettre tout comme possible, mais il faut tout vérifier.
Claude Bernard

SOMMAIRE

8.1 Introduction

Nous présentons dans ce chapitre l'évaluation de l'ensemble de nos contributions. Pour commencer, nous souhaitons mettre en avant le fait que ces contributions sont suffisamment

originales pour qu'il soit parfois difficile de mener une véritable étude comparative : nos contributions répondent à une problématique que personne n'avait intégralement résolu auparavant. Lorsque les caractéristiques que nous mettrons en avant seront communes à d'autres approches, nous ferons cette étude comparative.

Pour mener à bien cette évaluation, nous avons :

- réalisé un prototype mettant en œuvre l'approche théorique que nous avons présenté au chapitre 4.
- étudié les caractéristiques du code généré, en termes de quantité et de compacité ;
- testé les performances associées à ce code, en termes de latence, de gigue et de débit puis en termes de temps de reconfiguration ;
- évalué notre méthode de vérification formelle, et en particulier l'impact de la discrétisation sur la taille de l'espace d'état à explorer.

Plate-forme de test. Tous les résultats de tests de performance que nous allons présenter dans ce chapitre ont été obtenus sur une carte utilisée dans de nombreux projets de la société *Thales*, et dont les caractéristiques sont les suivantes :

- microprocesseur : PowerPC MPC5200B ;
- fréquence : 384 MHz ;
- mémoire : 256 MB SDRAM ;
- interface réseaux : ethernet 100 Mb/s.

Un Linux temps-réel commercial (ELinOS) a été installé sur cette carte.

Organisation du chapitre. Dans la suite de ce chapitre, nous commençons par rappeler l'ensemble des contributions que nous avons présenté au cours des chapitres précédents (section 8.2). Ce rappel sera l'occasion de résumer les résultats théoriques de notre travail. Nous décrirons ensuite notre contribution logicielle principale, qui constitue un premier prototype mettant en œuvre nos résultats théoriques (section 8.3). Nous nous penchons ensuite sur une série d'études quantitatives, concernant :

- les caractéristiques du code généré (section 8.4) ;
- les performances de ce code (section 8.5) ;
- la taille de l'espace d'état exploré pour la vérification formelle de l'architecture logicielle (section 8.7).

8.2 Rappel des contributions

8.2.1 Méthode de conception

Tout d'abord, nous avons proposé une méthode de conception qui améliore la productivité des équipes de développement logiciel en termes de qualité et de productivité. Cette méthode s'appuie sur deux principes :

- l'analyse doit être faite au plus tôt dans le processus de développement logiciel ;
- la production et l'analyse du logiciel s'appuient sur des techniques de génération de code et de modèles formels pour assurer que ces deux processus (analyse et exécution) utilisent une sémantique d'exécution commune.

A partir de ces deux principes, nous avons prouvé la faisabilité de notre approche :

– nous avons montré que les systèmes TR²E critiques et adaptatifs peuvent être modélisés par le biais d'un langage de description d'architecture à base de composants génériques (voir chapitre 5) ;
– nous avons ensuite montré comment automatiser la production du code technique, y compris le code d'adaptation, de ces systèmes (voir chapitre 6). Ceci améliore la productivité des équipes de développement logiciel ;
– enfin, nous avons démontré que la sémantique d'exécution ainsi implantée est analysable, ce qui améliore la qualité des développements logiciels en prouvant formellement que certaines propriétés sont vérifiées (voir chapitre 7).

Ainsi, notre solution améliore à la fois la qualité et la productivité des développements logiciels dans le domaine des systèmes TR²E critiques et adaptatifs. De plus, cette méthodologie répond à l'ensemble des problèmes et contraintes que nous avions identifiés au chapitre 3.

8.2.2 Langage de modélisation

Le langage de description d'architecture que nous avons défini (COAL), facilite la description conjointe de l'architecture à base de composants (type Lightweight CCM) d'un système TR²E. Pour ce faire, un modèle COAL :

– utilise les concepts propres à la modélisation système (interfaces, sous-systèmes, modes et changements de mode) ;
– décompose les fonctionnalités de chaque système en sous-ensembles de composants logiciels génériques avec des niveaux de granularité plus ou moins fins ;
– spécifie les propriétés non fonctionnelles de l'architecture (période des activités, accès à des données partagées, etc...) ;
– représente les configurations logicielles associées à chaque mode du système.

Pour chacune de ces caractéristiques, la sémantique d'exécution de COAL est inspirée des travaux existants autour de la définition des langages de description d'architecture D&C, et AADL. D&C a permis d'intégrer la notion de composant logiciel dans la description d'architecture, alors que AADL a été la source d'inspiration principale pour définir la sémantique d'exécution de COAL.

8.2.3 Analyse système, analyse et production logicielle

Analyse système et logicielle. Comme nous l'avons mentionné précédemment, un des principes de notre méthodologie consiste à analyser au plus tôt dans le processus de développement logiciel les propriétés du système TR²E réalisé. Nous avons donc montré que la sémantique d'exécution de COAL est analysable : nous avons utilisé des automates temporisés communicants, et plus particulièrement l'outil de vérification formelle UPPAAL qui permet d'analyser ce type de modèle. Nous nous sommes appuyé sur des résultats récents, qui automatisent l'analyse de l'ordonnançabilité d'applications logicielles dont la sémantique d'exécution est similaire à celle définie par COAL. Nous avons étendu ces travaux pour prendre en compte la sémantique d'exécution des protocoles de reconfiguration pseudo-dynamique. Ces travaux ont donné lieu à la définition d'un ensemble de règles de transformation qui automatisent la production d'un modèle UPPAAL équivalent à un modèle COAL. Ces règles de transformation ont été définies, d'une part pour analyser une spécification système, mais également pour analyser l'architecture logicielle pseudo-dynamiquement reconfigurable.

Production logicielle. L'automatisation de la production d'une application TR²E critique et adaptative a été une étape cruciale dans la réalisation de nos contributions. Elle a tout d'abord permis de définir précisément la sémantique d'exécution de COAL, ainsi que des différents protocoles de changement de mode associés. Elle a ensuite permis de montrer que le code correspondant pouvait être généré automatiquement. Enfin, cette étape a abouti à la production d'un *framework* à composant, MyCCM-HI, qui démontre la faisabilité de notre approche.

MyCCM-HI a ainsi été utilisé au cours d'un stage de Master pour réaliser un prototype d'UGV fonctionnant en mode automatique et manuel. Ce stage a donc permis de démontrer la maturité logicielle de MyCCM-HI, puisque ce *framework* a permis à un stagiaire de réaliser le prototype d'un robot mobile en moins de 6 mois de travail et sans avoir de connaissances approfondies concernant le contexte technologique que nous avons présenté.

Enfin, le code généré par MyCCM-HI respecte les limitations qu'impose la réalisation des systèmes critiques (absence d'allocation dynamique de mémoire, respect des règles du profile Ravenscar).

Utilisation de standards Le *framework* que nous avons réalisé utilise principalement deux standards : Lightweight CCM pour ce qui concerne la décomposition fonctionnelle d'une application en ensembles de composants logiciels, et AADL 1.0 pour ce qui concerne la description d'architecture logicielle. Certaines caractéristiques de l'architecture n'ont pu être modélisés grâce à AADL 1.0 du fait de limitations de cette version du langage. La nouvelle version d'AADL 2.0 permet de résoudre ces problèmes. L'utilisation du standard AADL permet de bénéficier des travaux réalisés autour ce langage, d'une part en termes de vérification formelle, et d'autre part en termes de génération de code technique.

Afin d'évaluer la faisabilité de ces contributions, nous avons réalisé une chaîne d'outils qui automatise la réalisation de chacune des étapes de l'approche que nous avons présenté au chapitre 4. Nous présentons en détail le prototype associé dans la sous-section suivante.

8.3 Intégration des contributions

8.3.1 Outillage du processus de développement

Au chapitre 4, nous avons présenté une approche qui se décompose en trois étapes (voir figures 4.2 et 4.3) :

1. une étape d'analyse des propriétés du systèmes ;
2. une étape d'analyse des propriétés des applications logicielles ;
3. une étape de production automatique et d'intégration du code des applications logicielles.

Nous avons réalisé un prototype de mise en œuvre de cette approche, appélé MyCCM-HI. Nous avons présenté au cours du chapitre 5 le langage de spécification sur lequel s'appuie ce prototype. Nous avons ensuite décrit (chapitre 6) la structure du générateur de code qui permet d'interpréter et d'utiliser ce langage. Enfin, nous nous sommes intéressé à l'analysabilité de la sémantique d'exécution définie par COAL (chapitre 7).

La figure 8.1 présente les différents prototypes que nous avons réalisé pour mettre en œuvre cette approche. Un premier prototype (en haut à gauche de la figure) trasforme une spécification système exprimée avec COAL en un ensemble d'automates UPPAAL analysables.

Le second prototype (au centre à droite de la figure) transforme la description de l'architecture logicielle, exprimée en COAL, en un nouvel ensemble d'automates UPPAAL dédiées aux analyses temporelles. Enfin, un troisième prototype (en bas et au centre de la figure) exploite cette même spécification logicielle pour générer le code technique et le code de reconfiguration pseudo-dynamique des applications.

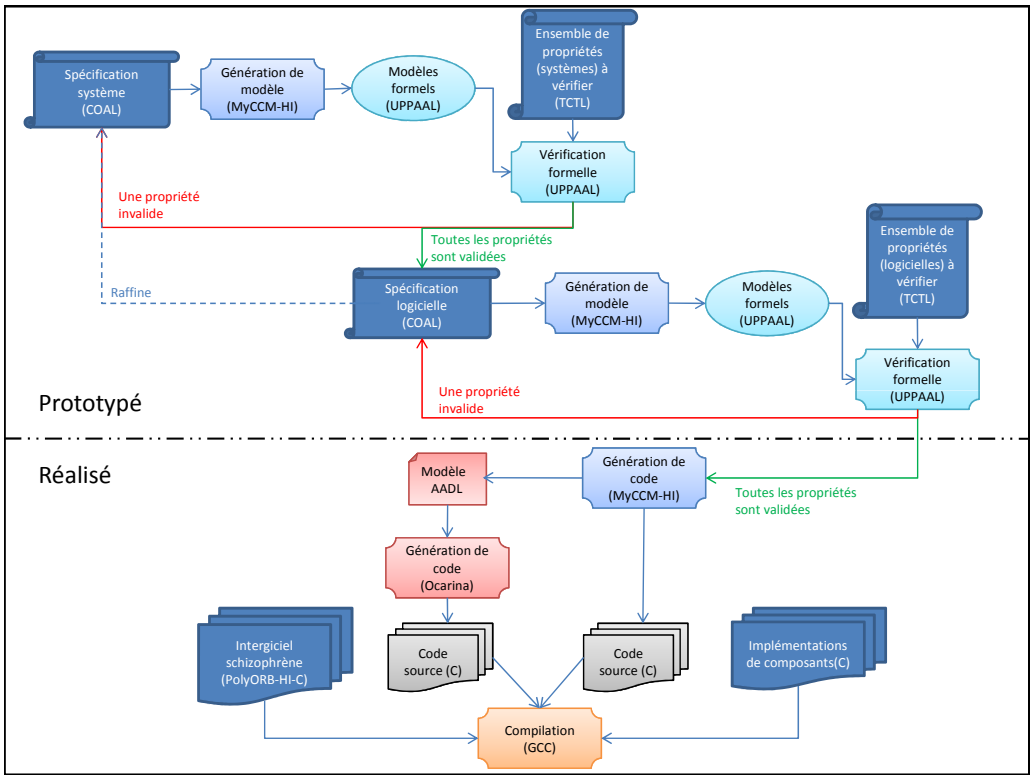

FIGURE 8.1 – Réalisation de l'approche théorique

Ces différents prototypes ont des niveaux de maturité très hétérogènes. Ceci s'explique principalement par la quantité importante de travail que représente la réalisation de chacun d'entre eux, mais également par la nécessité de valider en profondeur les prototypes dont les autres dépendent. En termes de dépendance, il est évident que la réalisation des prototypes dédiés à l'analyse dépend fortement de la réalisation des prototypes dédiés à la production logicielle. En effet, il faut fixer la sémantique d'exécution du langage COAL, et s'assurer que cette sémantique répond bien aux besoins des utilisateurs de notre *framework* avant de chercher à l'analyser formellement. Le prototype de génération de code est par conséquent d'une maturité logicielle nettement supérieure à celle du générateur de modèle formel. En effet, le prototype de génération de code (partie inférieure de la figure 8.1) a été utilisé par un étudiant de Master pour réaliser un prototype du système de pilotage d'un robot mobile. D'un autre coté, les prototypes de génération de modèles formels (partie supérieure de la figure 8.1) ne sont en fait constitués que des règles de transformation de modèle que nous avons présentées au chapitre 7. Cependant, une partie importante des générateurs de code que nous avons produit pourra être réutilisée pour implanter les générateurs de modèles UPPAAL. La partie frontale et centrale des générateurs (voir chapitre 6) seront intégralement réutilisables. Certains AST dédiés à la génération de code pourront également être réutilisés. Malgré cela, l'intégration

Type de code	*Nombre de lignes de code*
Générées par ANTLR (IDL3)	12 407
Générées par ANTLR (COAL)	13 764
Générées par EMF (MyCCM)	12 486
Générées par EMF (MyCCM-HI)	70 233
Ecrites à la main (MyCCM)	1 852
Ecrites à la main (MyCCM-HI)	20 662
TOTAL	**131 404**

TABLE 8.1 – Nombre de lignes de codes nécessaires à la réalisation des générateurs de MyCCM-HI

complète de ces nouveaux générateurs constitue un travail important. Ceci est illustré dans la suite de cette section, dédiée à l'étude du nombre de lignes de code nécessaire à la réalisation du générateur de code de MyCCM-HI.

8.3.2 Coût de réalisation du générateur de code

Nous avons présenté en détail la structure du générateur de code de MyCCM-HI au chapitre 6 (voir figure 6.1). Cette présentation a déjà permis de montrer la complexité d'un tel outil. Nous détaillons ici le nombre de lignes de code Java que nous utilisons pour réaliser chacune des étapes de génération. Ceci illustre globalement l'effort (ou le coût) dépensé dans cette réalisation.

Commençons par l'analyseur syntaxique du langage COAL. Sa réalisation a nécessité la création de 2940 lignes de grammaire ANTLR, à partir desquelles 13764 lignes de code Java ont été produites. MyCCM-HI réutilise les constructions du framework MyCCM pour l'analyse syntaxique du langage IDL3, ce qui représente 12407 lignes de code générées à partir de 2425 lignes de spécification ANTLR.

MyCCM-HI utilise également 82719 lignes de code Java générées à partie des métamodèles EMF (AST, modèles intermédiaires, etc...). Parmis ces lignes de code, 12486 correspondent au code généré à partir du métamodèle correspondant à l'AST du langage Idl3 (donc ces lignes de code sont réutilisées du *framework* MyCCM), alors que 70233 ont été générées à partir de métamodèles que nous avons produits pour MyCCM-HI. Ces différentes lignes correspondent au code généré pour représenter les différents AST et modèles intermédiaires représentés sur la figure 6.1.

Enfin, MyCCM-HI se compose de 22514 lignes de code écrites à la main. 1852 ont été défini pour MyCCM et réutilisées pour le développement MyCCM-HI. 20662 lignes de code ont donc été écrites à la main pour MyCCM-HI.

Le tableau 8.1 résume ces différents chiffres. Au total, le générateur de code de MyCCM-HI utilise 131 404 lignes de code Java, dont 104 659 ont spécifiquement été développées pour cette version du framework.

A ces lignes de code, il faut également ajouter que MyCCM-HI s'appuie sur Ocarina, qui correspond à une partie importante de la génération de code. Enfin, l'utilisation du langage *StringTemplate* n'est pas non plus comptabilisé ici. Les spécifications StringTemplate correspondent à une quantité importante de code, puisqu'ils contiennent l'ensemble des patrons de code C utilisés pour la génération.

Présentons maintenant les résultats que nous avons obtenus en utilisant le *framework* MyCCM-HI pour produire le système de pilotage de l'UGV.

8.4 Étude du code généré

8.4.1 Quantité relative de code généré

Intéressons nous ici à la quantité relative de code générée par les différents outils de notre chaine de compilation. Cette évaluation sera l'occasion de mesurer le surcoût lié à l'utilisation de composants logiciels de type Lightweight CCM ;

Nous nous intéressons ici au nombre de lignes de code généré pour mettre en œuvre le système de pilotage multi-modal. Le diagramme de la figure 8.2 représente le pourcentage de code généré par MyCCM-HI (72%), le pourcentage de code généré par Ocarina (12%), le pourcentage de code utilisé parmi le code PolyORB-HI-C (6%), le pourcentage de code fonctionnel (4%), et le pourcentage de code correspondant à la *"runtime"* de MyCCM-HI (2%).

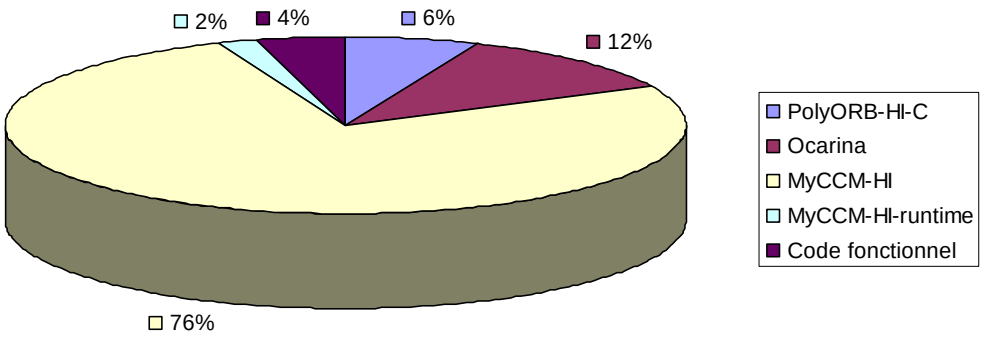

FIGURE 8.2 – Pourcentage de code généré/écrit à la main

Parmi le code généré par MyCCM-HI, il nous a semblé pertinent d'évaluer la quantité de code correspondant à la génération des structures de données correspondant aux composants, et la quantité de code spécifique au déploiement, à la gestion des modes, des données partagées, etc... Le diagramme de la figure 8.3 représente le pourcentage de code généré correspondant aux structures de données des composant (58%), à la gestion des modes de fonctionnement (24%), à l'initialisation des composants (17%), et enfin à la gestion des données partagées (1%).

En conclusion, il est intéressant de noter ici que sur cet exemple très simplifié en terme de gestion des modes de fonctionnement, 18,24% du code de l'application correspond au code de gestion des modes de fonctionnement. Ceci signifie que notre approche permet de générer une quantité de code importante qui correspond à la gestion des modes de fonctionnement. La réalisation, la maintenance, ainsi que les possibilités d'analyse correspondant à ce comportement multi-modal sont ainsi largement facilités par notre approche.

De plus, la mis en œuvre de ces modes s'appuie sur les fonctionnalités de l'intergiciel et interagit directement avec le code technique de l'application. Il s'agit donc d'un code difficile à produire, et source fréquente d'erreurs de conception. Enfin, notre approche facilite la mise en œuvre de la séparation des préoccupations : les besoins fonctionnels liés à l'utilisation des

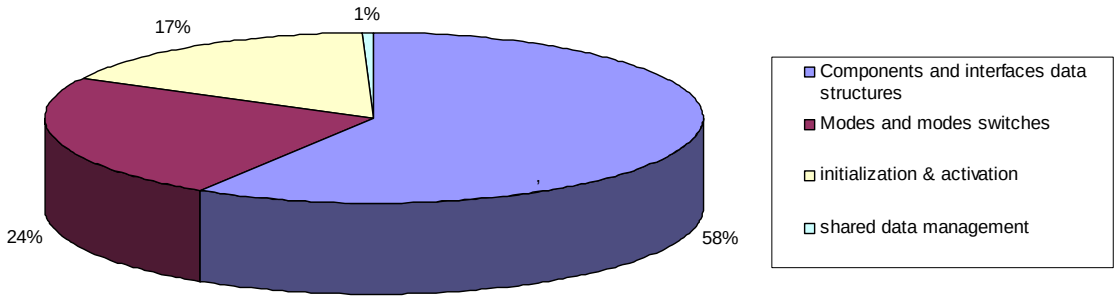

FIGURE 8.3 – Répartition du code généré par MyCCM-HI

modes de fonctionnement sont exprimés dans les assemblages de composants, alors que le code technique correspondant (code mettant en œuvre les politiques de synchronisation) est automatiquement produit par le générateur de code du *framework* MyCCM-HI. Notre approche accélère ainsi les developpements logiciels des systèmes TR2E critiques et adaptatifs.

Il est évident que les pourcentages de code que nous avons présentés ici ne reflètent pas tout à fait la réalité, dans la mesure où le résultat sera vraisemblablement différent avec un système réel : dans ce cas, la proportion de code fonctionnel, ainsi que la proportion de code correspondant à la gestion des modes devrait augmenter significativement. En revanche, la quantité de code technique restera identique.

Étudions maintenant les caractéristiques du code généré par le *framework* MyCCM-HI en terme d'empreinte mémoire.

8.4.2 Embarquabilité

Pour la réalisation de systèmes répartis, l'utilisation de MyCCM-HI permet un gain d'empreinte mémoire d'un **facteur 500** environ par rapport à l'empreinte mémoire obtenu avec MyCCM. Ceci s'explique facilement :

1. MyCCM utilise un intergiciel CORBA, dont le code généré ne prend pas en considération la spécification de l'architecure logicielle. Ce code est donc sous optimal [Zalila, 2008]. L'utilisation d'AADL et d'Ocarina sont donc pour beaucoup dans cette optimisation ;
2. l'utilisation d'une machine de déploiement indépendante, qui crée les composants en allouant dynamiquement la mémoire nécessaire. L'implantation de ce processus de déploiement consomme la mémoire disponible de façon significative.

A titre d'exemple, nous donnons ici l'empreinte mémoire associée à l'exemple du système de pilotage que nous avons utilisé tout au long de ce mémoire. Ces chiffres correspondent aux résultats obtenus en réalisant ce système grâce à notre démarche de génération de code. En réalisant la compilation statique de l'ensemble de l'application, nous obtenons un fichier exécutable dont l'exécution requiert au minimum 641 Ko. L'espace mémoire utilisé à l'exécution est de 860 Ko.

Ce résultat montre l'utilisabilité de notre approche dans un contexte fortement contraint en termes d'embarquabilité.

8.4.3 Discussion

Nous avons également évalué notre facteur de génération en comparant la quantité de code généré au nombre de lignes de modèles COAL et IDL3 fournis. Pour un total de 836 lignes de COAL + IDL3, nous obtenons un total de 20248 lignes de C générées, soit un facteur supérieur à 24. Une étude similaire nous donne le résultat suivant concernant le facteur de génération propre à la spécification des changements de mode : 160 ligne de COAL + IDL3 pour 4196 lignes de code générées, soit un facteur supérieur à 26. Enfin, concernant la compilation des structures de données correspondant aux composants, nous avons 346 lignes d'IDL3 pour 10367 lignes de code générées ; soit un facteur de génération supérieur à 29.

Ce facteur de génération donne un ordre de grandeur du ratio de code généré par ligne de spécification fournie. Il n'est évidemment pas possible d'affirmer qu'un ratio élevé est meilleur qu'un ratio faible ou inversement. En revanche, cette étude nous montre l'intérêt majeur d'une approche de génération : faciliter la réalisation et la maintenance d'une application logicielle. En effet, notre approche MDE permet d'obtenir en moyenne 24 lignes de code générées par ligne de modèle fournie.

Pourquoi un facteur trop élevé n'est-il pas souhaitable ? Si le facteur est trop élevé, cela peut signifier par exemple que le code produit est sous optimal (une quantité inférieure de code aurait rempli le même type de fonctionnalités), ou que le processus de génération est trop complexe (une modification du modèle entraîne une modification très significative dans le logiciel produit).

Or, les résultats obtenus avec MyCCM-HI montrent un bénéfice très important en termes d'embarquabilité par rapport aux résultats fournis par MyCCM. Ainsi, en créant MyCCM-HI, nous avons augmenté fortement le facteur de génération tout en améliorant la compacité (embarquabilité) du code généré. **MyCCM-HI produit donc un code fortement optimisé.**

Enfin, le code généré par MyCCM-HI **respecte l'ensemble des contraintes liées à la réalisation des systèmes critiques** (voir chapitre 3, section 3.4).

L'ensemble des résultats que nous avons présenté jusque là prouve l'utilisabilité du générateur de code de MyCCM-HI dans le contexte des systèmes TR²E critiques et adaptatifs fortement contraints. Pour clore notre étude des caractéristiques du générateur de code de MyCCM-HI, nous présentons l'évaluation des performances du code généré.

8.5 Performances réseaux

Nous nous intéressons dans cette section aux performances réseaux du code généré par MyCCM-HI.

Commençons par évaluer les caractéristiques du code généré en termes de temps de latence, de gigue, et de débit. Rappelons ici que la carte utilisée fourni une interface réseaux de type 100 Mbits/s. Pour procéder à ces tests de performance, nous avons réalisé une application qui envoie un message toutes les 2 millisecondes à un composant qui est instancié :

- soit sur le même nœud mais dans un processus différent,
- soit sur un nœud différent.

Le message envoyé contient une donnée de type entier long non signé (*unsigned long*). Lorsque le composant reçoit le message, il renvoie aussitôt un message au composant source.

8.5.1 Latence

On mesure ici le temps qui sépare l'émission du premier message et la réception de la réponse correspondante. On obtient donc un temps de latence "aller-retour". La figure 8.4 représente la répartition du temps de latence (nombre d'occurrences en ordonnées, temps de latence en abscisse) dans le cas où les deux composants sont instanciés sur une même carte.

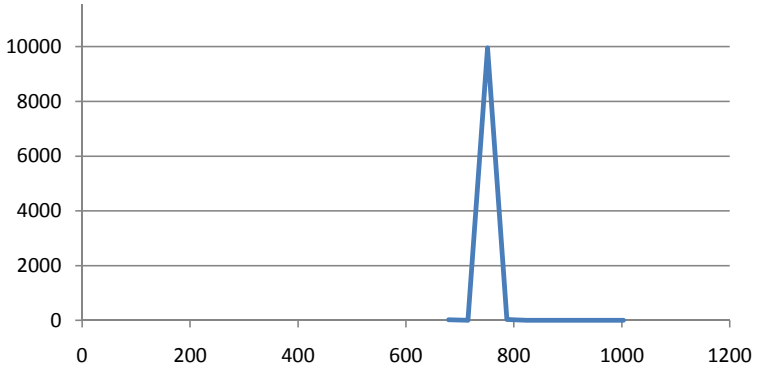

FIGURE 8.4 – Répartition des temps de latence avec une carte

Dans cette configuration, le temps de latence moyen est de 754 microsecondes, le temps de latence minimal est de 679 microsecondes, le temps de latence maximum est de 1003 microsecondes. Au delà de la valeur absolue de ces temps de latence, il est intéressant de noter le déterminisme du comportement de l'application. Ici, 99,98% des itérations fournissent un temps de latence compris entre 679 et 787 microsecondes.

La figure 8.5 représente la répartition des temps de latence dans le cas où les composants sont instanciés sur deux cartes différentes. Dans cette configuration, le temps de latence

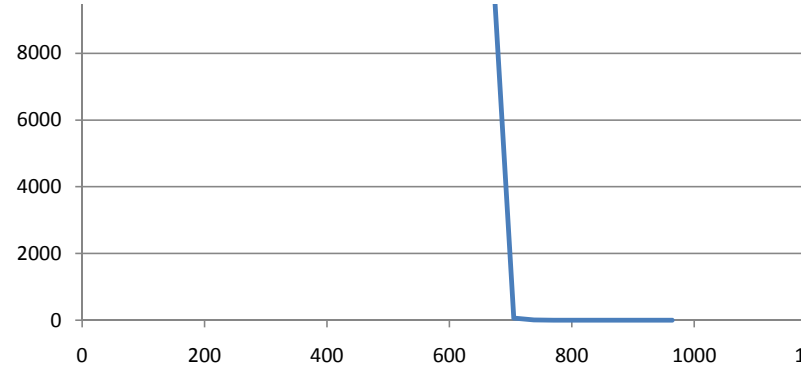

FIGURE 8.5 – Répartition des temps de latence avec deux cartes

moyen est de 696 microsecondes, le temps de latence minimal est de 673 microsecondes, le temps de latence maximum est de 964 microsecondes. 99,98% des itérations du test fournissent un temps de latence compris entre 673 et 770 microsecondes.

8.5.2 Gigue

Nous avons également mesuré le temps de gigue entre deux “aller-retour” successifs : on mesure la différence entre deux temps de latence successifs.

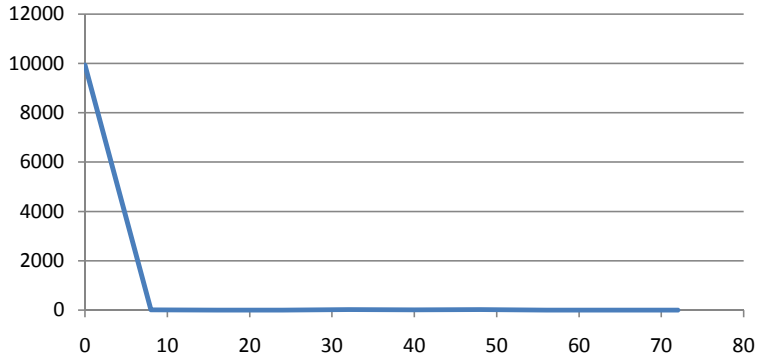

FIGURE 8.6 – Répartition des gigues avec une carte

Dans cette configuration, la gigue est en moyenne de 1 microsecondes et s’étend de 0 à 74 microsecondes. Les résultats obtenus lors de l’utilisation de deux cartes électroniques sont très similaires à ceux obtenus ici. La simulation de charge CPU n’a ici que peu d’impact sur les résultats obtenus.

Nous obtenons donc un temps de gigue inférieur à 9,8% du temps de latence moyenne, et un temps de gigue moyenne qui correspond à 0,13% du temps de latence moyenne.

Ces résultats montrent le déterminisme du code utilisé pour gérer les communications.

8.5.3 Débit

Le débit maximal obtenu sur notre carte électronique était de 72Mb/s, ce qui est un peu en dessous de la limite théorique de l’interface réseau (100 Mb/s). Cette différence peut s’expliquer d’une part par l’utilisation du protocole PCP, et d’autre part par les phases d’empaquetage et de dés-empaquetage des données au niveau de l’intergiciel.

Étudions maintenant les caractéristiques du code généré par le *framework* MyCCM-HI en terme de temps de reconfiguration.

8.6 Validation des politiques de reconfiguration

8.6.1 Temps de reconfiguration.

Nous présentons dans cette sous-section les mesures que nous avons effectuées afin d’évaluer le temps de reconfiguration moyen correspondant à chacune des politiques de reconfiguration que nous avons présentées précédemment (voir chapitre 4). Comme le laisse supposer les équations de pire temps de reconfiguration que nous avons présentées au chapitre 7, ce temps moyen de reconfiguration devrait dépendre fortement de la charge CPU de l’application en cours d’adaptation. Nous avons donc simulé cette charge CPU en ajoutant dans le code fonctionnel de l’application des instructions qui permettent d’occuper le CPU pendant un temps déterminé à l’avance. Par ailleurs, nous stimulons les mécanismes d’adaptation

du système en envoyant toutes les 210 millisecondes une requête de changement de mode. La figure 8.7 présente les résultats obtenus en terme de temps de reconfiguration moyen sur 1000 requêtes de changements de mode (500 M → A et 500 A → M).

Cette figure présente les résultats obtenus en suivant les quatre types de configuration que nous avons présentés au chapitre 4.

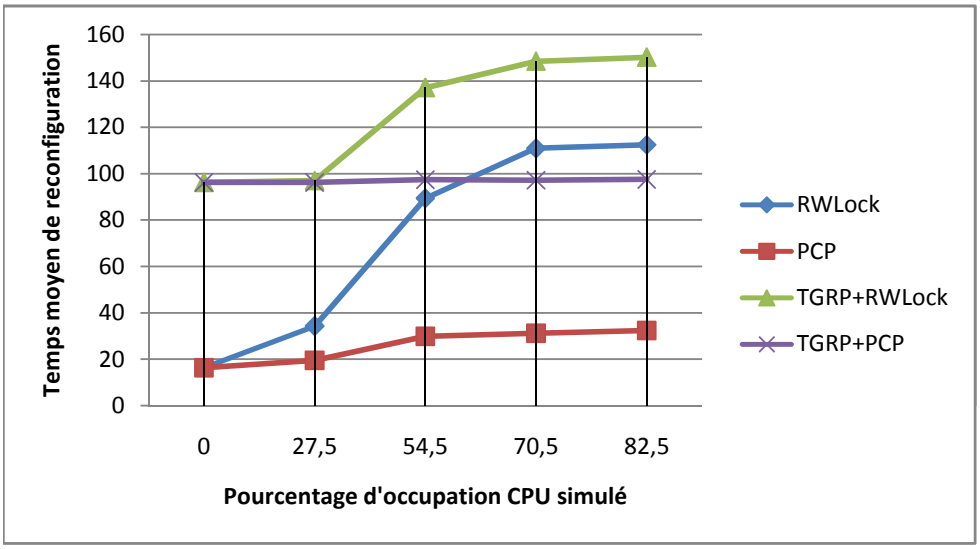

FIGURE 8.7 – Temps de reconfiguration mesuré, en fonction de la charge CPU simulée

Cette figure représente en abscisse le pourcentage d'occupation CPU simulé. Ainsi, lorsque le temps CPU sur la figure est de zéro, cela signifie que seul le code fonctionnel implanté dans les composants est exécuté : le code correspondant à la simulation de l'occupation CPU n'est pas exécuté. Le temps de reconfiguration est exprimé en millisecondes.

Les résultats présentés sur cette figure sont conformes aux attentes que nous avions exprimées dans la présentation de notre démarche :

- l'utilisation de la politique qui permet de favoriser la reconfiguration dynamique améliore fortement la performance de la reconfiguration, surtout si la charge CPU est élevée ;
- il est intéressant de noter que la différence de temps de reconfiguration entre la politique RWLock et PCP augmente avec la charge CPU : plus la charge CPU est élevée, plus la différence entre ces deux politiques est élevée. Ceci s'explique facilement en considérant les équations (7.1) et (7.2). En effet, si on considère ces équations, on s'aperçoit que dans le cas de la politique PCP (équation (7.2)), une seule exécution de chaque tâche impactée ne peut avoir lieu avant la mise en œuvre de la reconfiguration. En utilisant la politique RWLock (équation (7.1), le temps de reconfiguration dépend du nombre d'occurrence des tâches impactées, et ce nombre d'occurrence dépend du temps d'exécution des tâches de priorité faible. Ainsi dans notre exemple, lorsque le temps d'exécution de la tâche *guidance_activity* augmente, le nombre d'occurrence de la tâche *auto_pilot_activity* entre la réception de la requête de changement de mode et le changement de mode effectif augmente. La figure 8.8 illustre les résultats obtenus en utilisant les équation (7.1) et (7.2) pour calculer le pire temps de reconfiguration attendu compte tenu des valeur d'occupation CPU que nous avons simulé. Cette figure confirme que la différence entre le temps de reconfiguration avec les politique RWLock et PCP augmente avec la charge CPU.

– le temps de reconfiguration associé aux politiques utilisant la synchronisation à l'hyper-période provoque l'apparition d'un seuil dans le temps de reconfiguration. Ce seuil correspond à l'attente de la synchronisation des tâches à leur hyper-période (terme $Hyper(SG)$ dans les équations (7.3) et (7.4). Sur la figure 8.8, on constate un décalage du pire temps de reconfiguration attendu, dû à l'attente de l'hyper-période. Ici encore, la différence de pire temps de reconfiguration augmente avec la charge CPU selon qu'on utilise la politique TGRP combinée avec RWLock ou PCP.

Le résultat de cette expérience permet ainsi d'aiguiller la décision de l'architecte logicielle concernant la politique de reconfiguration à utiliser. Pour faire un tel choix, voici les critères sélectionnés :

1. la performance de la reconfiguration, ou le pire temps de reconfiguration acceptable ;
2. le besoin fonctionnel de conserver des ensembles de données cohérentes lors du changement de mode ;
3. la charge CPU et le temps d'exécution des tâches impactées par le changement de mode ;
4. la nécessité de garantir l'ordonnançabilité de l'application, même en cas de changement de mode.

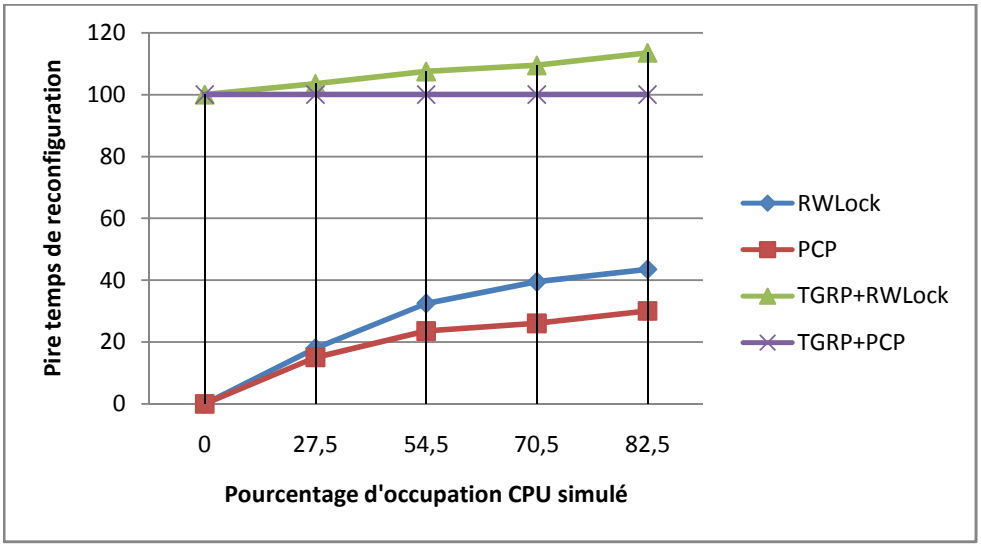

FIGURE 8.8 – Pire temps de reconfiguration, calculé en fonction de la charge CPU simulée

Comme nous l'avons expliqué précédemment, la figure 8.8 représente le pire temps de reconfiguration calculé selon les équations (7.1), (7.2), (7.3) et (7.2), respectivement pour les protocoles RWLock, PCP, TGRP+RWLock et TGRP+PCP.

Cette figure permet de confirmer que nos expérimetations fournissent des résultats conforme aux attentes théoriques. Les allures des courbes dans le cas expérimentale (voir figure précédente) et théorique sont très similaires, les différence venant du fait que le calcul théorique ne prend en compte que le temps CPU simulé, alors que l'expérimentation reflète un cas d'utilisation réel, dans lequel le nombre de tâche est plus importants (tâches pour lesquels aucune occupation CPU n'est simulée).

Cette expérimentation nous a permis de montrer que l'implantation des politiques de reconfiguration, générée par MyCCM-HI, était conforme aux attentes exprimées lors de la présentation de l'approche (voir chapitre 4).

8.6.2 Configuration de la priorité plafond.

Nous allons ici nous pencher sur deux questions intéressantes concernant l'utilisation de la politique PCP. La première concerne la façon dont est déterminée la priorité plafond. La seconde concerne la possibilité d'associer une priorité plafond non pas à l'ensemble des transitions d'un automate, mais seulement à un ensemble de port de celui-ci.

Déterminer la priorité plafond. Si l'utilisateur précise la valeur de la priorité plafond utilisé, MyCCM-HI vérifie que cette valeur est bien supérieur à la plus haute priorité de l'ensemble des tâche impactés par un changement de mode de cet automate. Si l'utilisateur ne spécifie pas de valeur, mais souhaite utiliser la politique PCP, alors MyCCM-HI fixe cette priorité plafond à la valeur de la plus haute priorité (plus un) des tâches impactées par la reconfiguration. Ensuite, MyCCM-HI vérifie que la priorité des tâches qui traitent les évènements de reconfiguration utilisant PCP sont plus prioritaires que les tâches impactées par ces reconfiguration. Ceci permet d'obtenir une configuration cohérente : utiliser PCP garantit que la reconfiguration est privilégiée.

Configurer les ports de l'automate Il est également possible d'associer la politique PCP à un ensemble de ports de l'automate plutôt qu'à l'ensemble de l'automate. Pour cela, il faut respecter la règle suivante : une activité sporadique ne peut pas traiter les évènements d'un port d'automate configuré avec PCP et ceux d'un port d'automate qui n'est pas configuré avec cette politique. En effet, utiliser PCP vise à privilégier la reconfiguration alors que ne pas l'utiliser favorise l'application. Une tâche ne peut pas être configurer pour répondre simultanément à ces deux besoins : quelle sera sa priorité dans ce cas ? Une priorité supérieure à la plus haute priorité des tâches impactées, pour privilégier la reconfiguration ? Ou une priorité intermédiaire pour privilégier certaines tâches de l'application ?

Enfin, insistons ici sur une caractéristique importante concernant l'utilisation de PCP : si deux ports d'un automate de mode impactent un groupe d'activités synchronisées, et sont configurés selon des politiques de reconfiguration différentes (PCP et non-PCP), il faut créer deux tâches périodiques de reconfiguration. Une pour traiter les reconfigurations avec PCP, l'autre pour traiter les reconfigurations sans PCP.

La configuration de la politique de reconfiguration associée aux ports d'un automate de mode n'est pas encore possible dans le prototype que nous avons réalisé. Pour ce faire, une évolution majeure consiste à générer les opérations de changement de mode indépendemment pour chaque port d'un automate.

8.6.3 Fusion de groupes d'activités impactées.

Si une tâche appartient à plusieurs groupe de tâches synchronisées, les reconfiguration qui impactent cette tâche sont réalisées à l'hyper-période de l'union de ces groupes de tâches. Autrement dit, la reconfiguration d'une tâche appartenant à l'intersection de plusieurs groupes de tâches synchronisées se fait à l'hyper-période de l'union de ces groupes de tâches. Cette fonctionnalité a été implantée dans le prototype de génération de code que nous avons réalisé.

Dans la section suivante, nous allons nous intéresser aux résultats que nous avons obtenu lors des expérimentations que nous avons faites autour de la vérification formelle.

8.7 Vérification formelle

Nous allons diviser notre étude en deux partie, respectivement dédiées à l'analyse système et la l'analyse temporelle de notre cas d'étude.

8.7.1 Analyse système

Détaillons ici l'ensemble des automates UPPAAL obtenus en utilisant les règles de transformation que nous avons présenté au chapitre 7.

Modélisation de l'automate de mode Nav_supervisor_impl

La figure 8.9 représente la modélisation formelle de l'automate de mode *Nav_supervisor_impl*, automate de mode du sous-système de navigation. Nous retrouvons sur cette figure les modes de cet automate, à savoir le mode *automatic* et le mode *manual*. Les transitions de modes illustre bien l'utilisation de la règle de transformation présentée au chapitre 7.

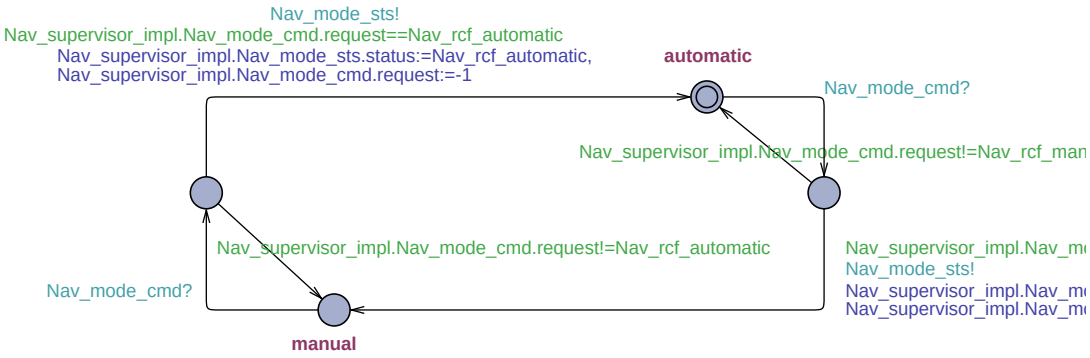

FIGURE 8.9 – Modélisation formelle de l'automate de mode Nav_supervisor_impl

Modélisation du comportement de l'utilisateur final

La figure 8.10 représente le scénario d'exécution que nous avons modélisé afin de fermer le modèle formel. Il s'agit de la modélisation de l'interface graphique, et du comportement de l'utilisateur susceptible d'envoyer des requêtes de changement de mode au système.

La spécification de cet automate permet de simuler les requêtes de changement de mode. Le comportement simulé ici consiste à envoyer une requête de changement de mode vers le mode automatique dès que l'automate est dans le mode manuel, et vice versa. Ce type d'automate peut être généré à partir de la spécification du comportement d'un système (ici, le système de communication).

Nous ne représentons pas ici d'automate correspondant à une connexion dans la mesure où ils sont identiques à celui que nous avons présenté au cours du chapitre 7.

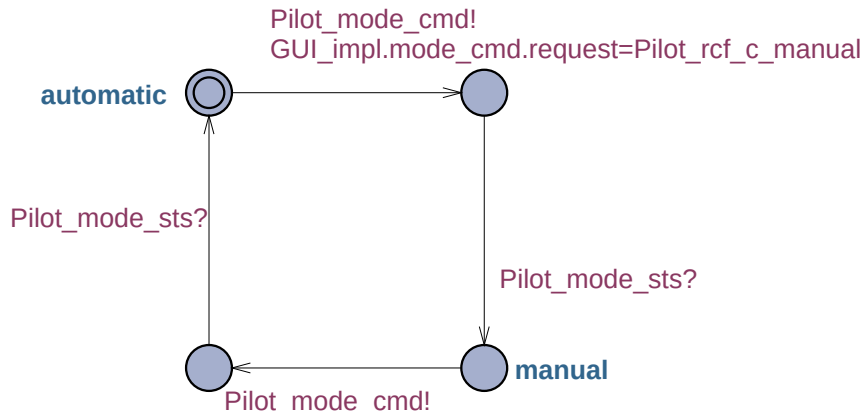

FIGURE 8.10 – Modélisation formelle de l'interface utilisateur

Vérifications efféctuées

A partir de ces différents modèles, nous avons pu vérifier les propriété formelles suivantes (exprimées en TCTL [Alur *et al.*, 1992]) :

- absence de blocage dans un mode donné. Expression logique : *A[] not deadlock* ;
- le mode *automatic* est atteignable. Expression logique : *E<> Pilot_supervisor_inst.automatic*
- l'instance d'automate de mode *Loc_supervisor_inst* peut être dans le mode *stopped* alors que l'instance d'automate de mode *Nav_supervisor_inst* est dans le mode *automatic*. Expression logique : *E<> Loc_supervisor_inst.stopped and Nav_supervisor_inst.automatic*.

Cette première phase de vérification nous a permi d'établir que la troisième propriété est fausse. En effet, les automates de mode du système de navigation et du système de localisation ne sont pas synchronisés. Nous avons facilement pu résoudre ce problème. Cependant, nous considérerons dans la suite de ce mémoire que ces deux modes sont faiblement incohérent entre eux, c'est-à-dire qu'ils peuvent être simultanément actifs pendant une période de temps donnée.

8.7.2 Analyse logicielle

Présentons maintenant les modèles UPPAAL obtenus par transformation de l'architecture logicielle du système de pilotage. Les modèles des horloges, des tâches et des connexions ne sont pas représentés car identique à ceux que nous avons présenté au chapitre 7

Modélisation de l'automate de mode Nav_supervisor_impl

La figure 8.11 illustre l'utilisation de ces règles de mapping dans le cas de l'automate de mode *Nav_supervisor_impl*.

Modélisation de l'ordonnanceur

La figure 8.12 représente le modèle UPPAAL de l'ordonnanceur du système. Comme prévu, cette figure se compose de deux états indépendant du nombre de tâches à ordon-

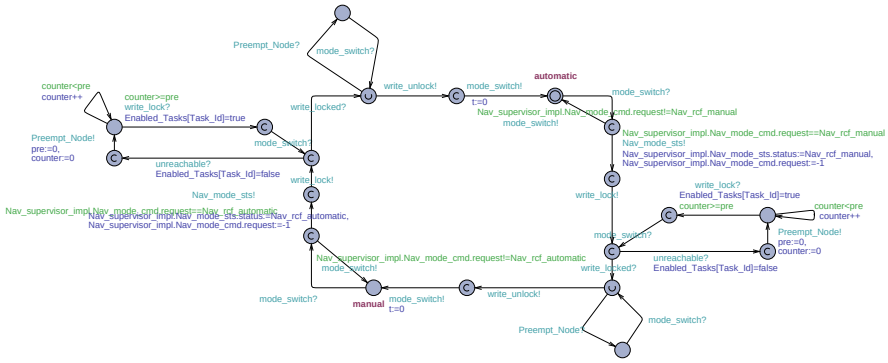

FIGURE 8.11 – Modélisation formelle de l'automate de mode Nav_supervisor_impl dans un contexte temporisé

nacer, et d'autant de paires d'états qu'il existe de de tâches à ordonnancer (ici cinq tâches : *guidance_activity*, *auto_pilot_activity*, *navigation_activity*, et deux tâches de reconfiguration.

Modélisation du comportement des tâches impactées

Pour expérimenter notre approche, nous avons modélisé le comportement des différentes tâches sans passer par ce processus de transformation. La figure 8.13 illustre le comportement d'une tâche impactée par la reconfiguration : acquisition du verrou en lecture, calcul pendant un temps *Delta*, puis envoie de message (transition set_pos !).

Vérifications effectuées

La modélisation formelle de l'architecture logicielle du système de pilotage a donné lieu à l'analyse des propriétés temporelles décrites aux chapitres 3 puis 7. Rappelons ici ces propriétés, et donnons leur expression en TCTL :

1. l'ordonnançabilité de la tâche Guidance_activity. Propriété logique : *A[] not Guidance_activity.Task_Error* ;
2. l'atteignabilité du mode *manual*. Propriété logique : *E<> Pilot_supervisor_inst.manual* ;
3. le passage du mode automatique au mode manuel se fait toujours dans un délai inférieur à 200 millisecondes. Propriété logique : *A[] GUI_inst.t<=200*
4. les automates *Nav_supervisor_inst* et *Loc_supervisor_inst* ne sont jamais dans des états incohérents entre eux (*automatic* et *stopped*) pendant plus de 100 millisecondes. Propriété logique : *A[] ((Nav_supervisor_inst.automatic and Loc_supervisor_inst.stopped) imply (Nav_supervisor_inst.t<=100 or Loc_supervisor_inst.t<=100))*.

Nous allons maintenant nous intéresser à l'impact de la discrétisation de la préemption sur la taille de l'espace d'état à explorer pour évaluer ces propriétés. Nous considérons pour cela deux cas de discrétisation. Dans un premier temps, nous considérons que l'unité de temps de la préemption est de 1 millisecondes. Dans ce cas, le nombre d'états explorés est environ de 2 700. Ensuite, nous avons discrétisé en considérant une unité de préemption de

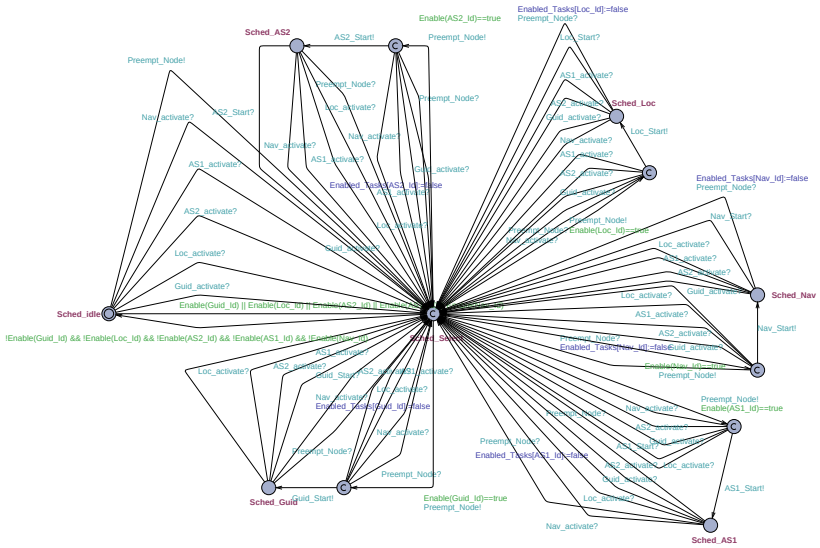

FIGURE 8.12 – Modélisation formelle de l'ordonnanceur du système

10 microsecondes. Dans ce cas, le nombre d'états explorés est environ de 51 000. Dans ces deux cas, l'analyse des propriétés se fait très rapidement.

Ces résultats montrent que la stratégie de modélisation formelle que nous avons mis en œuvre permet d'analyser très finement le comportement de l'architecture logicielle.

8.8 Synthèse

Augmenter la productivité. L'ensemble des résultats que nous avons présenté dans ce chapitre confirme la validité de notre approche. Nous avons ainsi montré qu'il est possible d'augmenter la productivité des équipes logicielles par le biais de techniques de génération de code. Rappelons que sur notre exemple, 836 lignes de spécification COAL donnent lieu à la production de 20248 lignes de code C et que ce code est optimisé, dans la mesure où l'exécutable produit ne nécessite que 860 Ko à l'exécution. Enfin, rappelons que le code généré respecte les règles de réalisation des systèmes critiques.

Améliorer la qualité du logiciel produit. Les résultats présentés dans ce chapitre ont également permis de montrer l'analysabilité des spécifications COAL, aussi bien du point de vue système que du logiciel. Outillée l'analyse formelle du comportement du système devient beaucoup plus facile, et peut ainsi être utilisé en cours de réalisation du système. Ceci permet d'augmenter fortement la qualité du logiciel produit, puisque le comportement de celui-ci est formellement vérifié.

Disposer de différentes politiques de reconfiguration. Les résultats numériques obtenus autour de la réalisation du cas d'étude, en termes de temps de reconfiguration, montrent la validité de notre implémentation des protocoles de reconfiguration et illustrent l'utilité de disposer de ces différents protocoles de reconfiguration. Rappelons un dernière fois que ces différents protocoles permettent de réaliser le compromis, en fonction de la charge CPU occupée par

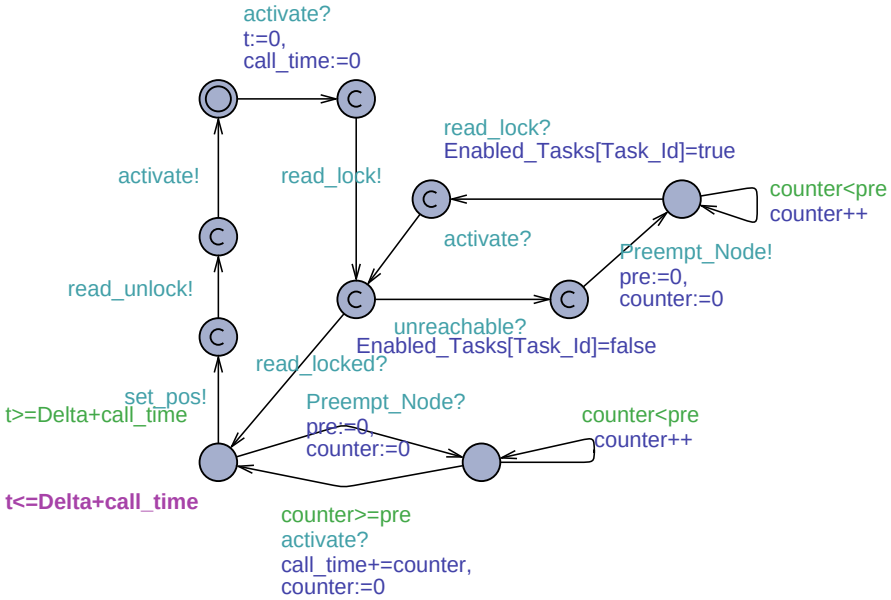

FIGURE 8.13 – Modélisation du comportement d'une tâche impactée

l'application, entre (i) la rapidité de la reconfiguration, (ii) l'analysabilité du changement de mode, et (iii) la disponibilité d'ensembles de données cohérentes entre elles.

Automatiser le processus de développement. Enfin, les expérimentations menées autour de la réalisation du prototype de générateur de code montrent (i) qu'il est possible d'automatiser la production du code technique des systèmes TR²E critiques et adaptatifs, et (ii) par extension, qu'il est possible de générer les modèles formelles sur lesquels s'appuie la vérification des propriétés temporelles de l'application.

En conclusion, l'ensemble des résultats expérimentaux que nous avons présenté ici confirment que notre méthodologie répond à l'objectif que nous nous étions fixé : fournir un démarche de génie logiciel dirigée par les modèles qui améliore l'efficacité des équipes de développements logiciels en termes de qualité et de productivité.

Sixième partie

Conclusion Générale

Chapitre 9

Conclusions et Perspectives

Chaque progrès donne un nouvel espoir, suspendu à la solution d'une nouvelle difficulté.
Le dossier n'est jamais clos.
Claude Lévi-Strauss

SOMMAIRE

Pour conclure cette étude, nous allons commencer par résumer l'ensemble des contributions que nous venons de présenter. Nous prendrons alors un peu de recul pour évaluer la qualité et la complétude de notre travail. Enfin, nous discuterons les perspectives qu'offre le travail recherche que nous avons réalisé.

9.1 Rappel des contributions et résultats

Nous avons présenté dans ce mémoire trois contributions logicielles, qui s'articulent toutes autour d'une quatrième contribution : une méthode de conception.

9.1.1 Méthode de conception

Contribution centrale de notre étude, la méthode de conception que nous proposons dans ce mémoire permet d'améliorer l'efficacité des équipes de développement logiciel, en termes

de qualité et de productivité. En effet, cette méthodologie automatise l'analyse et la production des systèmes TR²E critiques et adaptatifs en proposant de :

- modéliser l'ensemble des mécanismes d'adaptation du système ;
- générer les modèles formels représentatifs de ces mécanismes d'adaptation ;
- générer le code de mise en œuvre de ces mécanismes.

La mise en œuvre de cette spécification s'appuie sur des techniques de modélisation qui ont nécessité la proposition d'un nouveau langage de description d'architecture.

9.1.2 Langage de description d'architecture des systèmes TR²E critiques et adaptatifs

Notre méthodologie s'appuie sur un langage de description d'architecture (COAL) qui permet de modéliser le comportement adaptatif d'un système. De plus, COAL permet de représenter conjointement l'architecture logicielle à base de composants génériques, ainsi que les propriétés spécifiques au domaine des systèmes TR²E.

COAL permet ainsi de capturer les caractéristiques non-fonctionnelles de l'application, et plus particulièrement les caractéristiques temporelles, tout en facilitant la décomposition fonctionnelle par le biais de composants logiciels génériques.

Avec ce langage, les différents comportements possibles du système sont représentés par le biais de modes de fonctionnement, encapsulés dans des automates de mode dont le comportement est analysable. Nous avons montré au chapitre 7 une technique de modélisation formelle grâce à laquelle nous vérifions que le comportement du système adaptatif respecte les propriétés de sûreté identifiées lors de sa spécification. De plus, COAL permet de spécifier l'ensemble des fonctionnalités actives dans un mode donné en associant une configuration logicielle à chacun de ces modes. Enfin, ce langage permet de paramétrer le compromis nécessaire entre les différents enjeux de la mise en œuvre de mécanismes d'adaptation : l'analysabilité, la rapidité, et la cohérence des flux de données.

Pour automatiser l'exploitation de cette spécification, nous avons créé un *framework* à composants, appelé MyCCM-HI, qui automatise la production et l'analyse des applications logicielles d'un système TR²E critique et adaptatif.

9.1.3 MyCCM-HI, *framework* à composants pour les systèmes TR2E critiques et adaptatifs

Notre contribution autour du *framework* à composants MyCCM-HI se décompose en deux parties. Une partie dédiée à la production automatique du code de l'application, et une partie dédiée à la validation du comportement du système.

Production automatique de code

Une première partie, relativement mature, consiste en un générateur de code qui automatise la production de mécanismes d'adaptation "corrects par construction". En effet, les mécanismes d'adaptation que nous utilisons garantissent que l'ensemble des tâches qui ont commencé leur exécution dans un mode donné finit leur exécution dans ce mode. En fonction de l'urgence de l'adaptation et de la nécessité de conserver des flux de données cohérents,

nous utilisons la politique de reconfiguration adéquate. Nous avons montré au chapitres 7 et 7 que la sémantique de reconfiguration utilisée est analysable y compris concernant ses caractéristiques temporelles. De plus, le code généré par MyCCM-HI respecte les règles de réalisation des systèmes critiques que nous avons présenté au chapitre 3, ce qui facilite l'analyse des caractéristiques temporelles de l'application ainsi que l'analyse statique du code produit. Enfin, les résultats expérimentaux obtenus sur un cas d'étude industriel montre que le code obtenu est optimisé, alors que le facteur de génération est important : A partir de 836 lignes de spécification COAL, nous obtenons 20248 lignes de code C générées, pour une occupation mémoire à l'exécution inférieure à 860 Ko ; Le gain estimé en terme d'empreinte mémoire par rapport à MyCCM est d'un **facteur 500** environ. Ce gain est en grande partie imputable à l'utilisation de l'intergiciel PolyORB-HI et du générateur de code associé, Ocarina.

Ce générateur de code est le fruit d'un travail collaboratif impliquant à la fois des ingénieurs Thales et des chercheurs de TelecomParisTech. Le fruit de ce travail, qui représente (en dehors du code d'Ocarina) plus de 130 000 lignes de code Java, a été suffisamment mature pour constituer une livraison *"open source"* d'une part, et la réalisation d'un démonstrateur lors d'un stage de Master d'autre part.

Vérification formelle du système et de ses applications

La seconde partie de notre contribution consiste en un ensemble de règles de transformation qui permettent de générer automatiquement un modèle formel équivalent à la spécification système et/ou logicielle fournie. Ces modèles représentent à la fois la logique des changements de mode, ainsi que le comportement de l'ensemble de l'application en cours d'adaptation.

Nous avons formalisé ces règles de transformation au chapitre 7. Les expérimentations que nous avons réalisé autour du cas d'étude industriel nous ont permis de vérifier la validité de ces règles, et de vérifier le type de propriété de sûreté de fonctionnement spécifiées ci-dessous :

- un mode donné est accessible ;
- la spécification n'induit pas de blocage dans un mode donné ;
- deux modes fortement incohérents entre eux ne sont jamais simultanément actifs ;
- le passage d'un mode à un autre n'excède pas un certain intervalle de temps ;
- deux modes faiblement incohérents ne sont pas simultanément actifs plus d'une certaine période de temps ;
- le système reconfigurable est ordonnançable.

Ces propriétés correspondent par ailleurs aux propriétés de sûreté que nous avions identifiés en précisant notre problématique, au chapitre 3.

Une autre caractéristique importante des travaux que nous avons réalisé est l'intégration de langages de spécification standardisés. L'utilisation de standards garantit l'interopérabilité entre les outils qui s'appuient sur ces standards. De plus, cela permet de bénéficier des travaux existants autour de ces standards.

9.1.4 Intégration de langages standards de spécification

Un autre aspect important de notre contribution est l'intégration de langages standardisé : lightweight CCM pour la spécification des composants et de leurs interfaces, et AADL pour la description des caractéristiques de l'architecture propres aux systèmes TR²E.

Bien que l'intégration de AADL se limite aujourd'hui aux constructions de la première version du standard (AADL 1.0), l'effort que nous avons fait ici constitue une première étape d'intégration. Les construction définies dans la nouvelle version de ce standard (AADL 2.0) permettront sûrement d'étendre cette intégration aux mécanismes d'adaptation dynamique. L'utilisation d'AADL comme langage intermédiaire permet de bénéficier des travaux réalisés autour de ce langage de description d'architecture dédié à la spécification et à l'analyse des systèmes TR²E.

9.1.5 Evaluation expérimentale de nos contributions

Les résultats expérimentaux que nous avons obtenus illustrent le bien fondé de notre approche. Outre la génération de code optimisée, que nous avons déjà évoqué dans ce chapitre, les résultats que nous avons obtenus en évaluant le temps moyen de reconfiguration, ou encore en vérifiant formellement le comportement du système de notre cas d'étude industriel, confirment cette assertion.

9.2 Conclusion

L'approche que nous avons développé répond à l'ensemble des problématiques que nous avions identifiées au chapitre 3. Bien sûr, cette solution mérite d'être améliorée ; nous détaillerons ses limitations dans la section suivante. Cependant, les résultats expérimentaux que nous avons obtenus confirment son bien fondé car ils montrent qu'elle permet d'automatiser l'ensemble des étapes de production des systèmes TR²E critiques et adaptatifs.

En conclusion, notre contribution résout les questions de la configuration et de la reconfiguration des systèmes TR²E critiques et adaptatifs dans une approche "pragmatique". Elle automatise la production et l'analyse du comportement du logiciel en partant du principe que la spécification est intégralement fournie par un utilisateur final. Nous avons donc atteint notre objectif initial : améliorer l'efficacité des équipes de développement logiciel, en termes de qualité et de productivité. Le gain en productivité est assuré par le générateur de code, alors que le gain en qualité est à la fois assuré par le générateur de code et par l'automatisation de la vérification formelle de l'architecture système et logicielle.

9.3 Limites et perspectives

9.3.1 Limites

Décrivons maintenant les limites de notre contribution. Ces limites viennent principalement du fait que nous n'avons pas eu le temps d'implanter l'ensemble des caractéristiques de notre approche.

Premièrement, notons que la politique de synchronisation des tâches est spécifiée sur l'ensemble des automates, et non sur un port donné. Cette solution est pourtant meilleure, puisqu'elle permet de différencier la politique choisi pour différentes transitions d'un même automate. Ensuite, nous n'avons pas pu implanté les mécanisme de temporisation des transition d'un automate, qui permettent de ne pas attendre indéfiniment la réponse d'un système en panne. Enfin, bien que nous ayons fourni les règles de transformation de modèle, nous n'avons pas pu implanter l'outil de génération correspondant.

Malgré ces limitations, les contributions et résultats que nous avons présentés montrent la validité de notre approche. L'extension de la spécification de la politique de reconfiguration aux ports de l'automate ne constitue qu'une optimisation de ce que nous avons déjà réalisé et expérimenté. La proposition des mécanismes de temporisation est spécifique aux exigences de la tolérance aux pannes, ce qui ne fait pas intégralement partie du périmètre de notre travail de recherche. Enfin, les règles de transformation que nous avons proposés ont été expérimentées, de telle sorte qu'automatiser la production de ces modèles ne constitue pas un travail difficile.

9.3.2 Perspectives

Bien que nous n'ayons pas pu réaliser ces différentes améliorations, notre contribution ouvre un large spectre de perspectives. En effet, elles constitue une première implantation d'outils de génération de code et de modèles formel dédiés aux systèmes TR²E critiques et adaptatifs.

Une solution ouverte à tous

Open source, cette solution pourra ainsi être réutilisée par de nombreux ingénieurs ou chercheurs pour tester la mise en œuvre de leur solution dans le contexte de systèmes critiques et adaptatifs.

Plus particulièrement, notre approche pourra servir à l'évaluation de techniques visant à automatiser la configuration logicielle en amont du processus de développement : dans le domaine de la sûreté de fonctionnement par exemple, les changements de modes sont utilisés pour réagir à l'arrivée d'une panne. Imaginons qu'une solution permette de déterminer automatiquement la configuration cible sur survenue d'une panne donnée. Dans ce cas, l'ensemble des mécanismes de reconfiguration peut être décrit selon notre approche, et notre solution peut-être utilisée pour faciliter la mise en œuvre de l'ensemble de l'approche, depuis la spécification des pannes jusqu'à l'implantation finale.

De la même façon, lorsqu'une technique de résolution de contrainte est disponible pour automatiser le déclenchement de changements de modes en fonction des caractéristiques de l'application (charge CPU, niveau de batterie, etc...) le comportement associé peut-être décrit et implanté en utilisant notre approche.

Perspectives industrielles et scientifiques

Le travail de recherche que nous avons présenté dans ce mémoire constitue une contribution importante du projet de recherche Flex-eWare, piloté par l'Agence Nationale de la Recherche (ANR). Il a reçu un accueil très favorable auprès des experts techniques de la société Thales, et fut à la base de nouveaux travaux collaboratifs internes, principalement dans le do-

maine du spatial. D'autres collaborations, principalement dans le domaine de l'avionique, sont à l'étude.

Au sein de la communauté scientifique du domaine, ce travail sera également prolongé dans un nouveau projet de recherche, Parsec, financé par le pôle de compétitivité System@tic.

Annexe A

COAL ANTLR Grammar

```
1
2
3
4  definition
5          :       system';'
6          |       system_realization ';'
7          |       component_impl ';'
8          |       interface_impl  ';'
9          |       operational_automaton ';'
10         |       operational_automaton_impl ';'
11         |       technical_service ';'
12         |       technical_service_impl ';'
13         |       technical_service_inst ';'
14         |       host_dcl ';'
15         |       bus_dcl ';'
16         ;
17
18 specification
19         : ( definition )* ((import_idl3)+ ( definition )+)+
20         ;
21
22 import_idl3
23         : 'import' fileName=identifier ';'
24         ;
25
26 scoped_name returns [Contained target, int lineNumber, ScopedName sc, String strScopeName, Token token]
27         :       k= identifier    {$strScopeName = $k.str;} ('::' ihm=identifier { $strScopeName+= "::" + $ihm.str;} )*
28         | '::' l= identifier  {$strScopeName = "::" + $l.str ;}   ('::' m=identifier { $strScopeName += "::" + $m.str; })*
29         ;
30
31 ref_string returns [String strValue, Token token]
32         :       k= identifier    {$strValue = $k.str ;} ('::' ihm=identifier { $strValue += "::" + $ihm.str ;} )*
33         | '::' l= identifier  {$strValue = $l.str ;} ('::' m=identifier { $strValue += "::" + $m.str; })*
34         ;
35
36
37
38
39
40 system
41         : ch=system_header '{' system_body '}'
42         ;
43
44 system_header returns [Token token]
45         : c='system' id= identifier
46         ;
47
48 system_body
49         : (system_export)*
```

```
50          ;
51
52  system_export
53          : em=emits_dcl ';'
54          | pu=publishes_dcl ';'
55          | co=consumes_dcl ';'
56          ;
57
58  system_realization
59          : ch=system_realization_header '{' system_realization_body '}'
60          ;
61
62  system_realization_header returns [Token token]
63          : c='system' name=identifier 'realizes' ref=scoped_name
64          ;
65
66  system_realization_body
67          : (process_dcl ';')*
68          | (system_realization_export)*
69          subcomponents_specification ';'
70          (system_realization_export)*
71          ;
72
73  subcomponents_specification
74          : 'subcomponents' '{'
75                  software_components_specification+
76          '}'
77          |
78          'subsystems' '{'
79                  (subsystem_ref ';')+
80          '}'
81          ;
82
83  subsystem_ref
84          : 'system' ref=scoped_name
85          ;
86
87  software_components_specification
88          : component_inst';'
89          | component_impl ';'
90          ;
91
92  system_realization_export
93          : process_dcl ';'
94          | operational_automaton ';'
95          | operational_automaton_impl ';'
96          | operational_automaton_inst ';'
97          | target_process_assignment ';'
98          | internal_connections ';'
99          | internal_connection
100         | a=activity ';'
101         | synchronization_spec ';'
102         ;
103
104 synchronization_spec
105         : 'synchronization' '(' t_id = scoped_name
106         (',' t_id=scoped_name)+ ')'
107         ;
108
109
110
111
112
113 interface_impl
114         : interface_impl_forward_dcl
115         | interface_impl_dcl
116         ;
117
```

```
118 interface_impl_forward_dcl
119         :  ' interface '   'implementation'  identifier
120         ;
121
122 interface_impl_dcl  returns  [Token token]
123         : c = ' interface '   'implementation' name=identifier 'implements' spec=ref_string
124         ;
125
126
127
128
129
130 component_impl
131         :         component_impl_forward_dcl
132         |         component_impl_dcl
133         ;
134
135 component_impl_forward_dcl
136         :          'component' 'implementation'  identifier
137         ;
138
139 component_impl_dcl
140         :         ch=component_impl_header '{' component_impl_body '}'
141         ;
142
143 component_impl_header returns [Token token]
144         :         c='component' 'implementation' name=identifier 'implements' spec=ref_string
145         ;
146
147 component_impl_body
148         : (composite_impl_export)+
149         | (primitive_impl_export)*
150         ;
151
152 primitive_impl_export
153         : intfmap_dcl ' ; '
154         | data_dcl ' ; '
155         | data_access_dcl ';'
156         |  declaration_file_dcl   ' ; '
157         ;
158
159  declaration_file_dcl
160         : 'declaration'   ' file '   ' : '   file_ref = string_literal
161         ;
162
163
164 composite_impl_export
165         : component_inst';'
166         | component_impl ';'
167         | operational_automaton ';'
168         | operational_automaton_impl ';'
169         | operational_automaton_inst ';'
170         | a =  activity   ' ; '
171         | internal_connections ' ; '
172         | internal_connection
173         | synchronization_spec ';'
174         ;
175
176 intfmap_dcl
177         :    intf_ref =ref_string  c='.implementation' ' :='   intf_impl =scoped_name
178         ;
179
180 internal_connections
181         :ch=internal_connections_header '{' internal_connections_body '}'
182         ;
183
184 internal_connection
185         : 'connection' id= identifier   ' : '    internal_connection_export
```

```
186             ;
187
188 internal_connections_header returns [Token token]
189             : 'connections' id= identifier
190             ;
191
192 internal_connections_body
193             : (internal_connection_export)*
194             ;
195
196 internal_connection_export
197             : membrane_connection_dcl ';'
198             | subcpt_connection_dcl ';'
199             | bus_association ';'
200             ;
201
202 bus_association
203             : 'bus' ':' bus=scoped_name
204             ;
205
206 membrane_connection_dcl
207             : subcpt_ref=scoped_name '.' subcpt_port=identifier c='<->' composite_port=identifier (m=mode_ref_list)?
208             ;
209
210 subcpt_connection_dcl
211             : subcpt1_ref=scoped_name '.' subcpt1_port=identifier c='<->' subcpt2_ref=scoped_name '.' subcpt2_port=identifier (m=mode_ref_list)?
212             ;
213
214 uses_dcl: c='uses' intfc=ref_string port= identifier
215             ;
216
217
218
219
220
221 component_inst
222             :       component_inst_dcl
223             |       component_inst_forward_dcl
224             ;
225
226 component_inst_forward_dcl
227             :       'component' 'instance' identifier
228             ;
229
230 component_inst_dcl
231             :       ch=component_inst_header '{' component_inst_body '}'
232             ;
233
234 component_inst_header returns [Token token]
235             :       c='component' 'instance' name=identifier 'instantiates' impl=component_impl_ref
236             ;
237
238 component_impl_ref returns [ComponentImplementation impl]
239             : sc=scoped_name
240             ;
241
242 component_inst_body
243             : (component_inst_export)*
244             ;
245
246 component_inst_export
247             : attrInit=attribute_assignment ';'
248             ;
249
250
251 target_process_assignment
252             : c_inst=scoped_name '.' c='process' ':=' process=process_ref
253             ;
```

```
254
255 process_ref returns [TargetProcess proc]
256         : sc=scoped_name
257         ;
258
259 attribute_assignment returns [ AttributeInitialization attributeAssignment, String fileName, int lineNumber]
260         : attribute = identifier c=':=' value=value_ref (m=mode_ref_list)?
261         ;
262
263 value_ref returns [Contained inst, AttributeValue v]
264         : sc=ref_string
265         | idl_positive_int_value = positive_integer_literal
266         | idl_negative_int_value=idl_negative_int
267         | idl_float_value = floating_pt_literal
268         | char_value=character_decl
269         | string_value= string_literal
270         | '{' value=value_ref (',' value_ref)* '}'
271         ;
272
273 mode_ref_list returns [ArrayList<ModeRef> modes]
274         : c='in' 'mode' '(' instance=scoped_name '.'modeId=identifier
275         (',' i=scoped_name '.' modeId=identifier)*')'
276         ;
277
278
279
280
281
282 operational_automaton
283         : operational_automaton_dcl
284         | operational_automaton_forward_dcl
285         ;
286
287 operational_automaton_forward_dcl
288         : 'mode' 'automaton' identifier
289         ;
290
291 operational_automaton_dcl
292         : ch=operational_automaton_header '{' operational_automaton_body'}'
293         ;
294
295 operational_automaton_header returns [Token token]
296         :       c='mode' 'automaton' name=identifier
297         ;
298
299 operational_automaton_body
300         : (operational_automaton_export)*
301         ;
302
303 operational_automaton_export
304         : modes_dcl ';'
305         | em=emits_dcl ';'
306         | pu=publishes_dcl ';'
307         | co=consumes_dcl ';'
308         | attr_spec ';'
309         ;
310
311 modes_dcl
312         : 'mode' '{' m=mode_list_dcl '}'
313         ;
314
315 mode_list_dcl returns [ArrayList<Mode> modes]
316         : m=identifier '(' 'initial' ')'
317         (',' m=identifier)+
318         ;
319
320 emits_dcl returns [EmitsDef evt]
321         :       'emits' sc=ref_string id= identifier
```

```
322             ;
323
324 publishes_dcl returns  [PublishesDef evt]
325             :           'publishes'  sc=ref_string  id= identifier
326             ;
327
328 consumes_dcl returns [ConsumesDef evt]
329             :           'consumes' sc=ref_string id= identifier
330             ;
331
332 attr_spec
333             :           a=' attribute '   attr_type= identifier   decls=attr_declarator
334             ;
335
336  attr_declarator  returns  [ArrayList<String> names, ArrayList<Integer> lineNumbers]
337             :           sd=simple_declarator
338                         (   ' , '   sdc=simple_declarator )*
339             ;
340
341
342
343
344
345 operational_automaton_impl
346             : operational_automaton_impl_dcl
347             | operational_automaton_impl_forward_dcl
348             ;
349
350 operational_automaton_impl_forward_dcl
351             :  'mode' 'automaton' 'implementation' name=identifier
352             ;
353
354 operational_automaton_impl_dcl
355             : ch=operational_automaton_impl_header '{' operational_automaton_impl_body'}'
356             ;
357
358 operational_automaton_impl_header returns [Token token]
359             :           c='mode' 'automaton' 'implementation' name=identifier 'implements' impl=operational_automaton_ref
360             ;
361
362 operational_automaton_ref returns [OperationalAutomaton comp]
363             :           (  sc=scoped_name )?
364             ;
365
366 operational_automaton_impl_body
367             :  (operational_automaton_impl_export)*
368             ;
369
370 operational_automaton_impl_export
371             :  mode_transition_spec ';'
372             ;
373
374 mode_transition_spec
375             : ch=mode_transition_spec_header '{' mode_transition_spec_body'}'
376             ;
377
378 mode_transition_spec_header returns [Token token]
379             :           'in '   'mode' name=identifier ' : '
380             ;
381
382 mode_transition_spec_body
383             :  (cond=condition_conjunction '−−>' target_mode=identifier ';')+
384             |  (cond=condition_conjunction '−−>' target_mode=identifier act=action_spec ';')+
385             ;
386
387 condition_conjunction returns  [ArrayList<Condition> pre_cond_list, ArrayList<TriggerEvent> evt,  ArrayList<Condition> post_cond_list]
388             :  '['
389            (precondition= condition_list   'AND')? evt_spec=trigger_event_spec
```

```
390             ('AND' evt_spec=trigger_event_spec)*
391             ('AND' precondition2=condition_list } )?  ']'
392             ;
393
394  condition_list  returns  [ArrayList<Condition> cond_list]
395             : condition=condition_expr
396             (  'AND'condition=condition_expr )*
397             ;
398
399
400  condition_expr returns  [Condition cond]
401             : c=id_condition_expr
402             | c=const_condition_expr
403             ;
404
405  id_condition_expr returns  [Condition cond]
406             : '('  tested_elt= identifier  comparator=COMPARATOR testor_elt=identifier ')'
407             | 'NOT' '('  tested_elt= identifier  comparator=COMPARATOR testor_elt=identifier ')'
408             ;
409
410  const_condition_expr returns [Condition cond]
411             : '('  tested_elt= identifier  comparator=COMPARATOR testor_elt=positive_integer_literal ')'
412             | 'NOT' '('  tested_elt= identifier  comparator=COMPARATOR testor_elt=positive_integer_literal ')'
413             ;
414
415  trigger_event_spec returns [TriggerEvent evt]
416             : event_sink= identifier    '?'  '('
417             ( field = identifier '='value= identifier )*  ')'
418             ;
419
420  action_spec returns [ArrayList<Action> actions]
421             : '{'  act=action_export
422             (act=action_export)* '}'
423             ;
424
425  action_export returns[Action action]
426             : attrAs=attribute_assignment ';'
427             | attrUp=attribute_update ';'
428             | evt=event_emission_spec';'
429             ;
430
431  attribute_update  returns[AttributeUpdate attributeUpdate]
432             :  attr = identifier   '++'
433             |  attr = identifier   '--'
434             ;
435
436  event_emission_spec returns[EventEmission evtEmission]
437             : event_source=identifier  '!'  '('  values=eventtype_assignment ')'
438             ;
439
440  eventtype_assignment returns [EventtypeAssignment assignment]
441             :  field = identifier   ':='  value= identifier
442             ;
443
444
445
446
447
448  operational_automaton_inst
449             : operational_automaton_inst_dcl
450             | operational_automaton_inst_forward_dcl
451             ;
452
453  operational_automaton_inst_forward_dcl
454             : 'mode' 'automaton' 'instance'  name=identifier
455             ;
456
457  operational_automaton_inst_dcl
```

```
458         : ch=operational_automaton_inst_header '{' operational_automaton_inst_body'}'
459         ;
460
461 operational_automaton_inst_header returns [Token token]
462         :       c='mode' 'automaton' 'instance' name=identifier ' instantiates ' impl=operational_automaton_impl_ref
463         ;
464
465 operational_automaton_impl_ref returns [OperationalAutomatonImplementation comp]
466         :       ( sc=scoped_name )?
467         ;
468
469 operational_automaton_inst_body
470         : (operational_automaton_inst_export)*
471         ;
472
473 operational_automaton_inst_export
474         : target_process_assignment ';'
475         |   attribute_initialization    ';'
476         | priority_ceiling_assignment  ';'
477         ;
478
479  attribute_initialization
480         :   attrInit =attribute_assignment
481         ;
482
483 priority_ceiling_assignment
484         :'ceiling  priority '  ':'  value= idl_positive_int
485         ;
486
487
488
489
490
491  activity  returns [Activity  act]
492         : pa= periodic_activity
493         | sa=sporadic_activity
494         | aa=aperiodic_activity
495         | ia=it_based_activity
496         ;
497
498
499
500
501
502  periodic_activity  returns [Activity  act]
503         :ch=periodic_activity_header '{' periodic_activity_body '}'
504         ;
505
506 periodic_activity_header returns [Token token, Activity  act]
507         :'periodic'  ' activity ' name=identifier
508         ;
509
510
511 periodic_activity_body
512         : periodic_activity_export*
513         ;
514
515  periodic_activity_export
516         :   periodic_activity_inst_port_ref   ';'
517         | period_assignment ';'
518         | wcet_assignment ';'
519         | priority_assignment ';'
520         | stack_size_assignment ';'
521         ;
522
523
524  activity_inst_port_ref
525         :'configures' instance=scoped_name'.'port=identifier
```

```
526             |
527             'configures' '{' 'instance=scoped_name'.'port=identifier
528              (',' instance=scoped_name'.'port=identifier )*'}'
529             ;
530
531  periodic_activity_inst_port_ref
532             : 'configures' instance=scoped_name'.'port=identifier 'operation' operation= identifier
533             | 'configures' '{' 'instance=scoped_name'.'port=identifier 'operation' operation= identifier
534              (',' instance=scoped_name'.'port=identifier 'operation' operation= identifier )*'}'
535             ;
536
537 period_assignment
538             : 'period' ':' period= idl_int 'ms'
539             | 'period' ':' period= idl_int 'us'
540             ;
541
542 wcet_assignment
543             : 'Execution' 'Time' ':' min=idl_int '..' max=idl_int 'ms'
544             ;
545
546 priority_assignment
547             : ' priority ' ':' prio = idl_int
548             ;
549
550 stack_size_assignment
551             : 'stack' 'size' ':' stack_size= positive_integer_literal 'KByte'
552             ;
553
554
555
556
557
558  sporadic_activity returns [ Activity act]
559             : ch=sporadic_activity_header '{' sporadic_activity_body '}'
560             ;
561
562 sporadic_activity_header returns [Token token, Activity act]
563             : 'sporadic' ' activity ' name=identifier
564             ;
565
566
567 sporadic_activity_body
568             : (sporadic_activity_export)*
569             ;
570
571 sporadic_activity_export
572             : port_ref ';'
573             | ports_ref ';'
574             | period_assignment ';'
575             | wcet_assignment ';'
576             | priority_assignment ';'
577             | stack_size_assignment ';'
578             ;
579
580  port_ref
581             : 'configures' instance=scoped_name'.'port=identifier
582             ;
583
584 ports_ref
585             : 'configures' '(' ports= list_of_ports ')'
586             ;
587
588  list_of_ports
589             : instance=scoped_name'.'port=identifier
590             (',' instance=scoped_name'.'port=identifier )*
591             ;
592
593
```

```
594
595
596
597 aperiodic_activity returns [Activity act]
598         : ch=aperiodic_activity_header '{' aperiodic_activity_body '}'
599         ;
600
601 aperiodic_activity_header returns [Token token, Activity act]
602         : 'aperiodic' 'activity' name=identifier
603         ;
604
605
606 aperiodic_activity_body
607         : (aperiodic_activity_export)*
608         ;
609
610 aperiodic_activity_export
611         : port_ref ';'
612         | ports_ref ';'
613         | wcet_assignment ';'
614         | priority_assignment ';'
615         | stack_size_assignment ';'
616         ;
617
618
619 exception_list
620         : '('scoped_name')' (',' scoped_name)* ')'
621         ;
622
623 simple_declarator returns [String name, int lineNumber, Token token]
624         : i= identifier
625         ;
626
627 positive_integer_literal returns [String expValue, Token token]
628         : i= idl_positive_int
629         | o=octal
630         | h=hex
631         ;
632
633 idl_int         returns [String expValue, Token token]
634         : i=IDL_POSITIVE_INT
635         | i=IDL_NEGATIVE_INT
636         ;
637
638 idl_positive_int        returns [String expValue, Token token]
639         : i=IDL_POSITIVE_INT
640         ;
641
642 idl_negative_int        returns [String expValue, Token token]
643         : i=IDL_NEGATIVE_INT
644         ;
645
646 octal   returns [String expValue, Token token]
647         : i=OCTAL_NUMBER
648         ;
649
650 hex returns [String expValue, Token token]
651         : i=HEX
652         ;
653
654 value_base_type
655         : 'ValueBase'
656         ;
657
658 fixed_pt_literal
659         : FIXED_PT_LITERAL
660         ;
661
```

```
floating_pt_literal returns [String expValue, Token token]
        :       i = FLOATING_PT_LITERAL
        ;

string_literal returns [String strValue, Token token]
        :       (str=STRING_LITERAL)+
        ;

path returns [String strValue, Token token]
        : str=PATH_LITERAL
        ;

wide_string_literals
        :       wide_string_literal+
        ;

wide_string_literal
        :
        'L' string_literal
        ;


boolean_literal returns [String strValue, Token token]
        :       i='TRUE'
        |       j='FALSE'
        ;









process_dcl
        : ch=process_header '{' process_body '}'
        ;

process_header returns [Token token]
        : 'process' id= identifier
        ;

process_body
        : process_export*
        ;

process_export
        : host_association ';'
        | ports_number ';'
        ;

host_association
        : 'target' ':' cpu=scoped_name
        ;

ports_number
        : 'port' ':' port_number=idl_positive_int
        ;









host_dcl
        : ch=host_header '{' host_body '}'
        ;

host_header returns[Token token]
        : 'target' id= identifier
        ;
```

```
730 host_body
731         : host_export*
732         ;
733
734 host_export
735         : requires_bus ';'
736         ;
737
738 requires_bus
739         : id= identifier ':' 'uses' 'bus' bus=scoped_name 'at' 'address' address=string_literal
740         ;
741
742 address_list returns[ArrayList<String> addr_list]
743         : address= string_literal (',' address= string_literal)*
744         ;
745
746
747
748
749
750
751 bus_dcl
752         : ch=bus_header ;
753
754
755 bus_header returns [Token token]
756         : 'bus' id= identifier
757         ;
758
759
760
761
762
763 data_dcl
764          : 'data' id= identifier
765         ;
766
767 data_access_dcl
768         : data_readers_dcl
769         | data_writers_dcl
770         ;
771
772 data_writers_dcl
773         : operations=operation_ref 'write' data=scoped_name
774         ;
775
776 data_readers_dcl
777         : operations=operation_ref 'read' data=scoped_name
778         ;
779
780 operation_ref returns [ArrayList<OperationInterceptor> operationInterceptors]
781         : facet= identifier '.' operation= identifier
782         | sink= identifier
783         | '(' (sink= identifier
784         | facet= identifier '.' operation= identifier)
785         (',' facet= identifier '.' operation= identifier
786         | ',' sink= identifier)*
787         ')'
788         ;
789
790
791
792
793
794
795
796  identifier returns [String str, Token token]
797         :         i=IDENTIFIER
```

```
798             ;
799
800     character_decl returns [String str, Token token]
801             :       i=CHARACTER_DECL
802             ;
803
804     fragment
805     ZERO    :       '0'
806             ;
807
808     fragment
809     DIGIT   :       '0' .. '9'
810             ;
811     fragment
812     NONZERO_DIGIT
813             :       '1' .. '9'
814             ;
815     fragment
816     FIXED_VALUE
817             :       ('-')? NONZERO_DIGIT (DIGIT)* ('.' DIGIT*)?
818             |       '.' (DIGIT)+
819             ;
820
821     FIXED_PT_LITERAL returns [String expValue]
822             :       i=FIXED_VALUE ('d' | 'D')
823             ;
824
825     IDL_POSITIVE_INT    :       (NONZERO_DIGIT) (DIGIT)*
826             ;
827
828     IDL_NEGATIVE_INT    :       ('-') (NONZERO_DIGIT) (DIGIT)*
829             ;
830
831     fragment
832     OCTAL_DIGIT
833             :       '0' .. '7'
834             ;
835
836     OCTAL_NUMBER
837             :       (ZERO) (OCTAL_DIGIT)*
838             ;
839
840     fragment
841     HEX_DIGIT
842             :       '0' .. '9'
843             |       'a' .. 'f'
844             |       'A' .. 'F'
845             ;
846
847     HEX     :       ('0x' | '0X') (HEX_DIGIT)+
848             ;
849
850     IDENTIFIER
851             :       '_'? i=LETTER (LETTER | DIGIT | '_')*
852             ;
853     fragment
854     LETTER :        'a' .. 'z'
855             |       'A' .. 'Z'
856             ;
857
858     fragment
859     CHARACTER_LITERAL
860             :       LETTER | ESCAPE_SEQUENCE | PONCTUATION | DIGIT
861             ;
862
863     CHARACTER_DECL
864             :       ('\'')(CHARACTER_LITERAL)('\'')
865             ;
```

```
866
867 fragment
868 PONCTUATION
869         :       '.' | ':' | ',' | '\'' |'/' | '_' | '−'| '!' | '?' | '*' | '%'| ';' | ' '
870                 ;
871
872 fragment
873 WIDE    :       'L'
874         ;
875
876 STRING_LITERAL
877         :       '"' (CHARACTER_LITERAL) *'"'
878         ;
879
880 PATH_LITERAL
881         :       '"'
882         (
883         ('.')
884         | ('\\')
885         | (':')
886         | ('−')
887         | (' ')
888         | ('_')
889         | ('/')
890         | LETTER
891         | DIGIT) * '"'
892         ;
893
894 fragment
895 EXP     :       ('e' | 'E')? ('+' | '−')? (DIGIT)+
896         ;
897
898 FLOATING_PT_LITERAL
899         :       FIXED_VALUE (EXP)?
900         ;
901
902
903 WS :   (' '|'\r'|'\t'|'\u000C'|'\n')+ {skip ();}
904     ;
905
906 SL_COMMENT
907         :
908         '//' (~'\n')* '\n'
909         ;
910
911 fragment
912 NEW_LINE:     ('\r')? '\n'
913         ;
914
915 PRAGMA :      '#' ' ' + ln=IDL_POSITIVE_INT ' '+ str=STRING_LITERAL (' '+ IDL_POSITIVE_INT)? ' '* NEW_LINE
916         |
917                 '#' ' '+ j=IDL_POSITIVE_INT + ' '+ '"<' (~'>')+ '>"' ' '* NEW_LINE
918                                 |
919                 '#pragma' ' '+ IDENTIFIER ' '+ STRING_LITERAL ' '* NEW_LINE
920         ;
921
922 ML_COMMENT
923         :
924         '/*'
925         (
926                         '\n'
927                 |       ('*')+
928                         (       '\n'
929                         |       ~( '*' | '/' | '\n')
930                         )
931                 |       ~('*' | '\n')
932         )*
933         '*/'
```

```
934             ;
935
936     COMPARATOR
937             : '>' | '>=' | '<' | '<=' | '=='
938             ;
939
940     fragment
941     ESCAPE_SEQUENCE
942             :       '\\n'
943             |       '\\t'
944             |       '\\v'
945             |       '\\b'
946             |       '\\r'
947             |       '\\f'
948             |       '\\a'
949             |       '\\\\'
950             |       '\\?'
951             |       '\\\''
952             |       '\\\"'
953             |       '\\x' HEX_DIGIT HEX_DIGIT
954             |       '\\' OCTAL_DIGIT OCTAL_DIGIT OCTAL_DIGIT
955             |       '\\u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
956             ;
```

Exemple A.1 – COAL grammar

Bibliographie

[Allen, 1997] Robert J. Allen. *A formal approach to software architecture*. PhD thesis, Pittsburgh, PA, USA, 1997.

[Alur and Dill, 1994] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[Alur *et al.*, 1992] Rajeev Alur, Costas Courcoubetis, R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-toi. Minimization of timed transition systems. pages 340–354. Springer-Verlag, 1992.

[Apvrille *et al.*, 2004a] L. Apvrille, P. De Saqui-Sannes, P. Sénac, and C. Lohr. Verifying service continuity in a dynamic reconfiguration procedure : Application to a satellite system. *Automated Software Engg.*, 11(2) :167–191, 2004.

[Apvrille *et al.*, 2004b] L. Apvrille, P. De Saqui-Sannes, P. Sénac, and C. Lohr. Verifying service continuity in a satellite reconfiguration procedure. *Journal of Automated Software Engineering*, 11(2) :167–191, 2004.

[Armstrong, 1996] Joe Armstrong. Erlang - a survey of the language and its industrial applications. In *In INAP'96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, 1996.

[Barnes, 2008] John Barnes. Safe and secure software, an invitation to ada 2005, Avril 2008.

[Behrmann *et al.*, 2004] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems : 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[Bengtsson *et al.*, 1996] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal - a tool suite for automatic verification of real-time systems, 1996.

[Benveniste *et al.*, 2003] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, pages 64–83, 2003.

[Berry, 2000] Gérard Berry. *The foundations of Esterel*. MIT Press, Cambridge, MA, USA, 2000.

[Bertrand *et al.*, 2008] Dominique Bertrand, Anne-Marie Déplanche, Sébastien Faucou, and Olivier H. Roux. A study of the aadl mode change protocol. In *ICECCS '08 : Proceedings of the 13th IEEE International Conference on on Engineering of Complex Computer Systems*, pages 288–293, Washington, DC, USA, 2008. IEEE Computer Society.

[Birrell and Nelson, 1984] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1) :39–59, February 1984.

[Borde *et al.*, 2007] E. Borde, G. Haïk, , V.Watine, and L. Pautet. Really hard time developing hard real-time. In *2nd National workshop on Control Architecture of Robots*, june 2007.

[Borde *et al.*, 2008] E. Borde, G. Haïk, , V.Watine, and L. Pautet. *Les systèmes répartis en action : de l'embarqué au large échelle*, chapter 4. Lavoisier, 2008.

[Bouyer, 2002] Patricia Bouyer. *Modèles et algorithmes pour la vérification des systèmes temporisés*. Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France, April 2002.

[Burns, 1999] Alan Burns. The ravenscar profile. *ACM Ada Letters*, 4 :49–52, 1999.

[Chang and Collet, 2007] Hervé Chang and Philippe Collet. Compositional patterns of non-functional properties for contract negotiation. *Journal of Software (JSW)*, 2(2) :52–63, August 2007.

[Coulouris *et al.*, 2005] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems (4th ed.) : concepts and design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[Cui and Nahrstedt, 2001] Yi Cui and Klara Nahrstedt. Qos-aware dependency management for component-based systems. In *In HPDC '01 : Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01*, page 127. IEEE Computer Society, 2001.

[Cui *et al.*, 2001] Yi Cui, Dongyan Xu, and Klara Nahrstedt. Smart : A scalable middleware solution for ubiquitous multimedia service delivery. *Multimedia and Expo, IEEE International Conference on*, 0 :260, 2001.

[Dumant *et al.*, 1999] Bruno Dumant, François Horn, Frédéric Dang Tran, and Jean-Bernard Stefani. Jonathan : an open distributed processing environment in java. *Distributed Systems Engineering*, 6(1) :3–12, 1999.

[Fujii and Suda, 2004] Keita Fujii and Tatsuya Suda. Component service model with semantics (cosmos) : A new component model for dynamic service composition. *Applications and the Internet Workshops, IEEE/IPSJ International Symposium on*, 0 :348, 2004.

[Garavel and Thivolle, 2009] Hubert Garavel and Damien Thivolle. Verification of gals systems by combining synchronous languages and process calculi. In *SPIN*, pages 241–260, 2009.

[Gardey *et al.*, 2005] Guillaume Gardey, Didier Lime, Morgan Magnin, and Olivier (H.) Roux. Roméo : A tool for analyzing time Petri nets. In *17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, Edinburgh, Scotland, UK, July 2005. Springer.

[Gokhale *et al.*, 2003] Aniruddha Gokhale, Douglas Schmidt, Nanbor Wang, Tao Lu, Balachandran Natarajan, and Ran Natarajan. Cosmic : An mda generative tool for distributed real-time and embedded applications, 2003.

[Gorrieri and Siliprandi, 1994] Roberto Gorrieri and Glauco Siliprandi. Real-time system verification using p/t nets. In *CAV '94 : Proceedings of the 6th International Conference on Computer Aided Verification*, pages 14–26, London, UK, 1994. Springer-Verlag.

[Goudarzi and Kramer, 1996] Kaveh Moazami Goudarzi and Jeff Kramer. Maintaining node consistency in the face of dynamic change. In *In Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 62–69. IEEE Computer Society Press, 1996.

[Grondin *et al.*, 2006] G. Grondin, N. Bouraqadi, and L. Vercouter. MaDcAr : an Abstract Model for Dynamic and Automatic (Re-)Assembling of Component-Based Applications. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2006)*, volume 4063 of *LNCS*, pages 360–367, Västeras, Sweden, Jun 2006. Springer-Verlag.

[Haddad *et al.*, 2006] Serge Haddad, Fabrice Kordon, and Laure Petrucci, editors. *Méthodes Formelles pour les Systèmes Répartis et Coopératifs*. Hermes/Lavoisier, septembre 2006.

[Halbwachs *et al.*, 1991] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[Hamid *et al.*, 2008] I. Hamid, B. Zalila, E. Najm, and J. Hugues. Automatic framework generation for hard real-time applications. *Innovations in Systems and Software Engineering : A NASA Journal*, mar 2008.

[IEEE, 2004] IEEE. Portable operating system interface, ieee std 1003.1. Technical report, 2004.

[Krčál and Yi, 2004] Pavel Krčál and Wang Yi. Decidable and undecidable problems in schedulability analysis using timed automata. In *Proc. of TACAS'04*, pages 236–250. Springer–Verlag, 2004.

[Labbani *et al.*, 2005] Ouassila Labbani, Jean luc Dekeyser, and Pierre Boulet. Mode-automata based methodology for scade. In *In Springer, Hybrid Systems : Computation and Control, 8th International Workshop, LNCS series*, pages 386–401. Springer Verlag, 2005.

[Leclercq *et al.*, 2007a] Matthieu Leclercq, Ali E. Ozcan, Vivien Quema, and Jean B. Stefani. Supporting heterogeneous architecture descriptions in an extensible toolset. In *ICSE '07 : Proceedings of the 29th international conference on Software Engineering*, pages 209–219, Washington, DC, USA, 2007. IEEE Computer Society.

[Leclercq *et al.*, 2007b] Matthieu Leclercq, Ali Erdem Ozcan, Vivien Quema, and Jean-Bernard Stefani. Supporting heterogeneous architecture descriptions in an extensible toolset. In *ICSE '07 : Proceedings of the 29th international conference on Software Engineering*, pages 209–219, Washington, DC, USA, 2007. IEEE Computer Society.

[Liu and Layland, 1973] C.L. Liu and James Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment, 1973.

[Luckham *et al.*, 1995] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Walter Mann, Walter Mann, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4) :336–355, 1995.

[Madl *et al.*, 2009] Gabor Madl, Nikil Dutt, and Sherif Abdelwahed. A Conservative Approximation Method for the Verification of Preemptive Scheduling using Timed Automata. In *Proceedings of RTAS*, pages 255–264, April 2009.

[Mallet *et al.*, 2009] Frederic Mallet, Charles Andre, and Julien DeAntoni. Executing aadl models with uml/marte. In *ICECCS '09 : Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 371–376, Washington, DC, USA, 2009. IEEE Computer Society.

[Maraninchi and Rémond, 1998] Florence Maraninchi and Yann Rémond. Mode-automata : About modes and states for reactive systems. In *In European Symposium On Programming*. Springer Verlag, 1998.

[McManis and Varaiya, 1994] Jennifer McManis and Pravin Varaiya. Suspension automata : A decidable class of hybrid automata. In *CAV*, pages 105–117, 1994.

[Medina *et al.*, 2002] Gonzlez Harbour Medina, J. L. Medina, J. J. Gutiérrez, J. C. Palencia, and J. M. Drake. Mast : An open environment for modeling, analysis, and design of real-time systems, 2002.

[Moazami-Goudarzi, 1999] K. Moazami-Goudarzi. *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College London, March 1999.

[(OMG), 2003] Object Management Group (OMG). Light weight corba component model revised submission. Technical report, OMG (Object Management Group), May 2003.

[(OMG), 2004] Object Management Group (OMG). Common object request broker architecture : Core specification, version 3.0.3. Technical report, OMG (Object Management Group), March 2004.

[(OMG), 2005a] Object Management Group (OMG). Deployment and configuration of component-based distributed applications specification. Technical report, OMG (Object Management Group), January 2005.

[(OMG), 2005b] Object Management Group (OMG). Real-time corba specification version 1.2. Technical report, OMG (Object Management Group), January 2005.

[(OMG), 2006] Object Management Group (OMG). Corba component model specification version 4.0. Technical report, OMG (Object Management Group), April 2006.

[(OMG), 2007] Object Management Group (OMG). Unified modeling language (uml). Technical report, OMG (Object Management Group), November 2007.

[Pautet, 2001] Laurent Pautet. *Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition*. Habilitation à diriger des recherches, Université Pierre et Marie Curie – Paris VI, December 2001.

[Pedro and analysis for, 1998] P. Pedro and A. Burns. Schedulabillity analysis for. Mode changes in real-time systems. *Real-Time Systems, 10th Euromicro Workshop on*, 1998.

[Polakovic and Stefani, 2008] Juraj Polakovic and Jean-Bernard Stefani. Architecting reconfigurable component-based operating systems. *Journal of Systems Architecture*, 54(6) :562 – 575, 2008. Selection of best papers from the 32nd EUROMICRO Conference on ['']Software Engineering and Advanced Applications'' (SEAA 2006).

[Polakovic *et al.*, 2007] Juraj Polakovic, Sebastien Mazare, Jean-Bernard Stefani, and Pierre-Charles David. Experience with safe dynamic reconfigurations in component-based embedded systems. In *CBSE*, pages 242–257, 2007.

[Quinot, 2003] Thomas Quinot. *Conception et réalisation d'un intergiciel schizophrène pour la mise en oeuvre de systèmes répartis interopérables*. PhD thesis, École Nationale Supérieure des Télécommunications, 2003.

[Reade, 1989] Chris Reade. *Elements of functional programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[Real and Crespo, 2004] Jorge Real and Alfons Crespo. Mode change protocols for real-time systems : A survey and a new proposal. *Real-Time Syst.*, 26(2) :161–197, 2004.

[Real and Wellings, 1999] Jorge Real and Andy Wellings. Implementing mode changes with shared resources in ada. *Real-Time Systems, Euromicro Conference on*, 0 :0086, 1999.

[Rolland, 2008] Jean-François Rolland. *Développement et validation d'architectures dynamiques*. Thèse de doctorat, Université Paul Sabatier, Toulouse, France, décembre 2008.

[RTCA and EUROCAE, 1992] RTCA and EUROCAE. Do-178b, software considerations in airborne systems and equipment certification. Technical report, 1992.

[SAE, 2004] SAE. Architecture analysis and design language (aadl). Technical report, The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, November 2004.

[Schmidt and Cleeland, 2000] Douglas C. Schmidt and Chris Cleeland. Applying a pattern language to develop extensible orb middleware. In *ORB Middleware," in Design Patterns in Communications*. University Press, 2000.

[Schmidt *et al.*, 1997] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the tao real-time object request broker. *Computer Communications*, 21 :294–324, 1997.

[Schneider *et al.*, 2004] Etienne Schneider, Florentin Picioroagă, and Uwe Brinkschulte. Dynamic reconfiguration through osa+, a real-time middleware. In *DSM '04 : Proceedings of the 1st international doctoral symposium on Middleware*, pages 319–323, New York, NY, USA, 2004. ACM.

[Sha *et al.*, 1988] Lui Sha, Ragunathan Rajkumar, John Lehoczky, and Krithi Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1 :243–264, 1988.

[Sha *et al.*, 1990] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9) :1175–1185, 1990.

[Singhoff *et al.*, 2004] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar : a flexible real time scheduling framework. *Ada Lett.*, XXIV(4) :1–8, 2004.

[Singhoff *et al.*, 2005] Frank Singhoff, Jérôme Legrand, and Laurent tchamnda Nana. Aadl resource requirements analysis with cheddar. In *SAE AADL Working Group meeting, Paris, October 18-21, 2005*, Oct 2005.

[Sriplakich *et al.*, 2008] Prawee Sriplakich, Xavier Blanc, and Marie-Pierre Gervals. Collaborative software engineering on large-scale models : requirements and experience in modelbus. In *SAC '08 : Proceedings of the 2008 ACM symposium on Applied computing*, pages 674–681, New York, NY, USA, 2008. ACM.

[Stankovic, 1988] John A. Stankovic. Misconceptions about real-time computing : A serious problem for next-generation systems. *Computer*, 21(10) :10–19, 1988.

[Sun Microsystems, 2002] Inc Sun Microsystems. Java remote method invocation specification. Technical report, Sun Microsystems, Inc, 2002.

[Vergnaud, 2006] Thomas Vergnaud. *Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées*. PhD thesis, École Nationale Supérieure des Télécommunications, 2006.

[Weil *et al.*, 2000] Daniel Weil, Valérie Bertin, Etienne Closse, Michel Poize, Patrick Venier, and Jacques Pulou. Efficient compilation of esterel for real-time embedded systems. In *CASES '00 : Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 2–8, New York, NY, USA, 2000. ACM.

[Xavier Allamigeon and Charles Hymans, 2007] Xavier Allamigeon and Charles Hymans. Analyse Statique par Interprétation Abstraite. In Eric Filiol, editor, *5ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'07)*, Rennes, France, June 2007. To appear.

[Zalila *et al.*, 2008] B. Zalila, L. Pautet, and J. Hugues. Towards automatic middleware generation. In *11th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'08)*, pages 221–228, Orlando, Florida, USA, may 2008.

[Zalila, 2008] Bechir Zalila. *Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture*. PhD thesis, École Nationale Supérieure des Télécommunications, nov 2008.

