République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mentouri – Constantine
Faculté des Sciences de l'Ingénieur – Département d'Informatique

N° Ordre :

N° Série :

Une Approche Basée Architecture pour la Spécification Formelle des Systèmes Embarqués

Thèse

Doctorat en Sciences

Spécialité : Informatique

Par

Malika BENAMMAR

Soutenue le 06 mars 2011 devant le Jury composé de:

Mme Zizette BOUFAÏDA	Professeur, Université de Constantine	Président
Mr Med BENMOHAMED	Professeur, Université de Constantine	Examinateur
Mr Med Tahar KIMOUR	Maître de Conférences, Université d'Annaba	Examinateur
Mr Djamel MESLATI	Maître de Conférences, Université d'Annaba	Examinateur
Mme Faïza BELALA	Maître de Conférences, Université de Constantine	Rapporteur
Mr Kamel BARKAOUI	Professeur, CNAM, Paris France	Invité

Résumé

La complexité croissante de la technologie de l'embarqué fait que le processus de développement des systèmes embarqués doit suivre une démarche rigoureuse basée sur une sémantique formelle pour consolider la description structurelle des composants de ces systèmes et évaluer au plus tôt leur comportement. C'est dans ce contexte, que nous proposons une nouvelle approche, pour la conception architecturale et l'analyse formelle des systèmes embarqués temps réel, basée sur la logique de réécriture révisée. Nous nous sommes concentrés sur les descriptions architecturales offertes par le langage AADL, un ADL spécifique pour les systèmes embarqués temps réel, prenant en charge l'aspect dynamique et concurrent de ces systèmes et leur analyse temporelle, non considérés par le standard AADL. Pour ce faire, nous utilisons le cadre sémantique formelle de la logique de réécriture révisée pour étendre AADL avec une annexe baptisée ABAReL (**A**ADL **B**ehavioral **A**nnex based on revised **Re**writing **L**ogic). Celle-ci associe à chaque composant AADL, un modèle mathématique représenté par une théorie de réécriture étendue décrivant sa structure statique et son comportement. Nous proposons ensuite une extension de cette annexe pour la spécification et l'analyse des propriétés d'exécution temporelles. Le prototypage du modèle mathématique d'ABAReL sous RT-Maude, permet d'obtenir un modèle exécutable capable d'assurer la simulation du comportement et la vérification des propriétés à l'aide de l'outil LTL model checker du système RT-Maude.

Remerciements

El-Hamdou Li ALLAH le Tout Puissant pour m'avoir donné la force morale et physique pour achever cette thèse.

C'est avec gratitude que j'exprime ma profonde reconnaissance à ma directrice de thèse, le docteur Faïza BELALA. J'ai beaucoup appris à son contact, tant au point de vue scientifique en matière de rigueur et d'organisation que sur le plan humain. Qu'elle en soit ici, pour l'éternité, vivement remerciée.

Je tiens aussi à manifester ma profonde gratitude envers le professeur Kamel BARKAOUI, Responsable de l'équipe Vérification et Évaluation de Systèmes Parallèles et Asynchrones au laboratoire CEDRIC du CNAM Paris, de m'avoir si bien accueilli dans ce laboratoire à maintes reprises, pour m'avoir accordé de son temps précieux et m'avoir fait part de ses compétences et de son savoir faire.

Mes remerciements vont aussi à Mme Zizette BOUFAÏDA, Professeur à l'université Mentouri de Constantine, de m'avoir fait l'honneur de présider le jury de thèse.

Je remercie également Mr. M. BENMOHAMED, Professeur à l'université Mentouri de Constantine, Mr. M. T. KIMOUR et Mr. D. MESLATI, Maîtres de conférences à l'Université Baji Mokhtar d'Annaba, d'avoir accepté de juger ce travail.

Je tiens à remercier aussi les membres de l'équipe du laboratoire LIRE pour leur accueil, leurs conseils et leur gentillesse qui ont rendu ces années d'intenses efforts plus agréables.

Du fond du cœur, j'adresse tous mes remerciements à Salima Hacini et Samia Chikhi pour leur indéfectible et inaltérable amitié, malgré les kilomètres, à Nadia Chelali pour la délicate amitié dont elle m'honore, et à chaque membre de l'équipe GLSD pour le soutient moral qu'ils m'ont prodigué tout au long de la réalisation de cette thèse. Je remercie tout particulièrement Awatef Hicheur pour les passionnantes discussions que nous avons eu (au CNAM) sur le système RT-Maude, et qui m'en étaient plus d'une fois très précieuses.

Pour finir, je remercie bien sincèrement tous ceux qui m'ont encouragé, m'ont soutenu et, il faut bien l'admettre, m'ont supporté pendant toutes ces années. Spécialement mes enfants, et tout particulièrement mon époux.

Table des matières

Table des figures

Liste des tableaux

Introduction Générale

Sommaire

Les systèmes embarqués, connus dans le secteur automobile, aéronautique et militaire, touchent actuellement de nombreux autres domaines, par exemple les cartes à puce, les systèmes mobiles communicants (tels les téléphones mobiles, les agendas électroniques, etc.), les capteurs intelligents, la santé et l'électronique grand public. Ils apparaissent souvent comme des composants intégrés de systèmes complexes destinés à effectuer des tâches précises au niveau du dispositif qu'ils contrôlent. La conception et le développement d'un tel système diffèrent de ceux d'une application logicielle sur une plateforme standard du fait qu'ils possèdent des caractéristiques et des besoins qui les distinguent particulièrement des systèmes classiques. En effet, les systèmes embarqués sont soumis à des contraintes techniques strictes à la fois de performance fonctionnelles mais également temporelles, de consommation énergétique et de robustesse, qui pèsent sur eux du fait de leur taille et de leurs ressources limitées. Ces contraintes imposées par les besoins du système doivent être prises en compte dès les premières phases du cycle de développement. Généralement, les systèmes embarqués se composent de multiples sous-systèmes ou unités fonctionnelles qui peuvent être implémentés en utilisant une variété de composants logiciel et matériel combinés ensemble dans une plateforme d'exécution. Ces implémentations sont réalisées par une combinaison de flux de données et un contrôle orienté composants.

1 Contexte

Le défi pour pouvoir maitriser une telle complexité, au cours des différentes phases dans le processus de développement des systèmes embarqués, est sans doute de proposer une démarche rigoureuse basée sur une sémantique formelle pour consolider la description structurelle des différents composants de ces systèmes, évaluer au plus

tôt leur comportement et vérifier leurs propriétés d'exécution. Plusieurs approches de développement des systèmes embarqués ont été proposées dans la littérature dont certaines ont été appliquées dans l'industrie. L'approche classique suit une démarche informelle basée sur le prototypage et le débogage. L'approche synthèse niveau système concerne surtout les systèmes enfouis comme les systèmes on chip (SoC) où le développement est basé sur la synthèse d'un matériel dédié. L'approche de conception basée plateforme est essentiellement adoptée dans l'industrie des systèmes mobiles communicants par Sun Microsystems et son langage Java. L'approche basée modèle est beaucoup plus orientée vers la modélisation fournie par MARTE [OMG08], un profil UML spécifique pour les systèmes embarqués temps réel en tenant compte de leurs propriétés fonctionnelles et non-fonctionnelles. Néanmoins, avec UML il est difficile de disposer d'une vue globale du système à modéliser, il faut faire appel à plusieurs types de diagrammes.

D'un autre côté, l'architecture logicielle a émergé comme une notion centrale dans le génie logiciel. Sa principale caractéristique réside dans le fait qu'elle représente un modèle abstrait significatif de la structure et du comportement d'un système logiciel. Généralement, l'architecture logicielle d'un système définit son organisation comme une collection de composants, de connecteurs, et de contraintes d'interactions avec leurs propriétés additionnelles définissant le comportement prévu. De plus, les architectures logicielles permettent de valider très tôt les choix de conception en considérant les besoins fonctionnels ou non fonctionnels de l'application. Les langages de description d'architectures (ADL) offrent une manière formelle de spécifier les architectures logicielles des systèmes. AADL (Architecture Analysis and Design Language) [SAE04] est un langage de description d'architecture conçu particulièrement pour modéliser et analyser les systèmes embarqués temps réel. Celui-ci décrit l'architecture logicielle d'un tel système à différents niveaux d'abstraction. Il se distingue notamment par sa capacité à rassembler au sein d'une même notation l'ensemble des informations concernant l'organisation de l'application et son déploiement. Il offre plusieurs types de composants : les threads, les processus, les systèmes, les processeurs, les mémoires, les devices, et même les bus.

En outre, AADL est conçu pour être extensible afin de s'adapter à des analyses d'architectures d'exécution que le noyau du langage ne supporte pas complètement. Les extensions peuvent prendre la forme de nouvelles propriétés et de notations spécifiques d'analyse qui peuvent être associées à la description architecturale sous forme d'annexes. Les annexes permettent d'incorporer des éléments rédigés dans une syntaxe différente de celle d'AADL. Plusieurs annexes du noyau AADL, utilisant différents formalismes, ont été approuvées et publiées par le standard [FBV06]. Ces annexes sont : l'annexe des notations graphiques AADL, le méta-modèle AADL et l'annexe des formats d'échange XMI, l'annexe de conformité du langage et de l'Interface de Programme d'Application (pour Ada95 et C/C++), l'annexe du modèle d'erreur, le profil UML pour AADL, et l'annexe de modélisation du comportement.

Cependant, le standard AADL se focalise sur la description des aspects architecturaux tels que les composants et leurs connexions, mais ne traite pas directement de leur implantation comportementale, ni de la sémantique des données manipulées. Ainsi, le raisonnement formel sur une architecture AADL ou la vérification de ses propriétés est loin d'être envisageable par ce langage, d'où le besoin d'adopter des techniques d'analyse formelle pour décrire le comportement des composants AADL et leurs interactions.

Plusieurs tentatives de formalisation de la description architecturale AADL ont été publiées, pour établir une sémantique formelle pour les modèles AADL et pouvoir exécuter la simulation de leurs comportements et vérifier leurs propriétés. La plupart de ces formalisations sont couplées à une transformation de modèles AADL, soit dans un langage de modélisation intermédiaire dédié tels que : Fiacre [BBC09], IF [ACD+08], ou ATL [YHM+09] plus adaptés à l'analyse formelle, soit vers des langages possédant une sémantique formelle, comme BIP [Sif05] ou VTS [ABK+04], et plus adaptés aux outils de simulation et de vérification existants comme TINA [BRV04] ou CADP [GML+07]. Néanmoins, toutes ces contributions se limitent seulement à la formalisation de certains concepts AADL, en particulier, le comportement complexe des threads, cette unité d'exécution concurrente pour AADL, n'est pas formellement définie. D'où la nécessité d'un bon formalisme, en

l'occurrence la logique de réécriture, pour combler ces limites et offrir un cadre sémantique approprié pour spécifier et analyser le comportement complexe des threads et leurs interactions.

2 Objectifs

Nous exploitons l'aptitude de AADL à être étendu par des annexes pour lui définir une nouvelle annexe comportementale basée sur la logique de réécriture révisée, permettant d'associer une théorie de réécriture à un composant AADL afin d'en faire un modèle mathématique propre à son analyse ou sa vérification comportementale. ABAReL (AADL Behavioral Annex based on generalized Rewriting Logic) est le nom de cette annexe qui permet de décrire formellement la structure statique et le comportement d'un composant AADL. Nous nous intéressons principalement au type de composant thread qui représente le seul composant applicatif actif, et l'unité fondamentale d'exécution concurrente en AADL. Le standard AADL donne la possibilité de spécifier les conditions d'exécution des threads par la déclaration des propriétés tels que : le deadline, la politique d'expédition (`Dispatch_Protocol`), la période, …etc. La politique d'expédition implique que les threads peuvent être `Periodic`, `Aperiodic`, `Sporadic` ou `Background`. Le standard définit aussi, pour chaque thread, une sémantique dynamique à base d'automate décrivant ses états et les conditions de transition de ses états. Un `thread` dans un état donné, peut être stoppé, inactif, ou en activité. Un `thread` actif peut être en attente pour une expédition, ou en exécution. Dans son état d'exécution, le thread peut avoir d'autres sous états où il peut être prêt pour l'exécution, en exécution, ou bloqué sur l'accès d'une ressource. Cependant, un thread dans un système embarqué en exécution, reçoit des données et/ou des événements en entrée, il effectue le calcul et envoie des signaux (données et/ou événements) en sortie. Ceci, s'effectue au niveau de l'état d'exécution du thread, où particulièrement des changements de son état local et de l'état de chacun de ses ports de connexion, sont visibles à travers sa configuration.

Le premier objectif que nous nous sommes fixé concerne la définition d'un cadre sémantique approprié, basée sur la logique de réécriture révisée, permettant de formaliser la description architecturale et le comportement d'un composant AADL.

Nous nous concentrons particulièrement sur le composant thread et nous formalisons ensuite une architecture AADL d'un système embarqué temps réel composée de plusieurs threads en interaction. Cette formalisation s'effectue au niveau de l'annexe comportementale ABAReL que nous proposons pour AADL.

Le second objectif concerne l'extension du modèle mathématique formel obtenu d'ABAReL pour la description des propriétés AADL, un concept fondamental pour les descriptions de toute entité AADL. Cet enrichissement d'ABAReL permettra sans doute une meilleure évaluation du comportement des éléments architecturaux AADL du fait qu'il concerne par exemple les conditions d'exécution des threads, et de communication entre les threads. L'implémentation du modèle formel d'ABAReL sous RT-Maude offre un cadre sémantique approprié pour simuler et analyser le comportement des threads dans une architecture AADL. Les spécifications obtenues seront utilisés comme entrée de l'outil LTL model checker du système RT-Maude pour la vérification d'un ensemble de propriétés exprimées dans la syntaxe de la logique temporelle linéaire LTL.

3 Plan du mémoire

Dans le premier chapitre de cette thèse, nous introduisons d'abord les notions essentielles qui caractérisent le processus de développement des systèmes embarqués. Puis nous dégageons un état de l'art des principales approches de développement de ces systèmes recensées dans la littérature. Nous commençons par l'approche classique et nous exposons les différentes étapes du cycle de développement en se focalisant particulièrement sur la conception conjointe (co-design). Nous évoquons ensuite l'approche synthèse niveau système, l'approche conception basée plateforme et l'approche basée modèles. Une discussion autour de ces différentes approches a motivé le choix de la conception basée architecture.

Nous introduisons, dans le chapitre 2, les notions de base du langage de description d'architecture AADL. Nous nous concentrons particulièrement sur la description des aspects structurel et comportemental d'une architecture AADL. Nous

présentons ensuite un exemple illustratif d'une modélisation architecturale d'un système embarqué et nous montrons les manques et les limites de ce langage.

Dans le chapitre 3, nous présentons les principaux concepts du formalisme de la logique de réécriture, du système Maude et de l'outil RT-Maude qui nous utilisons comme environnement pour la spécification, l'exécution et l'analyse d'une description AADL.

Dans le chapitre 4, nous présentons notre principale contribution qui consiste à définir un cadre sémantique basé sur la logique de réécriture révisée, permettant d'intégrer les éléments architecturaux AADL dans le cadre formel de cette logique. Nous définissons ce support formel comme une annexe comportementale nommée ABAReL que nous intégrons au noyau du langage AADL. Cette annexe associe au composant `Thread`, unité d'exécution dynamique et concurrente, une théorie de réécriture orientée objet temps réel permettant de décrire la sémantique comportementale de ce composant. Elle permet en plus de raisonner correctement sur une architecture AADL composée de plusieurs threads en interaction.

Le chapitre 5 fait l'objet de l'étude des propriétés AADL et les possibilités d'extension d'ABAReL par ces propriétés. Nous évoquons l'importance des propriétés dans l'expression des différentes caractéristiques des composants. En effet, les propriétés font partie intégrante de la syntaxe AADL et complètent la description architecturale d'un système embaqué en fournissant toutes les informations descriptives des architectures. Nous montrons comment enrichir le modèle formel d'ABAReL par la spécification et l'analyse formelle de certaines propriétés AADL.

Le chapitre 6 est consacré à l'implémentation du modèle mathématique d'ABAReL sous RT-Maude. Le modèle exécutable résultant est ensuite utilisé pour l'analyse formelle d'une architecture AADL en bénéficiant des outils de l'environnement Maude. A travers une étude de cas, nous montrons, d'une part, comment nous pouvons simuler l'exécution concurrente de plusieurs threads en interaction tout en supervisant l'effet du passage du temps sur une connexion, et d'autre part comment vérifier formellement les propriétés d'exécution temporelle.

Finalement, un récapitulatif des principales contributions et des perspectives inhérentes à la poursuite de ces travaux est donné dans la conclusion générale.

Chapitre 1

Approches de Développement des Systèmes Embarqués

Sommaire

1.1 Introduction

L'avènement des microprocesseurs en 1970 a orienté le développement informatique vers un nouvel horizon. Il a élargie l'éventail des domaines de recherches et d'applications informatiques. Pour la première fois, des systèmes relativement complexes ont pu être construits en utilisant un dispositif simple: le microprocesseur [Ber02]. Initialement, les systèmes embarqués étaient des systèmes matériels et logiciels intégrés dans des navettes spatiales, des avions ou des trains. Par extension, les appareils portables ont été à leurs tours désignés comme des systèmes embarqués. Actuellement, les systèmes embarqués sont présents dans des applications de plus en plus nombreuses, par exemple les cartes à puce, les systèmes mobiles communicants (tels les téléphones mobiles, les mobiles dans les réseaux ad hoc), l'automobile, l'avionique, les capteurs intelligents, la santé et l'électronique grand public. Ils représentent un secteur important pour l'informatique, et chaque jour, notre vie en devient de plus en plus dépendante. La technologie de l'information digitale est désormais intégrée dans notre environnement quotidien.

L'objet de ce chapitre est de présenter le processus de développement des systèmes embarqués. D'abord, nous exposons les différentes étapes du cycle de développement classique de tels systèmes en se concentrant particulièrement sur la conception conjointe (co-design) hardware/software. Nous énumérons ensuite d'autres approches de développement, à savoir l'approche synthèse niveau système, l'approche conception basée plateforme, l'approche basée modèles et l'approche basée architecture.

1.2 Définition d'un Système Embarqué

On peut distinguer deux catégories de systèmes embarqués: les systèmes autonomes et les systèmes enfouis:

- Un système autonome correspond à un équipement autonome contenant une intelligence qui lui permet d'être en interaction directe avec l'environnement

dans lequel il est placé. Il s'agit des téléphones portables, agendas personnels électroniques ou GPS.

- Un système enfoui (souvent invisible à l'utilisateur) est un ensemble cohérent de constituants informatiques (matériel et logiciel), d'un équipement auquel il donne la capacité de remplir un ensemble de missions spécifiques [MC03]. Il s'agit d'un système physique sous-jacent avec lequel le logiciel interagit et qu'il contrôle.

Dans ce cas, les systèmes embarqués peuvent être vus comme des sous-systèmes de systèmes plus importants qu'ils sont chargés de commander et/ou de surveiller. Ils sont donc très étroitement liés à ces systèmes [MC03]. Ces derniers sont d'origines très diverses, mécanique, automatique, aéronautique, ferroviaires, nucléaire, télécommunication, etc. Il y a donc une informatique aéronautique, différente des informatiques ; automobile, spatiale, ferroviaire ou télé-communicante.

Tout ceci est spectaculaire, mais la question qui se pose est sans doute: comment peut-on maîtriser la complexité de ces systèmes au cours des phases de développement? D'autant que les systèmes embarqués sont soumis à des contraintes techniques très sévères qui pèsent sur eux incluant l'optimisation du code, d'énergie, d'espace, etc.

Une comparaison des systèmes embarqués avec notre PC est sans doute difficile à établir, vu la complexité et la dispersion de ces systèmes. Toutefois, sans que l'on puisse l'attester, une telle comparaison peut nous orienter, tant sur le plan technique des systèmes embarqués que sur le processus de développement de ses applications.

En quoi les systèmes embarqués diffèrent-ils de notre PC? Les systèmes embarqués sont dédiés à des tâches spécifiques, tandis que les PC ont des plateformes génériques. Ces systèmes ont de loin moins de ressources que les PC, et doivent souvent fonctionner dans des conditions environnementales extrêmes. Par exemple le cas des capteurs embarqués à bord des satellites. Les systèmes embarqués ont souvent des contraintes d'énergie et des contraintes temps-réel [Ber02]. En effet, les évènements temps-réel sont des évènements externes au système et ils doivent être

traités au moment où ils se produisent (en temps-réel). C'est pourquoi, si un système embarqué utilise un système d'exploitation, il doit probablement utiliser un système d'exploitation temps-réel RTOS, plutôt que windows, unix, etc.

Pourquoi le développement des systèmes embarqués est différent? Généralement, les systèmes embarqués sont supportés par une grande sélection de processeurs et architectures de processeurs, et stockent souvent tout leur code objet dans la ROM. Aussi, les défaillances logicielles sont beaucoup plus graves pour les systèmes embarqués que pour les applications sur PC. Pour cela, les systèmes embarqués exigent des outils spécialisés et des méthodes de conception plus efficaces.

Le développement des systèmes embarqués possède des caractéristiques et des besoins qui les distinguent particulièrement des autres systèmes. Significativement, les systèmes embarqués diffèrent des autres systèmes dans plusieurs aspects d'après [HT05], en exigeant:

- Un haut degré d'intégration des composants software et hardware.
- Un besoin temps-réel rigide.
- Un comportement très complexe, caractérisé par de nombreux scénarios compliqués.
- Un besoin en fiabilité et sécurité résistant, surtout pour les systèmes avec des missions critiques (safety-critical and/or mission-critical).
- Un traitement parallèle et souvent distribué.

Egalement, le développement de tels systèmes inclus l'utilisation de:

- Méthodes formelles pour la spécification, conception, vérification et test.
- Outils et méthodes de communication, synchronisation, etc.
- Co-conception (co-design) hardware/software.
- Langages de programmation spécialisés.
- Outils (des environnements) de développement spécialisés.

- Contraintes d'optimisation.

1.3 Développement Classique

1.3.1 Principe générale

À la différence du développement et de la conception d'une application logicielle sur une plateforme standard, la conception d'un système embarqué fait que le software et le hardware soient conçus en parallèle. En effet, le développement et la conception sont réalisés, comme le montre la figure 1, en décomposant et en assignant le système embarqué en software SW et hardware HW. Ainsi, on permet une conception séparé des composants software et hardware, et finalement l'intégration des deux. Le développement dans ce cas se fait selon les phases suivantes [Ber02, VR05]:

1. Spécification des besoins
2. Partitionner la conception des composants SW et HW
3. Itération et raffinement de ce partitionnement
4. Conception des tâches SW et HW séparément
5. Intégration des composants HW et SW
6. Production des tests et validation
7. Maintenance et revalorisation

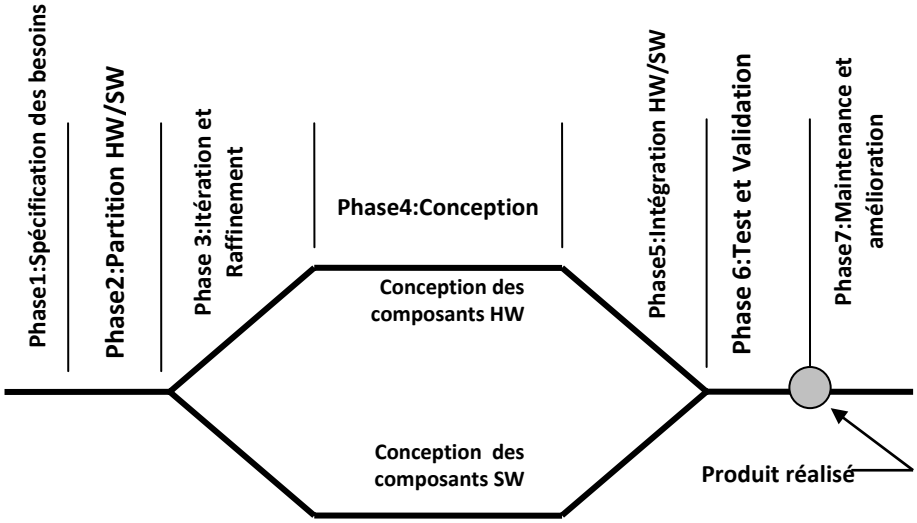


Figure 1.1 - Cycle de vie de conception embarquée [Ber02]

Spécification des besoins : Souvent, cette phase est établi selon le résultat d'un compromis entre le fabriquant du produit (système embarqué) et les concepteurs. En tenant compte d'une part des considérations de conception de tels produits, d'autre part des contraintes requises par les systèmes embarqués. Un facteur commun pour les produits réussis est que l'équipe de conception partage une vision commune du produit conçu. Beaucoup de produits échouent parce qu'il n'y avait pas une articulation cohérente des objectifs du projet [Ber02]. Souvent la phase de spécification des besoins demande le développement des outils de spécification, de génération de tests, et de validation.

Partitionnement : Puisque la conception embarquée implique les composants hardware et software, alors quelqu'un doit décider quelle partie du problème sera résolue dans le hardware et laquelle dans le software. Ce choix s'appelle la "décision du partitionnement". Décider comment partitionner la conception des fonctionnalités qui sont représentées par le hardware et le software est une partie principale de création d'un système embarqué. Le choix du partitionnement a un impact significatif sur le coût du projet, le temps du développement et le risque.

Itération et Implémentation : Cette phase du processus de développement représente un secteur brouillé entre l'implémentation et le partitionnement hardware/software dans lequel le chemin du hardware et celui du software divergent. La conception dans cette phase est délicate, bien que les principales parties à concevoir soient partitionnées entre les composants hardware et les composants software.

Conception conjointe hardware/software (Co-design): Avec le développement de la technologie embarquée, la conception moderne exige un concepteur qui a une vue unifiée du software et du hardware. Ces dernières années, la conception conjointe software/hardware "**Co-design**" émerge avec un développement constant de la technologie de conception. D'abord, une analyse globale du système est nécessaire, ensuite, elle est décrite dans un langage de programmation spécifique (généralement les langages C, C++ ou Java) malgré que chaque partie est implémentée dans le software ou le hardware et la simulation est exigé pour le partitionnement HW/SW approprié [DXW05]. Si la partition ne satisfait pas les besoins, une re-partition et une simulation peuvent être exécuté pour obtenir une meilleure partition.

Intégration software/hardware : Le processus d'intégration software et hardware embarqués est un exercice (souvent compliqué) de débogage, afin de prendre en charge les problèmes d'ambiguïté qui peuvent surgir. Dans certain cas, la conception doit combiner le premier prototype hardware, l'application software, le code du driver, et le logiciel système d'exploitation pour pouvoir tester cette intégration. Dans le cas des systèmes temps réel, ce scénario est si peu probable. La nature des systèmes embarqués temps réel mène à un comportement fortement complexe et non déterministe qui ne peut pas être analysé dans cette phase. Souvent on fait recours aux techniques de simulation pour simuler le comportement du système.

Test et Validation : Cette phase est généralement reléguée à une équipe distincte (les testeurs). Les conditions de test et de fiabilité pour les systèmes embarqués sont beaucoup plus important que la grande majorité des applications du bureau surtout s'il s'agit des systèmes de performances critiques. En plus, tester un système embarqué est

plus que s'assurer qu'il est performant, il doit aussi respecter les marges extrêmement serrées de conception en matière de coût et de contraintes de capacités optimales.

Maintenance et Amélioration : On trouve souvent une documentation bien rédigée pour les produits les plus commercialement développés. Mais, il n'y a pas un développement d'outils spécifiques aux produits déjà en service. Cependant, la majorité des concepteurs de systèmes embarqués maintiennent et améliorent les produits existants, plutôt que concevoir les nouveaux produits. Ils utilisent dans ce cas la documentation existante, et le vieux produit pour le maintenir et l'améliorer.

1.3.2 Commentaire et discussion

Ce processus de développement possède des caractéristiques particulièrement différentes de celui des systèmes classiques étant donné que la conception d'un système embarqué fait que le logiciel et le matériel soient conçus en parallèle. Donc, après les spécifications des besoins, le système est partitionné afin de permettre une conception conjointe (co-design) des composants software et hardware, et finalement l'intégration hardware et software.

Néanmoins, l'étude des différentes phases de ce cycle de développement nous a précisé qu'il suit une démarche informelle basée sur le prototypage et le débogage, et que certaines de ces phases sont brouillées, cas de la phase *itération implémentation*. Ils indiquent aussi qu'il y a une quantité considérable d'itération et de raffinement d'optimisation qui se produit dans les phases et entre les phases sans qu'on puisse détecter la méthode exacte de ces optimisations. Même si des développeurs arrivent à fabriquer certains produits en faisant recours à un prototypage et une simulation dans les phases de partitionnement pour confirmer le choix des composants, cette démarche reste très couteuse étant donné qu'elle n'est pas fondée sur des bases solides. La phase de test et validation occupe la majeure partie du coût de développement. Le fait de la déporter en dernier lieu rend ce coût exponentiellement élevé surtout s'il s'agit d'un débogage tardif dans le cycle de développement.

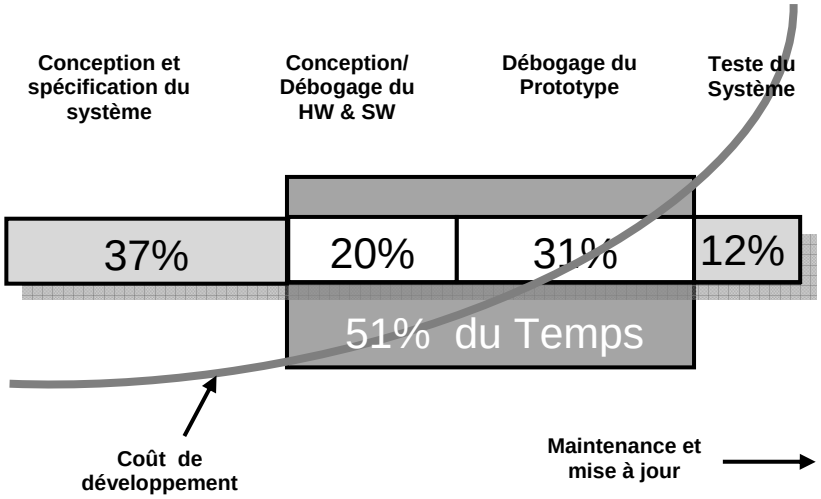


Figure 1.2 - Expansion du coût et du temps de conception

Avec la complexité croissante des systèmes embarqués, qui doivent obéir à des contraintes techniques très sévères, ce cycle de développement reste insuffisant pour satisfaire les besoins en matière de fiabilité et de performance de ces systèmes. La conception des systèmes embarqués exige beaucoup d'expertise et une vision globale des composants matériels et logiciels du système, puisque le tâtonnement ou le développement incrémentale sont des approches non faisables en électronique. Une fois le hardware est fait, il est très cher de le modifier.

1.4 Développement niveau système

Les systèmes embarqués sont généralement dédiés à une utilisation spécifique prévue dés les premières phases du processus de développement. Ils sont souvent utilisés pour n'effectuer qu'une seule tâche ou un ensemble fixe et bien défini de tâches. Ces spécificités orientent le concepteur à développer un système complet avec une architecture matérielle composée de ressources (processeurs, mémoires, bus de communication, etc.) et une partie applicatif exprimant le code des tâches du système dans un langage de programmation. Donc, le concepteur doit avoir une vue globale et unifiée de l'architecture matérielle et des composants logiciels. La complexité du processus de conception de tels systèmes est déterminée par la sémantique et la

syntaxe du langage de conception niveau système SLDL (System Level Design Language) adoptée pour véhiculer l'implémentation.

En général un SLDL exige deux attributs essentiels:

1) Il devrait supporter la modélisation à tous les niveaux d'abstraction.
2) Les modèles doivent être exécutables et simulables, de sorte que les fonctionnalités et les contraintes peuvent être validées.

L'objectif de la conception niveau système est de générer l'implémentation du système à partir de son comportement. Les approches de conception niveau système peuvent être classifiées d'après [TGP+05] dans trois groupes:

- la synthèse niveau système : cette approche concerne beaucoup plus les systèmes enfouis comme les systèmes en chip (SoC) ou cartes à puces.
- la conception basée plateforme : elle concerne essentiellement les systèmes embarqués autonomes comme par exemple les téléphones portables, les agendas électroniques, etc.
- la conception basée modèle.
- Et la conception basée architecture.

1.4.1 Synthèse niveau système

Dans la *synthèse niveau système*, la conception commence par décrire le comportement du système dans un langage haut niveau comme C, C++, SDL. Cette description est ensuite transformée vers une description structurelle sous un format qualifié par transferts de registres RTL (Register Transfer Level). Le niveau RTL utilise des langages de description de matériels (Hardware Description Langages ou HDL) comme VHDL, Verilog ou HardwareC [Fra01]. Ces langages de description intègrent la capacité d'exprimer le temps, par une ou plusieurs horloges, et la notion de concurrence matérielle. Ils sont différents des langages de description comportementale.

L'utilisation du langage de haut niveau C/C++, pour décrire le hardware et le software permet de garder la conception des activités hardware et software étroitement couplées [TGP+05]. Les deux langages de conception niveau système ou SLDL les plus utilisés dans l'ingénierie des systèmes embarqués sont: SystemC et SpecC.

SystemC fournit un langage de haut niveau commun pour modéliser, analyser et simuler un système embarqué. Il peut être également lié aux outils commercialisés tels que le compilateur de conception Synopsys, pour la synthèse du hardware. SystemC supporte aussi la modélisation hiérarchique du comportement du système. Dans ce cas, il isole respectivement la fonctionnalité et la structure dans des processus et des modules.

Le langage SpecC est une notation formelle destinée aux spécifications et à la conception des systèmes embarqués digitaux, y compris les parties hardware et software. SpecC supporte les concepts essentiels pour la conception des systèmes embarqués, incluant la hiérarchie comportementale et structurale, la concurrence, la communication, la synchronisation, les transitions d'états et les manipulations d'exception.

Quelque soit le langage utilisé, la conception dans ce cas doit essentiellement combiner la construction d'une architecture pour les calculs, les communications et le stockage du code et des données et la compilation de cette architecture, où la compilation sous entend la synthèse d'un matériel dédié [Fra01]. Dans cette approche, le compilateur devrait supporter l'accès convenable aux caractéristiques particulières du matériel cible en apportant les optimisations nécessaires pour la vitesse et la taille du code. Pour se faire, le compilateur doit connaitre les particularités de l'architecture cible pour en exploiter tout le potentiel, où il pourrait, par exemple, replacer des instructions, suivre des branches retardées et doit certainement se rendre compte de la structure particulière des registres du processeur [Fra01, Mig99].

1.4.2 Conception basée plateforme

Dans l'approche de la *conception basée plateforme,* le comportement du système est mappé sur une architecture prédéfini du système, au lieu d'être généré à partir du comportement comme dans l'approche de synthèse niveau système. Cette approche est adoptée, dans l'industrie des systèmes embarqués, par Sun Microsystems et son langage Java. Sun Microsystems propose l'environnement de spécification Java pour les systèmes embarqués, où la machine JVM (Java Virtual Machine) est adaptée aux besoins de chaque plateforme par le développeur de systèmes [IN05].

Ce processus de développement doit être suivi, comme le montre la figure 1.3, par plusieurs phases de compilation, simulation, débogage avant d'être déployé en ROM (ou Romization) [CPG+06].

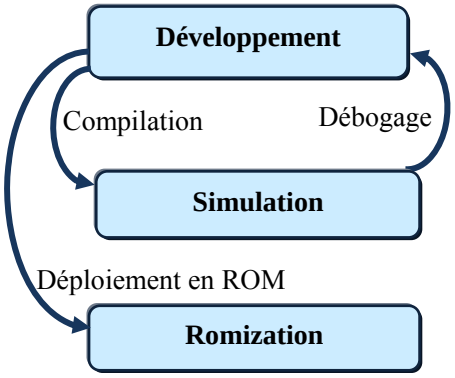


Figure 1.3 - Cycle de vie des logiciels embarqués

La compilation Java en code natif est une pratique courante dans le domaine de l'embarqué. Il existe aussi des compilateurs Java qui compilent directement le Java en code objet natif pour la machine cible, comme par exemple GCJ (GNU Compiler for Java). Pour la phase de simulation, les concepteurs du hardware peuvent utiliser des outils de simulation, tel que les simulateurs architecturaux pour modéliser la performance du microprocesseur. Les concepteurs du software exécutent probablement le code benchmark sur l'ordinateur single-board utilisant le microprocesseur cible. Ces ordinateurs single-board évaluent la performance du microprocesseur en lui faisant subir plusieurs tests.

Dans l'industrie des systèmes embarqués, la romization est le processus par lequel un système logiciel est pré-déployé par un outil spécifique le « romizer » utilisé pour instancier le programme initial d'un dispositif [CGV05]. Le romizer crée une image mémoire appropriée pour le dispositif cible qui contient le système avec les composants déployés dessus. L'image mémoire produite par le romizer est complètement prête à fonctionner et mappable à la mémoire physique du dispositif.

1.4.3 Conception basée modèle

L'approche basée modèle, qualifie aussi par le développement dirigé par les modèles MDD (Model Driven Development), place le modèle au centre du processus de développement des systèmes en général. Cette approche est beaucoup plus orientée vers la modélisation fournie par UML [OMG99]. Comme standard de l'OMG, UML offre un langage de modélisation sous forme d'un ensemble d'annotations graphiques et textuelles permettant de couvrir toutes les étapes du cycle de développement d'un système.

UML est reconnue par son caractère extensible grâce aux notions de *stéréotype* et *profil*. Un *profil* adapte UML aux besoins d'un domaine spécifique. On peut le présenter comme un métamodèle d'un domaine particulier. Un *profil* est exprimé en utilisant un ensemble cohérent de *stéréotypes* avec des valeurs étiquetées (*tagged values*) associées qui sont utilisées via des *contraintes*. Le *profil* peut être accompagné par une librairie de modèles spécifique au domaine d'intérêt. Dans le cas de développement des systèmes embarqués, le profil UML pour la modélisation et l'analyse des systèmes embarqués temps réel MARTE [OMG08] remplace et étend le profil UML pour la spécification de l'ordonnançabilité, la performance et le temps *SPT* [OMG05]. Cette extension concerne surtout de nouvelles annotations et des techniques d'analyse permettant la modélisation et l'analyse des aspects matériels et logiciels des systèmes embarqués temps réel en tenant compte de leurs propriétés fonctionnelles et non-fonctionnelles tels que la puissance d'énergie ou la capacité de la mémoire, etc.

Le profil MARTE [OMG08] est structuré autour d'un noyau nommé TCRM (Time, Concurrency and Resources Modeling), décrivant des constructions de modélisation pour le temps, les ressources, la concurrence, l'allocation et les propriétés non-fonctionnelles NFPs [EMD+06]. Le modèle des propriétés non fonctionnelles (NFP) fournit des constructions pour une modélisation généralisée de ces propriétés, incluant le temps et l'utilisation des ressources. Le modèle générique des ressources GRM permet de modéliser une plateforme générale pour exécuter des applications embarquées en temps réel. Le modèle du domaine de temps identifie un ensemble de concepts et de sémantiques relatifs au temps. Dans ce cas, le temps peut être perçu différemment dans les phases de développement (conception, analyse de performance, analyse d'ordonnançabilité, implémentation, etc.) d'un système embarqué temps réel.

TCRM est ensuite raffiné pour donner deux sous-profils : RTEM (Real-Time and Embedded Modeling) et GQAM (Generic Quantitative Analysis Modeling). RTEM modélise les caractéristiques de développement des systèmes embarqués temps réel en définissant des annotations standard pour décrire les caractéristiques des ressources. GQAM procure une base générique pour des sous domaines d'analyse quantitative différents. Un domaine d'analyse, dans ce cas, représente la description de la façon dont le comportement d'un système utilise les ressources [EMD+06]. Cette phase est d'une importance fondamentale et implique un certain nombre de décisions de conception. L'analyse concerne aussi l'ordonnancement et la performance. L'analyse de l'ordonnancement permet de prédire les contraintes temporelles pour un ensemble de tâches logicielles. L'analyse de performance détermine si un système avec un comportement non-déterministe peut aboutir à la performance adéquate.

1.4.4 Commentaires et discussion

La compilation classique d'un programme est la traduction de sa description écrite dans un premier langage ou langage source en un programme équivalent écrit dans un autre langage ou langage cible. La compilation, dans l'approche de synthèse niveau système, est une transformation du programme en préservant sa sémantique. Le travail du compilateur est de transformer le programme source en un programme optimisé en tenant compte des paramètres de chaque composant de l'architecture cible

[Fea00, DGQ+02]. Les techniques d'optimisations concernent le parallélisme de données, le parallélisme de flot, l'ordonnancement des instructions, la transformation de boucles, etc. Ces transformations sont très utiles pour une optimisation de la mémoire et par conséquent du code stocké. La transformation de boucle inclue la *fusion* et la *distribution*, le découpage en sous-blocs appelé *pavage* ou *tiling*, le *décalage d'instructions*, l'*échange* (ou *permutation*) et l'*inversion* [Ris00]. Les problèmes de transformations de boucles et les combinaisons de transformations sont le plus souvent posés sous forme d'équations linéaires et résolus par des programmes linéaires [QRW00, Fea96, Ras00].

Le concept de machine virtuelle Java JVM permet la portabilité où le code Java est totalement portable à condition qu'il existe une machine virtuelle sur la cible. Il facilite ainsi le développement, parce que le langage Java permet de développer plus vite une application, et qu'une grande partie du code peut être développée et testée indépendamment de la cible. Cependant, Java présente également des limitations pour le développement d'applications critiques. La portabilité théorique du langage Java se heurte au masquage du matériel et des couches basses (par exemple : le RTOS) issues de fournisseurs différents. La préconisation sur la qualité de service des plateformes Java pourrait permettre de réduire cette contrainte. En plus, une JVM nécessite plusieurs centaines de Koctets pour permettre le fonctionnement d'une application Java. Des techniques de compilation "intelligentes" ou de précompilation ont permis de réduire de manière significative les ressources mémoires nécessaires.

MARTE définit les constructions d'un langage seulement mais ne couvre pas les aspects méthodologiques pour offrir une démarche à suivre pour la modélisation. Le profil souffre de l'absence d'une sémantique formelle des modèles UML décrivant le comportement du système. Il est aussi difficile de disposer d'une vue globale du système à modéliser, il faut faire appel à plusieurs types de diagrammes. Les contraintes d'optimisation, dans MARTE, sont spécifiées par les propriétés non-fonctionnelles, incluant le temps et l'utilisation des ressources, nécessaires pour différentes sortes d'analyses quantitatives. Les sous-profils d'analyse de l'ordonnaçabilité et de performance permettent juste de faire des prévisions de conception.

1.5 Conception basée architecture

L'approche basée architecture ou l'adoption de la description architecturale joue un rôle important dans le développement de logiciels complexes et maintenables à moindre coût et avec respect des délais. Notons que l'architecture logicielle d'un système définit son organisation, à un niveau élevé d'abstraction, comme une collection de composants, de connecteurs, et de contraintes d'interactions avec leurs propriétés additionnelles définissant le comportement prévu. Les méthodes formelles s'imposent pour appréhender la complexité et la fiabilité de ces systèmes [Fei04]. Elles entraînent le développement de techniques d'analyse de systèmes et de génération automatique de tests à partir de leurs spécifications. Cela permet d'avoir une évaluation des comportements possibles d'un système au plus tôt, ce qui réduit à la fois le temps et les coûts de validation, et augmente la fiabilité du système. Pour se faire, nous avons besoin d'un formalisme capable de décrire les composants, les interactions entre les composants, et de spécifier des propriétés non fonctionnelles concernant la configuration, le déploiement …etc.

Les langages de description d'architectures (ou ADL) offrent une manière formelle de spécifier les architectures logicielles des systèmes. Un ADL est un langage informatique utilisé pour décrire les composants logiciels, ou les composants logiciels et matériels, d'un système et les interactions entre ces composants. Il peut décrire les interfaces fonctionnelles des composants (flots de contrôle et de données) et les aspects non fonctionnels (aspects temporels, sûreté, fiabilité). Cette description formelle est par la suite utilisée pour analyser les comportements possibles du système en question et de vérifier ces propriétés. Comme exemples d'ADL, nous pouvons citer : Wright [AG96], Rapid [Luc95], et ACME [GMW97]. Ces langages sont classés dans la catégorie des ADLs formels puisqu'ils supportent les notions de composants et de connecteurs de manière abstraite sans indiquer à quoi ils correspondent dans le système réel. Il y a aussi une autre catégorie de langages de description, il s'agit des ADLs concrets. Ces langages reflètent la réalité mieux que les langages formels. Ils spécialisent la notion de composants en introduisant plusieurs catégories correspondant chacune à une entité du monde réel (logiciel et matériel). Ils sont également mieux adaptés pour la génération automatique d'un système à partir de

ces modèles. Parmi les ADLs concrets, nous trouvons UNICON [Zel96], MetaH [Ves98] et par la suite AADL [SAE04].

6. Conclusion

Le développement des systèmes embarqués implique la conception d'un système dans lequel les ressources sont habituellement limitées, et lequel peut devoir s'exécuter sans intervention manuelle. En effet, les systèmes embarqués sont soumis à des contraintes techniques strictes à la fois de performance fonctionnelles mais également temporelles, de consommation énergétique et de robustesse, qui pèsent sur eux du fait de leur taille et de leurs ressources limitées. Ces contraintes imposées par les besoins du système doivent être prises en compte dés les premières phases du cycle de développement. Par exemple la plupart des systèmes embarqués doivent avoir tout leur code dans la ROM, avec des limitations sévères qui sont imposées sur la taille de ce code dans l'espace ROM. Comme ces ressources sont soumises à des contraintes, la conception des systèmes embarqués exige leurs optimisations.

Dans ce chapitre, nous avons présenté différentes approches de développement pour les systèmes embarqués. Ces approches s'accordent sur le fait que le développement de tels systèmes doit permettre une conception conjointe des composants software et hardware et doit prendre en considération les contraintes d'optimisation. Cependant, la façon de considérer ces contraintes diffère d'une approche à l'autre.

Dans l'approche classique, le système est construit de différents composants individuels, ce qui fournit une énorme quantité de différentes possibilités pour l'exécution. Si les tests indiquent que les performances du produit sont insuffisantes, alors les algorithmes doivent être réécrits, on dit qu'on fait une reconception. Le développement, dans ce cas, est basé sur le prototypage et le débogage comme il fait recours à la simulation pour confirmer le partitionnement et le choix des composants. Les optimisations des ressources sont traitées informellement, selon les cas, dans les phases et entre les phases de développement du système embarqué.

Donc, la conception des systèmes embarqués nécessite un niveau de fiabilité important. Ce besoin en fiabilité et en complexité exige l'utilisation de méthodes et langages de haut niveau pour garantir la sécurité des développements. Pour ce faire, le processus de développement a été orienté vers les approches de conception niveau système. La conception niveau système a pour objectif de générer l'implémentation du système à partie de son comportement. Nous avons présenté trois approches : l'approche synthèse niveau système, la conception basée plateforme et la conception basée modèle.

Dans l'approche synthèse niveau système, les optimisations concernant la vitesse de traitement et la taille du code sont faites au niveau de la compilation. Le compilateur, dans ce cas, doit supporter l'accès convenable aux caractéristiques particulières du matériel cible. Par contre dans l'approche de conception basée plateforme, le comportement du système est mappé sur une architecture cible. L'approche est adoptée par Sun Microsystems et son langage Java et utilise les compilateurs Java pour compiler le code source en code natif. Dans cette approche, ils élaborent souvent des expérimentations pour optimiser ce code nativement compilé par une suppression des exceptions d'exécution.

Cependant, les méthodes de conception et de vérification des systèmes embarqués sont souvent moins développées que pour le logiciel « classique ». Toutefois, la conception de ces systèmes demande des méthodes rigoureuses et fiables pour s'adapter à des contraintes en temps et coût de développement toujours plus critiques. L'adoption de la description architecturale dans le processus de développement d'un système embarqué permet de définir son organisation complète. À un plus haut niveau, la spécification d'une architecture décrit comment les composants sont combinés en sous-systèmes, comment ils interagissent (structure des flots de données et de contrôles), et comment ils sont alloués sur les composants matériels.

Le langage AADL (Architecture Analysis and Design Language) est un ADL concret permettant de spécifier une telle architecture et de générer automatiquement un système. Il a été développé pour satisfaire les besoins spéciaux des systèmes

embarqués temps réel. AADL permet également par la déclaration de propriétés d'enrichir les descriptions par des aspects fonctionnels (pointeur vers du code source ou le temps d'exécution d'un thread) et non fonctionnels (la période ou la politique de dispatch). Ce langage constitue la base de travail de cette thèse, il sera détaillé dans le chapitre suivant.

Chapitre 2

AADL : Langage de Description d'Architecture dédié aux Systèmes Embarqués

Sommaire

2.1 Introduction

C'est dans les années 70, lorsque les systèmes logiciels ont commencé à franchir un certain seuil de complexité que l'idée de conception architecturale, au sens d'une activité de conception séparée de la conception détaillée, a émergé. La notion d'architecture logicielle, en tant que perspective haut niveau d'un système logiciel, n'apparaît réellement qu'à partir des années 90 et est alors présentée comme le cœur d'une discipline à part entière. Depuis, la définition d'une architecture logicielle est devenue une étape importante dans la conception des systèmes. Elle permet de prendre en compte les descriptions haut niveau de systèmes complexes et de raisonner sur leurs propriétés. Donc, un langage de description est nécessaire pour pouvoir décrire le plan architectural à différents niveaux d'abstraction, allant d'un modèle très abstrait jusqu'à l'implantation du système sur une architecture plus concrète. MetaH [Ves98] est le premier ADL, dédié particulièrement aux systèmes avioniques. AADL [SAE04] est un langage de description d'architecture qui résulte du langage MetaH. Il s'agit d'un langage de description d'architectures destiné aux systèmes embarqués ayant des contraintes de ressources strictes (taille, poids, et puissance), et qui sont soumis à des besoins de réponse temps réel. AADL permet la conception et la description d'architectures temps réel embarquées, en considérant simultanément tous les aspects logiciel et matériel sous-jacents. La description d'une architecture en AADL consiste en la description de ses composants et leur composition sous forme d'une arborescence. De multiples propriétés permettent de considérer les caractéristiques des composants tels que : les propriétés temporelles, flux de données, modes de fonctionnement, etc.

Dans ce chapitre, nous présentons le langage de description d'architecture AADL en commençant par son historique et sa description générale. Nous exposons ensuite les principes généraux qui permettent la description d'une architecture en AADL. Nous introduisons les différents concepts d'AADL, les phases de description d'un composant (type et implantation) à travers des exemples, et les éléments d'interfaces et connexions pour décrire les interactions entre les composants. Nous achevons par un exemple et une conclusion.

2.2 Historique

MetaH a été développé par Honeywell depuis 1991 pour le DARPA (Defense Advanced Research Projects Agency) et AMCOM (Aviation and Missile COMmand of US Army). Aussitôt, un ensemble conséquent d'outils (éditeurs graphiques, analyseurs temporels, d'ordonnancement, de charge, générateurs de code...) ont été prototypés et utilisés dans le contexte de plusieurs projets expérimentaux utilisant le langage MetaH [Ves98].

En 2001, MetaH a été pris comme base pour un effort de standardisation visant à définir AADL *(Avionique Architecture Description Language)*, un langage de description d'architectures avioniques, sous l'autorité du SAE *(Society of Automotive Engineers)*. Ce langage a été développé pour répondre aux besoins spéciaux des systèmes embarqués temps-réel tels que les systèmes avioniques. En 2003, le comité de standardisation a décidé de changer la signification des initiales du nom du langage AADL *(Architecture Analysis & Design Language)* pour ne pas le limiter au domaine de l'avionique. Notons que ce comité de standardisation comporte les principaux acteurs de l'industrie avionique et aérospatiale, en Europe comme aux USA (Honeywell, ESA, Airbus, Rockwell Collins, EADS, etc.).

En novembre 2004, AADL est standardisé au niveau international par la SAE et une première version, AADL 1.0, a été publiée [SAE04]. Celle-ci définit la syntaxe textuelle et la sémantique du noyau d'un langage extensible. Dés lors, le développement des extensions AADL reste un point ouvert de recherche, permettant d'introduire des annexes et des propriétés additionnelles qui nécessitent de nouvelles approches d'analyse.

En mai 2006, plusieurs annexes du noyau AADL, utilisant différents formalismes, ont été approuvées et publiées [FBV06]. Ces annexes sont : l'annexe des notations graphiques AADL utilisée pour modéliser graphiquement des architectures avec AADL, l'annexe méta-modèle AADL et format d'échange XML/XMI pour la représentation abstraite et le format d'échange des modèles AADL,

l'annexe d'interface de programmation d'application définissant le mapping d'AADL aux langages de programmation Ada95 et C/C++, le profil UML pour AADL, qui est devenu ensuite une base pour MARTE (Modeling and Analysis of Real-Time Embedded systems) et aussi l'annexe du modèle d'erreur associé au noyau des composants AADL.

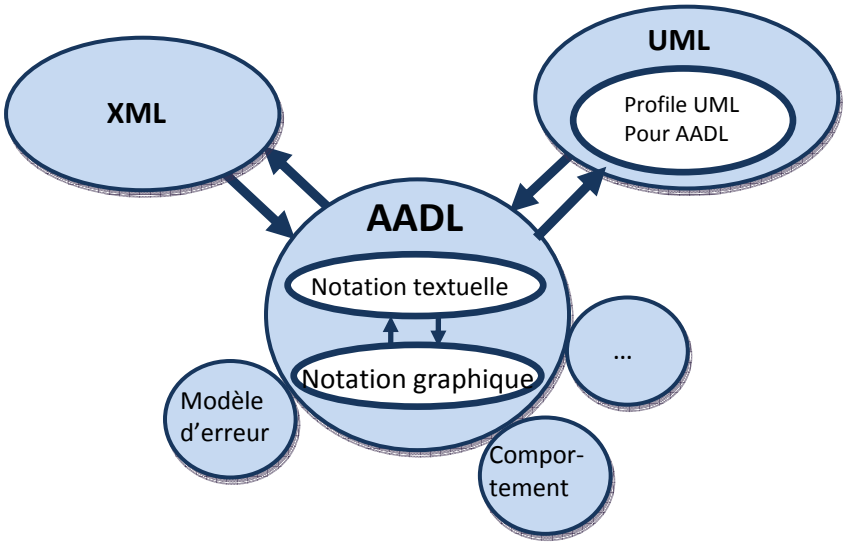


Figure 2.1 - Description générale du langage AADL

En janvier 2009, le standard publia la version 2.0 du langage AADL incluant l'annexe comportementale. Cette annexe a été formulée dans le cadre du projet COTRE par [DBF+06]. Elle offre des descriptions dynamiques du comportement des composants AADL en utilisant un langage d'automates approprié.

2. 3 Présentation générale du langage AADL

AADL est un langage textuel et graphique destiné à la modélisation des architectures logicielles et matérielles des systèmes embarqués temps réel et leurs caractéristiques de performances critiques. Il utilise des concepts de modélisation formels pour la description et l'analyse des architectures d'un système d'application en termes de composants distincts en interactions. Ainsi, le modèle AADL d'un système embarqué est présenté comme un assemblage de composants logiciels mappé sur une plate forme

d'exécution. AADL décrit les interfaces fonctionnelles des composants (tels que les données en entrée et en sortie), comment les composants sont combinés (c'est-à-dire comment les données en entrée et en sortie sont connectées ou bien comment les composants logiciels sont alloués aux composants matériels). Il décrit également les mécanismes de flux de données et des flux de contrôles utilisés dans les systèmes embarqués et les aspects non fonctionnels tels que des conditions de synchronisation, les comportements par défaut ou sur erreur, le partitionnement de temps et d'espace et les propriétés de sûreté et de certification.

AADL dispose d'un certain nombre d'outils développés dans le cadre des projets Européen ASSERT (Automated proof-based System Software Engineering for Real-Time systems) et COTRE (COmposants Temps REel) par l'initiative TOPCASED (Toolkit in Open-source for Critical Application & Systems) [Gau05]. L'outillage de TOPCASED a pour but d'intégrer les résultats des recherches académiques dans le processus de développement industriel. Il contribue à la simplification des langages de modélisation, comme UML2 et AADL, en fournissant des technologies du méta-niveau telles que des générateurs d'éditeurs syntaxiques (textuels et graphiques), des outils de validation statique et d'exécution des modèles [CRC+06]. Pour le langage AADL, les outils de modélisation textuelle et graphique, de simulation et de vérification sémantique sont fondés sur l'environnement OSATE (Open Source AADL Tool Environnement) [Fei05] basé sur la plateforme Eclipse.

En plus des outils TOPCASED, il existe aussi des logiciels de simulation du comportement, d'ordonnancement et validation d'architecture AADL, parmi eux nous pouvons citer : CHEDDAR [SLN+04], ADeS (Architecture Description Simulation) [AxG02] et Furness toolset.

2.4 Les éléments architecturaux de base

2. 4.1 Composants

Le langage AADL modélise l'architecture d'un système embarqué par une hiérarchie de composants, dont l'interaction est représentée par des connexions. Donc, les

composants sont les éléments de base d'une architecture AADL. Un composant AADL, selon le standard, représente une entité matérielle ou logicielle qui appartient au système en cours de modélisation. Un composant possède un type (`component type`), qui spécifie son interface, et une ou plusieurs implantations (`component implementation`). Le type d'un composant est constitué de trois parties : les éléments d'interface (`features`), les flots (`flows`) et les propriétés (`properties`). Cette spécification est utilisée par les autres composants du système pour interagir avec ce composant. Une implantation, par contre, décrit la structure interne du composant en termes de sous composants (`subcomponents`), connexions (`connections`) entre les éléments d'interface de ces sous-composants, et les propriétés (`properties`). Dans ce cas, le composant est généralement décrit sous forme d'un assemblage de sous-composants qui sont des instances de types ou d'implantation d'autres composants. La spécification d'une implantation englobe aussi les connexions qui lient les sous-composants, les flux (`flows`) traversant les sous-composants, les modes (`modes`) pour représenter les états opérationnels, et les propriétés qui spécifient les caractéristiques structurelles et/ou comportementales de ce composant.

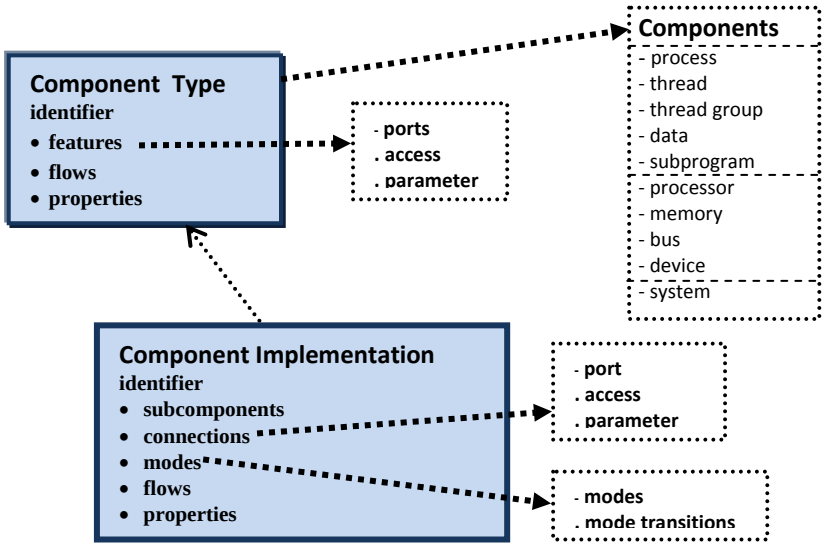


Figure 2.2 - Description des éléments architecturaux AADL

Chaque composant AADL appartient à une catégorie. Chaque catégorie a une représentation graphique. Ces catégories sont prédéfinies et réparties, comme le montre la figure 2.3, en trois grandes familles.

1) Les composants matériels décrivent des éléments de la plateforme d'exécution (processeurs, mémoires, etc.).
2) Les composants logiciels définissent les entités logicielles qui forment une application (processus, threads, groupe de threads, sous-programmes, etc.).
3) Les composants composites servent à hiérarchiser une description AADL. Ils permettent de regrouper différents composants en entités logiques pour structurer l'architecture.

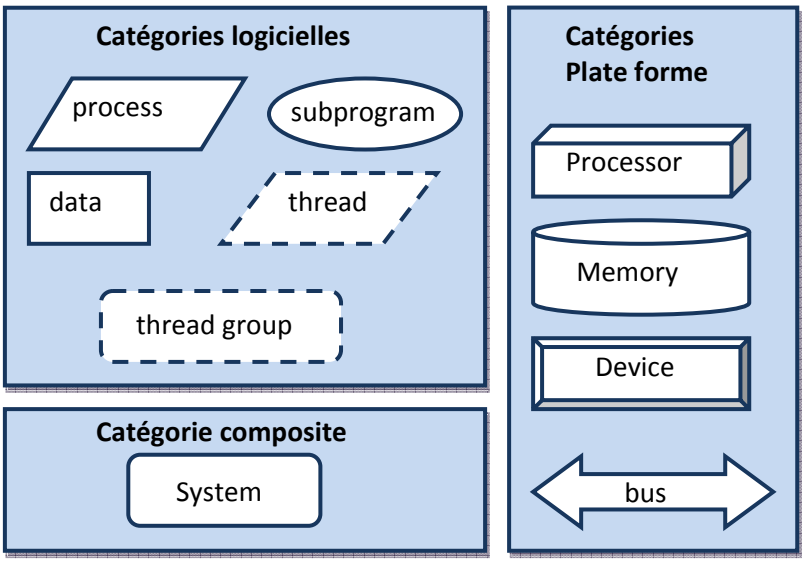


Figure 2.3 - Catégories de composants

Les composants logiciels : sont répartis en cinq catégories : les *processus*, les *threads*, les *groups de threads*, les *sous-programmes* et les *données*. Les *processus* sont modélisés en utilisant le composant `process`. Ils définissent des espaces mémoire pour contenir les threads et les données partagées entre eux. Les *threads* sont les éléments applicatifs actifs. Ils représentent les unités d'exécution concurrente et peuvent être comparés aux processus légers tels que définis dans les systèmes d'exploitation. Ils sont modélisés à l'aide du composant logiciel `thread`. Dans le cas

où de nombreux threads d'un système possèdent des caractéristiques proches et pour éviter la duplication de code, AADL introduit les *groupes de threads* pour décrire des tâches partageant un certain nombre de propriétés. Les *groupes de threads*, modélisés par le composant `thread group`, permettent d'introduire une hiérarchie entre les threads. Les *sous-programmes* modélisent, avec le composant `subprogram`, les procédures telles que définies dans les langages de programmation impératifs. Et les *données* représentent les structures de données qui peuvent être stockées ou échangées entre les composants. Elles sont modélisées en utilisant les composants `data` et spécifient des types de données, lorsqu'elles sont utilisées dans les éléments d'interfaces. Elles représentent des variables partagées lorsqu'elles sont instanciées sous forme de sous composants.

Les composants matériels : appartiennent à quatre catégories : les *processeurs*, les *mémoires*, les *bus* et les *dispositifs*. Les *processeurs* sont spécifiés à l'aide du composant `processor`. Celui-ci modélise un microcontrôleur plus un noyau constitué d'un système d'exploitation minimal (ordonnanceur, pilotes…). Les *mémoires* sont modélisées avec le composant `memory` qui représente une mémoire physique quelconque (disque dur, mémoire vive…). Elles servent à stocker les données et le code et sont accessibles par les threads en cours d'exécution. Les *bus* sont modélisés par l'intermédiaire du composant `bus`. Celui-ci représente toutes sortes de réseaux ou de bus, depuis le simple fil jusqu'à un réseau de type Internet. Le rôle du bus est de transporter les communications entre processeurs, mémoires et dispositifs. Si des connexions sont associées à un bus, celui-ci doit véhiculer les flots de données ou de contrôle entre les entités logicielles du système reliées par ces connexions. Les *dispositifs* sont décrits à l'aide des composants `device`. Ils modélisent une large variété de matériel allant des simples capteurs jusqu'aux appareils les plus complexes. Dans un modèle, un dispositif représente un élément dont la structure interne est ignorée ; seules son interface est visible pour les autres composants du système. L'ensemble de ces composants matériels permettent de définir l'architecture matérielle d'une application.

Les systèmes : ou composants composites permettent de regrouper différents composants en entités logiques pour former des blocs et faciliter ainsi la structuration de l'architecture. Contrairement aux autres composants, le composant système, décrit par l'intermédiaire du composant `system`, ne représente pas une entité concrète. Dans les premières phases de la modélisation, les systèmes servent pour remplacer les composants dont la nature n'a pas encore été décidée par le concepteur.

Structure interne des composants : où un sous composant est une instance d'une déclaration de composant (type ou implantation). Certains composants peuvent avoir des sous composants décrits dans la description de l'implantation. Ces mêmes sous-composants peuvent aussi contenir d'autres sous-composants pour former l'architecture de l'application. Cette hiérarchie de composants a comme racine un composant `system` contenant toutes les instances qui constituent un modèle AADL. Pour structurer cette hiérarchisation des composants, le standard a établit des règles sémantiques précises interdisant à des catégories particulières de composants de contenir certains catégories comme sous-composants. Donc, seules les combinaisons ayant une sémantique cohérente sont autorisées. Ces règles sont résumées dans le tableau suivant.

Composant	**Sous-composant**
system	System, processor, memory, bus, device, process, data
processor	memory
memory	memory
bus	—
device	—
process	Thread, thread group, data
thread	data
thread group	thread, thread group, data
subprogram	—
data	data

Tableau 2.1- Règles de composition des composants AADL

Ainsi, le standard interdit à un composant ***process*** d'être un sous composants du composant `processor`. En réalité, un programme est exécuté par un processeur,

mais n'est pas intégré dedans. Dans un modèle AADL, ces deux composants sont reliés par une relation d'attachement par le biais des propriétés. Comme il interdit au composant `data` de contenir un `process` ou encore un `processor` car ceci ne correspond pas à un assemblage réel. Nous pouvons constater également que les `subprograms` ne sont jamais instanciés, puisqu'un `subprogram` n'est pas une entité en soi, il représente juste une séquence de code stockée dans l'espace mémoire d'un programme. Ces règles sémantiques obligent certaines catégories de composants de contenir des sous-composants d'autres catégories : ainsi, un `process` doit contenir au moins un `thread`. Les composants `systems` peuvent contenir toutes les catégories de composants sauf les `subprograms` et les `threads` et `threads group`, qui doivent appartenir à un `process`.

2.4.2 Instance de système

Les instances de système sont créées et stockées comme des modèles d'instances de système. Une instance de système se compose de composants logiciels d'application et des composants de la plateforme d'exécution. Les déclarations de type et d'implantation de composants sont les plans architecturaux qui définissent la structure et la connectivité d'une architecture du système physique. Ils doivent être instanciés pour créer une instance du système complet. Une instance du système qui représente sa hiérarchie physique est créée par instanciation de l'implantation du système au plus haut niveau suivi par des instanciations récursives des sous-composants et leurs sous-composants. Une fois instancié, les instances des composants d'application peuvent être liées aux composants de la plateforme d'exécution. Par exemple, chaque `thread` doit être lié à un `processor` ; chaque texte source et composant `data` est relié à une mémoire et chaque connexion est relié à un `bus`. Les modèles obtenus peuvent être exploités par des outils d'analyse.

Exemple 1:

```
thread T_thread
end T_thread;

process P_process
end P_process;

process implementation P_process.imp
   subcomponents
        thread1: thread T_thread;
        thread2: thread T_thread;
end P_process.imp;
```

Figure 2.4- Exemple d'instances de composants

2.4.3 Interaction des composants

Les représentations des interactions entre les composants AADL se limitent aux connexions définies au niveau d'un modèle et établis entre les éléments d'interface. Les éléments d'interface, modélisés par les caractéristiques ou `features` d'un composant, représentent la partie visible par les autres composants du modèle. Ces éléments sont des entités nommées à travers lesquels s'effectue l'échange des données et des signaux. Il existe plusieurs sortes de `features` : les ports de connexion, les appels aux sous-programmes, et les accès aux sous-composants.

Les ports de communication

Les points de connexion logique entre les composants sont : des ports entrant (`in ports`), des ports sortant (`out ports`), ou des ports bidirectionnels (`in out ports`). En AADL, un port est une entité nommée qui représente une interface de communication pour échanger des données et/ou des événements entre les composants. Les ports sont répartis selon trois types : le port donné (`data port`), le port événement (`event port`), et le port événement-donnée (`event data port`). Les ports d'événement (`event ports`) correspondent à l'émission d'un signal. Ils peuvent déclencher l'exécution des threads. Les ports de données (`data ports`) correspondent à la transmission de données. Ils peuvent modéliser un registre mémoire mis à jour de façon asynchrone. Les ports événement-données (`event data`

ports) permettent de transporter des données tout en générant un événement à la réception.

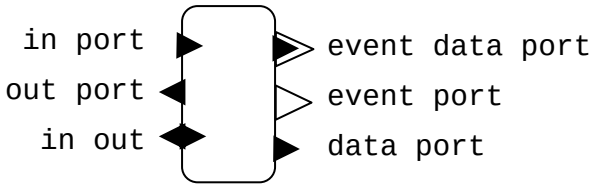


Figure 2.5 - Représentation graphique des ports

Les groupes de ports

Le standard définit la notion de groupe de ports (port groups) pour permettre de regrouper des ports et d'autres groupes de ports. Le but est de faciliter la manipulation des ports de connexion qui entrainent une déclaration répétée des ports. Cette notion permet aussi de ne pas multiplier les ports pour représenter, par exemple, un bus parallèle. Ainsi, les groupes de ports sont utiles pour réduire le nombre de connexions et par conséquent simplifier la représentation graphique du modèle AADL. La déclaration d'un groupe de ports est similaire à celle d'un type de composant.

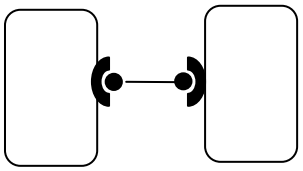


Figure 2.6 - Connexion par des groupes de ports

Les appels aux sous-programmes

Un sous-programme, qui représente un texte source séquentiellement exécutable, est un composant accessible par un appel avec ou sans paramètres. Les appels aux sous-programmes sont déclarés dans des séquences d'appels dans un thread ou dans les implantations d'un sous-programme. Ces appels peuvent inclure le transfert de données vers ou du sous-programme. Les paramètres, déclarés comme des caractéristiques (features) d'un sous-programme, fournissent l'interface pour le

transfert de données dans ou d'un sous-programme. Un sous-programme peut fournir des fonctions de serveur aux composants qui l'appellent, il modélise dans ce cas un sous-programme serveur (invocable à distance) par un support d'appel distant de type RPC (Remote Procedure Call).

Accès aux sous-composants

Ce sont des déclarations explicites des éléments d'interface (`features`) qui permettent un accès à un sous-composant spécifique. Il s'agit des types d'accès aux sous-composants : pour les bus et les données. Dans les deux cas, ils sont déclarés comme étant des accès fournis (`provides`) ou requis (`repuires`).

Dans le cas d'un composant de données, l'accès est soit `read_only` ou `write_only`. Il permet de modéliser une variable partagée : le composant de donnée est instancié une fois et manipulé par plusieurs composants. L'accès à un bus permet de représenter le partage d'un bus entre différents composants matériels, modélisant ainsi l'interconnexion entre les processeurs, les dispositifs et les mémoires.

2.5 Connexions des composants

Les connexions spécifient le chemin d'un flux (`flow`) de contrôle et/ou de données entre différents composants au moment de l'exécution. Une connexion sémantique est composée par une suite de connexions au niveau des composants et sous-composants qui sont traversés. Chaque connexion sémantique a une ultime source et une ultime destination. Les ultimes sources et destinations peuvent être des composants `thread` ou `device`. La description des flux (`flows`) de données et de contrôles entre composants se fait par le moyen de ports et de connexions. Un port de connexion est un point d'entrée et/ou de sortie d'un composant, par où peuvent transiter des données (`data`), des événements (`event`) ou même des événements associés à des données (`event data`).

Composant	Port de connexion
system	Sous-programme serveur, port, groupe de ports, accès requis ou fourni à une donnée ou un bus
process	Sous-programme serveur, port, groupe de ports, accès requis ou fourni à une donnée
thread	Sous-programme serveur, port, groupe de ports, accès requis ou fourni à une donnée
processor	Sous-programme serveur, port, groupe de ports, accès requis à un bus
memory	accès requis à un bus
bus	accès requis à un bus
device	port, groupe de ports, sous-programme serveur, accès requis à un bus

Tableau 2.2 - Les ports de connexion possibles pour chaque catégorie de composants

Une connexion permet de relier deux ports, soit les ports de deux sous-composants, soit le port d'un sous-composant avec le port du composant le contenant, avec une vérification de conformité de type et de sens entre les ports connectés. Les ports de connexion diffèrent d'une catégorie de composant à une autre, nous les résumons dans le tableau 2.2.

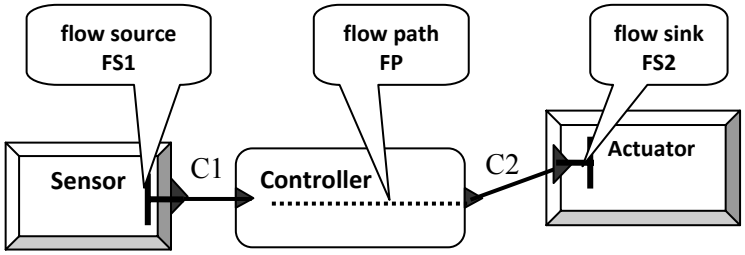
2.5.1 Description des flux

Les descriptions de flux décrivent des flux de contrôle et de données dans une architecture. Celles-ci indiquent que l'information logique du flux provient des ports de connexion sortant du composant de départ du flux, pour aller vers un port de connexion entrant du composant destinataire. Le composant de départ du flux est désigné comme une source du flux (`flow source`), et le composant d'arrivée est appelé puits du flux (`flow sink`).

```
process Position
   features
    InitCmd: in event port;
    Signal: in data port signal_data;
    Result1: out data port position.X;
    Result2: out data port position.Y;
    Status: out event port;
   flows
    Flow1 : flow path Signal -> Result1;
    Flow2 : flow path Signal -> Result2;
    Flow3 : flow sink InitCmd;
    Flow4 : flow source Status;
end Position;
```

Figure 2.7- Description AADL des flux

Les flux de bout en bout (end to end flows) : décrivent un flux complet. Celui-ci peut se décomposer, comme le montre la figure 2.8, en une source (`flow source`), des tronçons (`flow path`) et un puits (`flow sink`) qui sont assemblés en fonctions de la composition des composants.



Représentation graphique d'un flux de bout en bout

```
system system_A
end system_A;

system implementation system_A.Impl1
  subcomponents
    Sensor : device capteur_S;
    Controller : system controller.impl;
    Actuator : device actuator_T;
  flows
    flowC : end to end flow Sensor.FS1 -> C1 ->
        Controller.FP -> C2 -> Actuator.FS2 ;
end system_A.Impl1;
```

Figure 2.8 - Description AADL d'un flux de bout en bout

Pouvoir spécifier des flux de bout en bout (`end to end flows`), pour décrire un flux complet, permet des analyses temporelles, de latence, de fiabilité, de propagation d'erreur, de qualité de service, etc.

Les implantations de flux : Une implantation de flux permet d'expliciter la structure interne d'un tronçon d'un flux (`flow path`). Elle spécifie la source (`source`), le puits (`sink`) et le chemin (`path`) qui font intervenir des ports de connexion (`features`), des connexions et des flux traversant les sous-composants.

2.6 Description du comportement

Le langage AADL ne se limite pas à la description statique d'une architecture, il est possible d'introduire un certain dynamisme dans les modélisations architecturales AADL. La description du comportement dans AADL se fait par la déclaration des modes opérationnels et des transitions de mode.

2.6.1 Les modes et les transitions de mode

Les modes permettent de décrire des modes de fonctionnement de l'architecture. Ils sont déclarés dans les implantations des composants et représentent les états opérationnels du logiciel, de la plateforme d'exécution, et des composants compositionnels dans le système physique modélisé. Ceux-ci s'énoncent comme des configurations des composants et des connexions entre ces composants. Donc, l'ensemble de threads considérés active, c'est-à-dire prêt à répondre aux expéditions, et les connexions qui sont disponibles pour transférer des données et des évènements entre ces threads, représentent *le mode courant* du système. Un changement de mode peut changer l'ensemble des composants actifs et connexions. Dans ce cas, le changement de comportement est spécifié comme un diagramme d'états transitions, où les états sont des modes et les transitions sont déclenchées par les événements.

```
thread execution_thread
end execution_thread;

process a_process
  features
    multi_thread : in event port;
    mono_thread : in event port;
end a_process;

process implementation a_process.impl
   subcomponents
    thread1 : thread execution_thread;
    thread2 : thread execution_thread in mode (multitask);
  modes
    monotask: initial mode ;
    multitask: mode ;
    monotask -[ multi_thread ]-> multitask;
    multitask -[ mono_thread ]-> monotask;
 end a_process.impl;
```

Figure 2.9 – Description AADL des modes et des transitions de mode

Ces événements proviennent d'un port d'entrée (`event port`) d'un composant. Elles peuvent avoir l'effet d'activation ou désactivation des *threads* pour l'exécution, un changement du chemin de connexion entre les `threads`, et des changements des caractéristiques internes des composants. A un moment donné, un seul mode peut être actif et seuls les `threads` actifs peuvent exécuter leur code.

2.6.2 Les propriétés

Dans cette section, nous montrons l'importance des déclarations de propriétés dans la description architecturale AADL d'un système embarqué et le rôle qu'elles jouent dans la description de certains aspects du comportement des composants AADL. Une description plus détaillée des propriétés AADL et leur rôle dans l'analyse formelle de l'architecture AADL d'un système embarqué sera donnée dans le chapitre 5.

Les propriétés permettent une grande souplesse et peuvent être associées à quasiment tous les éléments architecturaux d'une description. Elles jouent un rôle très important dans la description du comportement et constituent donc un aspect fondamental pour AADL. À chaque composant on peut associer des propriétés et leur donner des valeurs. Il est donc possible de décrire des contraintes s'appliquant à

l'architecture par la déclaration de propriétés. Par exemple, la propriété `Propagation_Delay` permet de déclarer le temps de propagation d'un signal à travers un `bus`, et la propriété `Processing_Speed` spécifie la vitesse de traitement d'un processeur.

```
thread Thr1
end Thr1;

thread implementation Thr1
    properties
      Dispatch_Protocol => Periodic;
      Period => 100 ms;
      Compute_Deadline => value(Period);
      Compute_Execution_Time => 20 ms;
      Source_Text => "waypoint.java";
      Source_Code_Size => 12 KB;
      Source_Data_Size => 5 KB;
end Thr1;
```

Figure 2.10 – Déclaration de propriétés pour un thread AADL

De façon similaire, la description AADL de chaque thread peut contenir la déclaration des propriétés d'exécution temporelles (voir figure 2.10) comme : `Dispatch_protocol`, `Period`, `Compute_Deadline` et `Compute_Execution_Time` pour décrire ses condtions d'exécution. Il peut également contenir d'autres propriétés dont les valeurs permettent d'identifier les fichiers associant un code source au thread comme: `Source_Text`, `Source_Code_Size`, `Source_Data_Size`.

2.7 Exemple d'une architecture AADL : *description du système ABS*

Dans cette section, nous présentons le résultat d'une modélisation architecturale où nous avons exploité les notations et les techniques du langage AADL dans [BBF+09], à travers les outils de l'environnement OSATE, pour modéliser l'architecture du système ABS (Anti-lock Brake System), un exemple concret d'un système embarqué. Cette modélisation suit un processus de raffinement de l'architecture logicielle AADL abstraite au départ, pour générer un modèle composé d'un certain nombre de

composants AADL de catégories différentes, permettant de décrire la structure et le comportement du système ABS.

2.7.1 Principe de fonctionnement du système ABS

Le rôle du système antiblocage ABS [HM04] est d'empêcher le blocage des roues en cas de freinage brusque ou freinage sur un revêtement glissant en permettant au conducteur de conserver le contrôle et la stabilité de son véhicule. Il permet aussi de réduire les distances de freinage particulièrement sur une route mouillée. Pour ce faire, le *calculateur électronique* (voir figure 2.11) déclenche un cycle de contrôle, dés que le véhicule roule à une vitesse supérieure à 8 km/h, pour vérifier entre autre les *capteurs de vitesse des roues* et le *groupe hydraulique*. Ce cycle de contrôle est très court, il s'effectue en une fraction de seconde pour surveiller en permanence tous les composants nécessaire au bon fonctionnement du système ABS. Lorsque le conducteur appui sur la *pédale de freins*, le maître cylindre de frein transforme ce mouvement en pression hydraulique. Cette pression se diffuse alors par l'intermédiaire d'un liquide incompressible jusqu'aux roues pour les bloquer dans le but de stopper le véhicule.

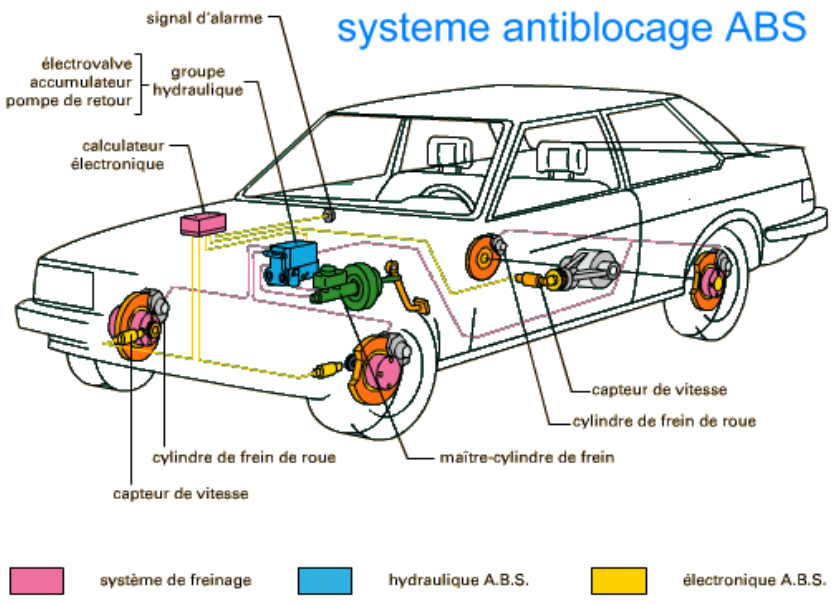


Figure 2.11 - Composants du système antiblocage ABS dans une voiture

Cependant, dés que les roues ne tournent plus mais glissent sur le sol, le ralentissement n'est plus contrôlé et perd de son efficacité, puisqu'il ne dépend que des frottements entre les roues et le sol. Dès lors, le système ABS entre en action pour empêcher le blocage des roues. Pour ce faire, la vitesse de rotation de chaque *roue* est mesurée par des *capteurs de vitesse*. Elle est transmise en permanence au système de contrôle de l'ABS via des impulsions électriques.

Dès qu'une roue amorce un blocage, le système d'antiblocage ABS réduit momentanément la pression hydraulique des étriers de freins. Il diminue ainsi la pression hydraulique des étriers de freins pour les relâchés et les roues sont libérées. Dès que la roue commence à reprendre de la vitesse, le système ABS augmente temporairement la pression hydraulique des étriers de freins. Le système ABS répète ce cycle autant de fois qu'il est nécessaire pour assurer un freinage efficace.

2.7.2 Description AADL de la structure du système ABS

La modélisation architecturale d'un tel système commence par la déclaration de type du composant *systeme_ABS*.

```
system systeme_ABS
end systeme_ABS;
```

Cette déclaration représente un modèle très abstrait où ce composant de catégorie `system` peut être vu comme une boite noire. La déclaration AADL de l'implantation de ce composant (figure 2.12) spécifie sa structure interne qui peut être décomposée en un ensemble de sous composants interconnectés.

Le composant `controleur_abs` de catégorie `system` représente le noyau du système ABS. Il modélise le calculateur électronique où s'exécutent les commandes de freinage. Ce composant est lié en entrée à des capteurs, modélisés par des composants de la catégorie `device`, pour détecter les états du moteur (*capteur_moteur*), de la pédale de frein (*capteur_pedale*), et de la vitesse des roues (*capteur_roue*). Il effectue des calculs selon le principe de fonctionnement du

système ABS et ses lois spécifiques de commande, et produit des valeurs de contrôle en sortie qui affectent le groupe hydraulique au niveau du composant `actateur_frein` et l'affichage de l'état du système ABS au niveau du composant `temoin_abs`.

```
system implementation systeme_ABS.SystemImpl1
  subcomponents
    controleur_abs: system controleur_abs.impl;
    capteur_moteur: device capteur_moteur;
    capteur_pedal: device capteur_pedal;
    capteur_roue: device capteur_roue;
    capteur_vitesse: device capteur_vitesse;
    temoin_abs: device temoin_abs;
    actuateur_frein: device actuateur_frein;
connections
    DataConnection1: data port capteur_roue.vitesse_r ->
controleur_abs.vitesse_r { Latency => 10Ms; };
    DataConnection2: data port capteur_vitesse.vitesse_V ->
controleur_abs.vitesse_V { Latency => 10Ms; };
    DataConnection3: data port controleur_abs.etat_abs ->
temoin_abs.etat_abs {
      Latency => 10Ms; };
    EventConnection1: event port capteur_pedal.etat_pedal ->
controleur_abs.etat_pedal { Latency => 10Ms; };
    EventConnection2: event port capteur_moteur.etat_moteur ->
controleur_abs.etat_moteur { Latency => 10Ms; };
    EventDataConnection1: event data port controleur_abs.cmd_frein
-> actuateur_frein.cmd_frein { Latency => 10Ms; };
end systeme_ABS.SystemImpl1;
```

Figure 2.12 - Description AADL de l'implantation du composant *system_ABS*

La représentation graphique de ce système est donnée par l'image d'écran de la figure 2.13.

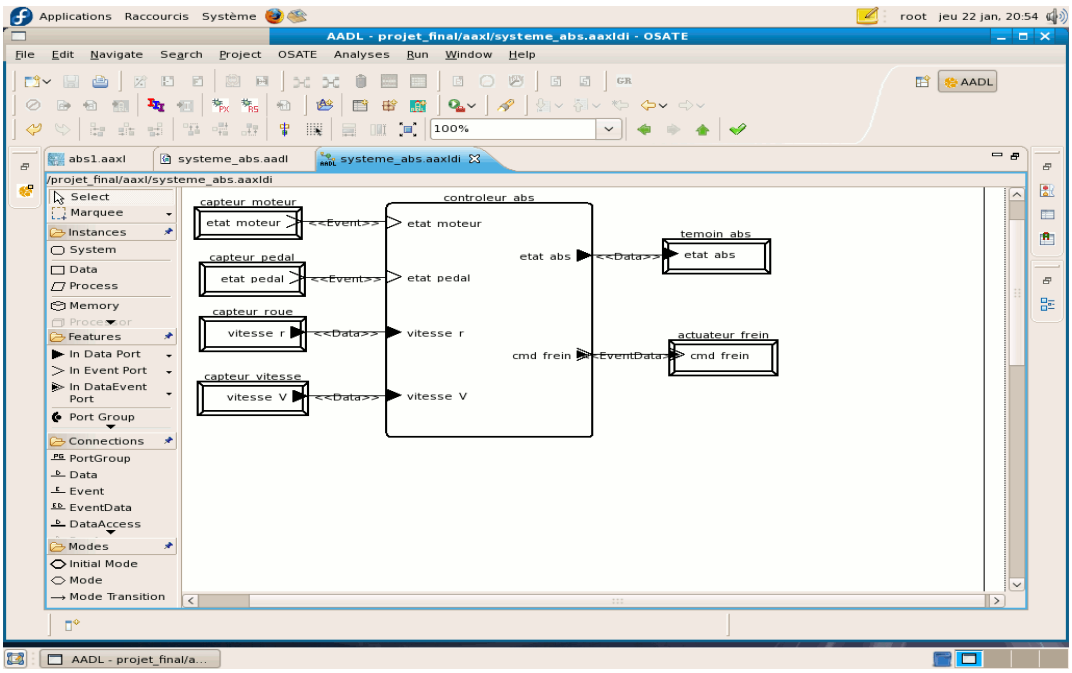


Figure 2.13 - Modèle graphique AADL du système ABS

Elle montre les sous composants du composant `system_abs` et leur interconnexion. Chaque sous composant représente une instance d'une catégorie de composant qui peut être déclarée à son tour par un type et une ou plusieurs implantations. Par exemple, la déclaration de type du composant `capteur_moteur` (figure 2.14) montre qu'il transmet un évènement au composant `controleur_abs` via son port de sortie `etat_moteur` pour indiquer l'état actuel du moteur.

```
device capteur_moteur
  features
    etat_moteur: out event port;
  flows
    flot1: flow source etat_moteur { Latency => 10Ms; };
  properties
    Device_Dispatch_Protocol => Background;
end capteur_moteur;
```

Figure 2.14 - Description AADL du composant *capteur_moteur*

Le rôle du composant `controleur_abs` est la régulation de freinage pour empêcher le blocage des roues d'un véhicule en état de marche dans le cas d'un

freinage brusque ou d'un freinage sur un revêtement glissant. Il doit d'abord vérifier les valeurs mesurées par les capteurs comme la vitesse de rotation des roues, l'état du moteur, et l'état de la pédale de frein. Ensuite, il contrôle l'action de freinage pour éviter le blocage des roues. Dans la figure 2.15, nous décomposons le composant `controleur_abs` en deux sous composants de catégorie `process` nommés respectivement `tester_entrees` et `actionner_frein`. Le premier teste les entrées : `etat_moteur` et `vitesse_v` et le deuxième calcule le taux de glissement et active le régulateur de pression pour éviter le blocage des roues en cas de freinage.

```
system implementation controleur_abs.impl
  subcomponents
    tester_entrees: process tester_entrees.ProcessImpl1
    actionner_frein: process actioner_frein.ProcessImpl2;

  connections
    DataConnection1: data port vitesse_V ->
                     tester_entrees.vitesse_V_in;
    EventConnection1: event port etat_moteur ->
                     tester_entrees.etat_moteur;
    EventConnection2: event port etat_pedal ->
                     actionner_frein.etat_pedal;
    DataConnection2: data port vitesse_r ->
                     actionner_frein.vitesse_r;
    DataConnection3: data port tester_entrees.vitesse_V_out ->
                     actionner_frein.vitesse_V_out;
    EventConnection3: event port tester_entrees.vitesse_V_ok ->
                     actionner_frein.vitesse_V_ok;
    EventDataConnection1: event data port
                     actionner_frein.cmd_frein -> cmd_frein;
    DataConnection4: data port actioner_frein.etat_abs2 ->
                          tester_entrees.etat_abs2;
    DataConnection5: data port tester_entrees.etat_abs -> etat_abs;
end controleur_abs.impl;
```

Figure 2.15 - Implantation du composant *controleur_abs*

La description AADL de l'implantation du processus `actionner_frein` (figure 2.16) est ensuite raffinée en utilisant deux threads nommés respectivement `calcul_glissement` et `calcul_cmd_frein` pour décrire les tâches effectuées par ce processus. Ce processus `actionner_frein` est activé par l'arrivée de

l'évènement `etat_pedale` si la pédale de frein est enfoncée. Ceci activera alors ses deux threads. Le thread `calcul_glissement` utilise la valeur de vitesse du véhicule (`vitesse_v`) et celle de la rotation des roues (`vitesse_r`) pour calculer le taux de glissement. Le thread `calcul_cmd_frein` reçoit la donnée `taux_glissement` et l'événement `etat_pedale` sous forme d'impulsions électriques indiquant la pression de la pédale de frein. Il calcule la pression hydraulique des étriers de frein pour donner une valeur sous forme de voltage électrique (2Volt pour maintenir la pression et 5Volt pour relâcher la roue). Cette valeur est transmise ensuite, au composant `actuateur_frein`, via le port data event `cmd_frein` pour régulariser le freinage et empêcher le blocage des roues. Le processus `actionner_frein` reste actif jusqu'à ce que le conducteur relâche la pédale de frein ou la vitesse du véhicule devient inférieure à 8 km/h.

```
process implementation actionner_frein.ProcessImpl2
  subcomponents
    calcule_glissement: thread calcule_glissement.ThreadImpl1;
    calcule_cmd_frein: thread calcule_cmd_frein.ThreadImpl1;
  connections
    EventConnection1: event port etat_pedal ->
                      calcule_glissement.etat_pedal;
    EventConnection2: event port vitesse_V_ok ->
                      calcule_glissement.vitesse_V_ok;
    DataConnection1: data port vitesse_V_out ->
                     calcule_glissement.vitesse_V_out;
    DataConnection2: data port vitesse_r ->
                     calcule_glissement.vitesse_r;
    EventDataConnection1: event data port
                     calcule_glissement.taux_glissement ->
                     calcule_cmd_frein.taux_glissement;
    EventDataConnection2: event data port
             calcule_cmd_frein.cmd_frein -> cmd_frein;
    DataConnection3: data port calcule_glissement.etat_abs2 ->
                     etat_abs2;
end actioner_frein().ProcessImpl2;
```

Figure 2.16 - description de l'implantation du composant *actionner_frein*

Le processus `tester_entrées` est composé à son tour en trois threads nommés respectivement `test_etat_moteur`, `afficher_etat_abs` et `test_vitesse`. Ce processus est activé par l'arrivée de l'évènement `etat_moteur` si le moteur est en marche. Le thread `test_etat_moteur`

détecte les valeurs de contrôle du système ABS. Il envoie ensuite le résultat sous forme «`data`» au thread `afficher-etat_abs`. Ce dernier prépare les données à afficher pour les envoyer périodiquement au composant `temoin_abs` pour afficher l'état du système ABS au conducteur.

Notre modélisation couvre également les composants de la plateforme d'exécution et ceci par la déclaration de deux processeurs nommés `cpu1_abs` et `cpu2_abs` où s'exécutent respectivement les calculs des deux processus `tester_entrées` et `actionner_frein`, un bus de nom `bus_abs` pour conduire les flux de contrôle et de données à travers les différentes connexions, et une mémoire nommée `sdRam` pour le stockage du code et des données. La figure 2.17 montre le modèle architectural raffiné du système ABS.

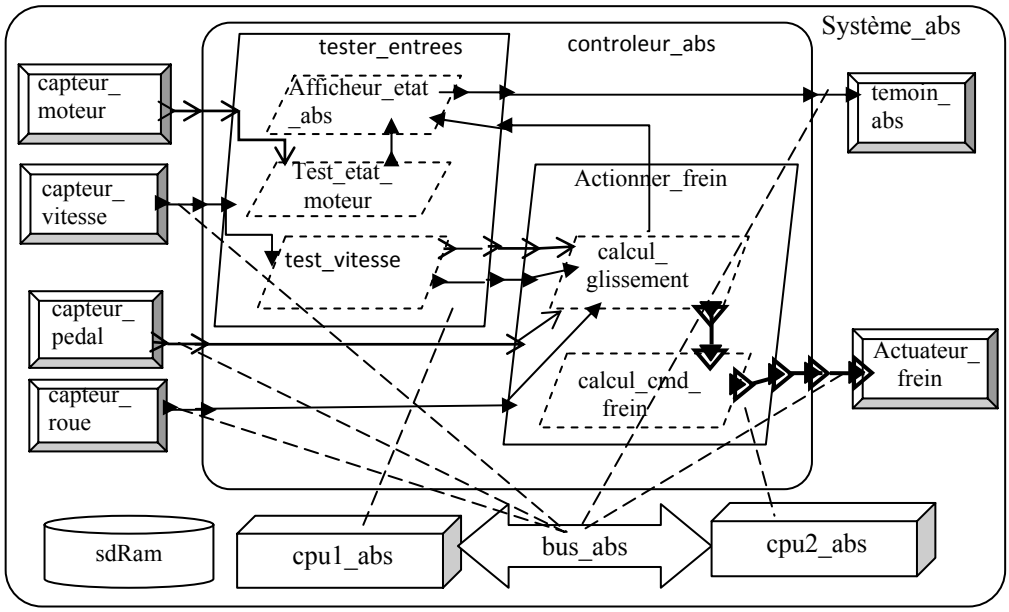


Figure 2.17 - Architecture AADL du système ABS

2.7.3 Description AADL du comportement du système ABS

Les états du système représentent les différentes configurations d'exécution où chaque configuration peut être déclarée comme un `mode` AADL et le passage d'une configuration d'exécution à une autre représente la transition de `mode`. Donc, l'identification des états du système et par conséquent la déclaration des `modes`

correspondant se fait selon les différentes configurations d'exécution du système. A cet effet, le système `controleur_abs` est à l'*état prêt* pour fonctionner si le moteur du véhicule est mis en marche (`mode Prêt`). Il est à l'*état actif* si la *pédale de frein* est enfoncée et la *vitesse des roues* est supérieure à 8 km/ h (le `mode Execution`) et à l'*état arrêt* si le *moteur* arrête de fonctionner (`initial mode`). Ces `modes` sont déclarés dans l'implantation du composant `controleur_abs`.

```
system implementation controleur_abs.impl
  subcomponents
   tester_entrees: process tester_entrees.ProcessImpl1 in modes
       (Pret , Execution);
   actionner_frein: process actioner_frein.ProcessImpl2 in modes
(Execution);
  modes
    Initiale: initial mode ;
    Pret: mode ;
    Execution: mode ;
    Initiale -[ etat_moteur ]-> Pret;
    Pret -[ etat_moteur ]-> Initiale;
    Pret -[ etat_pedal ]-> Execution;
    Execution -[ etat_pedal ]-> Pret;
end controleur_abs.impl;
```

Figure 2.18 - Description des modes d'exécution du système ABS

Le standard offre un mécanisme de propriétés pour déclarer les caractéristiques essentielles des composants. Pour les threads, la déclaration des propriétés permet de spécifier leurs conditions d'exécution et peut être utilisée pour simuler l'exécution des tâches modélisées au niveau des threads.

```
thread implementation test_vitesse_V.ThreadImpl1
  modes
    pret: mode ;
    execution: mode ;
    pret -[ etat_abs2 ]-> execution;
    execution -[ etat_abs2 ]-> pret;
  properties
    Dispatch_Protocol => Periodic ;
    Period => 14 ps in modes (pret);
    Period => 18 ps in modes (execution);
    Compute_Execution_Time => 2 ps .. 2 ps in modes (pret);
    Compute_Execution_Time => 3 ps .. 3 ps in modes (execution);
    Allowed_Processor_Binding_Class => processor cpu_abs.Processor
end test_vitesse_V.ThreadImpl1;
```

Figure 2.19 – Déclaration des propriétés AADL pour un thread du système ABS

Le standard AADL offre également, pour chaque catégorie de composant de la plateforme d'exécution, des propriétés prédéfinies tels que le temps d'exécution ou de propagation à travers les composants. Par exemple, la propriété `propagation_delay` correspond au temps de propagation d'un signal à travers un `bus`, et les propriétés `source_code_size` et `source-heap_size` permettent d'exprimer les contraintes en taille mémoire. La déclaration des propriétés apparait au niveau de la description textuelle AADL de chaque composant et permet de décrire ses caractéristiques.

2.8 Manques et limites du langage AADL

AADL est un langage de description d'architecture de systèmes logiciels ou matériels. La couche matérielle correspond au support d'exécution (*processeur*, *bus*, *mémoire*). Les composants logiciels d'application doivent être liés aux composants matériels. Les ultimes `threads` au `processor` et les images binaires à la mémoire. Ce langage fournit une syntaxe de référence textuelle possédant une sémantique et un modèle d'exécution. Il décrit également le comportement dynamique de l'architecture d'exécution en modélisant tout simplement des `modes` opérationnels et des transitions de `mode`. Les transitions de `mode` représentent la commutation entre les configurations d'exécution d'un système. Celles-ci peuvent avoir l'effet d'activation ou désactivation des threads pour l'exécution, ou encore un changement du chemin de connexion entre les threads, et enfin des changements des caractéristiques internes des composants. Le modèle d'exécution est principalement supporté, à un niveau élevé d'abstraction, par des descriptions des flots de contrôle ou de données et la déclaration de paramètres temporels comme la durée d'exécution au pire cas d'un sous-programme, des temps de commutations de tâches par les transitions de mode, etc. Les abstractions, dans ce cas, concernent surtout les fonctionnalités des composants logiciels d'application qui ne sont pas précisément connus au plus tôt durant les phases de développement. Cependant, il est nécessaire de savoir relier, dés la phase de conception, le comportement fonctionnel du système aux aspects logiciel et matériels relatifs aux contraintes temporelles d'exécution des composants (par

exemple, temps d'exécution, de communication et de réponse bornés). En effet, l'évaluation des comportements possibles d'un système au plus tôt est très importante pour réduire à la fois le temps et les coûts de validation, et augmente la fiabilité du système. Cette évaluation peut être réalisée en adoptant des techniques d'analyse formelle pour le comportement des composants AADL et leurs interactions.

2.9 Conclusion

Dans ce chapitre nous avons présenté la syntaxe textuelle et la représentation graphique du langage de description d'architecture AADL. Nous nous sommes concentrés sur sa description structurelle et comportementale. Dans la description de l'aspect structurel, nous avons explicité les éléments architecturaux de base permettant de modéliser un système embarqué. Dans la description de l'aspect comportemental, le standard propose la notion de modes permettant de décrire un ensemble de configurations dynamiques, bien que le système soit globalement statique.

A travers un exemple illustratif, nous avons présenté les étapes à suivre pour la modélisation architecturale AADL d'un système embarqué. Nous avons montré que le modèle d'exécution, définit par des descriptions des flots de données et/ou de contrôle et la déclaration de contraintes d'exécution des composants, demande une attention particulière et nécessite une formalisation. En effet, le langage AADL dispose de mécanismes d'extension permettant de l'enrichir pour de tels besoins.

Dans le chapitre suivant, nous décrivons les concepts de base de la logique de réécriture, sa vocation à fournir un cadre sémantique unificateur pour les modèles architecturaux AADL, et sa puissance à décrire leurs aspects structurels et comportementaux.

Chapitre 3

La Logique de Réécriture et le Système Maude

Sommaire

3.1 Introduction

La réécriture est un paradigme général d'expression de calcul dans différentes logiques computationnelles. Les calculs prennent la forme de règles dans une syntaxe donnée. Dans une logique équationnelle, les calculs résultent de l'interprétation d'équations entre termes. Dans une logique de satisfaction de contraintes, une règle de réécriture peut être interprétée de deux manières, soit comme une transformation syntaxique, soit comme une inférence logique d'une nouvelle formule [KKV95]. Par contre, dans la logique de réécriture, réécrire un terme consiste à le remplacer par un terme équivalent, en vertu des lois de l'algèbre des termes [Mes02]. La réécriture pour cette logique permet de calculer une relation de réécrivabilité entre des termes algébriques. L'idée essentielle de la logique de réécriture [Mes92] est que la sémantique de la réécriture peut être rigoureusement changée d'une manière très fructueuse. Il a été largement démontré qu'elle permet de raisonner parfaitement sur le comportement des systèmes concurrents. Donc la logique de réécriture est un modèle de calcul et un cadre sémantique expressif pour la concurrence, le parallélisme, la communication, et l'interaction.

Dans ce chapitre, nous présentons les concepts de base de la logique de réécriture, du système Maude et de l'outil RT-Maude permettant de faciliter la compréhension de notre travail. Pour plus de détails, le lecteur peut se référer à [MM 96] ou [CDE+99].

Le reste de ce chapitre est organisé comme suit. Dans les sections 3.2 et 3.3, nous introduisons la logique de réécriture et sa sémantique ainsi que son extension par les théories de réécriture généralisées et les théories de réécriture temps réel. Le principe de réflexion pour guider le processus de déduction est explicité dans la section 3.4. Dans les sections 3.5 et 3.6, nous présentons le système de réécriture Maude, ses différents niveaux de spécification ainsi que ses outils et méthodes d'analyse et de vérification. Nous exposons, dans la section 3.7 l'outil Real-Time Maude, une extension de Full Maude supportant la spécification et l'analyse des systèmes temps réel. La section 3.8 conclut le chapitre.

3.2 La logique de réécriture

La logique de réécriture est une logique de changement concurrent qui peut traiter l'état et le calcul des systèmes concurrents. Elle a été introduite par J.Meseguer [Mes90] comme une conséquence des travaux sur les logiques générales. Dés lors, cette logique a été largement utilisée pour spécifier et analyser des systèmes et langages dans différents domaines d'applications. Donc la logique de réécriture offre un cadre formel nécessaire pour la spécification et l'étude du comportement des systèmes concurrents. En effet, elle permet de raisonner sur des changements complexes possibles correspondant aux actions atomiques axiomatisées par les règles de réécriture. Le point clé de cette logique est que la *déduction* logique, qui est intrinsèquement concurrente, correspond au calcul dans un système concurrent [MM93, Mes 92].

Un autre aspect important de la logique de réécriture est qu'elle représente un cadre logique et sémantique général dans lequel des langages et des modèles de calcul de nature largement différentes ont été représentés. Dans ce contexte, nous pouvons citer sans être exhaustifs, les systèmes de transitions étiquetés [Mes96], les réseaux de Petri [SMÖ01], CCS [Mes96], etc. En plus, cette logique peut constituer une base formelle rigoureuse pour la description des architectures logicielles [MT97, Jer08, BLB08, BBC07]. Dans ce contexte nous proposons, dans le cadre de cette thèse, un cadre sémantique pour le langage AADL.

3.2.1 Théorie de réécriture

Dans la logique de réécriture, un système concurrent est décrit par une théorie de réécriture $\mathcal{R} = (\Sigma, E, L, R)$ où (Σ,E) désigne la signature (une théorie équationnelle) définissant la structure algébrique particulière des états du système (multi-ensemble, arbre binaire…) qui sont distribués selon cette même structure. La structure dynamique du système est décrite par les règles de réécriture étiquetées R (L est un ensemble d'étiquettes de ces règles). Les règles de réécriture précisent quelles sont les transitions élémentaires et locales possibles dans l'état actuel du système concurrent. Chaque règle (notée $[t] \rightarrow [t']$) correspond à une action pouvant survenir en

concurrence avec d'autres actions. Donc, la logique de réécriture est une logique qui capture clairement le changement concurrent dans un système.

Définition (Théorie de réécriture étiquetée)

Une théorie de réécriture $\mathcal{R}$ est un 4-uplet (Σ, E, L, R) tel que :

1. *Σ est un ensemble de symboles de fonctions, et de sortes.*
2. *E un ensemble de Σ-équations (l'ensemble des équations entre les Σ-termes).*
3. *L est un ensemble d'étiquettes.*
4. *R est un ensemble de règles de réécriture, défini ainsi $R \subseteq L \times (T_{\Sigma,E}(X))^2$; chaque règle est un couple d'éléments, le premier est une étiquette, le second est une paire de classes d'équivalence de termes $T_{\Sigma,E}(X)$ sur la signature (Σ,E), modulo les équations E, avec $X = \{x_1, \ldots, x_n, \ldots\}$ un ensemble infini et dénombrable de variables.*

La réécriture opère sur les classes d'équivalence de termes, modulo l'ensemble des équations *E*. Ainsi, la réécriture est libérée des contraintes syntaxiques de la représentation des termes pour bénéficier d'une grande flexibilité dans le choix des structures de données. Pour une règle de réécriture de la forme $r([t] , [t'] , C_1, \ldots, C_k)$, la notation suivante est utilisée,

$$r : [t] \rightarrow [t'] \ \textbf{if} \ C_1 \wedge \ldots \wedge C_k,$$

où une règle ***r*** exprime que la classe d'équivalence contenant le terme ***t*** peut se réécrire en la classe d'équivalence contenant le terme ***t'*** si la condition de la règle $C_1 \wedge \ldots \wedge C_k$ est vérifiée. Cette dernière est appelée condition de la règle et peut être abrégée par la lettre ***C***, et la règle de réécriture, dans ce cas, est dite conditionnelle. La partie conditionnelle d'une règle peut être vide, dans ce cas les règles sont appelées règles de réécriture inconditionnelles et sont notés par

$$r : [t] \rightarrow [t']$$

Une règle de réécriture peut être paramétrée par un ensemble de variables $\{x_1, \ldots, x_n\}$ qui apparaissent soit dans ***t, t'*** ou ***C***, et nous écrivons :

$$r : [t(x_1, \ldots, x_n)] \rightarrow [t'(x_1, \ldots, x_n)] \ \textbf{if} \ C(x_1, \ldots, x_n)$$

3.2.2 Déduction dans la logique de réécriture

Le calcul dans un système concurrent est une séquence de transitions (règles de réécriture) exécutées à partir d'un état initial donné. Il correspond à une preuve ou une déduction dans la logique de réécriture. Cette déduction est intrinsèquement concurrente et permet de raisonner correctement sur l'évolution du système d'un état à un autre [MM01, Mes90].

Définition (Principe de déduction). *Etant donné une théorie de réécriture* $\mathcal{R}_{\mathcal{C}} = (\Sigma, E, L, R)$, *nous disons que la séquence* $[t] \rightarrow [t']$ *est prouvable dans* $\mathcal{R}$ *et on écrit* $\mathcal{R} \vdash [t] \rightarrow [t']$ *si et seulement si* $[t] \rightarrow [t']$ *est obtenue par une application finie des règles de déduction suivantes:*

1. ***La réflexivité*** *: pour chaque terme* $[t] \in T_{\Sigma,E}(X)$, $\dfrac{}{[t] \rightarrow [t]}$ *où* $T_{\Sigma,E}(X)$ *est l'ensemble des Σ-termes avec variables construits sur la signature* Σ *et les équations E.*

2. ***La congruence*** *: pour chaque fonction* $f \in \sum_n$, $n \in N$,

$$\frac{[t_1] \rightarrow [t_1'] \ldots [t_n] \rightarrow [t_n']}{[f(t_1,\ldots,t_n)] \rightarrow [f(t_1',\ldots,t_n')]}$$

3. ***le remplacement*** *: pour chaque règle* $r : [t(x_1,\ldots,x_n)] \rightarrow [t'(x_1,\ldots,x_n)]$ *dans R :*

$$\frac{[w_1] \rightarrow [w_1'] \ldots [w_n] \rightarrow [w_n']}{[t(\overline{w}/\overline{x})] \rightarrow [t'(\overline{w'}/\overline{x})]}$$

Sachant que $t(\overline{w}/\overline{x})$ *dénote la substitution simultanée de* x_i *par* w_i *dans t avec* $\overline{x}$ *représentant* $x_1, \ldots, x_n$.

4. ***la transitivité*** *:* $\dfrac{[t_1] \rightarrow [t_2] \quad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$

De manière générale, la déduction dans la logique de réécriture est une itération des étapes suivantes :

1. La règle de *remplacement* identifie toutes les règles de réécriture dont le membre gauche correspond à un sous-terme de l'état global courant. Comme la logique de réécriture est une logique de changement, la règle de réflexivité, appliquées aux sous-termes non identifiés, les transforme mais en eux-mêmes.

2. Les règles de réécriture, identifiées par la règle de *remplacement* ainsi que la règle de *réflexivité*, sont exécutées en concurrence et indépendamment les unes des autres. La règle de *congruence* compose les effets, membres droits, de ces règles pour construire le nouveau terme global.

   Les étapes (1) et (2) sont réitérées jusqu'à ce qu'il n'y ait plus de règle applicable.

3. Enfin, la règle de *transitivité* construit la séquence des réécritures faites du terme initial jusqu'au terme final. La séquence ainsi construite correspond à un calcul possible dans le système concurrent.

Remarque : Les règles de *congruence* et de *remplacement* peuvent être vues sous un autre angle. La première règle exprime que des règles de réécriture disjointes (des règles de réécriture sont disjointes si elles n'ont pas de sous-termes communs) peuvent être exécutées en concurrences. Alors que la seconde règle permet des réécritures imbriquées, i.e., imbrication de la réécriture des sous termes $w_i \rightarrow w'_i$ dans celle du terme composite $t \rightarrow t'$. Elle indique que deux sous termes différents peuvent être réécrits en parallèle même si leurs racines, terme composite, ne sont pas disjointes [CGM95].

La *réflexivité* introduit le calcul « identité », c'est-à-dire que les sous-termes qui ne sont pas susceptibles de changer suite à l'application d'une étape de réécriture doivent être réécrits en utilisant la règle de *réflexivité*. Cette règle découle du fait que la logique de réécriture est une logique de « changement », tous les termes et sous-termes doivent être changés (substitués) durant une étape de réécriture. Les termes qui ne changent pas se réécrivent donc en eux même.

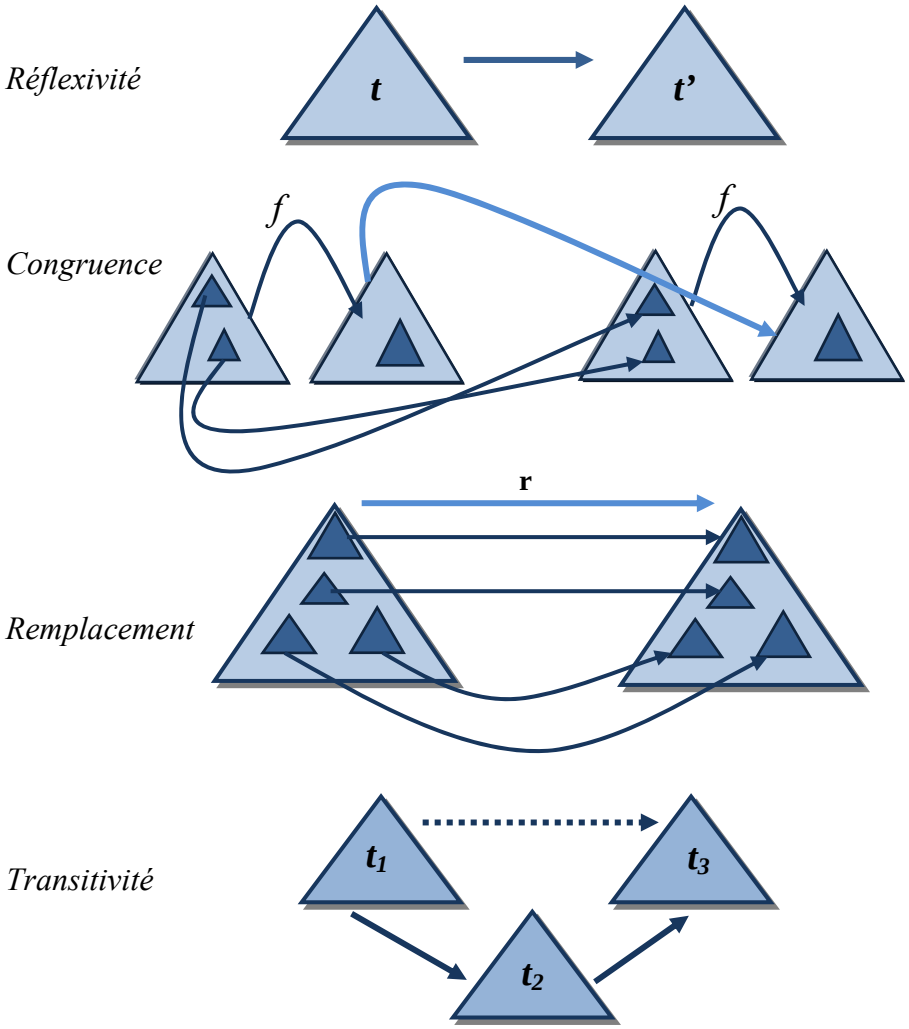


Figure 3.1 - Représentation graphique des règles de déduction

La *transitivité* détermine la composition séquentielle des étapes de réécriture. Cette règle offre la possibilité de construire la séquence de déduction à partir du terme initial jusqu'au terme final (à partir du quel plus aucune réécriture n'est possible).

La *congruence* spécifie que les réécritures peuvent être emboitées dans des contextes plus larges. Autrement dit, la composition parallèle d'un ensemble de transformations locales produit un état composite et cohérent et qui sera le nouveau terme global.

La règle de déduction qui semble la plus complexe est le *remplacement* qui identifie les termes à remplacer dans une expression d'un terme global représentant

l'état actuel du système concurrent en faisant les substitutions de variables appropriées.

3.2.3 Réécriture concurrente

Le calcul dans un système concurrent (représenté par une séquence d'étapes de réécritures concurrentes exécutées à partir d'un état initial donné) correspond à une preuve ou une déduction dans la logique de réécriture en utilisant les règles de déduction appropriées (*réflexivité*, *congruence*, *remplacement*, *transitivité*). Par conséquence, une théorie de réécriture est vue comme une spécification exécutable du système qu'elle formalise. Les règles de réécriture *R* de cette théorie décrivent les transitions élémentaires possibles dans un système. Les règles de déduction de la logique de réécriture permettent de raisonner correctement sur les transitions générales possibles dans un système.

Définition (réécriture concurrente) *Etant donné une théorie* $\mathcal{R}$ ***=(Σ,E,L,R),*** *une* ***(Σ,E)-formule [t]*** $\rightarrow$ ***[t']*** *est dite:*

1. $\mathcal{R}$ ***-réécriture concurrente 0-step*** *si et seulement si elle est dérivée à partir de* $\mathcal{R}$ *par une application finie des règles de déduction* ***1*** *et* ***2*** *(dans ce cas les termes [t] et [t'] coïncident nécessairement) ;*

2. $\mathcal{R}$ ***-réécriture concurrente 1-step*** *si et seulement si elle est dérivée à partir de* $\mathcal{R}$ *par une application finie des règles de déduction* ***1, 2*** *et* ***3,*** *avec au moins une application de la règle* ***3*** *(règle de remplacement). Si toutefois la règle* ***3*** *est appliquée une seule fois seulement, on appelle la formule une étape de* $\mathcal{R}$ ***– réécriture*** *séquentielle ;*

3. $\mathcal{R}$***-réécriture concurrente*** *(ou juste une* ***réécriture****) si et seulement si elle est dérivée à partir de* $\mathcal{R}$ *par une application finie des règles de déduction* ***1, 2, 3*** *et* ***4****.*

3.3 Extension de la logique de réécriture

La logique de réécriture dépend essentiellement de la logique équationnelle sous-jacente. Les auteurs de [BM06] précisent que la généralisation vers des logiques équationnelles plus expressives implique des versions plus expressives de la logique de réécriture en permettant notamment des conditions plus générales dans la partie conditionnelle des règles de réécriture et en introduisant formellement la notion d'interdiction de la réécriture de sous-termes relatifs à certaines positions dans des opérateurs. Ainsi, ces extensions de la logique de réécriture, proposées par Bruni et Meseguer, développent de nouvelles bases sémantiques pour une version révisée de cette logique qui supporte plusieurs caractéristiques nouvelles dont l'expressivité a été trouvée très utile dans la pratique. La révision de l'expression de son formalisme a été proposée selon plusieurs dimensions. D'abord en choisissant la logique équationnelle d'appartenance (membership) comme logique équationnelle fondamentale. Ensuite en permettant des conditions très générales dans les règles conditionnelles de réécriture. La troisième dimension permet de déclarer certains arguments d'opérateurs comme gelés (frozen), pour les bloquer en réécriture.

3.3.1 Théorie de réécriture généralisée

Une théorie de réécriture généralisée [BM06] est un 5-uplet $\mathfrak{R} = (\Sigma, E, \Phi, L, R)$, où (Σ, E) est une théorie équationnelle dans la logique équationnelle d'appartenance *MEL* (*Membership Equational Logics*), Φ est une fonction assignant à chaque opérateur $f : k_1, \ldots, k_n \to k$ dans Σ le sous ensemble $\Phi(f) \subseteq \{1, \ldots, n\}$ de ses arguments gelés, et R est un ensemble de règles de réécriture conditionnelles étiquetées qui peuvent être de la forme :

$$(\forall X) r : t \to t' \; if \bigwedge_{i \in I} p_i = q_i \wedge \bigwedge_{j \in J} w_j : s_j \wedge \bigwedge_{l \in L} t_l \to t_l'$$

où **r** : est la règle étiquetée, tous les termes $(\mathbf{p_i}, \mathbf{q_i}, \mathbf{w_j}, \mathbf{s_j}, \mathbf{t_l}, \mathbf{t_l'})$ sont des Σ-termes, et les conditions peuvent être des réécritures, des équations d'appartenance dans (Σ, E), ou n'importe quelle combinaison des deux.

3.3.2 Théorie de réécriture temps réel

Les théories de réécriture généralisées dites « *temps réel* » ont été introduites pour spécifier les comportements des systèmes hybrides et des systèmes temps-réel dans la logique de réécriture. Dans ce type de théories de réécriture, certaines règles appelées « *tick rules* » expriment l'écoulement du temps ou la durée dans un système tandis que des règles de réécriture ordinaires expriment les changements d'états du système qui ont lieu d'une manière instantanée. De plus, la théorie équationnelle d'appartenance sous-jacente à une théorie de réécriture temps-réel devrait contenir une axiomatisation (une interprétation abstraite) du domaine du temps et un opérateur {_} qui inclut l'état global du système. Cet opérateur est défini afin d'assurer que le temps s'écoule uniformément dans toutes les parties de ce système selon l'exécution de la règle « *tick rules* ». Une règle de réécriture « *tick rules* », qui modélise l'écoulement du temps, a la forme générale,

$$l : \{t\} \xrightarrow{\tau_l} \{t'\} \text{ if Cond}$$

où l est l'étiquette de la règle et le terme τ_l, de la sorte *Time*, exprime la durée de l'étape de réécriture.

Définition (Théorie de réécriture temps réel) *Une théorie de réécriture temps réel $\mathcal{R}_{\phi,\tau}$ est un tuple $\mathcal{R}_{\phi,\tau} = (\mathcal{R}, \phi, \tau)$ où $\mathcal{R} = (\Sigma, E, \Phi, L, R)$ est une théorie de réécriture généralisée tel que :*

- *ϕ est un morphisme de théorie équationnelle $\phi : TIME \rightarrow (\Sigma, E)$ où TIME est une théorie équationnelle* [ÖM96] *dans laquelle sont spécifiés, à l'aide d'équations, les besoins généraux du modèle du temps. La théorie équationnelle TIME défini le temps d'une manière abstraite comme un monoïde commutatif ordonné (Time, 0, +, <) avec des opérateurs additionnels tels que l'opérateur de soustraction $\dot{-}$ (où $x \dot{-} y$ dénote $x - y$ si $y < x$ et 0 si $y \geq x$) et l'opérateur de comparaison $\leq$;*

- *(Σ, E) contient une sorte désignée, appelée System (qui exprime l'état du système) et une sorte spécifique GlobalSystem n'admettant aucune sous-sorte ni aucune super-sorte et définie uniquement sur l'opérateur suivant :*

$\{\_\} : System \rightarrow GlobalSystem$

De plus, pour tout $f : s_1 \ldots s_n \rightarrow s$ *de* Σ, *la sorte GlobalSystem n'apparaît pas parmi* $s_1 \ldots s_n$,

- τ *est une attribution d'un terme* $\tau_l(x_1,\ldots,x_n)$ *de la sorte* ϕ*(Time) à chaque règle de réécriture de la forme :*

  $l : \{u(x_1,\ldots,x_n)\} \rightarrow \{u'(x1,\ldots,xn)\}$ *if* $Cond(x_1,\ldots,x_n)$

  où u et u' sont des termes de sorte GlobalSystem

  Ainsi, une règle l de sorte GlobalSystem et de durée τ_l *est notée :*

  $l : \{u(x_1,\ldots,x_n)\} \xrightarrow{\tau_l(x_1,\ldots,x_n)} \{u'(x1,\ldots,xn)\}$ **if** $Cond(x_1,\ldots,x_n)$

Etant donnée une théorie de réécriture temps réel $\mathcal{R}$, un calcul est une séquence non extensible $t_0 \rightarrow t_1 \rightarrow \ldots \rightarrow t_n$ (c'est-à-dire, une séquence pour laquelle t_n ne peut être réécrit) ou bien une séquence infinie $t_0 \rightarrow t_1 \rightarrow \ldots$ de réécritures $t_i \rightarrow t_{i+1}$, avec t_i et t_{i+1} des termes bornés, commençant avec un terme initial t_0 de sorte *GlobalSystem* .

L'état global du système doit être de la forme **{*u*}** de telle sorte que les règles « ***tick*** » assurent que le temps avance uniformément dans toutes les parties du système dont l'état est représenté par le terme ***u***.

3.4 Réflection et stratégie de réécriture

La logique de réécriture est réflective dans un sens mathématique précis, c'est-à-dire, il existe une théorie finie de réécriture *U* qui est universelle dans le sens où on peut représenter dans cette théorie *U* (comme des termes) toute autre théorie de réécriture finie *R* (y compris la théorie *U* elle-même) [CM97]. Ainsi, il est possible de simuler dans la théorie *U* toute réécriture inférée (déduite) dans la théorie de réécriture *R*. Par conséquent, il existe une représentation finie $\overline{R}$ sous forme de termes de *U* de toute théorie de réécriture *R*, une représentation $\overline{t}$ et $\overline{t'}$ sous forme de termes de *U* de tous termes *t* , *t*' et *R*, et une représentation $\langle \overline{R} ; \overline{t} \rangle$ de tout couple (*R,t*) vérifiant $R : t \rightarrow t'$ $\Leftrightarrow$ ($U : \langle \overline{R} ; \overline{t} \rangle \rightarrow \langle \overline{R} ; \overline{t'} \rangle$)

La réflection permet de guider le processus de déduction induit par une théorie de réécriture R au niveau objet par le biais de stratégies de réécriture dont la sémantique peut être définie à l'intérieur de la logique de réécriture par des théories de réécriture définies à un méta-niveau. Dans ce cas, de telles stratégies de réécriture constituent des procédures d'inférence particulières spécifiques à la théorie R. Ainsi, la propriété de reflection permet à la logique de réécriture de décrire fidèlement le comportement de certains systèmes dont la sémantique complète ne peut être spécifiée simplement par un ensemble de règles de réécriture mais nécessite aussi la spécification de procédures d'exécution particulières de ces règles (par exemple, en précisant un ordre d'exécution spécifique).

3.5 Système MAUDE

Maude est un langage déclaratif qui implémente correctement tous les concepts théoriques de la logique de réécriture. Il constitue un système de haute performance, défini par J. Meseguer [CDE+02], supportant à la fois la spécification exécutable et la programmation déclarative des logiques équationnelles et de réécriture pour un grand nombre d'applications. En général, un programme écrit dans le langage déclaratif Maude représente une théorie de réécriture, c'est-à-dire, une signature et un ensemble de règles de réécriture. Le calcul dans ce langage correspond à la déduction en logique de réécriture en utilisant les axiomes spécifiés dans ces théories/programmes.

Les unités basiques de spécification ou de programmation dans Maude sont appelées des *modules*. Par conséquent, on différencie trois types de modules : les modules fonctionnels pour implémenter les théories équationnelles. Les modules système implémentent les théories de réécriture et définissent le comportement dynamique d'un système. Les modules orientés-objet implémentent les théories de réécriture orientées objet (ils peuvent être réduits à des modules systèmes).

3.5.1 Modules fonctionnels

Un module fonctionnel est introduit par les mots clés `fmod` <*Corps du module*> `endfm` où le corps du module spécifie une théorie (Σ, EUA, Φ) dans la logique

équationnelle d'appartenance. La signature Σ inclut des sortes (indiqués par le mot clé `sort`), des sous-sortes (spécifiés par le mot clé `subsort`) et des opérateurs (introduits avec le mot clé `op`). La syntaxe des opérateurs est définie par les utilisateurs en indiquant la position des arguments par le symbole (_). Certains de ces arguments peuvent être spécifiés comme figés en utilisant le mot clé `frozen`(*PositionArgument*). L'ensemble $\boldsymbol{E}$ désigne les équations et les tests d'appartenance (qui peuvent être conditionnels) et $\boldsymbol{A}$ est un ensemble d'axiomes équationnels introduits comme attributs de certains opérateurs dans la signature $\boldsymbol{\Sigma}$ tels que les axiomes d'associativité (spécifiée par le mot clé `assoc`), de commutativité (spécifiée par le mot clé `comm`) ou d'identité (spécifiée par le mot clé `id` :). Ces derniers sont définis de manière à ce que les déductions équationnelles se fassent modulo les axiomes de A. Les équations sont spécifiées par le mot clé `eq` ou le mot clé **`ceq`** (pour les équations conditionnelles) et les tests d'adhésion ou d'appartenance sont introduits avec les mots clés `mb` ou `cmb` (pour les tests d'appartenance conditionnels). Une condition liée à une équation ou à un test d'appartenance peut être formée par une conjonction d'équations et de tests d'adhésions inconditionnels.

Dans un module fonctionnel, les équations sont utilisées comme des règles de simplification par lesquelles chaque expression, après substitution des variables, peut être évaluée et simplifiée à sa forme réduite dite *forme canonique*. Le résultat de la simplification d'un terme initial est unique quelque soit l'ordre d'application des équations. Les variables peuvent être déclarées dans les modules avec les mots clés `var` ou `vars`, ou introduites directement dans les équations et les tests d'adhésion, sous la forme d'une expression `var : sort`.

3.5.2 Modules systèmes

Les modules systèmes sont utilisés pour définir le comportement dynamique des systèmes concurrents en enrichissant les modules fonctionnels par un ensemble de règles de réécriture. Ils sont introduits par les mots clés `mod` <*Corps du module*> `endm` où le corps du module spécifie une *théorie de réécriture* $\Re = (\Sigma, E \cup A, \Phi, R)$ avec $(\Sigma, E \cup A, \Phi)$ la théorie équationnelle sous-jacente. Les règles de réécriture R sont

introduites avec les mots clés `rl` ou `crl`. Elles sont spécifiées dans Maude avec la syntaxe :

`crl` [*l*] : *t* => *t'* **`if`** *cond* .

Si la règle est non conditionnelle, le mot clé `crl` est remplacé par `rl` et la clause « **`if`** *cond* » est omise.

3.5.3 Modules orientés objet

Il est possible de présenter un système d'objets concurrents comme une théorie de réécriture avec une syntaxe plus appropriée pour décrire les concepts de base du paradigme objet. Les modules orientés objet sont supportés par le système Full-Maude [DM99]. Notons que Full Maude est une extension de Maude (Core Maude) dont le code est écrit en Maude, ce qui enrichit le langage Maude avec un module algébrique très puissant et extensible. Ces modules orientés objet sont introduits par les mots clés : (`omod` < *corps du module*> `endom`) où le corps du module est une *théorie de réécriture* $\Re = (\Sigma, E \cup A, \Phi, R)$. Ils supportent la spécification et la manipulation des *objets*, des *messages*, des *classes* et de l'*héritage*. Un système orienté objet concurrent dans ce cas est modélisé par un multi-ensemble d'objets et de messages juxtaposés, où les interactions concurrentes entre les objets sont régies par des règles de réécriture. Un objet est représenté par le terme $< O : C \mid a_1 : v_1 , \ldots , a_n : v_n >$, où O est le nom de l'objet instance de la classe C, $a_i, \in 1..n$, les noms des attributs de l'objet, et v_i, leurs valeurs respectives.

La déclaration des classes suit la syntaxe : *class* $C \mid a_1 : s_1 , \ldots , a_n : s_n .$, où C est le nom de la classe et s_i est la sorte de l'attribut a_i. Il est aussi possible de déclarer des sous classes et bénéficier ainsi de la notion d'héritage. Les messages sont déclarés en utilisant le mot clé *msg*. La forme générale d'une règle de réécriture dans la syntaxe orientée objet de Maude est :

$$crl\ [r] : M_1 \ldots M_n < O_1 : F_1 \mid a_1 > \ldots < O_m : F_m \mid a_m > \Rightarrow < O_{i1} : F'_{i1} \mid a'_{i1} > \ldots < O_{ik} : F'_{ik} \mid a'_{ik} > M_1' \ldots M_p' \quad \textbf{if}\ Cond .$$

Où r est l'étiquette de la règle, $M_s, s \in 1..n,$ et $M'_u, u \in 1..p$ sont des messages, $O_i, i \in 1..m,$ et $O_{il}, l \in 1..k$, sont des objets, et *Cond* est la condition de la règle. Si la règle est

non conditionnelle, nous remplaçons le mot clé *crl* par *rl* et nous enlevons la clause *if Cond*.

Notons que le système Full-Maude offre un support additionnel pour la programmation orientée objet avec les notions de classes, sous-classes et une syntaxe plus conviviale des règles de réécriture. Ainsi, il permet au système Maude de supporter la modélisation orientés objet en fournissant le module prédéfini *CONFIGURATION*. Dans ce module, les sortes représentant les concepts essentiels des objets, classes, messages et configurations, sont déclarés.

3.5.4 Modules prédéfinis

Les modules prédéfinis de Maude sont stockés dans une librairie spécifique et peuvent être importés par d'autres modules définis par l'utilisateur. Ils sont introduits dans les fichiers sources de Maude `prelude.maude` et `model-checker.maude` : comme par exemple les BOOL, STRING et NAT. Ces modules déclarent les sortes et les opérations pour manipuler, respectivement, les valeurs *booléennes*, les *chaînes de caractères* et les *nombres naturels*. Le fichier `model-checker.maude` contient les modules prédéfinis interprétant les outils nécessaires pour l'utilisation du *LTL Model Checker* de Maude.

3.6 Exécution et analyse formelle sous Maude

Dans un module écrit en Maude, les règles de réécriture constituent les unités élémentaires d'exécution. Elles interprètent les actions locales du système modélisé, et peuvent être exécutées dans un temps constant et de manière concurrente (à n'importe quel moment). Maude nous offre la possibilité de simuler l'exécution de telles réécritures (via des règles de réécriture) ou des réécritures équationnelles (via des équations) dans un module M par l'implémentation des deux commandes : `reduce` et `rewrite.`

La commande `reduce` (abrégée par `red`) permet à un terme initial d'être réduit par application des équations et des axiomes d'adhésion dans un module donné. Elle se présente sous la syntaxe suivante :

```
Reduce {in module :} term .
```

La commande de réécriture `rewrite` (abrégée par `rew`) et la commande de réécriture *équitable* `frewrite` (abrégée par `frew`) exécutent une seule séquence de réécriture (parmi plusieurs séquences possibles) à partir d'un terme initialement donné suivant la syntaxe :

```
rewrite {in module : } term .

frewrite {in module : } term .
```

Ces commandes permettent de réécrire un terme initial en utilisant les règles, les équations et les axiomes d'adhésion dans le module spécifié.

3.6.1 Analyse formelle et vérification des propriétés

La vérification formelle de modèles sur les systèmes spécifiés en Maude s'effectue à l'aide d'outils disponibles autour du système Maude. Parmi ces outils, on distingue un outil d'analyse d'accessibilité (la commande `search`) et un module de vérification par *model checking* de propriétés exprimées en logique linéaire temporelle (LTL) [EMS02]. Pour analyser toutes les séquences de réécritures possibles à partir d'un état (terme) initial t_0, on utilise la commande `search.` Celle-ci recherche si des états correspondants à des patterns donnés et satisfaisant certaines conditions, peuvent être accessibles à partir de t_0. L'exécution de cette commande effectue un parcours en profondeur de l'arbre de calcul (arbre d'accessibilité), généré lors de cette recherche, afin de détecter les violations d'invariants dans les systèmes à états infinis [CDE+07].

L'analyse formelle des systèmes par *Model Checking* [CGP01] est un ensemble de techniques pour vérifier automatiquement des propriétés temporelles relatives au comportement des systèmes. Le *Model Checking* permet de vérifier la satisfiabilité d'une propriété donnée dans un état ou un ensemble d'états, en faisant une exploration exhaustive de l'ensemble des états accessibles à partir d'un état initial. Il reçoit en entrée une abstraction du comportement du système (un système de transitions), représentée par une structure Kripke K, et une propriété φ de ce système, formulée dans une certaine logique temporelle, et répond si l'abstraction satisfait ou

non la formule φ c'est-à-dire, si K |= φ. L'intérêt du *Model Checking* est qu'il retourne une trace d'exécution du système violant la propriété lorsque cette dernière est non valide.

Logique temporelle linéaire (LTL)

La logique temporelle linéaire (LTL) est une extension de la logique classique avec des opérateurs temporels, tels que G et F (représentant respectivement « globalement », « finalement ou fatalement ») qui permettent d'exprimer des propriétés portant sur l'exécution d'une séquence d'états. Elle est dite temporelle car elle décrit le séquencement d'évènements observés dans un système. Cette logique permet de spécifier des propriétés intéressantes pour les systèmes concurrents notamment les propriétés de ***sûreté***, d'***accessibilité***, de ***vivacité*** et d'***équité***.

- ***Sûreté*** : quelque chose de mauvais n'arrive jamais.
- ***Accessibilité*** : une certaine situation peut être atteinte.
- ***Vivacité*** : quelque chose de bon est toujours possible.
- ***Equité*** : quelque chose se répètera infiniment souvent.

Les formules LTL sont construites à partir de variables propositionnelles appelées propositions atomiques, d'opérateurs booléens (¬, ∧, ∨, ⇒, ⇔), et d'opérateurs temporels, **F** (Futur), **G** (Global), **U** (Until), **X** (neXt). Les propriétés atomiques permettent de décrire les états du système : un état ***t*** est dit étiqueté par une proposition atomique φ si φ est vraie dans ***t***. Les opérateurs temporels permettent de relier des états du système au sein d'une séquence d'exécution.

- La formule **X***f* indique que la formule LTL *f* doit être vérifiée à l'instant suivant immédiat le long de l'exécution (*neXt*).
- La formule *f***U***g* indique que *f* est toujours vrai jusqu'à un état où *g* est vrai (*Until*).
- La formule **G***f* signifie *f* est toujours vérifiée (*Generally*) dans toute l'exécution.
- La formule **F***f* indique que *f* doit être vérifiée plus tard au moins dans un état de l'exécution (*Fatalement*).

3.6.2 Le Model Checker LTL de Maude

L'outil de model checking qu'offre Maude fait appel à la logique LTL. Il exige que le système à vérifier, décrit par une théorie de réécriture $\mathfrak{R} = (\Sigma, E, \Phi, R)$, soit à état fini, c'est-à-dire qu'à partir d'un état initial $[t_0]$, l'ensemble des états accessibles défini par $\{[u] \in T_{\Sigma,E} \mid \mathfrak{R} \vdash [t_0] \rightarrow [u]\}$ soit fini. Donc, le LTL model checker de Maude prend en entrée une théorie de réécriture $\mathfrak{R}$ dont il génère la structure Kripke $K(\mathfrak{R}, State)$ sous-jacente ainsi qu'une formule temporelle LTL φ et vérifie si cette structure $K(\mathfrak{R}, State)$ satisfait la formule φ. Si cette formule n'est pas valide, le LTL model checker retourne un contre-exemple qui montre une séquence d'états menant à la violation de cette formule. Cet outil est implémenté sous la forme d'un module, spécifié en termes de la logique de réécriture, dans le langage Maude. Ce dernier importe d'autres modules prédéfinis (spécifiés également dans le langage Maude) :

- Le module *LTL-SIMPLIFIER* implémente des procédures formelles pour simplifier la formule LTL exprimant une propriété.
- Le module SATISFACTION spécifie la syntaxe et la sémantique de l'opérateur de satisfaction (|=) indiquant si une formule donnée est vraie ou fausse dans un certain état.
- Le module SAT-SOLVER permet de vérifier la satisfiabilité et la topologie d'une formule spécifiée en logique LTL.

3.7 Real Time Maude

Le système Real Time Maude (RT-Maude) [ÖM00, ÖM04, Ölv07] est un langage et un outil permettant la spécification formelle et l'analyse de système temps et hybrides. Son formalisme de spécification, basé sur la logique de réécriture, est particulièrement approprié pour décrire les systèmes temps réel orientés objet. L'outil est implémenté en Maude comme une extension de Full Maude. Le langage de spécification RT-Maude permet de modéliser les systèmes temps réel en termes de théories de réécriture temps réel. Il permet d'implémenter ces théories de réécriture par des modules temporisés ou des modules orientés objet temporisés dont le corps est

encapsulé entre les mots clés `tmod` et `endtm` (spécifiant le début et la fin d'un module temporisé) et entre les mots clés `tomod` et `endtom` dans le cas des modules temporisés orientés objet. Notons que chaque module temporisé doit comporter essentiellement :

1. une abstraction du domaine temps (de sorte `Time`). Le domaine du temps géré par les modules temporisés peut être discret ou dense. Par exemple, pour manipuler le temps discret dans un module temporisé, l'utilisateur doit importer le module prédéfini *NAT-TIME-DOMAIN-WITH-INF* de RT-Maude qui défini le domaine de temps comme des nombres naturels en ajoutant la constante *INF* (dénotant ∞) de supersort `TimeInf`;

2. la sorte `GlobalSystem` et un constructeur libre {_}, englobant tout l'état du système modélisé ;

3. des règles de réécriture ordinaires modélisant les changements instantanés dans un système. Ces règles ont la même syntaxe que celle des modules système de Maude ; et

4. des règles « *tick* » pour modéliser l'écoulement du temps dans un système ;

   Ces dernières ont la syntaxe suivante :

```
crl [label] : {t} ⇒ {t'} in time D if cond .
```

où `t` et `t'` sont des termes de sorte `System` dénotant l'état du système, `cond` est la condition de la règle et `D` un terme, pouvant contenir des variables, de sorte `Time` désignant la durée de la règle. Un état initial d'un système modélisé dans RT-Maude, doit être un terme réductible à des termes de la forme `{t}` par application des équations de la spécification. La forme des règles « *tick* » assure ainsi un écoulement de temps uniforme dans toutes les parties du système. Dans ce cas, il est nécessaire de déterminer la stratégie d'avancement du temps pour guider l'application des règles « *tick* ». Le choix d'une telle stratégie se fait par la commande de RT-Maude suivante:

```
(set tick def r .)
```

où **r** est un terme de sorte `Time`, indiquant le pas d'avancement dans le temps défini par l'utilisateur et tenté par RT-Maude à chaque application d'une règle « *tick* ».

3.7.1 Spécification exécutable

RT-Maude dispose d'un ensemble de commandes de simulation, d'analyse formelle et de vérification de propriétés LTL par *model checking* [ÖM04]. L'exécution et l'analyse d'un module temporisé se fait par rapport à la stratégie choisie pour l'application des règles « *tick* » de ce module. Notons que les techniques d'avancement du temps de RT-Maude procèdent à un échantillonnage de l'espace des états d'un système temporisé de telle sorte qu'au lieu de couvrir tout le domaine du temps, seulement certains moments sont visités. Il est donc possible de vérifier seulement le comportement d'un système dans un sous-ensemble des états accessibles à certains instants selon le mode d'avancement du temps adopté par l'utilisateur à condition qu'un tel système n'admette pas de comportement Zénon (si la durée d'un nombre infini d'étapes temporisées est bornée) .

Pour simuler le comportement possible d'un système, RT-Maude dispose de deux modes de réécriture temporisée, implémentées respectivement par la commande de réécriture standard `timed rewrite` (ou `trew`) et la commande de réécriture équitable `timed fair rewrite` (ou `tfrew`). Chacune de ces commandes simule un comportement possible du système, selon sa propre stratégie, jusqu'à une durée de temps donnée.

Pour l'analyse d'accessibilité, RT-Maude offre également une commande de recherche temporisée (`tsearch`) pour vérifier si un état ***t*** dans un système est accessible à partir d'un état initial et dans une limite de temps donnée. Cette commande adopte par défaut la stratégie de recherche en largeur dans l'arbre de calcul (arbre d'accessibilité).

3.7.2 Analyse formelle avec le LTL Model Checker (borné dans le temps)

Le système RT-Maude étend également le LTL *model Checker* du système Maude pour vérifier si toutes les exécutions faites dans une durée de temps donnée, satisfait une formule exprimée dans la logique temporelle linéaire LTL (model checking bornée dans le temps). Se restreindre à des calcules faits dans une borne de temps donnée, assure que le nombre d'états accessibles à partir de l'état initial est fini pour les systèmes n'admettant pas de comportement *Zénon*. On note que la vérification de propriétés LTL par *model checking* (borné dans le temps) se fait toujours selon la stratégie choisie par l'utilisateur pour appliquer les règles « tick ». Les formules LTL à vérifier doivent être définies dans un module qui importe le module spécifiant le système à analyser ainsi que le module prédéfini TIMED-MODEL-CHECKER.

Les propositions (qui doivent être paramétrées) doivent être déclarées de sorte `Prop`, et leur sémantique doit être exprimée par des équations de la forme :

```
{état} |=  prop  =  b
```

Avec `b` un terme booléen, ce qui signifie que la proposition `Prop` tient dans tous les états {`t`} tel que `{t} |= prop` est évaluée à vraie.

Une formule de la logique temporelle est construite à partir de formules atomiques (temporisées et non temporisées), des constantes *True* et *False* et d'opérateurs de la logique temporelle (hormis l'opérateur Next) tels que la négation (¬), la conjonction (/\), la disjonction (\/), until (**U**), l'opérateur *Fatalement* (<>), l'opérateur *Globalement* ([]), …etc. La commande de *model checking* (borné dans le temps) se présente avec la syntaxe suivante :

```
(mc t0 |= formule timeLimit .)
```

Celle-ci vérifie si une formule de la logique temporelle `formule` tient dans tous les états accessibles à partir de l'état initial t_0 et dans la limite de temps `timeLimit`.

3.8 Conclusion

Nous avons présenté, dans ce chapitre, les notions relatives à la compréhension des concepts de base de la logique de réécriture, du système Maude et de son extension RT-Maude, utilisés dans le cadre de la réalisation de cette thèse. Dans la section réservée à la logique de réécriture, nous nous sommes concentrés sur la présentation de son aspect théorique. Nous avons montré son pouvoir expressif pour décrire naturellement et d'une manière intuitive le comportement des systèmes concurrents. Cette description permet de révéler toutes les actions qui peuvent se produire en parallèle, décrivant ainsi correctement le comportement de ces systèmes selon une sémantique de vraie concurrence. Les déductions faites sur l'évolution des états d'un système représentent les calculs dans le système concurrent.

Nous avons présenté également, le système Maude. Il s'agit d'un système de haute performance et d'un langage de spécification formelle implémentant correctement tous les concepts théoriques de la logique de réécriture. Le système Maude supporte aussi une large gamme de techniques de vérification formelles, implémentés en langage Maude.

Chapitre 4

Un Support Formel pour la Sémantique du Comportement dans AADL

Sommaire

4.1 Introduction

Le processus de développement des systèmes embarqués temps réel et de sûreté critiques doit suivre une démarche rigoureuse afin de répondre aux exigences de ces systèmes. Entre autres, il doit fournir une prévisibilité accrue dans l'intégration de systèmes par l'analyse des modèles d'architecture de ces systèmes très tôt dans le cycle de vie de développement. Cette analyse concerne surtout les caractéristiques opérationnelles de la réalisation d'un système logiciel embarqué par des modèles de son architecture d'exécution, de sa plateforme de calcul, de son interface avec l'environnement physique, et du déploiement du logiciel sur la plateforme matériel [Fei08]. Ceci mène à la création de différents modèles pour le même système. Ces modèles doivent être considérés à un niveau d'abstraction approprié à la réalisation de l'analyse.

Il s'agit, dans ce cas, de l'analyse de pannes, de fiabilité, d'utilisation de ressources, d'ordonnancement, et de sécurité. Par exemple, l'analyse de l'utilisation de ressources est basée sur des demandes de ressources et des capacités de ressources, et l'analyse de l'ordonnancement est basée sur un modèle de synchronisation des tâches. Afin de réaliser ces caractéristiques opérationnelles, l'architecture doit fournir des données valides, au bon moment, dans les limitations des ressources, avec une sécurité appropriée. Elle doit fournir également les niveaux exigés de tolérance aux fautes et de fiabilité.

Néanmoins, la concrétisation des processus d'analyse de cette ampleur, au niveau des modèles architecturaux, est loin d'être atteinte par le noyau du standard AADL d'où la nécessité d'extensions du langage. En effet, AADL est conçu pour être extensible afin de s'adapter à des analyses des architectures d'exécution que le noyau du langage ne supporte pas complètement. Ses extensions peuvent prendre la forme de nouvelles propriétés ou de notations spécifiques d'analyse qui peuvent être associées à la description architecturale sous forme d'annexes. L'objectif de ce chapitre est d'exploiter ce mécanisme d'extension du langage AADL pour définir une annexe comportementale pour AADL basée sur la logique de réécriture révisée,

permettant d'interpréter la description d'un composant AADL afin d'en extraire une description formelle de son comportement utilisable lors de sa phase analyse.

Le chapitre, commence par présenter quelques modèles formels existants dans la littérature, proposés pour formaliser le langage AADL. Une classification de ces modèles est alors dégagée afin de situer notre contribution et motiver la définition de notre annexe. Nous détaillons ensuite, notre support formel proposé pour AADL, nommé ABAReL (**A**ADL **B**ehavioral **A**nnex based on revised **Re**writing **L**ogic). Celle-ci associe au composant AADL une théorie de réécriture orientée objet temps réel permettant de décrire son aspect structurel et comportemental.

4.2 Modèles formels pour AADL

Le langage AADL est riche par ses concepts architecturaux pour spécifier un système temps réels, par contre, sa sémantique n'est pas formellement définie. Plusieurs tentatives de recherche sont apparues dans la littérature pour pallier à ce manque. Dans ce langage, la description des comportements d'un système se limite à la déclaration des modes opérationnels et des propriétés. La spécification des transitions possibles entre les modes est déclenchée par réception d'événements. La spécification des propriétés décrit les conditions d'exécution des composants où elles sont déclarées.

Ainsi, le langage AADL se focalise sur les aspects architecturaux : il permet la description des dimensions des composants et leurs connexions, mais ne traite pas directement leur implantation comportementale, ni la sémantique des données manipulées. Ceci nécessite une étape d'évaluation des comportements possibles d'un système au plus tôt pour augmenter sa fiabilité, d'où le besoin d'adopter des techniques d'analyse formelle pour le comportement des composants AADL et leurs interactions.

L'émergence du langage AADL en tant que standard, durant la dernière décennie, a attiré l'attention de plusieurs équipes de recherche. Plusieurs chercheurs

se sont penchés sur la question de l'analyse formelle du comportement dans AADL.

Dans l'état de l'art, nous avons recensé un bon nombre de travaux publiés sur l'application des approches formelles à la description architecturale AADL, mais aucune étude n'a tenté d'effectuer une comparaison ou une classification de ces approches.

Pour parvenir à situer notre approche de formalisation parmi les recherches actuellement publiés, une étude comparative de ces différentes approches s'est imposée. Un effort considérable à été déployé dans le but de définir une classification des approches de formalisation des descriptions AADL. Très peu sont les publications qui contiennent une section « travaux connexes », mais dans la plupart des cas, cette section est très réduite. Cependant, plusieurs de ces publications ont été particulièrement utiles dans l'organisation de notre classification. Nous avons ainsi dégagé quatre classes de modèles formels utilisés autour de AADL. Ceux à base des systèmes de transition et de réseaux de Petri temporisés (TPN). D'autres se basent sur les machines à états abstraits (TASM) et d'autres utilisant les algèbres de processus telle que ACSR. Certains de ces travaux évitent l'utilisation directe de ces formalismes qui s'avèrent difficiles à manier et font référence à des outils de transformation du modèle AADL en un langage intermédiaire. La classification que nous proposons dans le tableau 4.1 est principalement basée sur le type de formalisme utilisé mais elle peut être influencée par l'objectif même de la formalisation, les éléments AADL spécifiés, le langage intermédiaire adopté en cas d'une transformation, le type d'analyse entrepris et les outils utilisés.

4.2.1 Modèles à base de Système de Transition Etiqueté (LTS)

Les systèmes de transitions étiquetées (LTS : Labelled Transition System) ont été à la l'origine de plusieurs travaux de formalisation du langage AADL dans la littérature. Tous ces travaux adoptent une transformation vers un langage intermédiaire plus adapté à l'analyse formelle dans le but de pouvoir utiliser des outils d'analyse et de vérification existants.

Les auteurs de [BBC+09] (1ère ligne du tableau 4.1) utilisent le langage intermédiaire FIACRE pour représenter le comportement et les aspects de synchronisation des systèmes. FIACRE (Format Intermédiaire pour les Architectures de Composants Répartis Embarqués) [BPB+08] est un langage intermédiaire formel basé sur les systèmes de transitions temporisés (TTS). Il a été développé pour factoriser les transformations entre les langages de modélisation comme AADL, SDL et UML, et les dialectes des outils de vérification comme TINA (Time petri Net Analyser) [BRV04] et CADP [GML+07]. Le langage Fiacre est aussi intégré dans l'environnement TOPCASED. En particulier, l'outil de vérification TINA, intégré à FIACRE, a permis aux auteurs d'entreprendre l'analyse Model-checking des descriptions temporisées.

La transformation d'un modèle AADL, adopté par les auteurs de [ACD+08], utilise le langage intermédiaire IF (2ième ligne du tableau de classification) pour produire un modèle d'automate temporisé. IF est un langage d'implantation intermédiaire des modèles, dont la sémantique opérationnelle est exprimée en termes de systèmes de transitions étiquetés en vue d'effectuer des vérifications formelles de propriétés. Le but de cette transformation est de pouvoir intégrer un processus d'analyse d'accessibilité et d'interblocage en exploitant le simulateur de modèles IF. Ce dernier permet de générer un graphe d'accessibilité représentant l'ensemble de toutes les actions possibles du modèle.

Une autre méthodologie de translation de plusieurs constructions du langage AADL (tels que les threads, les processus et les processeurs), ainsi que la spécification même de l'annexe de comportement, dans le langage intermédiaire BIP [CRB+08], a aussi utilisé les systèmes de transitions comme base sémantique. BIP (Behavior Interaction Priority), étant un langage formel développé par [Sif05] qui a une sémantique opérationnelle formelle définie en termes de systèmes de transitions étiquetées. L'outil développé autour de BIP fournit un atelier de développement logiciel basé sur une théorie de composition incrémentale de composants hétérogènes, ainsi que la génération de code. La translation en BIP a permis la simulation des

systèmes décrits en AADL [CRB+08] ainsi que la détection de l'inter-blocage basé composant [PBF09].

4.2.2 Modèles à base de Réseaux de Petri Temporisés (TPN)

Plusieurs autres travaux, dans la littérature, tentent de formaliser la description architecturale AADL tout en la transformant à d'autres langages possédant une sémantique formelle et plus adaptés aux outils de simulation et de vérification existants. Cette transformation peut être vue comme une base pour établir une sémantique formelle pour les modèles d'exécution AADL et exécuter la vérification des modèles AADL. Nous recensons dans cette classe de formalisation deux types de travaux. Les premiers utilisent une transformation de VTS (Visual Timed event Scenarios) aux réseaux de Petri temporisés (TPN) proposée dans [MOY+08]. VTS [ABK+04], est un langage visuel pour définir des propriétés complexes sur l'occurrence d'événements dans le temps, comme la réponse bornée, la corrélation d'événements, etc. Le formalisme sous-jacent est basé sur des ordres partiels. Dans le cas de AADL, la description VTS concerne des scénarios modélisant les interactions entre composants AADL, par exemple, l'envoi ou la réception d'un message, une interaction sur un changement interne d'état. Cette transformation permet la vérification des propriétés décrites en VTS avec des outils de model-checking basés sur les réseaux de Petri temporisés TPN tels que la sûreté et la vivacité.

Parallèlement à ce premier type de travaux, les auteurs dans [RKH09a] et [RKH09b] proposent une approche globale pour la construction des modèles de RdP à partir d'une architecture décrite en utilisant AADL. Il s'agit, dans ce cas, d'une transformation directe des spécifications AADL vers les notations des réseaux de Petri. Le but de cette transformation est d'élaborer une abstraction mathématique appropriée pour vérifier des propriétés comme l'absence d'inter-blocage (deadlock) ou les conditions de sécurité, tout en profitant pleinement des outils et techniques d'analyse spécifiques comme l'outil de vérification TINA.

Modèle Formel de Base	Eléments AADL Spécifiés	Langage Utilisé	Type d'Analyse	Outil
TTS (Système de Transition Temporisé	X	FIACRE	Model-Checking	TINA, CADP
Système de Transition Etiqueté	X	IF	Accessibilité, Interblocage	Simulateur IF
Système de Transition Etiqueté	Thread, Processus, Processeurs	BIP	Interblocage	Aldebaran Toolset
TPN (Ordres Partiels)	Interactions	VTS	Accessibilité, Interblocage	Outils Model Checking pour TPN
TASM	X	ATL	Consommation de Ressources, Synchronisation	UPPAAL & Toolset TASM
ACSR	X	X	Ordonnançabilité, Utilisation de Ressources, Contraintes de Synchronisation	Furness Toolset

Tableau 4.1 : Classification des approches de formalisation d'une description architecturale AADL

4.2.3 Modèles à base de la Machine à Etats Abstraits Temporisés (TASM)

La machine à états abstraits temporisés (TASM) étend le formalisme ASM pour permettre l'expression explicite de la synchronisation, des ressources, la communication, la composition, etc. dans le but de définir de manière formelle le comportement d'un système. Concernant l'utilisation de ce formalisme pour AADL, nous avons recensé une seule contribution où les auteurs proposent dans [YHM+09] une sémantique formelle pour l'annexe de comportement AADL basée sur les modèles TASM. Il s'agit là encore d'une transformation du modèle AADL en utilisant le langage de transformation ATL (Atlas Transformation Langage). Ce langage a été utilisé comme un moteur de transformation automatique pour exprimer la translation du modèle AADL vers un modèle TASM (Timed Abstract State Machine). Le moteur de transformation est intégré au plug-in OSATE-BA et offre un support de modélisation à l'annexe comportementale (Behavior Annex) dans l'environnement OSATE. Cette contribution a montré l'utilisation du modèle checker UPPAAL pour vérifier les propriétés de synchronisation et le Toolset TASM pour analyser les consommations de ressources.

4.2.4 Modèles à base d'algèbre de processus ACSR

De la même manière, la translation d'AADL dans ACSR est donnée dans [SLC09], où les auteurs fournissent une représentation sémantique d'AADL en utilisant une algèbre de processus en temps réel [BW90], utilisée comme base commune pour un simulateur d'AADL et un outil d'analyse d'ordonnançabilité. Ce simulateur se présente comme une technique d'analyse pour AADL permettant à l'utilisateur de suivre visuellement, à un niveau élevé d'abstraction, l'exécution du système et l'utilisation des ressources. L'analyseur d'ordonnançabilité permet de déterminer si le système a assez de ressources pour satisfaire les contraintes de synchronisation. Le simulateur et l'outil d'analyse ont été implémentés dans Furness toolset, un plug-in OSATE. Furness toolset utilise un codage de sémantique AADL dans une algèbre de processus temps réel ACSR, qui fournit une base sémantique commune pour les

différents outils d'analyse dans toolset.

Nous notons que les travaux de formalisation de AADL à base de ce type de modèles n'a pas nécessité le recours à un langage intermédiaire comme c'est le cas des modèles précédents.

4.2.5 Limites des modèles

Plusieurs tentatives de formalisation de la description architecturale AADL ont été publiées, pour définir une sémantique formelle aux modèles AADL permettant l'exécution et la simulation de leurs comportements, ainsi que la vérification formelle de leurs propriétés. La plupart de ces formalisations se limitent à quelques éléments architecturaux du langage AADL (voir colonne 2 du tableau 4.1). De plus, elles sont toutes couplées à des transformation de modèles AADL, soit dans un langage de modélisation intermédiaire dédié tels que : FIACRE, IF, ou ATL plus adaptés à l'analyse formelle, soit vers des langages possédant une sémantique formelle, comme BIP ou VTS, et plus orientés vers les outils de simulation et de vérification existants (voir colonne 3 du tableau 4.1).

Par ailleurs, les formalismes utilisés dans toutes ces approches influent sur le choix des méthodes d'analyse adoptées pour vérifier les propriétés comportementales des systèmes modélisés. Notons d'après la colonne « type d'analyse » du tableau de la figure 4.1 que la plupart des contributions, citées dans la section précédente, se sont concentrées sur la détection des inter-blocages (qui se produisent quand les threads sont de manière permanente bloqués pour attendre une ressource) et l'accès concurrent aux données partagées entre les composants.

De toute évidence, le langage AADL offre une multitude de notations pour spécifier l'architecture logicielle des systèmes. Une telle richesse du langage peut le rendre difficile à manipuler, et trop puissant pour certaines tâches et/ou utilisateurs. En conséquence, plusieurs chercheurs se sont penchés sur la formalisation de la sémantique de ce langage. À la lumière des travaux étudiés et de la classification

établie, nous constatons que tous ces travaux se limitent seulement à la formalisation de certains concepts AADL. En effet, le comportement complexe des threads, ces unités d'exécution concurrentes pour AADL, n'est pas formellement défini. Tous les formalismes utilisés sont plus adaptés aux outils de simulation et de vérification existants.

Nous estimons que *la logique de réécriture*, une autre catégorie de formalisme, est apte à formaliser tous les éléments architecturaux d'AADL quelque soit leur type, composant, connecteur, flux, mode, propriété, etc. Ceci est facilité par sa flexibilité dans la représentation syntaxique des éléments modélisés. D'autre part, un élément architectural ainsi modélisé fournit un modèle mathématique catégoriel définissant son comportement qui peut être concurrent et distribué. En fait, cette logique présente aussi l'avantage d'être un cadre sémantique unificateur de la plupart des formalismes utilisés pour formaliser le langage AADL, à savoir les réseaux de Petri [SMÖ01], les automates [ÖM02], les systèmes de transitions étiquetés [Mes96]. Il existe des outils pratiques implémentant tous les concepts théoriques de la logique de réécriture, tel que le système Maude [CDE+02] qui offre un environnement d'exécution et d'analyse des modèles AADL intégrés dans cette logique.

4.3 ABAReL : Une Annexe Comportementale pour AADL

AADL fournit des concepts de modélisation pour décrire l'architecture d'exécution des systèmes d'application en termes de taches concurrentes et leurs interactions. Toutefois, les descriptions comportementales plus fines sont reléguées à des extensions du langage. Les extensions peuvent prendre la forme de nouvelles propriétés et de notations spécifiques d'analyse qui peuvent être associées à la description architecturale sous forme d'annexes.

4.3.1 Les annexes AADL

Les annexes permettent d'incorporer des éléments rédigés dans une syntaxe différente de celle d'AADL. Elles permettent d'étendre la syntaxe AADL tout en utilisant des

outils existants. Plusieurs annexes du noyau AADL, utilisant différents formalismes, ont été approuvées et publiées par le standard. Nous résumons dans ce qui suit les annexes les plus connues. Nous accordons une attention particulière à la présentation de l'annexe comportementale [DBF+06], pour montrer les apports de notre annexe comportementale qui semble avoir des motivations similaires.

Annexe des notations graphiques : elle définit un ensemble de symboles graphiques pouvant être utilisés pour exprimer des relations entre composants, dispositifs, et connexions dans un modèle AADL. Les diagrammes graphiques d'AADL apportent plus de clarté et de compréhension au plan architectural d'un système embarqué. Ils permettent une présentation visuelle claire de la hiérarchie structurelle d'un système et de sa topologie de communication. Ces diagrammes sont parfaitement légaux selon la norme de noyau AADL si les symboles graphiques (voir chapitre 2, section 4.1.1) sont utilisés correctement dans le modèle graphique AADL.

Annexe des formats d'échange XMI : elle contient la définition du méta modèle AADL et les formats d'échange basés-XML pour les modèles AADL [SAE06]. Le méta modèle AADL décrit la structure des modèles AADL, c'est-à-dire, une représentation objet des spécifications AADL qui correspond sémantiquement à un arbre syntaxique abstrait. Il a été développé en utilisant la notation Ecore de l'environnement de modélisation Eclipse (EMF). Ce méta modèle est représenté comme ensemble de diagrammes de classes UML avec des propriétés additionnelles spécifiques-EMF qui supportent la génération automatique de méthodes pour la manipulation des modèles objet AADL et pour le stockage persistant et la récupération de tels modèles dans un ou plusieurs documents XML. En effet, cet environnement EMF produit également un schéma XML et un méta modèle XMI du méta modèle AADL. Ceci permet aux différents outils de l'environnement EMF, qui supportent le schéma AADL XML ou les spécifications XMI du méta modèle, d'interopérer sur des modèles AADL.

Annexe d'Interface de Programme d'Application : définit des règles pour que le texte source d'un langage de programmation spécifique soit conforme avec des spécifications d'architecture écrites en AADL. Cette annexe fournit des directives aux utilisateurs permettant d'assurer la transition entre les modèles AADL et le texte source d'un langage de programmation tels que ADA ou C. Elle préconise l'utilisation d'une API (Application Program Interface) entre le logiciel d'application et l'environnement d'exécution pour faciliter l'utilisation des modules contenant le code source d'application du langage dans un environnement d'exécution de l'architecture AADL.

Annexe du modèle d'erreur : définit des dispositifs pour permettre les spécifications de la gestion de redondance et des méthodes de mitigation de risque dans une architecture. Elle permet également des évaluations qualitatives et quantitatives des propriétés du système telles que la sûreté, la fiabilité, l'intégrité, la disponibilité, et la maintenabilité. Cette annexe définit un sous-langage qui peut être utilisé pour déclarer des modèles d'erreur dans une bibliothèque d'annexe d'erreur pour les associer aux composants dans des spécifications d'architecture. Elle définit également un sous-langage qui peut être utilisé dans une clause de l'annexe d'erreur d'une déclaration de l'implantation du noyau AADL standard.

L'annexe de comportement : permet d'exprimer les concepts de composition haute niveau, tels que les modes de synchronisation HRT-HOOD dans les modèles AADL par des annotations comportementales [DBF+06]. Ces annotations préservent la sémantique et les règles d'utilisation du langage AADL définies par le standard. Les extensions sont introduites par l'intermédiaire de nouvelles propriétés déclarées dans une clause « `Property Set` » ou via le mécanisme d'annexes. Les comportements décrits dans cette annexe sont vus comme des spécifications de comportements réels: ils peuvent donc être non déterministes. Ils sont basés sur les variables d'état dont l'évolution est spécifiée par les transitions exprimées dans l'annexe comportementale. Généralement, une spécification de cette annexe se présente comme suit :

```
annex behavior_specification {**
    <state variables> ?
    <initialization> ?
    <states> ?
    <transistions> ?
    <connections> ?
**} ;
```

Exemple 1 : Pour expliquer les fonctionnalités de cette annexe, nous présentons un exemple de description AADL d'un thread nommé `test` contenant la déclaration d'une annexe comportementale décrivant l'envoi et la réception de messages.

```
thread test
  features
    p_in : in event data port Behaviour ::integer ;
    p_out: out event data port Behaviour ::integer ;
end test;

thread implementation test.default
  subcomponents
    x: data Behaviour:: integer;
annex Behaviour_behavior {**
   states
     s0: initial state;
     s1: state;
   transitions
     s0 -[p_in?(x)] -> s1;
     s1 -[p_out!(x+1)] -> s0;
**}
end test.default;
```

Figure 4.1 – Exemple d'une annexe de comportement dans la description d'un thread

Le thread `test` reçoit un message à travers son port d'entrée `p_in.` La réception de ce message est spécifiée, dans cette annexe de comportement, par la première transition. Celle-ci fait passer le thread de l'état `s0` à l'état `s1` en défilant une donnée de la variable `x` au niveau de la garde `[p_in ?(x)].` La deuxième transition spécifie l'envoie de message à travers le port de sortie `p_out` où la

variable x de la garde `[p_out!(x+1)]` est incrémentée par 1 et l'état du thread transite vers `s0`.

Notons que cette annexe ne change pas la sémantique existante du noyau du langage, elle offre juste des extensions optionnelles. Elle est basée sur le modèle d'exécution AADL dont la sémantique doit être formalisée. Des tentatives de formalisation de ce modèle d'exécution ont été abordées dans les travaux de [YHM+09] en utilisant comme formalisme la machine d'état abstraite temporisée (TASM). Néanmoins, cette contribution et plusieurs autres (voir section 4.2) se limitent uniquement à la formalisation de certains concepts AADL et le comportement complexe d'un thread n'est pas formellement défini. Nous présentons dans la section suivante une définition alternative d'une annexe comportementale basée sur la logique de réécriture permettant de compléter les insuffisances constatées lors de l'utilisation de l'annexe existante.

4.3.2 Principe d'ABAReL

Le standard décrit le comportement dynamique de l'architecture d'exécution d'un système embarqué en modélisant tout simplement des modes opérationnels et des transitions de mode (SAE04). Les transitions de mode représentent la commutation entre les configurations d'exécution d'un système. Celles-ci peuvent avoir l'effet d'activation ou désactivation des threads pour l'exécution, un changement du chemin de connexion entre les threads, ou des changements des caractéristiques internes des composants. Par conséquent, un thread peut être actif dans un mode et inactif dans un autre, seul les threads actifs exécutent leurs instructions.

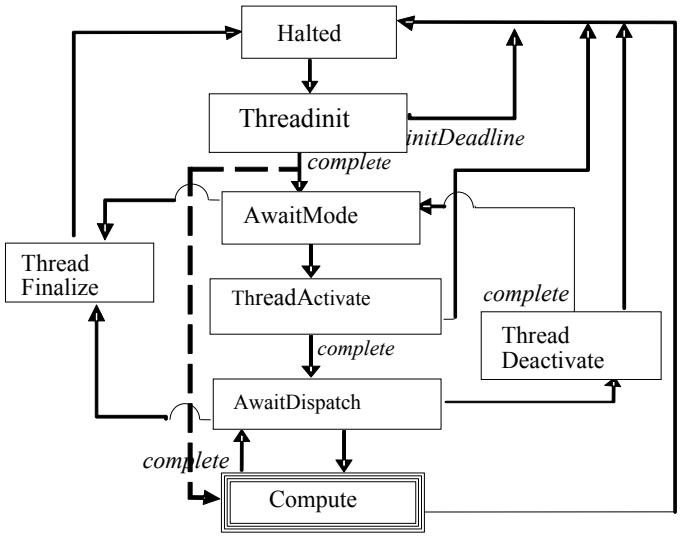


Figure 4.2 - Le modèle états-transitions pour un thread

Le standard AADL définit aussi le comportement des threads, ces unités d'exécution concurrente, en associant à chaque thread une sémantique dynamique à base d'automate décrivant ses états et les conditions de transition de ses états. La figure 4.2 montre qu'un *thread* peut être stoppé, inactif, ou en activité. Un *thread* actif peut être en attente pour une expédition, état *AwaitDispatch,* ou en exécution, état *Compute*.

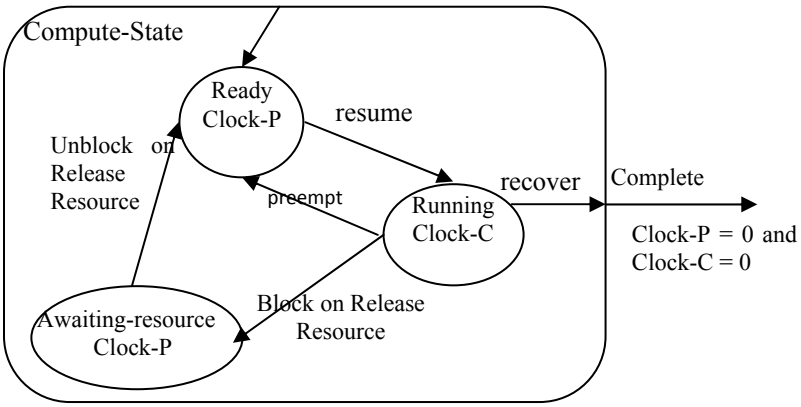


Figure 4.3 - Etat *Compute* d'un thread

De plus, dans l'état *Compute* de cet automate, qualifié de hiérarchique, le thread peut avoir d'autres sous états (figure 4.3) où il peut être prêt pour l'exécution (*Ready*), en exécution (*Running*), ou bloqué sur l'accès d'une ressource (*Awaiting-Resource*). Cependant, un thread dans une architecture logicielle d'un système embarqué en exécution, reçoit des données et/ou des événements en entrée, il effectue le calcul et envoie des signaux (données et/ou événements) en sortie. Ceci, s'effectue au niveau de l'état *Compute* de cet automate, où particulièrement des changements de l'état local du thread et de l'état de chacun de ses ports de connexion, sont visibles à travers sa configuration. Or, toute cette sémantique d'exécution n'apparaît pas au niveau de la description AADL du thread. En effet, au niveau d'un modèle AADL n'apparait pas la sémantique d'exécution d'un composant en général ni celle d'un thread, cette unité d'exécution concurrente qui se distingue par son comportement assez complexe. Une description AADL étendue par les concepts de l'annexe comportementale du standard se contente d'une clause de déclaration des différents modes possibles, suivie par une autre clause pour les propriétés d'exécution (*Compute-Execution-Time* et *Period*) spécifiques à chaque mode.

Dans ce contexte, nous définissons un cadre sémantique formel pour la description architecturale AADL. Il s'agit d'ABAReL (**A**ADL **B**ehavioral **A**nnex based on revised **Re**writing **L**ogic), une annexe comportementale pour AADL basée sur la logique de réécriture révisée [BBL08, BB^{+}09, BBB10]. Cette annexe est proposée dans le but de combler les limites cernées lors de l'utilisation de l'annexe du standard. Elle permet d'associer un seul modèle formel à la définition de la structure architecturale et le comportement d'un composant AADL. En particulier, la sémantique d'exécution d'un composant Thread telle qu'elle se présente dans le standard est prise en charge de façon naturelle. De plus, ce modèle qui hérite des concepts d'une catégorie mathématique fournit un prototype exécutable d'un composant AADL sous l'environnement Maude, facilitant ainsi son analyse formelle. Dans les sections suivantes nous détaillons de façon incrémentale l'approche adoptée pour définir ABAReL.

4.3.3 Formalisation d'une architecture AADL

Dans cette section, nous recensons les éléments architecturaux AADL les plus importants pour décrire le modèle formel, à base de concepts de la logique de réécriture révisée, d'une architecture AADL d'un système embarqué temps réel. Pour ce faire, nous rappelons que le langage AADL modélise l'architecture d'un tel système en termes de système d'application lié à une plateforme d'exécution. La majeure partie de l'effort de modélisation est dirigée vers le logiciel spécifique de l'application à un niveau élevé d'abstraction indépendamment des détails d'implémentation. Dans ce langage, la syntaxe est basée sur trois concepts de base : les *composants*, les *connexions*, et les *configurations*. De plus, l'architecture logicielle est modélisée en termes de *composants* et *interactions*, où l'*interaction* représente le *flux* de *données* et/ou d'*événements* traversant une ou plusieurs *connexion*s avec un temps de latence lié au flux.

Partant de ce constat, nous proposons des règles génériques permettant d'abstraire les éléments architecturaux du langage AADL pour les exprimer en termes des théorie de réécriture étendues, plus précisément des théories de réécriture orientées objet temps réel.

Notre approche de formalisation [BBB11] associe à chaque élément architectural une sémantique à base de concepts orientés objet temps réel de la logique de réécriture révisée (tableau 4.2). Donc, une configuration $\boldsymbol{A}$, dans une architecture AADL, est modélisée par une théorie de réécriture orientée objet temps réel $\boldsymbol{T_A}$. La spécification d'une telle théorie se fait en termes d'objets, de messages, de classes et d'héritage. Nous explicitons, dans ce qui suit, la spécification de cette théorie de réécriture orientée objet temps réel $\boldsymbol{T_A}$ en adoptant la syntaxe de l'outil RT-Maude [Ölv07].

Elément Architectural AADL	**Concept orienté objet temps réel de la logique de réécriture révisée**
Configuration AADL ***A***	Théorie de réécriture orientée objet temps réel $\mathbf{T_A}$
Composant ***C*** (de la configuration ***A***)	Classe *Component* ***C***
Une instance de composant ***O***	Un objet ***O*** de la classe ***C***
Catégorie de composant	Attribut *Category* de la classe ***C***
Interfaces fonctionnelles ou *Features* d'une instance de composant	Attributs de ***C***: *IPortN* , *OPortN*
Flux ***F***	Message ***F***
Interaction entre composants	Règle de réécriture Rl : m < $O_1 : C_1$ > $\rightarrow$ < $O_2 : C_2$ >
Connexion ***Cx***	Classe Connection ***Cx***

Tableau 4.2 - Sémantique des éléments architecturaux AADL en termes de logique de réécriture révisée

Plus précisément, un ensemble de composant ***C***, dans sa déclaration de type (*component type*) en AADL, est modélisé par la classe **`Component`** dont les attributs sont respectivement : 1) `Category` pour déterminer la catégorie du composant, 2) `IPortN` et `OPortN` pour représenter ses interfaces fonctionnelles. La valeur de l'attribut `Category` peut être : `system, process, thread, device.` L'ensemble des connexions, entre les différents composants O_i dans une configuration ***A***, est spécifié par la classe **`Connection`**. Chaque connexion ***Cx*** possède les attributs `TypeConx`, `source`, `Dest`, et `TimeTrans` dénotant le *type* de donnée ou d'événement qu'elle doit véhiculer d'un port sortant du composant *source* vers un port entrant du composant *destination,* avec un temps de transmission `TimeTrans`. Les composants *source* et *destination* sont des objets O_i de la classe `Component`. Le type de l'information à transporter est déclaré par la sorte `DataType` et peut prendre une des valeurs : `event`, `data`, ou bien `event data`. L'ensemble des flux traversant les connexions de la configuration ***A*** est modélisé par l'envoi de messages. Autrement dit le passage d'un flux ***F*** de données et/ou

d'événements à travers une connexion ***Cx*** se fait par la transmission d'un message **Msg** entre les composants O_i et O_j reliés par la connexion Cx_k. Nous introduisons aussi le délai de transmission du message par la sorte `DlyMsg` qui modélise le temps de latence du flux, où l'opérateur `dly(m,t)` indique qu'il reste **`t`** unité de temps pour l'arrivée du Msg **m** à destination. Nous illustrons l'approche de formalisation d'une architecture AADL par l'annexe ABAReL à travers la théorie de réécriture générique présentée dans la figure 4.4.

```
**********structure de la classe des composants **********
sorts Cat PortName .
ops system process thread device : -> Category .
class Component | Category : Cat , IPortN : PortName, OPortN : PortName .
**********structure de la classe des connexions **********
sort DataType .
ops data event data-event : -> DataType .
class Connection | TypeConx : DataType , Source : Oid , Dest : Oid ,
                   TimeTrans : Time .
**********spécification de flux par l'envoi de messages**********
sort DlyMsg .
subsorts Msg < DlyMsg < Configuration.
op dly : Msg Time -> DlyMsg [ctor] .
msg from_to_transfer_ : Oid Oid DataType -> Msg .
var ms : Msg .
eq dly(ms, 0) = ms .
```

Figure 4.4 - Théorie de réécriture générique $\boldsymbol{T_A}$ formalisant une configuration AADL ***A***

Exemple 1 : Pour illustrer notre approche de formalisation, nous considérons dans un premier temps un exemple d'une architecture AADL dont la représentation graphique est donnée dans la figure 4.5. Cette architecture est constituée de quatre composants dont trois sont de catégorie `device` nommés respectivement : `capteur_pedal`, `capteur_vitesse` et `actuateur_frein`, et un composant de catégorie `system` nommé `controleur_abs`. Nous avons sélectionné cette configuration du modèle architectural AADL du système ABS présenté dans le chapitre 3.

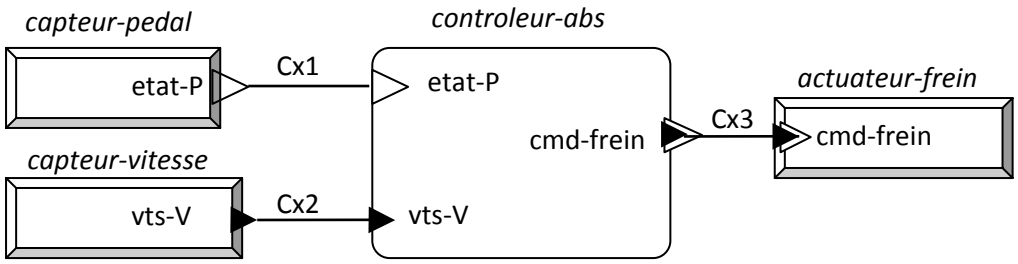


Figure 4.5 - Exemple d'une architecture AADL

Architecture AADL de l'exemple	Spécification Formelle dans ABAReL

`device capteur-pedal`
`  features`
`  etat_P: out event port;`
`  flows`
`  F1: flow source etat-P {`
`  Latency => 10Ms;`
`  };`
`end capteur-pedal;`
	< *capteur-pedal* : Component | Category : device , OPortN : etat_P >
	< Cx1 : Connection | TypeConx : event, Source : *capteur-pedal* , Dest : *controleur-abs* , TimeTrans : 10 >

`device capteur-vitesse`
`  features`
`  vts-V: out data port;`
`  flows`
`  F2: flow source vts-V {`
`  Latency => 15Ms;`
`  };`
`end capteur-vitesse;`
	< *capteur-vitesse* : Component | Category : device , OPortN : *vts-V* >
	< Cx2 : Connection | TypeConx : data, Source : *capteur-vitesse* , Dest : *controleur-abs* , TimeTrans : 15 >

`system controleur-abs`
`  features`
`  etat-P: in event port;`
`  vts-V: in data port;`
`  cmd-frein: out event data port;`
`end controleur-abs;`
	< *controleur-abs* : Component | Category : system , IPortN : *etat_P* , IPortN : *vts-V*, OPortN : *cmd-frein* >
	< Cx3 : Connection | TypeConx : event, Source : *controleur-abs*, Dest : *actuateur-frein* , TimeTrans : 10 >

`device actuateur-frein`
`  features`
`  cmd-frein: in event data port;`
`  flows`
`  F3: flow sink cmd-frein {`
`  Latency => 10Ms;`
`  };`
`end actuateur-frein;`
	< *actuateur-frein* : Component | Category : device , IPortN : cmd-frein >
	**exemple de transmission d'un message **
	dly (**from** *capteur-pedal* **to** *controleur-abs* **transfer** event 1, 10)

Tableau 4.3 - Formalisation d'un exemple d'une architecture AADL dans ABAReL

Dans un deuxième temps, nous présentons, dans le tableau 4.3, la description AADL de l'architecture de la figure 4.5, et sa transcription en termes de concepts de la théorie de réécriture orientée objet temps réel de la figure 4.4.

A cette étape de la formalisation d'une architecture AADL, nous avons considéré une configuration constituée d'un ensemble de composants de catégories différentes reliés par des connexions. La communication entre ces composants a été formalisée par la transmission de message à travers les connexions établies entre les composants. Pour chaque composant, nous avons pris en considération sa déclaration AADL de type (*component type*) où il n'est caractérisé que par sa catégorie et ses interfaces fonctionnelles (*features*). Rappelons que l'implantation d'un composant AADL (*component implementation*) décrit sa structure interne par la déclaration d'autres clauses comme par exemple les propriétés, les modes et les sous composants. Notons également que chaque catégorie de composant a un rôle bien déterminé qu'elle doit jouer dans l'organisation de la structure de l'architecture ou dans l'interprétation de son comportement. Par exemple, le composant *process* définit des espaces mémoire pour contenir les threads et les données partagées entre eux. Les threads représentent les composants applicatifs actifs.

Notre approche de formalisation est générique et extensible, elle permet de formaliser l'implantation de chaque catégorie de composant tout en considérant le rôle qu'il doit interpréter au niveau de la structure ou du comportement d'une l'architecture AADL. Par exemple, pour spécifier la sémantique de la clause des sous composants (*subcomponent*) dans l'implantation d'un composant, on doit respecter les règles du tableau 2.1 (Chapitre 2, Section 2.4) établit par le standard interdisant certaines combinaisons dans la structuration de la hiérarchie. La logique de réécriture révisée permet de décrire naturellement et intuitivement de telles sémantiques ceci grâce à sa flexibilité et son extensibilité.

4.3.4 Sémantique d'un thread dans ABAReL

Nous nous intéressons à présent à la formalisation de la sémantique du composant AADL *Thread* étant donné qu'il représente le seul composant applicatif actif dans une architecture AADL. La formalisation de ses aspects structuraux et comportementaux contribue massivement à la spécification du comportement dans une architecture AADL. Pour ce faire, ABAReL offre un cadre sémantique approprié. Nous commençons par définir la sémantique d'un composant de type Thread dans ABAReL. Ensuite, nous considérons plusieurs threads en interaction après avoir instancié l'architecture AADL générique déjà formalisée.

Formalisation des aspects architecturaux

Le thread est un composant AADL, autrement dit, un objet de la classe `Component`, selon notre formalisation, dont la valeur de l'attribut `Category` est égale à `thread`. Généralement l'implantation d'un thread est décrite par la déclaration des *modes* et des *propriétés*. En guise de formalisation de cette description AADL, nous proposons dans le tableau suivant (tableau 4.4) des règles génériques pour formaliser tous les aspects architecturaux d'un thread.

L'implantation d'un thread **Th** est modélisée par la classe `ThreadImpl` (voir tableau 4.4), comme étant une sous classe de la classe `Component` pour bénéficier de la notion d'héritage. Donc, certains attributs peuvent être hérités de la classe `Component`, comme par exemple l'attribut `Category` et les attributs spécifiant les interfaces de communication d'un thread (`IPortN`, `OPortN`). Les attributs de la classe `ThreadImpl` prennent en considération les *modes* pour spécifier le mode opérationnel du thread, les *propriétés temporelles* et les *états*.

Un Thread AADL Th	**Théorie de réécriture orientée objet temps réel $\mathfrak{R}_{Th}$**
Thread **Th**	Valeur de l'attribut *Category* de la classe C*omponent = Thread*
Interfaces de communication	Attributs : *InBufferPort, AccessData, OutBufferPort* Attributs hérités : *IPortN, OPortN*,
Implantation	Classe *ThreadImpl*, sous classe de la classe *Component*
Modes	Attribut *mode* de la classe *ThreadImpl*
Propriétés temporelles	**Attributs de la classe *ThreadImpl***
Period	Attribut *Period,* Attribut *Clock-P*
Compute_Execution_Time	Attribut *Execution-time,* Attribut *Clock-C*
Etats	**Attributs de la classe *ThreadImpl***
Etats des ports de communication	Attributs *IPort , OPort*
Etats du thread	Attribut *ThreadState*
Etats composite du thread	Attribut *SubState*

Tableau 4.4 - Sémantique d'un thread dans ABAReL

Les modes sont directement modélisés par l'attribut `Modes`. Pour prévoir l'utilisation des interfaces de communication au cours de l'exécution d'un thread, nous ajoutons les attributs `InBufferPort`, `OutBufferPort` et `AccessData` pour représenter des composants `Data` sous forme de buffers reliés à chaque port de connexion.

Pour les propriétés d'exécution temporelles, nous spécifions à travers ABAReL les propriétés `Period` et `Compute-Execution-Time` par les attributs `Period`

et `Execution-Time`. Les valeurs capturées de ces propriétés sont prises en considération par ABAReL pour assurer les conditions d'exécution d'un thread dans une configuration architecturale AADL. Pour la mise en garde de la période et du temps d'exécution d'un thread, ABAReL associe à chacune de ses propriétés une horloge. Ces horloges sont nommées respectivement `Clock-P` et `Clock-C`.

Nous enrichissons également la description AADL d'un thread par la spécification des états (tableau 4.4). Nous considérons les états et les sous états (ou états composites) d'un thread, et ceux des ports de communication liés au thread déclarés par les attributs `IPort` et `OPort`. Les états d'un thread sont spécifiés par l'attribut `ThreadState.` L'attribut `SubState` spécifie ses sous états.

```
sorts  modes  Sstate  PortState  Tstate  Data .
var  subS : Sstate .
vars T  T1  T2  Imp : Oid .
vars  M1  M2  :  Modes .
op EmptyFile : -> File .                    *** constant pour exprimer  la  file vide***

****structure de la classe ThreadImpl incluant les états et les propriétés***

class ThreadImpl |  Modes : modes,     ThreadState :  Tstate ,    SubState : Sstate,
                    IPort : PortState ,   OPort : PortState ,
                    InBufferPort : File ,   AccessData : Data , OutBufferPort : File ,
                    Period : Time,              Clock-P : Time,
                    Execution-time : Time,    Clock-C : Time  .
subclass ThreadImpl < Component .

**********déclaration des états et sous états d'un thread**********
ops wait  compute : -> Cstate .
op none : -> PortState .
ops waitIn  waitOut receive send : -> PortState .
subsort Sstate < CompState .
ops  noSub Running Ready Awaiting-resource Complete Recover : -> Sstate [ctor] .
```

Figure 4.6 - Formalisation de l'implantation d'un thread AADL par ABAReL

La spécification générique de l'implantation d'un thread dans ABAReL est donnée, dans la figure 4.6, par la classe `ThreadImpl` déclarée comme une sous classe de la classe `Component`. Les attributs de cette classe sont de sortes : *Modes* décrivant le mode opérationnel du thread, *Sstate* spécifiant les sous états de l'état composite *Compute*, et *Time* (sorte prédéfinie) permettant de spécifier d'une part, les propriétés d'exécution temporelle (*Period* et *Execution-time*) et d'autre part, les horloges *Clock-P* et *Clock-C*.

Nous définissons donc la sémantique d'un thread telle qu'elle a été définie dans le standard, par une théorie de réécriture orientée objet temps réel comme suit.

Définition 1 : *A chaque composant Thread AADL Th, est associé une théorie de réécriture orientée objet temps réel* $\mathfrak{R}_{Th} = (\Sigma_{Th}, E_{Th}, \Phi_{Th}, R_{Th})$ *où :* (Σ_{Th}, E_{Th}) *est une théorie équationnelle d'adhésion (membership) décrivant la structure statique du thread Th.*

1. *La signature* Σ_{Th} *spécifie l'ensemble des sortes et sous sortes, et l'ensemble des opérateurs utiles pour décrire chaque clause de sa description globale: ports de communication (features), propriétés, flux, modes, états, sous états, type et implantation.*

2. *L'ensemble des équations* E_{Th} *contient entre autre les attributs associés à certains opérateurs.*

3. *La fonction* Φ_{Th} *représente l'ensemble des opérateurs considérés dans* Σ_{Th} *pouvant geler, dans certains cas, leurs arguments.*

4. *Les règles de réécriture* R_{Th} *décrivent le comportement dynamique du thread Th selon un suivi de ses configurations comportant dans chaque étape l'état de chacun de ses ports de connexion, son état local et les sous états de cet état local.*

Formalisation de l'aspect comportemental

La formalisation de l'aspect comportement d'un thread doit commencer par la spécification des changements visibles des états d'un thread suivi par la spécification de ses états internes en tenant compte de ses états concurrents et hiérarchiques. Nous déclarons donc, dans la figure 4.6, avec des opérateurs constructeurs, les états du thread (`wait` ou `compute`), les états des ports de communication (`waitIn`, `waitOut`, `receive` et `send`) et les sous états de l'état `compute` (`noSub, Running, Ready, Awaiting-resource, Complete, Recover`).

La spécification des changements visibles des états d'un thread, lié à ses ports de connexion, est donnée dans la figure 4.7 par les règles de réécriture `Data-Receive` et `Data-Send`. La règle `Data-Receive` spécifie la réception d'un message contenant des données à travers une connexion, liée au port d'entée `IPort`. La donnée reçue est aussitôt enfilée dans le buffer `InBufferPort` lié au port d'entrée. Cette règle de réécriture prépare le thread pour l'exécution d'une nouvelle période en faisant passer son état (de `wait` à *`compute`*), son sous état (de `noSub` à `Ready`) et initialise les horloges par les valeurs capturées à partir des propriétés d'exécution dans le mode de fonctionnement actif.

Après écoulement de la période et du temps d'exécution, la règle de réécriture `Data-Send` remet le thread à son état initial (`TState` à `wait`, `Substate` à `noSub` et `OPort` à `waitOut`) et génère un message pour transmettre le résultat de son exécution. Elle défile donc la donnée à transmettre du buffer `OutBufferPort` lié au port de sortie du thread, pour l'expédier au thread voisin. On peut également expédier ce résultat d'exécution du thread à plusieurs threads voisins en générant plusieurs messages si des connexions entre ce thread et les threads voisins sont établies.

```
var cx : Oid .
vars DT Temp : Data .
vars L  L1  L2 : File .
vars D D1 : DataType .
vars R  R' R'' R1 R2 R3 R4 : Time .

rl [Data-Receive] : (from T1 to T  transfer DT)
    < T : ThreadImpl | IPort : waitIn,  TState : wait , Substate : noSub , Clock-P : 0, Clock-
C : 0,  Period : R ,  Execution-time : R1 ,  InBufferPort : L2 >
        < cx : Connection | TypeConx  : D ,  Source : T1 ,  Dest  : T >
 =>      < T : ThreadImpl | IPort : receive ,  TState : compute , Substate : Ready ,
     Period : R ,  Execution-time : R1 ,  Clock-P : R, Clock-C : R1 ,  InBufferPort : DT  ; L2 >
      < cx : Connection | TypeConx  : D ,  Source : T1 ,  Dest  : T >   .

rl [Data-Send] :  < T : ThreadImpl |  OPort : send , TState : compute , Substate :
           Complete , OutBufferPort : L ; DT ,  Clock-P : R1,  Clock-C : R2 >
    < cx : Connection | TypeConx  : D ,  Source : T ,  Dest  : T2 , TimeTrans :  R >

 =>    < T : ThreadImpl |  OPort : waitOut , TState : wait , Substate : noSub ,
           OutBufferPort : L ,  Clock-P : 0,  Clock-C : 0  >
    < cx : Connection | TypeConx  : D ,  Source : T ,  Dest  : T2 ,  TimeTrans :  R >
    dly(from T to T2 transfer DT, R)  .
```

Figure 4.7 - Spécification des changements visibles des états d'un thread AADL

Ces règles considèrent un thread dans la configuration d'une architecture logicielle où le passage d'un flux de données est spécifié par la transmission de messages entre threads (ou entre un thread et un composant de catégorie `device`) si une connexion est déclarée entre eux.

Pour compléter la formalisation du comportement, nous considérons les états composites de l'état hiérarchique actif `Compute` d'un thread. Nous spécifions, par les règles de réécriture de la figure 4.8, ces sous états et leurs transitions en tenant compte des valeurs capturées à partir des propriétés d'exécution (`Compute-Execution-Time` et `Period`). La première règle de réécriture, de la figure 4.8, fait passer le thread du sous état `Ready` au sous état `Running`.

```
Crl [init] : < Imp : ThreadImpl | InBufferPort: L, AccessData:
      EmptyFile, Substate: Ready, Clock-C: R'>
=>  < Imp: ThreadImpl | InBufferPort: Queu(L), AccessData: Head(L),
      Substate: Running,  Clock-C: R' > if (R' > 0) .
```

```
crl[resume]: <Imp : ThreadImpl | AccessData : L , Substate : Ready,
      Clock-C: R'>
=>  < Imp: ThreadImpl | AccessData: L, Substate: Running, Clock-C:
      R' > if (R' > 0) /\ (L =/= EmptyFile) .
```

```
crl[preempt]: < Imp: ThreadImpl | Substate: Running, Clock-C : R',
      MaxPreempt : N >
=> < Imp : ThreadImpl | Substate : Ready, Clock-C : R', MaxPreempt:
      N + 1 >   if (R' > 0) /\ (N < 2) .
```

```
crl[block-on-Release-Resource]: < Imp: ThreadImpl | Substate:
      Running, Clock-C: R', MaxData : N >
=>  < Imp: ThreadImpl | Substate: Awaiting-resource , Clock-C: R',
      MaxData: N + 1 > if (R' > 0) /\ (N < 2) .
```

```
crl[Unblock-on-Release-Resource]: < Imp : ThreadImpl | Substate:
      Awaiting-resource, InBufferPort : L, Clock-C : R' >
=>  < Imp : ThreadImpl | Substate: Ready, InBufferPort : Queu(L) ,
      AccessData : Head(L),   Clock-C : R' > if (R' > 0) /\
        (L =/=  EmptyFile) .
```

```
crl[recover]:  < Imp: ThreadImpl | TState: compute , Substate :
      subS , Clock-P : R, Clock-C : R' >
=>  < Imp: ThreadImpl | TState: compute, Substate: Recover,
      Clock-P: R, Clock-C: R'> if ((R > 0) /\ (R' ==  0) /\
      (subS =/= Recover)) .
```

```
crl [complete-Rec] : < Imp: ThreadImpl | IPort : receive , TState :
      compute, Substate: subS, AccessData: Temp2, OutBufferPort: L,
      OPort: waitOut, Clock-P: R, Clock-C : R' >
=>  < Imp: ThreadImpl | IPort: waitIn , TState: compute,
AccessData:
      EmptyFile , OutBufferPort: Temp2 ; L , Substate: Complete,
      OPortS :  send, Clock-P : R, Clock-C : R' >  if (R == 0) .
```

```
crl[complete-Rec2]: < Imp : ThreadImpl | IPortS : receive, TState :
      compute, Substate : subS, AccessData: Temp, OPortS: NoPort ,
      Clock-P : R, Clock-C : R' >
=> < Imp : ThreadImpl |  IPortS: receive, TState: compute,
      AccessData: EmptyFile , Substate: Complete, OPortS : NoPort ,
      Clock-P : R,Clock-C : R' > if (R == 0) .
```

```
rl[finish]: <Imp : ThreadImpl | IPort: receive, TState: compute,
      Substate: Complete , OPort: NoPort , Clock-P: R, Clock-C: R',
      MaxPreempt: N, MaxData: N1 >
=>  < Imp: ThreadImpl | IPort: waitIn, TState: wait, Substate:
      noSub, OPort: NoPort, Clock-P : 0, Clock-C : 0 , MaxPreempt :
      0 ,  MaxData : 0  > .
```

Figure 4.8 - Spécification des sous états de l'état '*Compute*' et leurs transitions

La règle de réécriture `complete-Rec` finalise l'écoulement de la période du thread si le temps d'exécution est terminé (`Clock-C = 0`) et remet le sous état du thread à `Complete`. Elle prépare également la transmission du résultat, qu'elle défile de son buffer `OutBufferPort`, à travers son port de sortie `OPort` en remettant son état à `send`. De la même manière, la règle de réécriture `complete-Rec2` termine l'écoulement de la période en considérant le cas où le thread ne possède pas de port de sortie (`OPortS = NoPort`).

Les règles de réécritures conditionnelles `preempt`, `block-on-Release-Resource`, `Unblock-on-Release-Resource` et `recover` représentent les transitions entre les sous états de l'état `Compute` (figure 4.3) avec une mise en garde de la période et du temps d'exécution par les horloges `Clock-P` et `Clock-C`. Cette mise en garde est spécifiée, dans une théorie de réécriture temps réel, par les opérations `delta` et `mte`. L'opération `delta` calcule l'effet du passage d'un temps R sur la configuration du thread. L'opération `mte` évalue le temps maximal pouvant s'écouler avant qu'une action importante ne s'exécute (ici c'est le minimum des valeurs des deux horloges).

La règle `tick` fait usage des deux opérations `delta` et `mte` pour interpréter l'écoulement du temps. La règle `tick` et les opérations `delta` et `mte` sont spécifiés dans la figure 4.9.

L'écoulement du temps sur la configuration d'un thread est interprété, dans notre cas, par la décrémentation synchrone des horloges. L'attribut `nonexec` de la règle `tick` exprime le fait que cette règle fait avancer le temps lorsqu'aucune règle (non `tick`) n'est exécutable.

```
**********Spécification de la règle tick **********

crl [tick] : {C:Configuration}  =>  {delta(C:Configuration, R)} in time R
              if R <= mte(C:Configuration) [nonexec] .
op delta : Configuration Time -> Configuration [frozen (1)] .
op mte : Configuration -> TimeInf [frozen (1)] .
vars NeC  NeC' : Configuration .

********** Spécification de l'opération delta**********

ceq delta(NeC NeC', R) =  delta(NeC, R)  delta(NeC', R)  if
                (NeC =/= none) /\  (NeC'  =/=  none)  .

eq delta(none , R) = none  .

eq delta(< cx : Connection | TypeConx : D, Source : T , Dest : T1  , TimeTrans : R' >, R) =
    < cx : Connection | TypeConx : D, Source : T , Dest : T1 , TimeTrans : R' > .

eq delta(dly(ms, R), R') = dly(ms, R monus R') .

ceq delta(< Imp : ThreadImpl | Substate : subS, Clock-P : R, Clock-C : R' >, R'') =
      < Imp : ThreadImpl | Substate : subS, Clock-P : R monus R'', Clock-C : R'  >
      if (subS =/= Running) .
eq delta(< Imp : ThreadImpl | Substate : Running, Clock-P : R , Clock-C : R' >, R'') =
< Imp : ThreadImpl | Substate : Running, Clock-P : R monus R'' , Clock-C : R' monus R'' >  .

**********Spécification de l'opération mte **********

eq mte(< Imp : ThreadImpl | Substate : noSub, Clock-P : R'', Clock-C : R' >) = INF   .
eq mte(< Imp : ThreadImpl | Substate : Recover, Clock-P : R'', Clock-C : R' >) = R''   .
eq mte(< Imp : ThreadImpl |  Substate : subS , Clock-P : R'', Clock-C : R' >) = min(R'', R')   .
eq mte(< cx : Connection | TypeConx : D, Source : T , Dest : T1 ,
          TimeTrans : R' > , R) =  min( R',R) .
eq mte(ms) = INF .
eq mte(dly(ms, R)) = INF .
eq mte(none) = INF .
ceq mte(NeC NeC') = min(mte(NeC), mte(NeC')) if (NeC =/= none) /\ (NeC'  =/=  none) .
```

Figure 4.9 - Spécification de l'écoulement du temps sur une configuration

L'opération `delta` modifie uniquement les attributs de sorte `Time`. L'attribut `frozen (1)` de cette opération permet de figer les paramètres de l'opération au cours de la substitution de la totalité des équations. Ceci, permet d'éviter les substitutions non nécessaires et non confluentes. Les équations de la

figure 4.9 concernant la spécification de l'opération `delta` calculent l'effet de cette opération sur les composants, les connexions et sur la transmission des messages, où nous introduisons un temps d'envoi et un temps de réception pour chaque message. Elles expriment ensuite, la distribution de l'opération `delta` sur l'ensemble de la configuration pour uniformiser le passage du temps.

L'évaluation de l'opération `mte` est donnée par les équations de la figure 4.9. Cette évaluation considère le calcul du `mte` sur : un thread initialisé (s'il est à l'état `compute`) en initialisant ses horloges par les valeurs des propriétés d'exécution, un composant non initialisé, les messages et les connexions. Elle considère ensuite la distribution de l'opération *mte* sur la configuration.

Exemple 2 : la spécification du comportement d'un thread par ABAReL est illustrée à travers l'exemple du thread `test-vitesse-V` suivant. Il s'agit d'un thread pris du modèle architectural du système ABS (chapitre 2, section 7) dont sa description AADL est comme suit :

Exemple d'un thread AADL	**Spécification ABAReL**

Description de type d'un thread
`thread test_vitesse_V`
`  features`
`  vitesse_in: in data port;`
`  vitesse_out: out data port;`
`end test_vitesse_V;`
 	
Description de l'implantation d'un thread
`thread implementation test_vitesse_V.Impl`
`  properties`
`  Period => 22 Ms in modes (pret);`
`  Compute_Execution_Time => 12 Ms`
`  in modes prêt ;`
`end test_vitesse_V.ThreadImpl1;`
	< test-vitesse-V : Component|
	Category : thread , mode : prêt ,
	spécification des ports de connexion
	IPort : vitesse-in , InBufferPort : EmptyFile , OPortN : vitesse-out, OutBufferPort : EmptyFile , AccessData : EmptyFile,
	spécification des propriétés
	Period : 22, Clock-P: 0, Execution-Time : 12 , Clock-C : 0 >
	spécification des états
	IPort : waitIn, OPort : waitOut, ThreadState : wait , SubState : noSub>

Tableau 4.5: Exemple d'un thread AADL formalisé dans ABAReL

Le résultat de cette transcription est une configuration d'exécution du thread `test-vitesse-V` mentionnant son état initial. Nous pouvons par la suite appliquer facilement les règles de réécriture du modèle mathématique associé au composant thread (figures 4.7 et 4.8) pour spécifier son aspect comportemental.

4.4 Apport sémantique d'ABAReL pour AADL

La logique de réécriture offre à l'annexe comportementale ABAReL un cadre sémantique puissant pour décrire les aspects statique et comportemental d'une architecture AADL composée d'une collection de composants. En effet, la sémantique complexe du composant applicatif Thread a été naturellement définie par ABAReL. L'illustration de notre approche de formalisation à travers ce type de composant a montré d'une part la **généricité** de la solution adoptée et d'autre part son pouvoir d'**extension**. Nous montrerons, dans le chapitre 6, comment étendre notre modèle afin qu'il puisse associer une sémantique rigoureuse à une configuration AADL composée de plusieurs threads en interaction. Cette sémantique permet l'exécution de cette architecture AADL et la vérification de ses propriétés.

Nous résumons dans ce qui suit les points jugés les plus importants que l'annexe comportementale ABAReL procure à une description architecturale AADL :

1. Un cadre opérationnel qui sert à la spécification des comportements atomiques et concurrents de tout composant AADL en particulier le Thread. Il existe actuellement des outils et des systèmes de réécritures qui supportent l'implémentation des concepts théoriques de cette logique. Le plus connu est le système Maude ce qui permettra d'obtenir rapidement des prototypes exécutables des spécifications produites.

2. Un cadre d'analyse formelle dans lequel toute réécriture correspond à une preuve de déduction permettant ainsi d'établir des procédures d'analyse correctes pour les modèles architecturaux AADL.

3. Une approche de formalisation générique et facilement extensible pour prendre en considération tous les aspects comportementaux d'un modèle architectural AADL d'un système embarqué temps réel.

4.5 Conclusion

Le langage AADL se focalise sur la description architecturale des dimensions des composants et leurs connexions, mais ne traite pas directement de leur implantation comportementale, ni de la sémantique des données manipulées. Nous nous sommes intéressés dans la première partie de ce chapitre à la présentation détaillée et la classification des quelques tentatives de formalisation de la description architecturale AADL publiées dans la littérature. Ces contributions qui sont censées établir une sémantique formelle pour les modèles AADL afin de les exécuter et vérifier leurs propriétés, se sont avérées très limitées quant aux concepts architecturaux AADL considérés ainsi que la description du comportement complexe des threads, les unités d'exécution concurrentes pour AADL. La deuxième partie de ce chapitre a été alors consacrée à la présentation de la solution adoptée pour la définition formelle de la sémantique d'exécution d'un thread. Nous avons présenté l'annexe ABAReL tout en motivant son existence pour la spécification et l'étude du comportement dynamique d'un thread. En effet, le formalisme de base de cette annexe qui est la logique de réécriture a permis de capturer correctement la sémantique d'exécution d'un thread AADL y compris le comportement complexe lié d'une part, aux modes d'interactions entre les threads dans une plateforme d'exécution et d'autre part, à la nature de leurs états (états hiérarchiques), ainsi que leurs comportements concurrents.

Chapitre 5

ABAReL et les Propriétés AADL

Sommaire

5.1 Introduction

AADL a été développé pour décrire l'architecture d'un système embarqué en termes de système d'application, formé d'une collection de composants en interaction, lié à une plate forme d'exécution. Dans cette description, chaque entité AADL possède des contraintes spécifiques pour caractériser son comportement au moment de l'exécution. Pour parvenir à spécifier les caractéristiques et les contraintes d'exécution et de déploiement associées à différentes entités, AADL introduit la notion de propriété. Les propriétés fournissent des informations descriptives au sujet des composants, des sous-composants, des dispositifs, des flux, des modes, et même des appels de sous-programmes. Ces informations jouent aussi un rôle très important dans la description du comportement des composants AADL.

Les propriétés de déploiement par exemple, permettent aux composants logiciels d'application d'être liés aux composants de la plateforme d'exécution. Donc, les valeurs assignées aux propriétés de déploiement sont utilisées pour déterminer les composants de la plateforme d'exécution (à savoir les processeurs, les bus et les mémoires) auxquels sont liés les threads, les connexions et où sont stockés le code source des threads et les données manipulées par les threads.

Les valeurs assignées aux propriétés d'exécution permettent de déterminer par exemple la période et le temps d'exécution que peut prendre un thread pour s'exécuter, la politique de déclenchement utilisé pour le thread, etc.

La valeur assignée à la latence d'un flux permet d'évaluer le temps de transmission d'un flux de contrôle (événement) et/ou de données à travers une connexion. Toutes ces valeurs et d'autres peuvent être utilisées pour simuler le comportement d'un système embarqué et évaluer ses performances durant les premières phases du processus de développement, à savoir l'étape de conception architecturale. Cependant, la description des propriétés dans AADL reste textuelle et nécessite une grande attention dans la formalisation et l'analyse du comportement d'une architecture AADL.

Dans ce chapitre, nous explicitons cette notion de propriété. Nous présentons d'abord la démarche à suivre pour déclarer une propriété dans une description architecturale AADL d'un système et lui assigner des valeurs. Nous montrons ensuite comment ces propriétés peuvent être prises en charge par l'annexe comportementale ABAReL pour une éventuelle formalisation et nous évoquons le besoin de leur analyse.

5.2 Expression des propriétés dans une description AADL

La version courante du standard AADL [SAE09] fournit un grand nombre de propriétés qui offre la possibilité de décrire des contraintes s'appliquant à l'architecture et au comportement des composants. Dans cette section, nous présentons le format d'expression des différentes propriétés AADL, à travers des exemples.

5.2.1 Où déclarer une propriété ?

Dans une description architecturale, les propriétés sont déclarées soit dans la section « `properties` » de la description de type ou d'implantation d'un composant, soit sont attachées directement à une entité telle une connexion, un flux ou un mode. Comme elles peuvent être attachées à un sous composant ou à un dispositif.

a) Dans la section « `properties` »

La déclaration des propriétés, où sont assignées les valeurs qui s'appliquent au composant et à ses sous-composants, apparait au niveau de la section « `properties` » dans la description de type ou d'implantation de chaque composant. Le standard AADL offre pour chaque catégorie de composant des propriétés prédéfinies. Pour les composants de la plateforme d'exécution ces propriétés permettant de décrire leurs caractéristiques d'exploitation. Pour les **`processors`** qui sont des abstractions du hardware et du système d'exploitation, les propriétés spécifient par exemple la vitesse de traitement (`processing speed`) et le protocole d'ordonnancement (`scheduling Protocol`). Les `buses` peuvent

représenter les interconnexions physiques ou les couches protocole. Leurs propriétés identifient la sortie et la latence des transferts, les formats de données, le temps d'exécution ou de propagation à travers les composants, etc. Par exemple, la propriété `propagation_delay` correspond au temps de propagation d'un signal à travers un `bus`. Pour les composants `memory`, les propriétés `source_code_size` et `source-heap_size` par exemple permettent d'exprimer les contraintes en taille mémoire.

Exemple 1. Pour le composant processeur `cpu_bus` de la figure 5.1, la propriété `Scheduling_Protocol` spécifie le protocole d'ordonnancement en lui assignant la valeur `RATE_MONOTONIC_PROTOCOL.`

```
processor cpu_abs
 features
 controleur_cpu: requires bus access bus_abs;
end cpu_abs;

processor implementation cpu.imp
  properties
  Scheduling_Protocol => RATE_MONOTONIC_PROTOCOL;
end cpu.imp;
```

Figure 5.1 : Exemple de déclaration de propriétés pour un processeur

b) Propriétés attachées à une entité

La déclaration des propriétés, dans ce cas, peut être attachée directement à la description d'un mode, d'un flux, d'une connexion, etc. Les propriétés attachées aux modes dans la description d'un thread indiquent des informations concernant les conditions de son activation ou désactivation. Les propriétés attachées aux flux spécifient les valeurs qu'on peut associer aux latences de flux.

Exemple 2. Dans la description du dispositif `capteur_vitesse` de la figure 5.2, la propriété attachée à la description du flux `flot1` associe la valeur `10Ms` à la latence de ce flux.

```
device capteur_vitesse
  features
    vitesse_V: out data port;
  flows
    flot1: flow source vitesse_V {
      Latency => 10Ms;
      };
end capteur_vitesse;
```

Figure 5.2 : Exemple d'une propriété attachée à un flux

5.2.2 Comment déclarer une propriété ?

AADL offre une grande souplesse dans la déclaration des propriétés. Ces propriétés peuvent appartenir à la grande gamme de propriétés prédéfinies par le standard, comme il est possible de définir de nouvelles propriétés par l'utilisateur. Dans tous les cas, la déclaration d'une propriété permet d'assigner une valeur au nom de la propriété en respectant la syntaxe suivante :

$$\texttt{PropertyName} \Rightarrow \texttt{PropertyValue}$$

où `PropertyName` peut donc être un nom prédéfini, ou le nom d'une nouvelle propriété définie par l'utilisateur. Les noms prédéfinis sont par exemple, `Dispatch_Protocol`, `Period`, `Compute_Execution_Time`, etc. pour les propriétés temporelles d'un thread, et `Processing Speed`, `Scheduling Protocol`, etc. pour les propriétés d'un processeur.

`PropertyValue` peut également appartenir à l'ensemble de valeurs dont les types sont prédéfinis dans le standard ou à d'autres types définis par l'utilisateur. Les déclarations de ces types de valeurs sont faites dans la section « `property set` ».

Les types prédéfinis sont généralement déclarés à partir de types de base (chaine de caractères, booléen, entier, réel, énumération, plage de valeurs, etc.). Les nouveaux types doivent être définis par l'utilisateur selon le besoin de son modèle architecturale.

5.3 Exploitation des propriétés

Pratiquement, une valeur assignée à une propriété peut être, soit une valeur numérique (cas des propriétés d'exécution temporelle ou code source), soit le nom d'une politique d'expédition (comme c'est le cas des valeurs `periodic`, `aperiodic`, etc. attribuées aux propriétés d'exécution) ou encore le nom du protocole d'ordonnancement choisi (tel que la valeur `Rate_Monotonic_Protocol` assignée à la propriété `Scheduling_Policy`). Cette valeur assignée au nom d'une propriété dans une déclaration donnée, peut être exploitée directement au niveau du composant dans lequel elle a été déclarée ou bien indirectement à un autre niveau par ses sous-composants. Donc, une valeur peut être attribuée à une propriété et exploitée directement dans la description d'un composant, implicitement par une valeur héritée, ou explicitement par propagation.

5.3.1 Assignation directe

L'assignation directe d'une valeur pour une propriété est introduite directement dans la clause « `properties` » de la description de type ou d'implantation d'un composant ou dans une description de dispositifs, de connexions, de flux, de modes, etc.

Exemple 3. Dans la figure 5.3, nous présentons deux exemples de déclaration de propriété dans la description AADL du composant `capteur_vitesse` de catégorie `device`. Le premier exemple concerne les deux propriétés standard (`Period` et `Dispatch_Protocol`). La propriété `Period` associe une valeur de `30 ms` et la valeur assignée à la propriété `Dispatch_Protocol` est `Periodic`. Le deuxième exemple inclus la déclaration de la latence du flux `flot1` et lui associe la valeur de 10 ms. Celle-ci est attachée directement à la déclaration du flux dans la sous-section `flows`.

```
device capteur_vitesse
  features
    vitesse_V: out data port;
  flows
    flot1: flow source vitesse_V {
      Latency => 10ms;
      };
  properties
    Device_Dispatch_Protocol => Periodic;
    Period => 30ms;
end capteur_vitesse;
```

Figure 5.3 : Exemple d'association de propriété directe

5.3.2 Assignation indirecte

- **Par héritage**

Il s'agit de la forme implicite d'une assignation de valeurs pour une propriété qui peut être déclarée pour s'appliquer aux composants contenus (les sous composants du composant contenant ou parent). Dans ce cas, la valeur de la propriété déclarée dans la clause « `properties` » d'un composant est assignée à n'importe quel sous-composant auquel la propriété peut s'appliquer.

Exemple 4. Dans la figure 5.4, la valeur 20 ms a été assignée à la propriété `Period` dans la section « `properties` » de la déclaration de type du composant `systeme_ABS`. Par héritage, cette valeur est attribuée implicitement aux périodes de chaque thread contenu dans la hiérarchie du composant `systeme_ABS.` Par conséquent, la période du thread `calcul_glissement` prend par héritage la valeur 20 ms. Si l'intention avait été d'avoir une période de 10 ms, il devrait y avoir une déclaration explicite pour la période du sous-composant `calcul_glissement` dans sa section `properties`.

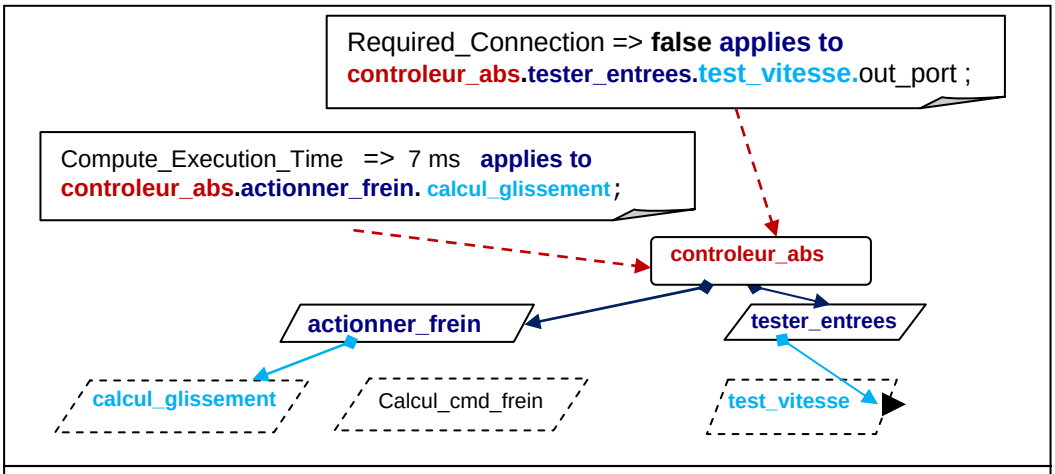
```
system systeme_ABS
  properties
   Period => 20 ms;
end systeme_ABS;
system implementation systeme_ABS.Impl
  subcomponents
   capteur_pedal: device capteur_pedal;
   controleur_abs: system controleur_abs.impl;
   actuateur_frein: device actuateur_frein;
end systeme_ABS.Impl;

system implementation controleur_abs.impl
  subcomponents
    tester_entrees: process tester_entrees.Impl;
    actionner_frein: process actionner_frein.Impl;
end controleur_abs.impl;

process implementation actionner_frein.Impl
  subcomponents
    calcul_glissement: thread calcule_glissement.impl;
    calcule_cmd_frein: thread calcule_cmd_frein.impl;
end actioner_frein.Impl;

thread implementation calcule_glissement.impl
  properties
   Compute_Execution_Time => 7ms;
   Dispatch_Protocol => Periodic;
end calcule_glissement.impl;
```

Figure 5.4: Assignation de valeurs aux propriétés par héritage

- **Par propagation**

La propagation d'une propriété concerne les propriétés déclarées dans un composant qualifié de parent et font référence à une autre partie dans la hiérarchie du modèle. Pour appliquer cette propriété, l'assignation de propriété doit spécifier un chemin relatif à l'endroit référencié du modèle par le mot clé `applies to`. Dans ce cas la propriété s'applique à la déclaration désignée par la référence du `applies to`. Cette déclaration dépend du niveau de la hiérarchie où se situe le composant référencié.

Exemple 5. Dans la figure 5.5, nous présentons deux propriétés de ce type déclarées dans la section `properties` de la description d'une implantation du composant systeme_ABS. La première propriété est `Compute_Execution_Time` assignée à la valeur 7 ms.

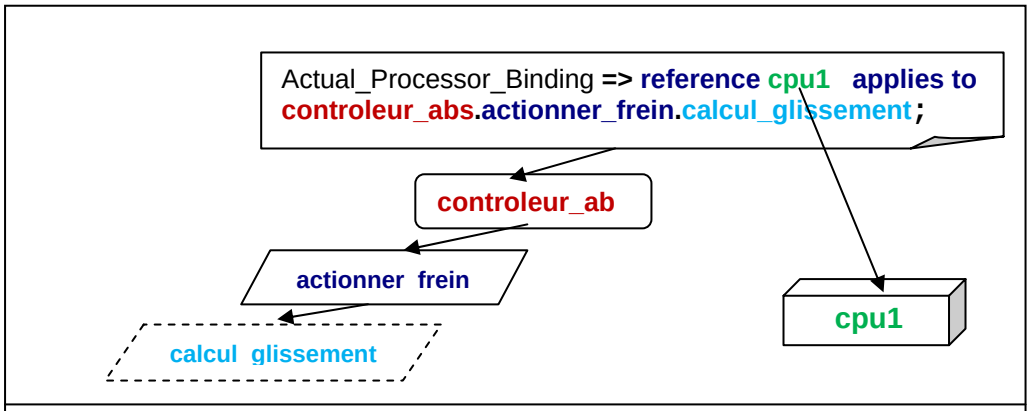



```
system systeme_ABS
end systeme_ABS;

system implementation systeme_ABS.Impl
  subcomponents
   capteur_pedal: device capteur_pedal;
   controleur_abs: system controleur_abs.impl;
   actuateur_frein: device actuateur_frein;
  properties
   Compute_Execution_Time => 7ms applies to
    controleur_abs.actionner_frein.calcul_glissement;

   Required_Connection => false applies to
    controleur_abs.tester_entrees.test_vitesse.out_port ;
end systeme_ABS.Impl;

system implementation controleur_abs.impl
  subcomponents
    tester_entrees: process tester_entrees.Impl
    actionner_frein: process actionner_frein.Impl;
end controleur_abs.impl;

process actionner_frein
  features
    etat_pedal: in event port;
    vitesse_r: in data port;
    cmd_frein: out event data port;
end actioner_frein;

process implementation actionner_frein.Impl
  subcomponents
    calcul_glissement: thread calcul_glissement.impl;
    calcul_cmd_frein: thread calcul_cmd_frein.impl;
end actioner_frein.Impl;

thread implementation calcul_glissement.impl
end calcul_glissement.impl;
```

Figure 5.5 : Exemple d'une assignation de propriétés par propagation

Cette propriété s'applique au thread `calcul_glissement` référencié par le chemin **`controleur_abs.actionner_frein.calcul_glissement`** qui indique que le thread `calcul_glissement` est un sous-composant du `process actionner_frein` lui-même étant un sous-composant du composant `system controleur_abs.` La deuxième propriété assigne la valeur *fausse* à la propriété `Required_Connection` permettant au port de sortie du thread `test_vitesse` d'être non lié.

L'assignation par propagation concerne également les propriétés de déploiement. Le déploiement est spécifié par les propriétés AADL: `Actual_Connection_Binding`, `Actual_Memory_Binding` et `Actual_Processor_Binding` pour indiquer respectivement par quel bus, mémoire ou processeur est porté une connexion, une donnée ou un processus. La valeur assignée à de telles propriétés est spécifié par une référence, via le mot clé `reference,` à un composant de la plateforme d'exécution (à savoir : `bus`, `memory`, `processor`) suivi par `applies to` pour désigner le chemin relatif à l'endroit de la hiérarchie auquel l'association de propriété est indiquée.




```
system implementation systeme_ABS.Impl
  subcomponents
   controleur_abs: system controleur_abs.impl;
   cpu1: processor cpu1.impl;
    actuateur_frein: device actuateur_frein;
  properties
    Actual_Processor_Binding => reference cpu1 applies to
    controleur_abs.actionner_frein.calcul_glissement;
end systeme_ABS.Impl;

system implementation controleur_abs.impl
  subcomponents
    tester_entrees: process tester_entrees.Impl
    actionner_frein: process actionner_frein.Impl;
end controleur_abs.impl;

process implementation actionner_frein.Impl
  subcomponents
    calcul_glissement: thread calcul_glissement.impl;
    calcul_cmd_frein: thread calcul_cmd_frein.impl;
end actioner_frein.Impl;

thread implementation calcul_glissement.impl
end calcul_glissement.impl;
```

Figure 5.6 : Exemple de propriété de déploiement

5.3.3 Analyse et vérification des propriétés AADL

A chaque entité architecturale AADL, nous pouvons ainsi associer des propriétés présentant des valeurs les caractérisant. Les propriétés constituent donc un aspect fondamental et important dans la description du comportement des entités AADL. Cependant, leur description reste textuelle et aucune sémantique ne leur a été proposée. Certains travaux dans la littérature [SLN+04] [Axl07] ont accordé une

certaine importance à la prise en charge des valeurs de quelques propriétés d'exécution pour simuler le comportement des threads, ou celui d'architectures de systèmes décrit par AADL. Jusqu'à présent, aucune recherche n'a abordé la formalisation de la sémantique d'une propriété rattachée à un composant donné quelque soit son type. Nous contribuons à travers la section suivante à la solution de ce problème, en montrant la capacité du modèle proposé (ABAReL) pour d'une part, formaliser la sémantique d'une propriété AADL déclarée et d'autre part, l'analyser et la vérifier formellement en exécutant le comportement du composant auquel elle se rattache.

5.4 Prise en charge des propriétés AADL par ABAReL

ABAReL modélise les aspects syntaxiques du langage AADL ainsi que la sémantique de comportement de ses composants en particulier celle d'un thread. Ce modèle formel est extensible et permet de prendre en charge la formalisation de toutes ces propriétés AADL grâce à l'expressivité et la flexibilité de la logique de réécriture. Donc, formaliser la sémantique d'une propriété déclarée dans une description AADL implique d'une part, un enrichissement de la signature d'ABAReL (aspects syntaxiques) et d'autre part, associer une interprétation sémantique à chaque concept syntaxique introduit (aspects sémantiques). L'implémentation du modèle mathématique étendu obtenu sous RT-Maude, permettra l'analyse et la vérification de ces propriétés en tirant profit des outils qui existent autour du système Maude. Nous nous restreignons dans le cas de notre étude à la formalisation de trois types de propriétés, les plus utilisées dans le cas de l'exécution d'un thread. Notre démarche est assez générique et pourra s'appliquer à n'importe quelle déclaration de propriété.

5.4.1 Sémantique d'une propriété de type `Period`

Dans la déclaration AADL ci-dessous de cette propriété, nous recensons les éléments les plus importants qui peuvent être pris en charge de façon assez naturelle par notre modèle pour procéder à sa formalisation.

```
thread implementation  Nom du composant Thread
  properties
   Period => valeur;
```

Dans une première étape, nous proposons une extension [BB+09, BBB10] de la signature du modèle mathématique ABAReL pour la prise en charge des valeurs assignées à cette propriétés, dans ce cas il s'agit des valeurs de **type temps**. Ces valeurs sont alors spécifiées par la théorie équationnelle TIME [ÖM96]. Dans cette théorie qui sera importée par la théorie de réécriture orientée objet temps réel $\mathfrak{R}_{\mathbf{Th}}$ formalisant le comportement du thread spécifié, nous retrouvons des opérations et des équations spécifiques pour gérer l'aspect temps (telles que `delta` et `mte` pour l'interprétation de l'écoulement du temps sur la configuration d'un thread).

Dans une seconde étape, nous enrichissons la spécification des configurations de ce thread par un paramètre dit `Clock-P` modélisant sa période capturée à partir de la déclaration de la propriété associée (`Period)`. Toutes les règles de réécriture modélisant le comportement concurrent de ce thread doivent prendre en considération l'évolution de ce paramètre.

5.4.2 Sémantique d'une propriété de type `Compute_Execution_Time`

De façon assez similaire, le modèle étendu d'ABAReL traite la formalisation de la sémantique d'une propriété AADL du type `Compute_Execution_Time` associée à un composant de type thread. Le paramètre à rajouter dans la configuration d'un thread donné est nommé `Clock-C`, il sert à modéliser son temps d'exécution. Evidemment, l'évolution de ce paramètre doit également être prise en considération par toutes les règles de réécriture modélisant le comportement concurrent de ce thread.

5.4.3 Sémantique d'une propriété de type `Latency`

Cette propriété AADL est déclarée pour une connexion établie entre deux threads, contraignant ainsi le passage de flux de données et/ou d'événements (transmission de messages) entre ces threads. Dans une théorie de réécriture formalisant une description architecturale constituée de plusieurs threads reliés par des connexions, nous introduisons alors la sorte `DlyMsg` qui modélise le temps de latence du flux (récupéré de la déclaration de la propriété `Latency`). Nous définissons aussi

l'opérateur `dly(m,t)` qui indique qu'il reste **`t`** unité de temps pour l'arrivée du message **`m`** à destination [BBB10, BB10].

Notons que pour cette étude, les valeurs des trois types de propriétés considérées sont déclarées dans la description des composants où elles sont exploitées. Il s'agit d'une assignation directe, les valeurs des propriétés sont donc capturées directement par la spécification. Quand la valeur d'une propriété nécéssite une référence à une autre partie dans la hiérarchie du modèle, on parle de la propagation des propriétés. Pour capturer les valeurs de ces propriétés et les affecter ensuite à la spécification du composant où elles doivent être exploitées, il faut expliciter leur chemin de propagation. Pour ce faire, nous proposons une extension du modèle mathématique d'ABAReL par une théorie de réécriture spécifiant le chemin de propagation pour pouvoir assigner la valeur déclarée à l'entité AADL référenciée. Le chemin de propagation doit être spécifié en fonction des sous composants déclarés dans la section `subcomponents` de chaque composant dans la hiérarchie.

Exemple 6. Pour clarifier la formalisation des propriétés à valeurs de type temps, nous considérons la description AADL du thread `test-etat-moteur` (figure 5.7) avec la déclaration des propriétés `Period` et `Compute-Execution-Time` assignées respectivement aux valeurs `28 ms` et `12 ms`. Ce thread est connecté au composant `capteur-moteur` de catégorie `Device` contenant la déclaration de la propriété `Latency` assignée à la valeur `10 ms`.

La spécification des propriétés d'exécution temporelles capture des valeurs de chaque propriété pour les associer aux horloges correspondantes dans la spécification du thread. La propriété `Latency` est spécifiée par une transmission d'un message où le temps de transmission correspond à la valeur de cette propriété.

Description AADL d'une configuration d'un thread	Formalisation des valeurs de type temps

device capteur-moteur
features
etat_moteur: out event port;
flows
flot1: flow source etat_moteur {
Latency => 10 ms;
};
end capteur-moteur;
 	
thread test_etat_moteur
features
etat_abs1: out data port;
etat_moteur: in event port;
end test_etat_moteur;
 	
thread implementation test-etat-moteur.Impl1
properties
Period => 28 ms;
Compute-Execution-Time => 12 ms ;
end test-etat-moteur.Impl1;
	******Propriété: Latency******
	
	dly (**from** capteur-moteur **to** test-etat-moteur **transfer** event1, 10)
	
	Composant source du flux
	
	<capteur-moteur| Category : Device , OPort : etat-moteur >
	
	Composant destination du flux
	
	<test-etat-moteur| Category : thread , IPort : etat-moteur, OPort: etat-abs1,
	...
	*****Propriétés temporelles*****
	
	Period : 28, Clock-P: 0,
	Execution-Time : 12 , Clock-C : 0
	... >

Tableau 5.1: Exemple de formalisation des propriétés à valeur de type temps

Notre approche d'extension de l'annexe comportementale ABAReL pour la prise en charge des propriétés, offre une solution générique à base de théories de réécriture orientées objet temps réel pour formaliser la sémantique de n'importe quel type de propriété. Notre étude a porté sur les propriétés qui présentent des valeurs de type réel pour contraindre les paramètres d'exécution des threads, mais nous pouvons élargir l'éventail de ces propriétés à celles qui présentent des valeurs d'un autre type, il suffit de prévoir leur cadre sémantique (théorie équationnelle) dans la théorie de réécriture modélisant le(s) composant(s) comportant ces déclarations de propriétés. En effet, les concepts de la logique de réécriture ont permis de décrire naturellement cette sémantique d'exécution en offrant un cadre sémantique formel pour l'analyse du comportement et la vérification de ces propriétés.

5.5 Analyse de propriétés par ABAReL

La prise en charge des propriétés AADL par ABAReL a permis leur formalisation en vue d'entreprendre leur analyse et par conséquent l'analyse du comportement des

composants auxquelles sont liées. Dans cette architecture, nous nous sommes intéressés, dans le cadre de ce travail, à l'analyse du comportement des composants threads AADL en considérant d'une part, les spécifications des propriétés d'exécution temporelle (`Period` et `Compute_Execution_Time`), et autre part, les latences de flux de transmission de données et/ou d'évènements entre les threads en exécution. Ainsi, l'exécution concurrente de plusieurs threads et l'effet des interactions entre les threads sont définis dans cette extension d'ABAReL.

Ces propriétés décrivant les conditions d'exécution d'un thread dans le temps sont formalisées, dans ABAReL, par des horloges permettant de superviser son comportement tout en considérant l'écoulement de sa période et de son temps d'exécution. La simulation d'un tel comportement permet de suivre les étapes d'exécution du thread tout en considérant ses états internes ou sous-états où le thread peut être prêt pour s'exécuter (sous-état `Ready`), en exécution (sous-état `Running`) ou bloqué sur l'accès d'une ressource (sous-état `Awaiting-Resource`). L'analyse de la propriété `Compute_Execution_Time` doit considérer le suivi de l'écoulement du temps, supervisé par l'horloge `Clock-C`, sur le thread en exécution (sous-état `Running`). L'analyse de la propriété `Period`, supervisée par l'horloge `Clock-P`, considère le thread dans tous les sous-états de l'état `Compute`.

L'implémentation du modèle mathématique d'ABAReL sous RT-Maude ouvre la voie pour l'analyse des latences de flux dans une architecture AADL d'un système embarqué temps réel en suivant les étapes de transmission d'un message. En particulier, nous avons exploité respectivement les commandes *trew (Timed rewrite))* pour simuler un comportement possible du système jusqu'à une durée de temps bien déterminée, la commande *reduce* pour réduire un terme par application des équations et des axiomes d'adhésion (membership) dans le module temporisé.

5.6 Conclusion

Dans ce chapitre, nous avons d'abord montré l'importance et la diversité des propriétés dans la description des entités architecturale AADL. Nous avons ensuite présenté le rôle que peut jouer chaque classe de propriété dans la détermination du

comportement des composants. Nous nous sommes concentrés sur le comportement des composants threads et nous avons considéré les propriétés à valeurs de type temps, à savoir les propriétés d'exécution temporelles et les latences de flux. Cependant, la déclaration de ces propriétés diffère d'une description architecturale à une autre. Ces propriétés sont soit déclarées et exploitées directement au niveau du même composant, soit déclarées dans la description d'un composant pour être exploitées indirectement à un autre niveau de la hiérarchie de ce composant. L'assignation de valeurs, dans ce cas, se fait implicitement par héritage ou explicitement par propagation.

A ce propos, nous avons montré comment étendre le modèle mathématique de l'annexe comportementale ABAReL, pour la spécification formelle de propriétés AADL afin de pouvoir les analyser lors de l'exécution des entités architecturales les contenant. Nous avons illustré notre approche de formalisation sur trois cas de propriétés à valeurs de type *Time,* déclarées souvent dans des composants de catégorie thread. Ainsi, nous avons associé à chaque thread deux horloges pour spécifier les propriétés `Period` et `Compute-Execution-Time`. Les latences de flux sont modélisées par le passage de messages entre deux threads si une connexion est préalablement déclarée avec un délai de transmission d'un message capturé à partir de la valeur assignée à la latence du flux (Propriété `Latency`). La prise en charge des valeurs assignées à ces propriétés est faite lors de l'exécution des threads pour pouvoir les analyser. La démonstration de quelques propriétés, en utilisant l'outil RT-Maude et son model checker, illustrée à travers une étude de cas, sera présentée dans le chapitre suivant.

Chapitre 6

Implémentation d'ABAReL sous RT-Maude

Sommaire

6.1 Introduction

Dans le chapitre 4, nous avons proposé, via l'annexe comportementale ABAReL, une approche basée sur la logique de réécriture révisée pour la formalisation d'une description architecturale AADL. Dans ce chapitre, nous prouvons que les théories de réécriture orientées objet temps réel de la logique de réécriture offrent à ABAReL non seulement une description précise de la sémantique des threads interconnectés en respectant leurs contraintes temporelles, mais rendent également la vérification formelle possible des propriétés comportementales. Pour ce faire, nous implémentons, dans un premier temps, le modèle mathématique d'ABAReL sous RT-Maude. Nous utilisons ensuite le modèle exécutable déduit pour analyser le comportement d'une architecture AADL, composée de plusieurs threads interconnectés, et nous vérifions ses propriétés comportementales tout en bénéficiant des outils d'analyse et de vérification du système RT-Maude. Le prototype obtenu de cette implémentation est considéré comme un outil pour spécifier formellement, exécuter et analyser une architecture AADL.

Ainsi, l'outil proposé pour AADL se caractérise par sa base sémantique appropriée. Comme nous l'avons présenté dans le chapitre 4, le langage AADL dispose de plusieurs outils logiciels permettant la vérification syntaxique et l'analyse des modèles AADL comme Osate [SEI04], Topcased [Gau05], Furness et les outils Cheddar [SLN+04] et ADeS [AxG02] pour l'analyse et la simulation du comportement des threads et celui d'une architecture AADL. Néanmoins, tous ces outils sont bâtis sur OSATE comme plug-in Eclipse et sont souvent adaptés à des besoins spécifiques. Aucun de ces outils n'a été conçu spécifiquement pour la spécification et la vérification formelles d'un système modélisé par AADL sans passer par plusieurs transformations.

Le reste de ce chapitre est organisé comme suit : la section 6.2, montre comment exploiter la version RT-Maude pour obtenir un modèle exécutable pour AADL à base d'ABAReL, la section 6.3 explique, à travers un cas d'étude, comment

nous pouvons spécifier une architecture AADL, analyser son comportement et vérifier ses propriétés, et la section 6.4 conclue le chapitre.

6.2 Un outil pour spécifier, exécuter et analyser une description AADL

Dans cette section, nous montrons comment nous implémentons les théories de réécriture orientées objet temps réel de la logique de réécriture, constituant le modèle mathématique de l'annexe comportementale ABAReL, sous RT-Maude [Ölv07]. Le prototypage de ce modèle mathématique fournit un outil pour spécifier formellement un système embarqué modélisé par AADL, analyser son comportement et vérifier ses propriétés comportementales.

6.2.1 Implémentation d'ABAReL sous RT-Maude

Nous exploitons la syntaxe du langage RT-Maude pour implémenter le modèle mathématique d'ABAReL. Le résultat de cette implémentation est un module orienté objet temporisé $\mathscr{R}_{Th}$ mis entre les mots clés `tomod` et `endtom`. En respectant cette syntaxe, la spécification du modèle mathématique d'ABAReL sous RT-Maude est déclarée comme suit :

```
(tomod   AADL-SPEC   is
protecting NAT-TIME-DOMAIN-WITH-INF .
protecting   LIST .
subsort Qid < Oid .

    …
endtom)
```

Il s'agit d'un module orienté objet temporisé nommé `AADL-SPEC` qui implémente la spécification des conditions d'exécution d'un thread dans une configuration architecturale [BB10]. $\mathscr{R}_{Th} = (\Sigma_{Th}, E_{Th}, L_{Th}, R_{Th})$ représente la théorie de réécriture orientée objet temps réel associée au composant thread ***Th*** (voir chapitre 4, section 4.3.4) où (Σ_{Th}, E_{Th}) est une théorie équationnelle d'adhésion (membership) décrivant tous les éléments déclarés dans la description du thread

AADL (flux, propriétés, etc.). Les règles de réécritures étiquetées (L_{Th}, R_{Th}), les règles *instantanées* et la règle *tick*, décrivent le comportement du thread ***Th*** selon l'évolution de ses configurations, en tenant compte de ses états locaux et ceux de ses ports de connexions, aussi bien que ses conditions temporelles déclarées.

Notons que chaque module temporisé importe implicitement une abstraction du domaine temps avec la déclaration de la sorte `Time`, et du constructeur libre {_} de sorte `GlobalSystem`. RT-Maude supporte les domaines de temps discret et dense. Dans notre cas, nous appliquons le temps discret. Donc notre spécification importe le module LIST pour introduire la structure liste et le module `NAT-TIME-DOMAIN-WITH-INF` de RT-Maude qui défini le domaine de temps comme des nombres naturels en ajoutant la constante `INF` (dénotant ∞) de `supersort TimeInf.`

Cette implémentation d'ABAReL sous RT-Maude offre :

- d'une part, un analyseur syntaxique pour les descriptions AADL à formaliser. En effet, la syntaxe du langage AADL est prise en charge de façon inhérente par la signature du modèle grâce à la flexibilité de la logique de réécriture.
- d'autre part, un modèle formel exécutable approprié pour une architecture AADL composée de plusieurs threads en interaction.

Dans ce modèle toute réécriture correspond formellement à une preuve de déduction permettant d'établir des procédures d'analyse correctes. Dans le système RT-Maude ceci est réalisé par entre autre les commandes `Timed rewrite` (`trew`), `reduce` (`red`) et `tsearch.` La commande de réécriture temporisée `(trew)` permet de simuler un comportement possible dans une architecture AADL spécifié par ABAReL jusqu'à une durée de temps bien déterminée. La commande `reduce` permet de réduire un terme par application des équations et des règles de réécriture dans le module temporisé `AADL-SPEC`. La commande de recherche temporisée `tsearch` permet de vérifier, sous certaines conditions, si un état quelconque dans un système architectural spécifié par ABAReL est accessible à partir d'un état initial et dans une limite de temps donnée.

Cependant, avant d'entreprendre n'importe quelle analyse, nous devons choisir une stratégie d'avancement du temps pour guider l'application de la règle « *tick* ». Nous optons pour la stratégie qui avance le temps par une unité de temps dans chaque application d'une règle de réécriture « *tick* » et nous déclarons cette stratégie par la commande suivante de RT-Maude.

```
Maude > (set tick def 1 .)
```

Donc, l'exécution et l'analyse du module temporisé `AADL-SPEC` se fait par rapport à cette stratégie pour l'application de la règle « *tick* » de ce module.

6.2.2 Analyse model checking d'une architecture AADL

Nous utilisons les spécifications RT-Maude du modèle exécutable d'ABAReL pour simuler et analyser formellement l'exécution concurrente de plusieurs threads dans un système architectural AADL, et également l'effet des interactions entre les threads avec le respect des contraintes d'exécution déclarées. Pour ce faire, nous définissons les propriétés comportementales inhérentes à l'exécution d'un thread, nous montrons ensuite l'utilisation du LTL model checker de RT-Maude pour vérifier ces propriétés comportementales.

Rappelons que le comportement concurrent d'un thread AADL a été défini dans le chapitre 4, en tenant compte de la description formelle de son comportement dans son état « `Compute` » et en utilisant les valeurs capturées à partir des propriétés d'exécution temporelles (`Period` et `Compute-Execution-Time`). Ces propriétés sont supervisées par les horloges (`Clock-P` et `Clock-C`). Chaque thread dans l'état "`Compute`" doit avoir les sous états suivants: Prêt pour s'exécuter (`Substate = Ready`), en exécution (`Substate = Running`) et en attente d'une ressource (`Substate = Awaiting-Resource`).

Nous nous intéresserons dans cette étude à trois types de propriétés que nous jugeons les plus représentatives dans ce type d'analyse formelle et les plus considérées dans les analyses existantes autour du langage AADL. Il s'agit des

propriétés d'*accessibilité* **(P1)**, de *sûreté* **(P2)** et de *vivacité* **(P3)**. Dans ce cas de figure, ces propriétés selon le comportement des threads s'expriment comme suit :

– La propriété **(P1)** exprime l'*accessibilité* de l'état final du processus d'exécution d'un thread. Cet état est accessible après l'écoulement du temps d'exécution du thread (`Clock-C =0`), de sa période (`Clock-P =0`), si l'état du thread est (`ThreadState = Compute`) et son sous état est (`Substate = Complete`).

– La propriété **(P2)** exprime la *sûreté,* assurant que le thread en exécution (`ThreadState = Compute`) n'est jamais préempté indéfiniment et/ou en attente d'une ressource indéfiniment.

– La propriété **(P3)** exprime la *vivacité* par le fait que le thread finira par s'exécuter. Cet état est atteint dans le temps si le thread est en exécution (`ThreadState = compute`) et son sous état est (`Substate = running`), en respectant le temps d'exécution déclaré par la propriété `Compute-Execution-Time`.

Ces propriétés représentent pour ABAReL des propriétés comportementales. Leur vérification implique que le thread s'exécute correctement dans une configuration architecturale AADL et que ses propriétés d'exécution temporelles (`Period` et `Compute-Execution-Time`) sont vérifiées.

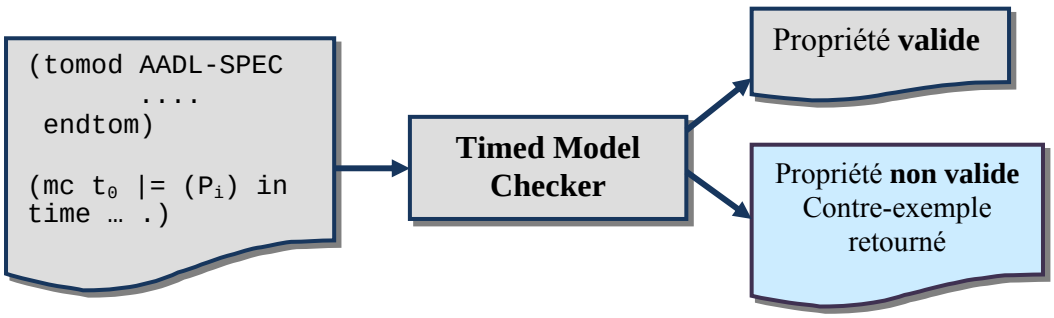


Figure 6.1. Le model checking borné dans le temps

Nous utilisons le LTL model checker de RT-Maude pour vérifier que l'exécution des spécifications du module temporisé `AADL-SPEC`, dans une durée de

temps donnée, satisfait les propriétés {**Pi** / i = 1..3} exprimées dans la logique temporelle linéaire, ou obtenir un contre-exemple montrant que la propriété en question est violée.

6.3 Etude de cas: Un système de contrôle de navigation

Dans cette section, à travers un cas d'étude, nous explicitons l'intérêt du modèle exécutable d'ABAReL pour une architecture AADL, l'analyse de son comportement et la vérification de ses propriétés comportementales

Dans ce cas d'étude, nous considérons un exemple AADL modélisant un système de contrôle de navigation (figure 6.2) que nous avons emprunté à [Ifr05]. Il s'agit d'un système AADL (`system`) nommé `flight Control System`. Ce système devrait afficher les informations de navigation au pilote et indiquer l'état actuel de l'Autopilote. Il est composé de deux sous-systèmes logiciel : le système de navigation autopiloté `S-NAP` et le système d'interface `S-HCI`. Chaque sous-système est lié à un processeur séparé à savoir `NAP-Processor` et `HCI-Processor`. Les deux sous-systèmes sont connectés ensemble par un bus nommé `LAN-Bus` qui permet la transmission des données et des événements entre les composants des deux sous-systèmes. Ces liens sont de types `data`, `event` et `event data` et sont portés respectivement à travers les connexions `Pos-Data`, `AP-Toggle` et `AP-Param`. Le système `S-NAP` contient un ensemble de déclencheurs (`Aileron`, `Elevator`, `Rudder`, `Engine`), que nous avons regroupé dans un seul composant de type `device` nommé `Actuators`, un `GPS` et un processus nommé `P-Nav-Con`. Le processus `P-Nav-Con` contient trois threads. Les connexions entre les composants `Actuators` et `GPS` et le processus `P-Nav-Con` sont reliées au bus `Dev-Bus`.

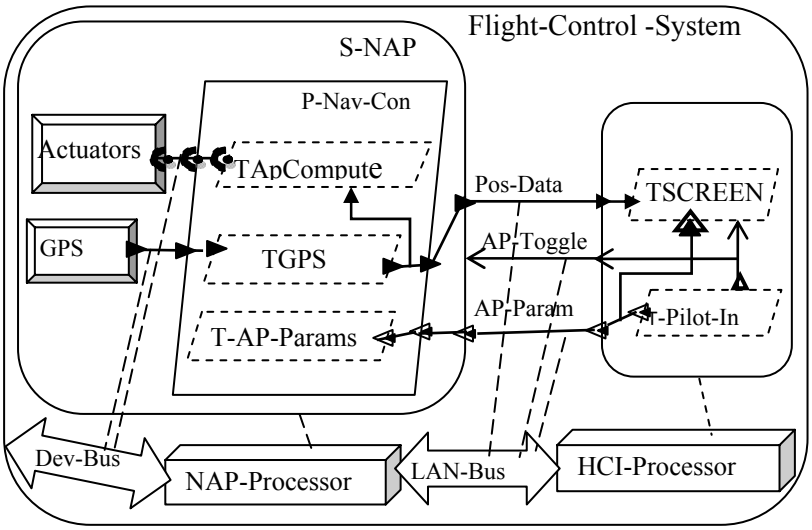


Figure 6.2 Un système de contrôle de navigation en AADL

6.3.1 Configurations architecturales AADL

Dans cette partie, nous présentons la description architecturale AADL de quelques composants qui nous paraissent les plus pertinents pour expliquer, par la suite, les procédures d'analyse et de vérification. Nous présentons la description AADL du composant *process* `P-Nav-Con` et celle des threads `TGPS et TSCREEN`. La description AADL du *process* `P-Nav-Con` fait apparaître la description de trois modes (voir figure 6.3) :

1) le mode initial noté `GPSupAPdown` qui correspond à un `GPS` mis en marche (positionné à `up`) et un arrêt de l'Autopilote (indiqué par `down`),
2) le mode `GPSupAPup` pour décrire la mise en marche du `GPS` et de l'Autopilote. Ce mode est activé par l'événement `AP-Toggle` provenant du thread `T-Pilot-In` exprimant le choix du pilote,
3) le mode `GPSdown` où le `GPS` s'arrête de fonctionner suite à une erreur. Dans ce cas, l'Autopilote s'arrête également puisqu'il ne peut fonctionner sans le `GPS`.

```
process  P-Nav-Con
  features
    GPSerror: in event port;
    APtoggle: in event port;
end P-Nav-Con;
```

```
process implementation P-Nav-Con
  subcomponents
     TGPS:  thread in mode (GPSupAPdown, GPSupAPup);
     TAPCompute: thread in mode (GPSupAPup);
  modes
     GPSupAPdown: initial mode; GPSupAPup: mode; GPSdown: mode;
     GPSupAPdown  -[ APtoggle ] ->   GPSupAPup;
     GPSupAPdown  -[ GPSerror ] ->  GPSdown;
     GPSupAPup  -[ GPSerror ] -> GPSdown;
end P-Nav-Con;
```

Figure 6.3 - Description AADL du processus 'P-Nav-Con'

La description AADL du thread `TGPS` montre qu'il s'agit d'un thread périodique qui fonctionne selon deux modes, le mode `GPSupAPup` et le mode `GPSupAPdown`. Ce thread lit la position courante du GPS et la convertit en une représentation interne pour l'envoyer au thread `TSCREEN` dans le système `HCI-system` si le mode courant est `GPSupAPdown`, et au thread `TApCompute` si le mode courant est `GPSupAPup` (voir figure 6.4). Les valeurs déclarées des propriétés `Compute_Execution_Time` et `Period` changent selon le mode de fonctionnement du thread.

```
thread TGPS
  features
    GPSPositionInput : in data port;
    GPSPositionOutput : out data port;
end TGPS;
```

```
thread implementation TGPS.impl
  modes
    GPSupAPdown :  initial mode;  GPSupAPup :  mode;
  properties
    Dispatch_Protocol => Periodic;
    Compute_Execution_Time => 6 ms in mode (GPSupAPup);
    Period => 10 ms in mode (GPSupAPup);
    Compute_Execution_Time => 3 ms in mode (GPSupAPdown);
    Period => 6 ms in mode (GPSupAPdown);
end TGPS.impl;
```

Figure 6.4 Description AADL du thread `TGPS`

Le thread TSCREEN doit afficher la position courante du GPS, provenant du thread TGPS à travers une connexion sémantique les reliant au pilote. Cette connexion nommée TGPS-TSCREEN véhicule un flux de données partant du port de connexion sortant du thread TGPS (source du flux), pour aller vers le port de connexion entrant du thread TSCREEN (destination du flux). TSCREEN est un thread périodique avec des valeurs déclarées pour les propriétés Period et Compute_Execution_Time.

```
thread TSCREEN
   features
      CurrentPositionInput : in data port;
      AP_Toggle : in event port;
      AP_Position_Input : in event data port;
end TSCREEN ;
```

```
thread implementation TSCREEN.impl
   properties
      Dispatch_protocol => periodic;
      Period => 4 ms;
      Compute_Execution_Time => 2 ms
end TSCREEN.impl;
```

Figure 6.5 Description AADL du thread TSCREEN

6.3.2 Modules RT-Maude correspondants

Notre approche de formalisation est générique, valable pour n'importe quel exemple considéré, nous changeons uniquement les noms des objets et les valeurs des attributs instanciées. En effet, le module de la figure 6.6 offre la possibilité d'instancier l'état initial de n'importe quelle configuration choisie du modèle d'architecture précédent (figure 6.2). Nous choisissons pour cela deux exemples de configurations déclarées chacune dans une équation à part (figure 6.6). La première configuration spécifie le composant GPS, les threads TGPS et TScreen et les connexions GPS-TGPS et TGPS-TScreen. La deuxième configuration considère juste le transfert de données entre le GPS et le thread TGPS. Nous pouvons déclarer plusieurs exemples de configuration de ce type, comme nous pouvons aussi ajouter la déclaration des propriétés en fonction des modes d'exécution.

```
ops initstate StoreConf : -> GlobalSystem .
ops GPSupAPup GPSupAPdown : ->  Modes [ctor] .

eq initstate  = {< GPS : Component | Cat : Device, OPort : send >
  < TGPS : ThreadImpl | Cat : thread, IPort : waitIn,  OPort : send,
TState : wait, mode : GPSupAPdown,   Substate :  noSub,
Period : 6, Execution-time : 3, Clock-P : 0,   Clock-C : 0 >
           dly (from GPS to TGPS transfer data1 , 5)
< GPS-TGPS : Connection | TypeConx : data,  Source : GPS, Dest : TGPS, TimeTrans : 5 >
 < TSCREEN : ThreadImpl | Cat : thread, IPort : waitIn,  OPort : waitOut, TState : wait,
 mode : GPSupAPdown,   Substate : noSub,  Period : 4, Execution-time : 2, Clock-P : 0, Clock-C : 0 >
          dly (from TGPS to TSCREEN transfer data2 , 7)
< TGPS-TScreen : Connection | TypeConx : data, Source : TGPS , Dest : TSCREEN , TimeTrans :  7 > } .

eq StoreConf = (from GPS to TGPS  transfer data1 )
   < TGPS : ThreadImpl |   Cat : thread,  mode : GPSupAPdown,    IPort : waitIn,
     TState :    wait ,   Substate : noSub ,   OPort : waitOut,   InBufferPort : EmptyFile,
      AccessData : Nill,   OutBufferPort : EmptyFile ,   Period : 10,   Execution-time : 4,
       Clock-P : 0, Clock-C : 0  >
 < GPS-TGPS : Connection | TypeConx : data, Source : GPS , Dest : TGPS , TimeTrans : 5 > .
```

Figure 6.6 - Instanciation de l'état initial des exemples de configurations de tests

6.3.3 Exécution comportementale

Pour analyser le comportement dans l'architecture AADL du système de contrôle de navigation en utilisant l'outil d'ABAReL, nous avons exploité les commandes de simulation et d'analyse formelle. Plusieurs tests ont été entrepris pour simuler l'effet des interactions entre les composants, l'exécution concurrente de plusieurs threads, l'effet du passage du temps sur une configuration, etc.

Par exemple, la commande `trew` dans la figure 6.7(a) permet de réécrire le terme `StoreConf` déclaré comme une configuration (du cas d'étude) dans les équations d'instanciation de la figure 6.6. Ceci permet de simuler le transfert de la donnée `data1` entre le composant `GPS` (de catégorie `device`) et le thread TGPS à travers la connexion `GPS-TGPS`.

(a)	(trew { StoreConf1 } in time <= 70 .)
(b)	(reduce delta(< TGPS : ThreadImpl | Cat : thread, mode : GPSupAPdown, IPort : receive, TState : compute , Substate : Running , OPort : NoPort, InBufferPort : data1, AccessData : Nill, OutBufferPort : EmptyFile , Period : 10, Execution-time : 4, Clock-P : 10, Clock-C : 4 >, 2) .)
(c)	(reduce mte(< TGPS : ThreadImpl | Cat : thread, mode : GPSupAPdown, IPort : receive, TState : compute , Substate : Ready , OPort : NoPort, InBufferPort : data1, AccessData : Nill, OutBufferPort : EmptyFile , Period : 10, Execution-time : 4, Clock-P : 10, Clock-C : 4 >) .)

Figure 6.7 Simulation et analyse du comportement d'un Thread

L'exécution de cette commande (`trew`) montre la réécriture de ce terme par application des règles de réécriture et des équations du module `AADL-SPEC` et en particulier la règle de réécriture `Data-Receive`.

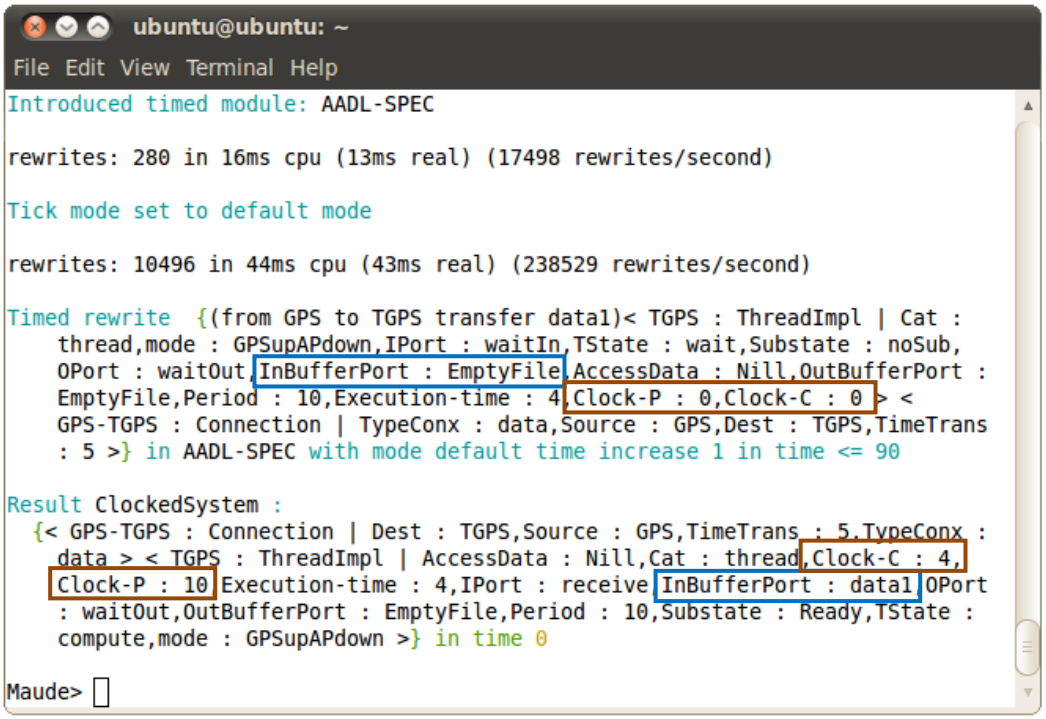


Figure 6.8 Exemple d'exécution d'une réécriture temporisée `trew`

L'image d'écran de la figure 6.8 montre le résultat de cette exécution. Nous constatons que la donnée `data1` est enfilée dans le buffer `InBufferPort` lié au

port d'entrée du thread `TGPS`, et que les horloges (`Clock-P` et `Clock-C`) sont initialisées pour l'exécution d'une nouvelle période de ce thread.

Nous appliquons ensuite la commande `reduce` respectivement sur les opérations `delta` (figure 6.7(b)), et `mte` (figure 6.7(c)). Cette commande permet de simuler les réécritures équationnelles, via les équations des opérations `delta` (pour la première commande) et `mte` (pour la seconde) dans le module temporisé `AADL-SPEC`. Le résultat d'exécution de la figure 6.9, montre l'effet du passage du temps sur la configuration du thread `TGPS` avec une décrémentation des valeurs des horloges `Clock-C` et `Clock-P`.

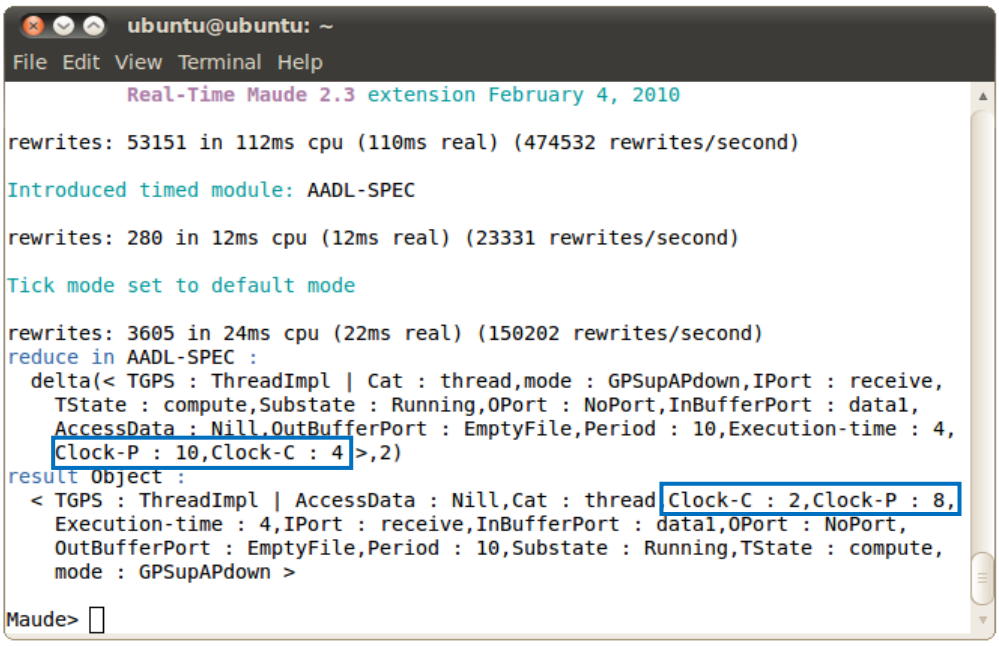


Figure 6.9 Effet du passage du temps sur la configuration du thread TGPS

Le résultat d'exécution de la figure 6.9, montre l'effet du passage du temps sur la configuration du thread `TGPS` avec une décrémentation des valeurs des horloges `Clock-C` et `Clock-P`.

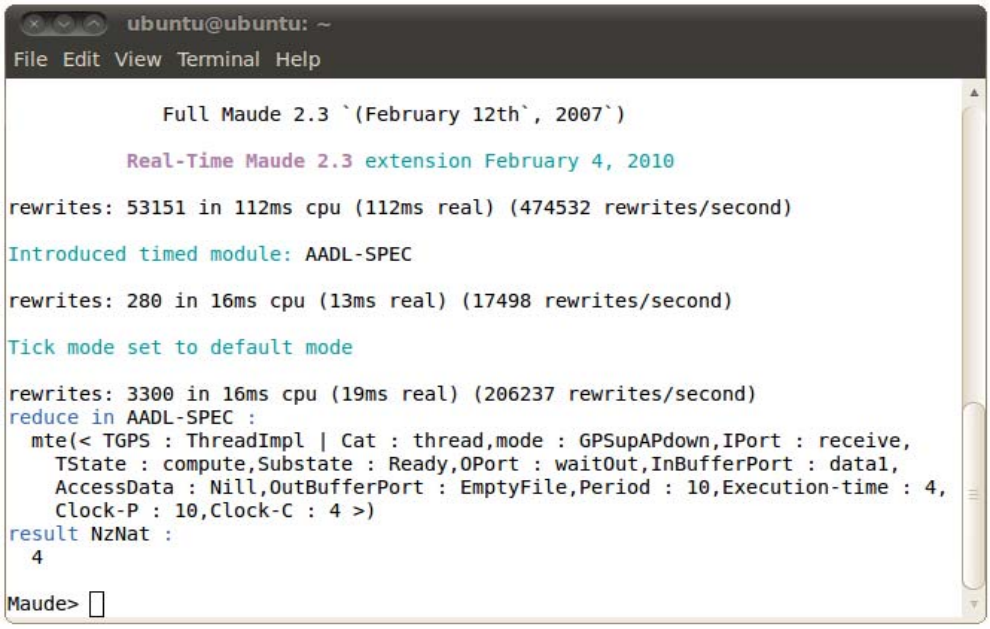


Figure 6.10 Effet de l'opération `mte` sur la configuration du thread `TGPS`

L'effet de l'opération `mte` sur la configuration du thread `TGPS`, à travers la figure 6.10, montre que cette opération a évalué le temps au minimum des valeurs des deux horloges.

6.3.4 Analyse model checking

Nous devons d'abord exprimer les propriétés à vérifier dans la syntaxe de la logique temporelle linéaire par des formules LTL. Nous spécifions ensuite ces formules dans le module `MODEL-CHECK-AADL-PROP` (figure 6.11) qui importe le module prédéfini `TIMED-MODEL-CHECKER` et le module `AADL-SPEC` pour être analysé. Dans ce module, la spécification des formules LTL, exprimant les propriétés à vérifier définies dans la section 6.3.3, se fait par la déclaration des propositions de sort `Prop`. Cette spécification est faite, en considérant les spécifications RT-Maude des configurations architecturales du module `AADL-SPEC`, par les propositions atomiques suivantes: `CompleteStateTGPS` **(P1)**, `CompleteStateTSCREEN` **(P1)** et `RunningState` **(P3)**.

```
(tomod MODEL-CHECK-AADL-PROP is
 including TIMED-MODEL-CHECKER .
 protecting AADL-SPEC .
var REST : Configuration .
 vars Imp : Oid .
 vars R R' R'' : Time .
 ops RunningState  CompleteStateTGPS

    CompleteStateTSCREEN : -> Prop [ctor].

eq {REST  < TGPS :  ThreadImpl | Substate : Complete   >}
   |=   CompleteStateTGPS = true .

eq {REST  < TSCREEN : ThreadImpl | Substate :  Complete >}
   |=  CompleteStateTSCREEN = true .

eq {REST < Imp : ThreadImpl| TState : compute, Substate :
   Running   >} |= RunningState = true.
endtom)
```

Figure 6.11- Spécifications RT-Maude des formules LTL

La vérification de la propriété **(P1)** par le LTL model-checker de RT-Maude est lancée par cette commande :

```
(mc initState1 |=t (<> CompleteStateTGPS)/\
(<> CompleteStateTSCREEN)  in time <= 70 .)
```

L'exécution de cette commande a donné le résultat de la figure 6.12 précisant que la propriété est non vérifiée. Un contre-exemple est alors retourné donnant une trace d'exécution, indiquant que le thread exhibe un comportement Zénon, signifiant qu'un nombre infini de transitions discrètes se produit dans un intervalle de temps fini.

En effet, la propriété **(P1)** n'est pas satisfaite parce que le thread est indéfiniment préempté (il y a une infinité de transitions d'état à travers le cycle : `resume`, `preempt` et `block-on Release-Resource`) pendant une période de temps limitée sans pouvoir finir son exécution. Par conséquence, le sous état (`Substate = Complete`) du thread ne peut jamais être atteint.

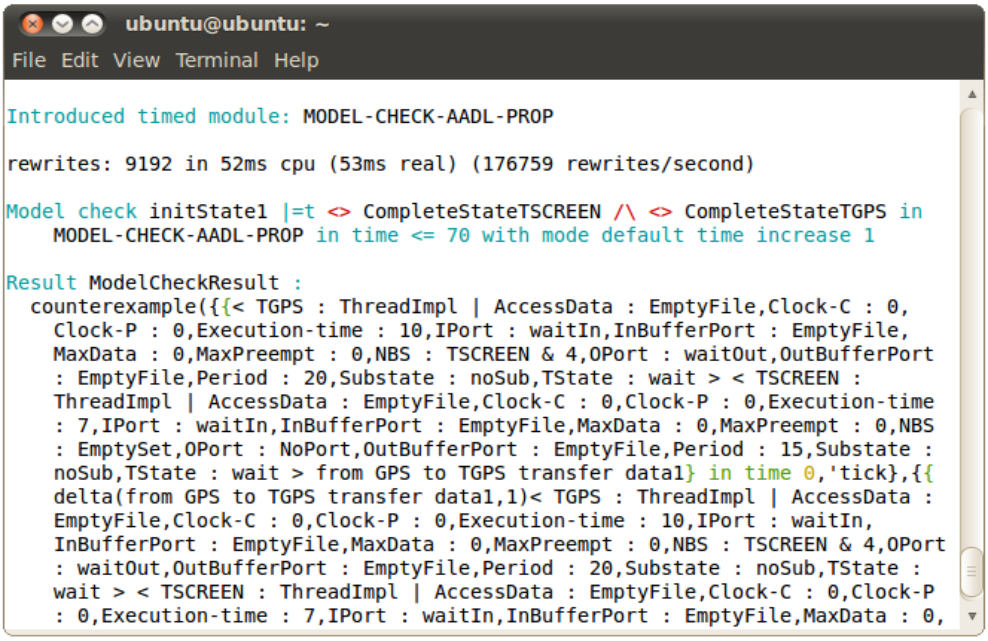


Figure 6.12 Vérification formelle des propriétés d'exécution AADL: "Comportement Zénon"

Cette première tentative de vérification de la propriété **(P1)** montre bien le comportement Zénon de l'automate du standard. Pour surmonter ce comportement incorrect, nous avons essayé d'adopter, dans un premier temps, une stratégie naïve. Celle-ci consiste à associer un intervalle de temps limité aux transitions `resume`, `preempt` et `block-on-Release-Resource`, en limitant le nombre de transitions durant l'exécution du thread (pas plus de deux).

Le résultat obtenu reste incorrect où d'autres comportements inattendus apparaissent. Cela est dû au fait que l'exécution des règles de réécriture du module temporisé `AADL-SPEC` se fait de façon aléatoire, il fallait donc guider le système de réécriture. La deuxième stratégie adoptée est donc d'ajouter des priorités à l'exécution de certaines règles.

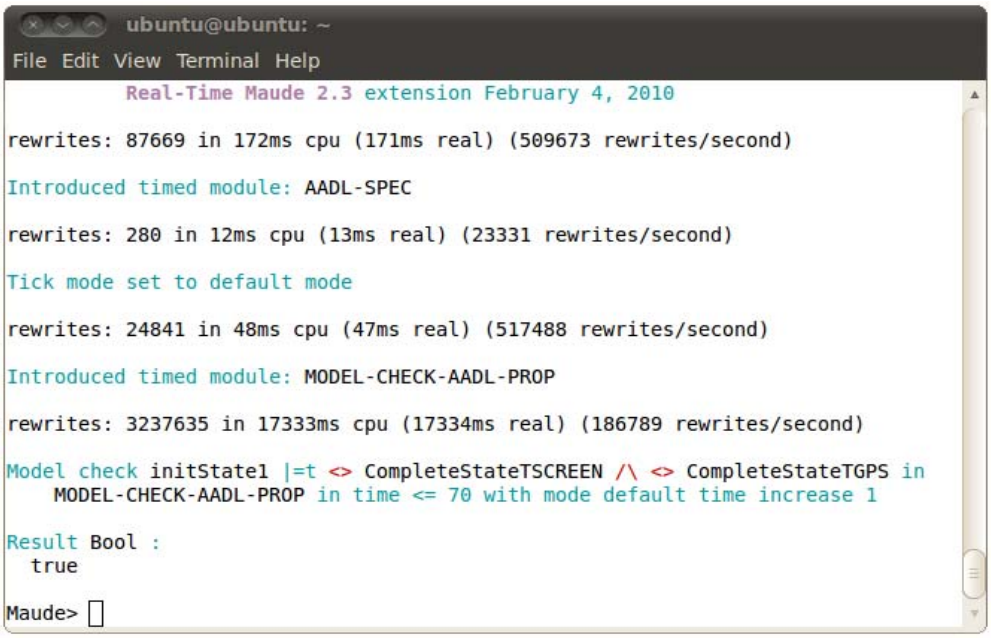


Figure 6.13 : Vérification formelle des propriétés d'exécution AADL

L'image d'écran de la figure 6.13 montre qu'après avoir adopté la deuxième stratégie, le problème du comportement Zénon est résolu et la propriété **(P1)** est évaluée à vrai. La vérification de cette propriété (`Substate = complete` est accessible dans le temps) pour les threads `TGPS` et `TSCREEN` implique que ces threads finissent par s'exécuter correctement et de façon concurrente. Nous pouvons considérer également que les propriétés d'exécution temporelles (`Period` et `Compute-Execution-Time`) sont vérifiées. Etant donné que le problème du comportement Zénon est résolu donc l'exécution d'un thread n'est pas préemptée indéfiniment et/ou en attente d'une ressource indéfiniment. Par conséquent la propriété **(P2)** est également vérifiée.

Pour vérifier la propriété de *vivacité* **(P3)** et justifier que le thread a fini par s'exécuter, nous appliquons la commande de recherche temporisée (`tsearch`) pour savoir si le sous état (`Substate = running`) est accessible à partir de l'état initial `initState` (instancié dans la figure 6.6) dans une durée de temps donnée.

```
(tsearch [10] {initState} =>*  {C:Configuration
 < TGPS : ThreadImpl | Substate : running  > }
                in time <= 50 .)
```

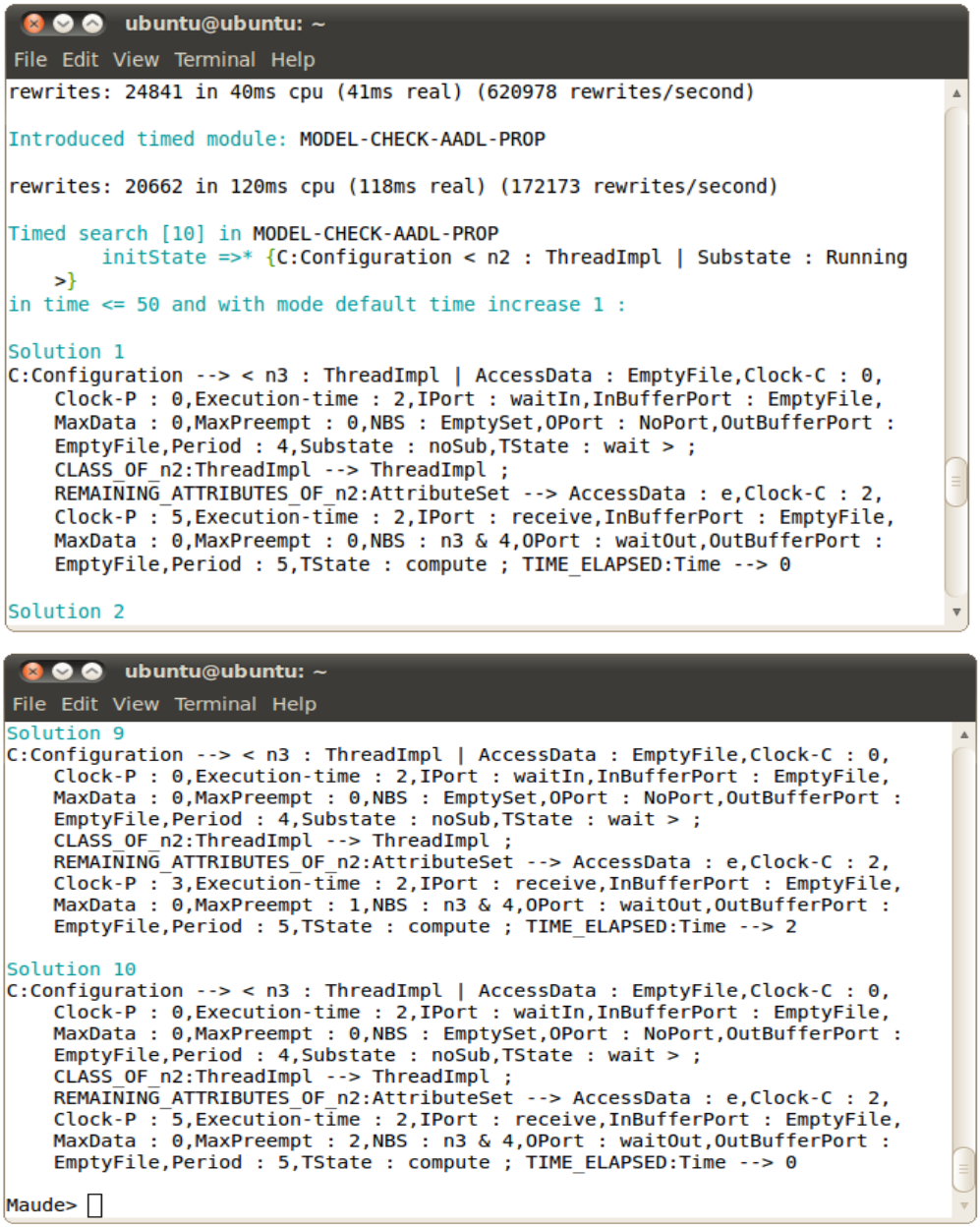


Figure 6.14 -Trace d'une recherche temporisée (`tsearch`)

Le résultat obtenu implique que l'exécution de chaque thread, dans la configuration de l'état initial, passe par le sous état (`Substate = running`). La recherche de cet état est matché à n'importe quel état qui contient (`Substate =`

`running`). La trace de la recherche temporisée (figure 6.14) montre que l'ensemble des états accessibles de l'état initial est fini.

6.4 Conclusion

L'annexe comportementale ABAReL offre une solution générique à base de théories orientées objet temps réel. Le prototypage de son modèle mathématique sous RT-Maude, propose un outil pour ABAReL permettant de spécifier formellement une architecture logicielle d'un système embarqué respectant la syntaxe du langage AADL, analyser son comportement et vérifier ses propriétés.

Etant donné que tous les outils d'analyse et de vérification supportant le langage AADL n'ont pas été conçus pour la spécification formelle et la vérification des propriétés sans passer par plusieurs transformations, nous avons présenté, dans une première partie de ce chapitre, une autre alternative qui consiste à implémenter le modèle sémantique ABAReL, proposé dans les chapitres précédents sous le système RT-Maude. Cette implémentation d'ABAReL permet d'autre part la vérification formelle des propriétés comportementales d'une architecture AADL composée d'un ensemble de composants de types threads en tirant profit des outils d'analyse et de vérification du système RT-Maude.

Nous avons illustré l'utilisation de l'outil issu d'ABAReL par une étude de cas reposant sur le modèle architectural AADL d'un système de contrôle de navigation. Nous avons exploité les procédures d'analyse et de simulation du comportement de RT-Maude pour montrer les différentes étapes de simulation du comportement de plusieurs exemples de configuration. Nous nous sommes particulièrement concentrées sur l'utilisation du model-checker LTL de RT-Maude pour la vérification des propriétés comportementales. Les premières tentatives de vérification ont échoué à cause du problème de comportement Zénon de l'automate du standard. Nous avons pu résoudre ce problème par une stratégie qui guide le système de réécriture de notre outil. Ainsi les propriétés comportementales exprimées dans la syntaxe de la logique temporelle linéaire LTL ont été vérifiées.

Conclusion générale

Sommaire

Actuellement, les systèmes embarqués sont présents dans des applications de plus en plus nombreuses (les cartes à puce, les téléphones mobiles, l'automobile, l'avionique, les capteurs intelligents, la santé et l'électronique grand public, etc.). Cependant, la complexité croissante de la technologie de l'embarqué fait que le processus de développement des systèmes embarqués doit suivre une démarche rigoureuse afin de répondre aux exigences souvent critiques de ces systèmes. Toutes les approches de développement classiques telles que l'approche synthèse niveau système, l'approche conception basée plateforme et l'approche basée modèles sont spécifiques à des exemples de systèmes particuliers et ne suggèrent pas de solutions génériques et efficaces. De plus, la conception des systèmes embarqués nécessite un niveau de fiabilité important qui ne peut être obtenu que par l'utilisation de méthodes et langages de haut niveau. Pour ce faire, le processus de développement a été orienté vers les approches basées architecture où l'adoption de la description architecturale joue un rôle important dans le développement de logiciels complexes et maintenables à moindre coût et avec respect des délais. Il est absolument nécessaire de pouvoir évaluer très tôt, durant les phases de développement de tels systèmes, les choix de conception et de raisonner correctement sur les comportements de ces systèmes. C'est dans ce contexte que se situe notre travail. Il présente comme objectif principal la contribution au développement d'une nouvelle approche pour la conception architecturale et l'analyse formelle des systèmes embarqués temps réel, basée sur la logique de réécriture révisée.

Dans le cadre de ces travaux, l'intérêt a été porté sur les descriptions architecturales offertes par le langage AADL, un langage de description d'architecture développé pour satisfaire les besoins spéciaux des systèmes embarqués temps réel. Ce langage se distingue notamment par sa capacité à rassembler au sein d'une même

notation l'ensemble des informations concernant l'organisation de l'application et son déploiement. Il définit également ses comportements par la déclaration des modes et des transitions de mode mais ne révèle aucun détail concernant la sémantique de son modèle d'exécution. L'intégration des méthodes formelles appropriées pour la prise en charge de l'aspect dynamique et concurrent de ces systèmes ainsi que leur analyse temporelle s'avère très utile et motive l'utilisation de la logique de réécriture à travers sa version étendue. Nous recensons dans les sections suivantes nos principales contributions et quelques perspectives futures pouvant constituer les continuations envisageables pour ce travail.

1. Contributions

Nos contributions dans le cadre de cette thèse concernent essentiellement :

Classification des approches de conception des systèmes embarqués : La réalisation d'un état de l'art touchant le développement des systèmes embarqués a été entreprise pour motiver le besoin d'adopter une nouvelle approche intégrée basé sur la description d'architectures et de méthodes formelles.

Définition d'une nouvelle annexe comportementale ABAReL pour AADL : Dans cette annexe nous avons [BBF08, BB08] associé à chaque composant AADL, un modèle mathématique représenté par une théorie de réécriture étendue décrivant sa structure statique, englobant toutes les structures déclarées au niveau de sa description AADL (éléments d'interface, flux, modes, propriétés : la période et le temps d'exécution, etc.). Nous nous sommes intéressées particulièrement au composant thread de AADL vu son importance dans la description de la dynamique d'un système embarqué. Nous avons spécifié alors son comportement par un suivi de ces configurations correspondant à des changements visibles dans le thread. En effet, un thread, dans une architecture logicielle d'un système embarqué en exécution, reçoit des données et/ou des événements en entrée, il effectue le calcul et envoie des signaux (données et/ou événements) en sortie. Ceci se concentre dans l'état *Compute* de l'automate associé au thread. En donnant une sémantique de réécriture à la description architecturale AADL, nous bénéficions du cadre unificateur de la logique de réécriture

[BM06], [MR04] qui permet à la fois la spécification des comportements atomiques d'un thread, en tenant compte de ses états et de ses sous états ou états hiérarchiques (sous états de l'état *Compute),* et les comportements concurrents de plusieurs threads en interaction dans une architecture d'exécution AADL.

Extension d'ABAReL par la spécification et l'analyse des propriétés AADL : Dans cette contribution [BB+09, BBB10], nous avons proposé une extension de l'annexe comportementale ABAReL par l'expression des propriétés d'exécution temporelles capturées à partir de la déclaration AADL d'un thread. Cette extension a permis d'enrichir le modèle mathématique d'ABAReL, basé sur la logique de réécriture révisée, par des constructions supplémentaires permettant de définir d'une part les configurations successives d'un thread périodique en exécution, et d'autre part un de ses états composites, à savoir l'état '*Compute*'. Une définition de ses sous états et leurs transitions ainsi que ses propriétés d'exécution temporelles ont été alors données. Les constructions formelles déduites sont utiles pour raisonner et analyser le comportement des threads dans une architecture AADL.

Implémentation du modèle mathématique d'ABAReL sous RT-Maude : Pour obtenir un modèle exécutable capable d'assurer la simulation du comportement et la vérification des propriétés d'exécution d'une architecture AADL, nous avons construit des théories de réécritures orientées objet temps réels en Maude implémentant l'essentiel d'ABAReL. Le modèle mathématique a été prototypé en utilisant le système RT-Maude [Ölv07]. Ceci a permis de décrire le modèle d'exécution d'une composition architecturale AADL de threads et leurs connexions à travers un multi-ensemble d'objets (*threads*) et de messages (*flux*) juxtaposés, où les interactions concurrentes entre les objets sont régies par des règles de réécriture. Le délai de transmission d'un message représente le temps de latence lié au flux. Nous nous sommes intéressés particulièrement aux propriétés d'exécution temporelles d'un thread, et à l'exécution concurrente de plusieurs threads, tout en supervisant l'effet du passage du temps sur la transmission des flux de données et/ou événements à travers les connexions entre les threads.

Nous avons illustré notre approche par une étude de cas reposant sur le modèle architectural AADL d'un système de contrôle de navigation. L'outil d'analyse et de simulation de RT-Maude nous a permis de voir les différentes étapes de simulation du comportement d'un exemple de configuration. La réalisation de cette formalisation en RT-Maude offre une spécification exécutable et extensible, dû à la flexibilité et la puissance de ce langage. Nous avons ensuite utilisé cette spécification comme entrée à l'outil LTL model checker du système RT-Maude pour la vérification d'un ensemble de propriétés exprimées dans la syntaxe de la logique temporelle linéaire LTL.

2. Perspectives

L'approche de formalisation proposée, à base de la logique de réécriture révisée, offre un cadre sémantique général, approprié pour raisonner sur le comportement des unités d'exécution concurrentes les `threads`. Nous avons procédé au raffinement de la description formelle du thread AADL de façon générique et naturelle pour prendre en compte d'une part, la modélisation des états concurrents et hiérarchiques d'un thread et d'autre part, la déclaration de ses propriétés d'exécution et des propriétés de propagation ainsi que leur intervention dans les exécutions à travers le temps. A ce stade de notre travail, nous proposons deux types de perspectives pertinentes : la première concerne l'extension du modèle ABAReL et la seconde se rapporte à l'analyse plus détaillée des systèmes embarqués via ABAReL.

L'extension d'ABAReL par la spécification d'autres types d'éléments architecturaux AADL enrichira cette vue architecturale d'un système embarqué. En particulier, les autres politiques de déclenchement d'un thread, apériodique, sporadique ou background, à l'aide de la propriété `Dispatch-Protocol` non considérés dans le cadre d'ABAReL peuvent être naturellement intégrées dans la spécification comportementale du thread. Notre approche de formalisation est générique et facilement extensible pour considérer tous ces concepts dans nos futurs travaux.

Pour la spécification des propriétés, le standard AADL considère plusieurs types de propriétés permettant d'exprimer des contraintes spatiales et temporelles, les descriptions des implantations et d'autres caractéristiques sur les entités AADL. Or, dans ce travail, nous nous sommes penchées uniquement aux propriétés temporelles

pour tester d'abord la faisabilité de l'approche. L'intégration des propriétés de descriptions des implantations `Source-Language`, `Source-Text`, `Source-Name` permettant d'associer un code source aux composants AADL fera l'objet d'une extension assez naturelle du modèle sémantique proposé.

Bibliographie

[ACD+08] T. Abdoul, J. Champeau, P. Dhaussy, P.-Y. Pillain, and J.-C. Roger. AADL execution semantics transformation for formal verification. In 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008), pp. 263-268, 2008.

[AG96] R. J. Allen, D. Garlan. The Wright Architectural Specification Language. Technical Report, MU-CS-96-TBD, Carnegie Mellon University, School of Computer Science, Pittsburg, PA, USA, September 1996.

[AxG02] Axlog-Geensyde. ADeS. Un simulateur pour AADL. Agence spatiale européene (ESA) 2002. http://www.axlog.fr/aadl/ades_fr.html.

[ABK+04] A. Alfonso, V. Braberman, N. Kicillof, A. Olivero. Visual Timed Event Scenarios. In Proceeding of the 26th ACM/IEEE ICSE'04. ACM Press, 2004.

[BB08] M. Benammar, F. Belala. ABAReL: Une Annexe Comportementale pour AADL. Séminaire sur les Systèmes Numériques Embarqués, SSNE 2008, EMP Bordj El-Bahri 05-06 Mai 2008.

[BB+09] M. Benammar, F. Belala, K. Barkaoui, N. Benlahrache. Extension d'ABAReL par les Propriétés d'Exécution. Revue de la Nouvelle Technologie de l'Information RNTI-L-4, Cépaduès éditions, 3ième Conférence Francophone Sur les Architectures Logicielles CFP-CAL'09 Nancy, France, pp. 45-58, ISSN : 1764.1667, ISBN : 978.2.85428.887.2, Mars 2009.

[BB10] M. Benammar, F. Belala. How to Make AADL Specification More Precise. International Journal of Computer Applications: IJCA, Volume 8-N°. 10, pages 16-23, ISSN 0975-8887, ISBN: 978-93-80746- 09-2. October 2010.

[BBB11] M. Benammar, F. Belala, K. Barkaoui. Implémentation Orientée Objet d'ABAReL en Maude. Rapport interne soumis à la Revue de Technique et Science Informatiques (TSI), en cours d'évaluation.

[BBC07] C. Bouanaka, F. Belala, A. Choutri. On Generating Tile System for a Software Architecture: Case of a Collaborative Application Session. In Proceeding of ICSOFT'2007 (the Second Conference on Software and Data Technologies), pp. 123-128, 2007.

[BBC+09] B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. Dal-Zilio, M. Filali, F. Vernadat. Formal Verification of AADL Specifications in the Topcased Environment. In Ada-Europe, pp. 207-221, 2009.

[BBF08] M. Benammar, F. Belala, F. Latreche. AADL Behavioral Annex Based on Generalized Rewriting Logic. In Proceeding of IEEE International Conference on Research Challenges in Information Science, RCIS'08, pp.7-13, Marrakech, Morocco June 3-6, 2008.

[BBF+09] M. Benammar, F. Belala, S. Fedali, F. Bensakhria, I. Sengouga. Modélisation Architecturale AADL du Système ABS. Embedded Systems Conference, ESC'09, EMP Bordj El-Bahri 05-06 Mai 2009.

[Ber02] A. Berger. Embedded System Design-An Introduction to Processes, Tools, and Techniques. CMP Books, ISBN: 1578200733, Lawrence, Kansas, USA 2002.

[BLB08] F. Belala, F. Latreche, M. Benammar. Vers l'Intégration des Propriétés non Fonctionnelles dans le Langage SADL. Revue de la Nouvelle Technologie de l'Information RNTI-L-2, Cépaduès-Editions, 2ième Conférence Francophone Sur les Architectures Logicielles CFP- CAL2008, pp.91-105, ISSN : 1764-1667, ISBN : 978.2.85428.826.1, Montréal, Canada, Mars 2008.

[BM06] Bruni, R., J. Meseguer. Semantic Foundations for Generalized Rewrite Theories. Theoretical Computer Science 360, pp. 386-414, 2006.

[BPB+08] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, F. Vernadat. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. Proceedings of the 4th European Congress on Embedded Real-Time Software ERTS'08 (Toulouse, France), January 2008.

[BRV04] B. Berthomieu, P.-O. Ribet, F. Vernadat. The tool TINA-construction of abstract state spaces for Petri nets and time Petri nets. International Journal of Production Research, Vol. 42, No 14, July 2004.

[BW90] J. C. M. Baeten, W. P. Weijland. Process algebra. Cambridge University Press, New York, NY, USA, 1990.

[CDE+99] M. Clavel, F. Duran, S. Eker, N. Martı-Oliet, P. Lincoln, J. Meseguer, J. Quesada. Maude: Specification and Programming in Rewriting Logic. SRI International Lab., (1999). http://maude.csl.sri.com,

[CDE+02] M. Clavel, M., F. Duran, S. Eker, N. Marti-Oliet, P. Lincoln, J. Meseguer, C. Talcott. Maude Manual. Version 2, 2002.

[CDE+07] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, C. Talcott. All About Maude – A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic. Lecture Notes in Computer Science, Springer-Verlag New York, Inc., Secaucus, NJ, USA 2007.

[CGP01] E. M. Clarke, O. Grumberg, D. A. Peled. Model Checking. MIT Press, Cambrigde, Massachusetts 2001.

[CGV 05] A. Courbot, G. Grimaud, J.-J. Vandewalle. Romization: Early Deployment and Customization Java Systems for Restrained Devices. Rapport de recherché INRIA N°5629 – Juillet 2005.

[CM97] M. Clavel, J. Meseguer. Reflection and Strategies in Rewriting Logic[1], Electronic Notes in Theoretical Computer Science 4, Elsevier Science B. V. 1997.

[CPG+ 06] A. Courbot, M. Pavlova, G. Grimaud, J.-J. Vandewalle. A Low-Footprint Java-to-Native Compilation Scheme Using Formal Methods. In Seventh Smart Card Research And Advanced Application IFIP Conference (CARDIS'06), Tarragona, Spain, Avril 2006.

[CRB+08] M. Chkouri, A. Robert, M. Bozga, J. Sifakis. Translating AADL into BIP - Application to the Verification of Real-time Systems. In Proc. of MoDELSACES-MB Model Based Architecting and Construction of Embedded Systems, pp. 39-54, Sept. 2008.

[CRC+06] B. Combemale, S. Rougemaille, X. Cregut, F. Migeon, M. Pantel, C. Maurel. Expériences pour décrire la sémantique en ingénierie des modèles. IDM'06, Secondes journées sur l'Ingénierie des modèles, Lille, France, 2006.

[DBF+06] P.Dissaux, J.P. Bodeveix, M. Filali, P. Gaufillet, F. Vernadat. AADL Behavioral annex. In Proceedings of the DASIA 2006 – DAta Systems In Aerospace, 2006.

[DGQ+02] S. Derrien, A. C. Guillou, P. Quinton, T. Risset, C. Wagner. Automatic Synthesis of Efficient Interfaces for Compiled Regular architectures. In International Samos Workshop on Systems, Architectures, Modeling and Simulation (Samos), Samos, Grece, Juillet 2002.

[DM99] F. Duran, J. Meseguer. The Maude specification of Full-Maude. Technical report, SRI International, Computer Science Laboratory, February 1999.

[DXW+05] Q. Deng, H. Xu, S. Wei, Y. Han, G. Yu. An Embedded SOPC System Using Automation Design. In Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPPW'05), IEEE Computer Society 2005.

[EMD+06] H. Espinoza, J. Medina, H. Dubois, S. Gerard, F. Terrier. Towards a UML-based Modeling Standard for Schedulability Analysis of Real-Time Systems. International Workshop on Modeling and Analysis of Real-Time and Embedded

Systems, MARTES at MoDELS, Research Rapport 343, ISBN 82-7368-299-4, ISSN 0806-3036, October 2006.

[EMS02] S. Eker, J. Meseguer, A. Sridharanarayanan. The Maude LTL Model Checker. In Fourth Workshop on Rewriting Logic and its Applications (WRLA'02), Vol. 71 of Electronic Notes in Theoretical Computer Science. Elsevier 2002.

[FBV06] P. H. Feiler, A. L. Bruce, S. Vestal. The SAE Architecture Analysis & Design Language (AADL) A Standard for Engineering Performance Critical Systems. In Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design, Munich, Germany, October 4-6, 2006.

[Fea00] P. Feautrier. Les Compilateurs. Revue de Technique et Science Informatiques, TSI, volume 19, pages 223-232, 2000.

[Fea96] P. Feautrier . Automatic Parallelization in the Polytope Model. In Guy-René Perrin and Alain Darte editors, The Data-Parallel Programming Model , LNCS volume 1132 , pages 79-103 , Springer 1996 .

[Fei04] P. H. Feiler. Pattern-Based Analysis of an Embedded Real-Time System Architecture. World Computer Congress, Workshop on Architecture Description Languages (WADL04), Toulouse, August 2004.

[Fei05] P. H. Feiler. Open Source AADL Tool Environment (OSATE). AADL Workshop, Paris, October 2005.

[Fei08] P. H. Feiler. The SAE Architecture Analysis & Design Langage (AADL) Standard. Software Engineering Institute, January 2008.

[Fra01] A. Fraboulet. Optimisation de la Mémoire et de la Consommation des Systèmes Multimédia Embarqués. Thèse de doctorat de l'université Lyon I, Soutenue le 23 Nov 2001.

[Gau05] P. Gaufillet. TOPCASED-Toolkit In Open source for Critical Applications & systems Development. AADL Workshop, Paris, October 2005.

[GML+07] H. Garavel, R. Mateescu, F. Lang, W. Serwe. CADP 2006 : A toolbox for the construction and analysis of distributed processes. In Werner Damm and Holger Hermanns, editors, CAV, Lecture Notes in Computer Science, Vol. 4590, pages 158-163. Springer 2007.

[GMW97] G. Garlan, R. T. Monroe, D. Wile. Acme : An Architecture Description Interchange Language. In Proceeding of CASCON'97, pages 169-183, Toronto, Ontario, November 1997.

[HM04] Honda Motors. SYSTEME ABS (ANTI-LOCK BRAKE SYSTEM). Les dossiers techniques Honda, Europe-France, 2004.

[HT05] B. Haberman, M. Trakhtenbrot. An Undergraduate Program in Embedded Systems Engineering. In Proceedings IEEE Computer Society of the 18th Conference on Software Engineering Education & Training (CSEET'05), 2005.

[Ifr05] H. Ifran. Flight Control System: Modeling of a Hardware/Software System in AADL. ASSERT AADL Workshop, 2005.

[IN05] H. Ishikawa, T. Nakajima. EarlGray: A Component-Based Java Virtual Machine for Embedded Systems. In Proceeding of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05), IEEE Computer Society 2005.

[Jer08] C. Jerad, K. Barkaoui. On the use of Real-Time Maude for Architecture Description and Verification: A Case Study. In BCS International Academic Conference "Visions of Computer Science". 2008.

[KKV95] C. Kirchner, H. Kirchner, M. Vittek. Designing constraint logic programming languages using computational systems. In: Principles and Practice of Constraint Systems: The Newport Papers, The MIT Press, pp. 133 160, 1995.

[Luc95] D. C. Luckham. Rapid : A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events. In Proceeding of the DIMACS Workshop on Partial Order methods in Verification, August 1996.

[MC03] F. Maraninchi, P. Caspi. La place de l'informatique dans l'enseignement des Logiciels et Systèmes Embarqués. Laboratoire Verimag, Avril 2003.

[Mes02] J. Meseguer. Rewriting Logic Revisited. Illinois University of Urbana-Champaign, USA, 2002.

[Mes90] J. Meseguer. Rewriting logic as a unified model of concurrency. In Lecture Notes in Computer Science, Volume 458, pages 384-400. August 1990.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency, Theoretical Computer Science, Volume 96(1), pages 73-155, 1992.

[Mes96] J. Meseguer. Rewriting logic as a semantic framework for concurrency. In Lecture Notes in Computer Science, editor, 7th International Conference on Concurrency Theory, volume 1119. Springer Verlag August 1996.

[Mes92] J. Meseguer. Multiparadigm Logic Programming. In: H. Kirchner and G. Levi (eds.), Proc. Third Int. Conf. on Algebraic and Logic Programming, LNCS 632, Springer-Verlag, pages 158-200, 1992.

[Mig99] A. Mignotte. Compilation sur silicium ou conception conjointe matérielle-logicielle. Thèse d'habilitation à diriger des recherches, Université Claude Bernard de Lyon, mars 1999.

[MM96] N. Marti-Oliet, J. Meseguer. Rewriting logic as a logical and semantic framework for concurrency. 7th International Conference on Concurrency Theory, Volume 1119 of Lecture Notes in Computer Science. Springer-Verlag. 1996.

[MM01] N. Marti-Oliet, J. Meseguer. Rewriting Logic: Roamap and Bibliography, In Theoretical Computer Science, June 2001.

[MM93] N. Marti-Oliet, J. Meseguer. Rewriting logic as logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993.

[MR04] J. Meseguer, G. Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In "IJCAR", pages 1-44, 2004.

[MT97] J. Meseguer, C. Talcott. Formals Foundations for Compositional Software Architectures. Position Paper, OMG-DARPA-MCC Workshop on Compositional Software Architectures, 1997.

[MOY+08] D. Monteverde, A. Olivero, S. Yovine, V. Braberman. VTS-based Specification and Verification of Behavioral Properties of AADL Models. Verimag Research Report n° TR-2008-11, August 12, 2008.

[Ölv07] P. C. Ölveczky. Real-Time Maude 2.3 Manual. Department of Informatics, University of Oslo, 2007.

[ÖM00] P. C. Ölveczky, J. Meseguer. Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems. In 3rd International Workshop on Rewriting Logic and its Applications (WRLA'00), Vol. 36 of Electronic Notes in Theoretical Computer Science. Elsevier 2000.

[ÖM96] P. C. Ölveczky, J. Meseguer. Specifying Real-Time Systems in Rewriting Logic. 1st International Workshop on Rewriting Logic and its Applications (WRLA'96), In Lecture Notes in Computer Science, editor, Vol. 4. Springer Verlag 1996.

[ÖM04] P. C. Ölveczky, J. Meseguer. Specification and Analysis of Real-Time Systems using Real-Time Maude. Fundamental Aspects of Software Engineering (FASE'2004), In Springer, editor, Vol. 2984 of "Lecture Notes in Computer Science", 2004

[ÖM02] P. C. Ölveckzy, J. Meseguer. Specification of Real-Time and Hybrid Systems in Rewriting Logic. Theoretical Computer Science, Volume 285, Issue 2, pp. 359-405, 2002.

[OMG05] OMG. UML Profile for Schedulability, Performance, and Time Specification. OMG document number: formal/05-01-02 (v1.1). January 2005.

[OMG99] OMG. Unified Modeling Language Specification. Version 1.3, June 1999.

[PBF09] L. Pi, J.P. Bodeveix, M. Filali. A Comparative study of different formalisms to define AADL data communication. UML&AADL'2009-14th IEEE International Conference on Engineering of Complex Computer Systems, 2009.

[QRW00] F. Quilleré, S. Rajopadhye, D. Wilde. Generation of Efficient Nested Loops From Polyhedra. Kluwer Academic Publishers. Printed in the Netherlands, 2000.

[Ras00] F. Rastello. Partitionnement : optimisation de compilation et algorithmique hétérogène. Thèse de doctorat, ´ Ecole Normale Supérieure de Lyon, septembre 2000.

[RIS00] T. Risset. Contribution à la compilation de nids de boucle sur silicium. Rapport d'habilitation à diriger des recherches, Université de Rennes 1, soutenue le 22 Nov 2000.

[RKH09a] X. Renault, F. Kordon, J. Hugues. Adapting models to modelcheckers, a case study : Analyzing AADL using Time or Colored Petri Nets. In Proceedings of the 20th International Symposium on Rapid System Prototyping, IEEE Computer Society, pages 26-33, Paris June 2009.

[RKH09b] X. Renault, F. Kordon, J. Hugues. From AADL architectural models to Petri Nets: Checking model viability. 12th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'09), IEEE Computer Society, pages 313-320, Tokyo, Japan March 2009.

[SAE04] SAE International Avionics Systems Division (ASD) AS-2C Subcommittee. Avionics Architecture Description Language Standard. SAE Document AS 5506, Nov. 2004. http://www.sae.org

[SAE06] SAE International. Annex C: AADL Meta Model and Interchange Formats. SAE AADL Meta Model/XMI V0.999, October 2006.

[SAE09] SAE International. Architecture Analysis and Design Language (AADL) Standard, Version 2. SAE Draft Standard AS5506 V2. 2009.

[SEI04] SEI. OSATE: an extensible Source AADL Tool Environment. SEI AADL team Technical Report, December 2004.

[Sif05] J. Sifakis. A Framework for Component-based Construction. IEEE SEFM'05, Keynote talk, September 7-9, 2005.

[SLC09] O. Sokolsky, I. Lee, D. Clarke. Process-Algebraic Interpretation of AADL Models. 14th International Conference on Reliable Software Technologies, LNCS 5570, pages 222-236, (Ada-Europe), Brest, France June 8-12, 2009.

[SLNM04] F. Singhoff, J. Legrand, L. Nana, L. Marcé. Cheddar: a flexible real time scheduling framework. ACM SIGAda Ada Letters, Volume XXIV Issue 4, pages 1-8, 2004.

[SMÖ01] M.O. Stehr , J. Meseguer, P.C. Ölveckzy. Rewriting logic as a unifying framework for Petri Nets. In Unifying Petri Nets (Advances in Petri Nets), Volume 2128 of Lecture Notes in Computer Science, pages 250-304. Springer-Verlag. 2001.

[TGP+05] C. Talarico, A. Gupta, E. Peter, J. W. Rozenblit. Embedded System Engineering Using C/C++ Based Design Methodologies. In Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05), IEEE Computer Society 2005.

[Ves98] S. Vestal. Software Programmer's Manual for the Honeywell Aerospace Compiled Kernel (MetaH Language Reference Manual). Technical report, Honeywell Technology Center, Minneapolis, USA, 1998.

[VR05] T. Vallius, J. Röning. Embedded Object Architecture. In Proceedings of the 8th Euromicro Conference on Digital System Design (DSD'05), IEEE Computer Society 2005.

[YHMP09] Z. Yang, K. Hu, D. Ma, L. Pi. Towards a Formal Semantics for The AADL Behavior Annex. In Design, Automation & Test in Europe DATE 2009, pages 1166-1171, Nice, France, 20-24 April 2009.

[Zel96] G. Zelesnik. The UNICON Language Reference Manual. Technical Report, Pittsburg, 1996.