ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2012-ENST-0036

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « Informatique et Réseaux »

présentée et soutenue publiquement par

Gilles LASNIER

le 27 Août 2012

Une Approche Intégrée pour la Validation et la Génération de Systèmes Critiques par Raffinement Incrémental de Modèles Architecturaux

Directeur de thèse : **Laurent PAUTET**
Co-encadrement de la thèse : **Jérôme HUGUES**

Jury

M. Yvon KERMARREC, Professeur, TELECOM Bretagne	Rapporteur
M. Frank SINGHOFF, Professeur, Université de Bretagne Occidentale	Rapporteur
M. Fabrice KORDON, Professeur, Université de Pierre et Marie Curie	Examinateur
M. Elie NAJM, Professeur, TELECOM ParisTech	Examinateur
M. Laurent PAUTET, Professeur, TELECOM ParisTech	Directeur de thèse
M. Jérôme HUGUES, Maître de conférences, ISAE	Co-encadrant de thèse

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

THÈSE

Á Anésie,
Á Alain.

© 2012 Gilles LASNIER

© 2012 Gilles LASNIER

Remerciements

Ce moment tant espéré est enfin arrivé. L'écriture des remerciements scelle ce voyage important avec pour origine l'île de la Réunion et arrivée la métropole (et Paris). Ce voyage fut exaltant et très enrichissant notamment à travers de nombreuses étapes internationales comme les États-unis. Mais celui-ci n'aurait pu aboutir sans l'aide de personnes remarquables. Cette section n'est qu'une modeste tentative de reconnaissance à ces personnes qu'il m'a été donné de (re-) connaître ou de rencontrer pendant ces années.

Je tiens à remercier sincèrement mes directeurs de thèse Laurent PAUTET, professeur à TELECOM ParisTech et Jérôme HUGUES, maître de conférences à l'Institut Supérieur de l'Aéronautique et de l'Espace. Les idées, les conseils et les ressources qu'ils n'ont cessé de m'apporter m'ont tout simplement permis de mener à terme mes travaux de recherches. La confiance qu'ils m'ont accordée m'a permis de révéler certaines de mes capacités bien réelles qui me serviront pendant de nombreuses années. Ma reconnaissance envers eux est un peu comme l'histoire du toit de l'Océan Indien, le "Piton des Neiges".

Je remercie Yvon KERMARREC, professeur à TELECOM Bretagne et Frank SINGHOFF, professeur à l'Université de Bretagne Occidentale pour avoir accepté d'être les rapporteurs de ce mémoire. Les remarques pertinentes qu'ils ont soulevées ont assurément amélioré et consolidé ce manuscrit.

Je remercie également Fabrice KORDON, professeur à l'Université de Pierre et Marie Curie, qui, pour l'anecdote, aura été à l'origine et à la fin de ce voyage, me faisant l'honneur de présider le jury de ma soutenance de thèse. Merci également à Elie NAJM, professeur à TELECOM ParisTech pour avoir accepté de faire partie de mon jury de thèse.

Je tiens également à remercier Peter FEILER, directeur de recherche du Software Engineering Institute - Carnegie Mellon pour l'intérêt qu'il a porté à mes recherches et pour son invitation à Pittsburgh pour l'intégration de mes travaux de recherches.

Effectuer cette thèse au sein du département INFRES de TELECOM ParisTech m'a donné l'opportunité de faire la connaissance de plusieurs collègues et maintenant amis. Le temps passé avec vous, nos nombreuses discussions souvent vives et intéressantes, m'ont amené à apprécier encore plus ces trois années de thèse. Merci à Bechir ZALILA, Irfan HAMID, Hoa HA DUONG, Raja CHIKY, Nesrine GABSI, Billel GUENI, Nora DEROUICHE, Olivier GILLES, Khaled JOUINI, Asmah SOUIHLI, Khalil MAHRSI, Michael LAFAYE, Fabien CADORET, Xavier JEAN et Cuauthemoc CASTELLANOS. Je ne vous oublierai jamais.

Je remercie chaleureusement Olivier MARIN, maître de conférences à l'Université de Pierre et Marie Curie et Isabelle DEMEURE (chef) professeur à TELECOM ParisTech pour m'avoir permis d'exercer mes activités d'enseignement au sein des parcours et modules dont ils ont la responsabilité et pour les conseils pertinents qu'ils m'ont donnés tout au long de ces trois années. Olivier a été un tuteur exemplaire et a grandement facilité mon introduction dans le domaine de l'enseignement.

Ma gratitude va également à Philippe DAX, Serge GADRET, Hayette SOUSSOU, Florence BESNARD et tout le personnel de TELECOM ParisTech pour avoir toujours été disponibles et pour m'avoir fourni les supports matériels, logiciels et administratifs nécessaires pour effectuer mes recherches dans les meilleures conditions.

Je n'oublie pas mes "dalons la kour" de la Réunion qui auront su faire de mes retours au "péi" de précieux moments que ce soit lors de nos grillades, sandballs ou de nos séjours Saint-Gillois. A ceux-ci s'ajoutent mes amis métropolitains du Master SAR de l'UPMC et de l'équipe de hand de TELECOM ParisTech grâce à qui le mal du pays a été considérablement amoindri durant ces années passées à Paris. Merci donc à Hayder, Laurent, Lam, Youliang,

© 2012 Gilles LASNIER

Sabrina, Jean-Louis, à la “dreamteam974”, Sarah, Florian, Nadir, et à tout ceux que je n’ai pas pu citer ici, j’espère qu’ils me pardonneront !

Je tiens à remercier très chaleureusement ma (très grande) famille, qui m’a toujours encouragé et m’a permis de me ressourcer dans mon île quand le besoin se faisait plus pressant. Je remercie particulièrement Evelyne et Pascal pour ces fous rires inoubliables autour de nos trop nombreux repas gargantuesques. Merci également à mes frères Laurent et Yves pour leur soutien.

Enfin, je ne saurai jamais exprimer à sa juste valeur ma reconnaissance et mes pensées à mes parents Anésie et Alain. Même s’ils n’ont pas toujours compris ce que je faisais, ils n’ont jamais cessé de m’encourager et de m’apporter un soutien indéfectible pendant ces nombreuses années pas toujours faciles et ce malgré la distance. C’est donc tout naturellement que ce document leur est dédié.

“Oh yeah, there’s one more thing !”, toutes mes pensées vont enfin à Souad pour avoir été à mes côtés une très (trop) longue partie de cette thèse et particulièrement dans les moments difficiles - mao.

© 2012 Gilles LASNIER

Résumé

L'augmentation de la complexité des systèmes temps-réel répartis embarqués (TR^2E) et leur implication dans de nombreux domaines de notre quotidien imposent de nouvelles méthodes de réalisation. Dans les domaines dits critiques (transport, médecine...) ces systèmes doivent satisfaire des contraintes dures pour garantir leur bon fonctionnement et éviter toutes défaillances qui engendreraient des conséquences financières ou humaines dramatiques.

L'*Ingénierie Dirigée par les Modèles* (IDM) introduit le "modèle" - *i.e.* une description abstraite du système - et un ensemble d'outils (édition, transformation...) permettant la simplification et l'automatisation des étapes de conception, de validation et de génération du système. Ainsi, différentes abstractions du système sont élaborées dans des formalismes spécifiques de manière à couvrir un aspect du système et à permettre la réutilisation des outils d'analyse et de génération existants. Cependant, ces multiples représentations évoluent à des niveaux d'abstractions différents et il n'est pas toujours évident de mettre en corrélation système analysé et système généré.

Ce travail de thèse exploite les concepts et les mécanismes offerts par l'IDM pour améliorer la fiabilité du processus de réalisation des systèmes critiques basé sur les modèles. L'approche que nous avons définie repose sur la définition du langage de modélisation architecturale et comportementale AADL-HI Ravenscar - un sous-ensemble du langage AADL (*Architecture Analysis & Design Language*) et de son annexe comportementale - contraint pour permettre conjointement l'analyse et la génération de l'ensemble des composants de l'application y compris de son exécutif, avec une sémantique proche d'un langage de programmation impératif. En effet, les descriptions architecturales permettent d'exprimer l'ensemble des éléments du système à un niveau d'abstraction pertinent pour l'analyse et la génération, mais il est nécessaire de les étendre et de les enrichir avec des descriptions comportementales de manière à expliciter les composants intergiciels de l'application et à réduire les différences sémantiques entre le modèle et le langage de programmation cible.

Nous proposons un nouveau processus de réalisation qui (1) raffine la description architecturale AADL initiale de l'application vers un modèle architecturale et comportementale AADL-HI Ravenscar, (2) génère et intégre automatiquement les composants intergiciels configurés statiquement en fonction des besoins de l'application et (3) analyse et produit automatiquement le code source de l'ensemble des composants constituants l'application et sa plate-forme d'exécution personnalisée. Ce processus est implanté à l'aide de techniques et outils de transformation de modèles facilitant la sélection des raffinements et des règles de transformation en fonction de la plate-forme d'exécution et du langage cible.

Mots-clés: systèmes répartis, systèmes embarqués, systèmes temps-réel, systèmes critiques, spécification comportementale, transformation de modèles, POLYORB-HI-AADL, AADL-HI Ravenscar, AADL-HI, AADL, IDM

Abstract

The increasing complexity of distributed realtime and embedded (DRE) systems and their implication in various domains imply new design and development methods. In safety-criticial domains such as space, aeronautical, transport or medicine, their failure could result in the failure of the mission, or in dramatic damages such as human losses. This particular class of systems comes with strong requirements to satisfy safety, reliability and security properties.

The Model-driven Engineering (MDE) introduces the concept of « model » - an abstract description of the system and a set of tools (editor, transformation engine, code generator) to simplify and automatize the design, the validation and the implementation of the system. Thus, various abstractions are realized using different domain-specific modeling languages in order to assess one particular aspect of the system and to re-use model-based analysis tools and generative technologies. These various representations may share some commonalities but the consistency between them is hard to validate (for example : Is the analyzed system the same as the generated one ?).

This PhD thesis leverages MDE concepts and mechanisms, to enhance the reliability of the model-based development process of DRE systems. Our approach is based on the definition of the architectural and behavioral modeling language AADLHI Ravenscar, a restriction of AADL (Architecture Analysis & Design Language) and its behavioral annex. This subset of AADL constructs, comes up with a semantic close to the one of an imperative programming language, to drive both the analysis and the code generation of the application components and its relying execution platform (middleware) components. We claim that architectural descriptions express all system component information (architecture, requirements, properties) relevant for those activities, but it is necessary to extend them with behavioral specifications to make explicit middleware components and artefacts and to reduce the semantic gap between the model and the targeted programming language.

Our approach defines a new design and development process which 1) refines the initial AADL architectural description of the application to an AADLHI Ravenscar architectural and behavioral model ; 2) generates and integrates middleware components, deployed and configured statically from the analysis of the application requirements and its properties ; and 3) validates and produces the source code of the whole system, i.e. the application components and the dedicated middleware components. This process is implemented using model-based transformation methods and frameworks that ease the selection of refinement steps and transformation rules according to the targeted execution platform and the programming language.

Keywords: distributed, realtime, embedded, safety-critical, behavioral specification, model transformation, POLYORB-HI-AADL, AADLHI Ravenscar, AADL-HI, AADL, MDE

Table des matières

© 2012 Gilles LASNIER

© 2012 Gilles LASNIER

© 2012 Gilles Lasnier

© 2012 Gilles LASNIER

© 2012 Gilles LASNIER

V Conclusion et Perspectives

VI Annexes

© 2012 Gilles LASNIER

© 2012 Gilles LASNIER

Liste des illustrations

© 2012 Gilles LASNIER

Liste des exemples de code

© 2012 Gilles LASNIER

© 2012 Gilles LASNIER

© 2012 Gilles LASNIER

Liste des tableaux

© 2012 Gilles LASNIER

Liste des Algorithmes

© 2012 Gilles LASNIER

Première partie

Introduction Générale

Chapitre 1

Introduction

SOMMAIRE

1.1 Présentation du contexte général

La conception des systèmes temps-réel répartis embarqués est une tâche rendue de plus en plus complexe de par leur implication dans des domaines variés de notre quotidien (transport, loisirs, etc.). Que ce soit pour parcourir dix mille kilomètres par avion pour atteindre une île tropicale, se localiser dans le désert ou écouter de la musique en attendant le métro, ces systèmes doivent se conformer à des exigences hétérogènes (fiabilité, performance, sûreté, etc.) dépendantes du contexte environnemental dans lequel ils évoluent. Notamment, certains systèmes dits *critiques*, issus par exemple du domaine avionique ou militaire, doivent garantir leur sécurité d'utilisation, leur bon fonctionnement ou encore l'intégrité des données qu'ils manipulent. Une défaillance de ce type de système dûe à une erreur de conception pourrait engendrer des conséquences dramatiques, financières voire humaines.

Pour répondre à la complexité croissante de ces systèmes dûe à l'augmentation du nombre de fonctionnalités qu'ils doivent fournir, de nombreuses méthodes ont été proposées pour faciliter leur réalisation. Dernièrement, l'ingénierie dirigée par les modèles a permis l'automatisation du processus d'analyse et d'implantation de ces systèmes à partir de la description de leur architecture matérielle et logicielle. Des représentations différentes du système sont ainsi élaborées dans l'objectif de couvrir un aspect du système (faisabilité, performance, comportement, déploiement, etc.). Les modèles d'analyse permettent de vérifier et de valider le système vis-à-vis de ces spécifications. Les modèles de déploiement permettent de spécifier le déploiement des composants logiciels sur la plate-forme matérielle et automatisent la production du code source de l'application.

Pour autant que ces représentations expriment chacune des propriétés pertinentes du système, les formalismes employés pour leur description ne sont pas forcément les mêmes. Ainsi, chaque formalisme dédié à un aspect du système ne permet pas forcément d'exprimer les mêmes hypothèses qu'un autre ni le même niveau de détail. Le passage d'un formalisme à un autre s'effectue parfois manuellement et rien ne garantit que les propriétés du système soient préservées lors de cette transformation. Cette problématique est aujourd'hui traduite par la difficulté d'assurer la cohérence des représentations qu'elles soient dédiées à l'analyse du système ou utilisées pour la génération et le déploiement automatique des composants du système. De plus, les analyses sont souvent effectuées sur une représentation partielle du système, sa partie applicative, omettant l'impact de sa partie exécutive (par exemple, des ressources d'un intergiciel) pouvant même compromettre sa faisabilité. Ce dernier point rend difficile l'implantation du processus de génération de code qui nécessite d'effectuer des hypothèses sur le support d'exécution des composants et d'élaborer des transformations complexes. Ceci augmente les différences sémantiques entre le modèle et le code généré.

Ce travail de thèse s'articule autour de la définition et de la réalisation d'un processus automatisé pour l'analyse et la génération de systèmes distribués dans les domaines temps-réel embarqués critiques, renforçant la cohérence entre ces différentes représentations et, par conséquent, la fiabilité du processus.

Dans la suite de ce chapitre, nous définissons le périmètre d'activité et les problématiques de ces travaux de recherche. Puis, nous présentons les contraintes de réalisation et nous identifions les objectifs à atteindre. Enfin, la présentation de l'approche définie pour la réalisation de ces objectifs conclut ce chapitre.

1.1.1 Périmètre d'étude

Le travail de recherche que nous présentons propose une méthode pour renforcer la fiabilité du processus d'analyse et de production automatique des systèmes temps-réel répartis embarqués (TR^2E) basé sur une approche d'*Ingénierie Dirigée par les Modèles* (IDM).

Pour déterminer notre périmètre d'étude, intéressons-nous premièrement aux différents aspects de ces systèmes.

Définition 1.1 (Systèmes critiques) *[Rushby, 1994] Un système critique est un système dont l'échec ou la panne peut avoir de lourdes conséquences financières, environnementales, ou humaines.*

Ces systèmes sont évalués en fonction de leur niveau de criticité relatif à l'impact possible des défaillances. On distingue les catégories suivantes : *Business critical*, provoque des pertes coûteuses ; *Mission critical*, provoque l'abandon de la mission ; *Life critical*, provoque

© 2012 Gilles LASNIER

des pertes humaines ou environnementales pouvant être graves. Ces systèmes doivent respecter des contraintes particulières tout au long de leur développement pour offrir des garanties sur leurs aspects critiques. Ils doivent être documentés, suivis depuis leur conception jusqu'à leur implantation et leur déploiement, testés et validés.

Définition 1.2 (Systèmes TR^{2}E) *Un système TR^{2}E est un système qui supporte une application répartie, devant respecter des contraintes temps-réel, et s'exécutant dans un environnement contraint. De tels systèmes peuvent être critiques.*

Ces systèmes doivent être analysés et validés pour s'assurer de leur adéquation aux exigences et aux contraintes de leur environnement d'exécution. Ainsi, en plus des contraintes de criticité, ils doivent se soumettre aux contraintes suivantes :

- Temps-réel : les fonctionnalités du système doivent être rendues dans un délai fixe, au-delà cela peut correspondre à un cas d'erreur. Elles doivent donc être déterministes et prédictibles de façon à respecter les exigences temporelles spécifiées.
- Réparti : les fonctionnalités du système sont réparties sur différents nœuds de calcul qui communiquent entre eux par envoi de messages sur un bus de communication. Ceci nécessite la mise en œuvre de mécanismes de répartition si possible standardisés pour assurer l'interopérabilité entre des nœuds hétérogènes.
- Embarqué : le système doit répondre à des contraintes fortes en terme d'empreinte mémoire, de ressources matérielles (autonomie énergétique...) et de fiabilité.

L'*Ingénierie Dirigée par les Modèles* (IDM) centre le processus de développement autour de la modélisation du système. A partir de cette modélisation, il est possible de raisonner sur les propriétés du système avant son implantation, d'automatiser ces analyses et la production du système. Intéressons-nous aux notions principales que définit l'IDM :

Définition 1.3 (Méta-modèle) *Un méta-modèle décrit la structure qu'un modèle doit respecter afin d'être valide.*

Un méta-modèle peut être comparé à la grammaire pour la construction d'un langage. Il sert à exprimer les concepts communs à l'ensemble des modèles d'un même domaine.

Définition 1.4 (Modèle) *Un modèle est une représentation abstraite d'un système permettant une meilleure compréhension de celui-ci.*

Un modèle définit ainsi une abstraction du système. Il est souvent exprimé dans un langage de modélisation généraliste comme UML ou dans un langage dédié à un domaine précis. On parle alors de *Domain-specific modeling language* (DSML) comme AADLv2 [SAE Aerospace, 2009b] pour la modélisation de systèmes critiques ou VHDL [IEEE, 2009] pour la description d'architecture matérielle.

Définition 1.5 (Niveau d'abstraction d'un modèle) *Le niveau d'abstraction d'un modèle définit le niveau de détail requis pour la description des éléments (ou fragments) d'un modèle.*

Le niveau d'abstraction définit à quelle granularité le modèle créé modélise le système réel. Ainsi, trop élevée, la vision du système devient généraliste et la quantité d'informations utilisables pour procéder à des analyses est trop faible. Trop fine, la vision du système est trop détaillée et les informations peuvent parasiter le modèle. Les outils manipulant ces modèles

© 2012 Gilles LASNIER

sont, souvent, plus efficaces avec des modèles de taille raisonnable : trop d'informations peut augmenter la taille des modèles et complexifier l'extraction des informations pertinentes pour l'analyse en cours.

Au cours du processus de développement, ce niveau d'abstraction peut évoluer en fonction des analyses ou des actions que l'on souhaite effectuer. La transformation de modèles est le moyen permettant de faire évoluer ce niveau d'abstraction.

Définition 1.6 (Transformation de modèles) *Génération d'un ou plusieurs modèles à partir d'un ou plusieurs modèles sources selon une description de transformation.*

En conclusion, la réalisation d'un système TR^{2}E critique est complexe car nous devons nous assurer qu'il satisfait différents types de contraintes propres à chacun de ces aspects (embarqués, temps-réel, répartis et critiques). L'IDM est une approche permettant de vérifier ces contraintes et de valider le système TR^{2}E par l'analyse d'informations structurées spécifiées au sein d'une abstraction (modèle) du système. Différentes abstractions sont ainsi réalisées pour analyser un aspect du système mais aussi pour produire de manière automatique l'implantation de l'application.

1.2 Problématiques

Notre problématique vise à améliorer la cohérence entre les modèles (pour l'analyse, le déploiement, etc.) et le code généré afin de renforcer la fiabilité du processus pour la réalisation des systèmes TR^{2}E critiques et les rendre plus sûrs.

Précisément, il s'agit de définir et de construire un processus qui inclut l'analyse de l'application à partir d'une description complète de sa partie applicative (composants applicatifs) et de sa partie exécutive (composants intergiciels). Le niveau d'abstraction de cette description doit limiter les différences sémantiques entre la modélisation et le code généré de façon à rendre les analyses plus pertinentes et à faciliter la génération du système. Enfin, ce processus doit aussi assurer le déploiement (sélection des composants), la configuration (paramétrisation des composants) et l'intégration des composants applicatifs, intergiciels et utilisateurs (code métier) pour la production automatique de l'application prête à être exécutée.

1.2.1 Modéliser pour l'analyse et la génération de code

Les représentations du système sont élaborées par des formalismes différents et évoluent à des niveaux d'abstraction différents. La sémantique des constructions d'un langage de programmation utilisées pour l'implantation du système définit un niveau d'abstraction trop bas (trop de détails) qui complexifie l'exploitation du code source pour l'analyse des propriétés du système. Les formalismes dédiés à l'analyse d'un système réparti tel les réseaux de Petri (comportement) définissent des abstractions trop élevées pour permettre l'exploitation du modèle d'analyse à des fins de génération de code. Afin de renforcer la cohérence entre les modèles d'analyse et le code généré, il est donc nécessaire de déterminer une représentation intermédiaire contenant à la fois des informations pour l'analyse du système et un niveau de détail suffisant pour une démarche de génération de code efficace.

Notre première problématique est donc : *Comment déterminer un niveau d'abstraction "intermédiaire" limitant les transformations pour (1) diminuer la granularité du modèle et rendre les analyses réalisables et (2) augmenter la granularité du modèle et permettre une démarche de génération de code efficace ?*

© 2012 Gilles LASNIER

1.2.2 Analyser une représentation complète du système

Un second problème consiste à réaliser les analyses à partir d'une description complète de l'application incluant aussi bien les composants applicatifs que les composants de son exécutif. Dans le contexte des systèmes TR^2E critiques, les composants de l'exécutif correspondent aux composants d'un intergiciel.

L'introduction des intergiciels et des standards de répartition a simplifié la conception des applications réparties. Le développeur n'a pas à se charger des aspects dits *bas-niveau*, notamment liés à la communication (réception des tampons de communication, extraction des données...) et dépendants de la plate-forme d'exécution de l'application. Des routines dites de plus *haut-niveau* pour l'envoi et la réception des données lui sont fournies et masquent la complexité de la mise en œuvre de la répartition.

La spécification des composants intergiciels dès les premières étapes de la modélisation va ainsi à l'encontre de la philosophie des modèles où l'abstraction des détails d'implantation est prépondérante lors des premières étapes d'analyse. Cependant, les ressources déclarées par l'intergiciel ont un impact sur l'architecture logicielle et l'implantation de l'application (gestion des ressources de communication, donnée partagée...). Par conséquent, il est nécessaire de prendre en compte ces ressources lors des différentes analyses (faisabilité, performance...) réalisées sur un modèle final de l'application.

Notre seconde problématique est donc la suivante : *Comment modéliser les composants intergiciels et à quel moment les intégrer au modèle initial pour une analyse complète du système ?*

1.2.3 Processus de réalisation automatique et complexité

Les processus de réalisation de systèmes critiques basés sur une approche IDM automatisent les étapes d'analyse, de déploiement, de configuration et d'intégration des composants constituant l'application.

Le déploiement revient à sélectionner et placer explicitement les différents constituants de l'application sur leurs emplacements physiques respectifs et préparer leur exécution (par exemple, le placement des entités responsables de l'envoi ou de la réception des messages sur un nœud). La configuration consiste à personnaliser ces composants sélectionnés en fonction des besoins et des propriétés de l'application (par exemple, l'ouverture des canaux de communication requis par un nœud donné). Enfin, l'intégration est la dernière phase de la réalisation permettant l'assemblage des différents composants (applicatifs, intergiciels et utilisateurs) qui forment l'application. Celle-ci s'effectue lors de la compilation de l'application.

Cependant, les composants intergiciels sont implantés dans le langage de programmation cible et nécessitent le développement et la maintenance d'un intergiciel par générateur de code. Ceci implique un processus de génération automatique complexe, effectuant plusieurs transformations et introduisant des mécanismes supplémentaires pour assurer l'assemblage correct des composants applicatifs et des composants intergiciels. De plus, ce processus automatique peut être différent en fonction du langage de programmation cible.

La seconde problématique que nous avons présentée suggère que les étapes de déploiement et de configuration soient réalisées lors de la phase de modélisation de l'application complète pour que les analyses soient effectuées sur une représentation cohérente avec l'implantation physique.

Ceci limite donc la difficulté de maintenir plusieurs versions d'intergiciel et permet la simplification du processus de génération automatique en autorisant la production de l'ensemble

des composants applicatifs et intergiciels. En revanche, nous devons être capables d'adresser différents langages de programmation cible.

Notre troisième problématique est donc : *Comment produire automatiquement le code de l'application en limitant le plus possible les différences entre les générateurs ?*

1.3 Contraintes

Nous présentons, dans cette section, les contraintes pour la réalisation des systèmes critiques selon une démarche basée sur l'IDM.

1.3.1 Eléments de modélisation parasites

Les formalismes de description introduisent de nombreuses constructions facilitant la tâche de modélisation aux développeurs. Il s'agit de mécanismes pour l'abstraction, la réutilisation, le raffinement et le paramétrage des composants de l'application. Par exemple, les composants abstraits permettent de modéliser un composant ou un assemblage de composants avec un très fort niveau d'abstraction. Ces composants sont utilisés lors des premières étapes de la modélisation et subissent un raffinement lors de l'évolution du modèle du système. Ce raffinement précise le rôle, le déploiement et la configuration du composant abstrait. Par exemple, un composant abstrait modélisant un calculateur de pression peut être raffiné en un composant matériel (par exemple, un périphérique d'entrée/sortie) ou en un assemblage de composants logiciels et matériels (un capteur et des entités actives pour traiter les valeurs reçues du capteur).

Si ces éléments ont des avantages pour les premières étapes de la modélisation, les résidus de ces mécanismes laissent des informations qui peuvent parasiter ou complexifier les analyses effectuées sur le modèle ou les générateurs de code. L'implantation de ces outils doit ainsi supporter une phase préliminaire pour résoudre ou supprimer ces mécanismes avant d'extraire l'information pertinente.

1.3.2 Mise en œuvre des systèmes critiques

Nous avons vu que les domaines d'utilisation des systèmes critiques nécessitent la vérification et la validation de ces systèmes pour assurer le fonctionnement correct de l'application. La mise en œuvre des systèmes critiques impose ainsi des règles rigoureuses pour leur développement.

Modèle de concurrence analysable

Dans un système TR^{2}E, les entités applicatives interagissent selon un modèle de concurrence. Le critère critique de ces systèmes implique que ce modèle de concurrence soit analysable afin de garantir au minimum l'analyse d'ordonnançabilité du système (que l'ensemble des tâches qui s'exécutent sur un nœud rendent leur calcul à l'échéance programmée) et l'absence d'interblocage. Ces analyses sont la base pour la vérification des systèmes critiques.

Nous devons ainsi être capables d'exprimer précisément ce modèle de concurrence lors de l'élaboration des représentations du système de façon à renforcer l'analysabilité du système et la cohérence entre les modèles et l'implantation.

© 2012 Gilles LASNIER

Restrictions architecturales et comportementales

La mise en œuvre des systèmes critiques nécessite beaucoup plus d'attention (sûreté de fonctionnement, sécurité...) que pour le développement d'un système classique. L'implantation du système doit ainsi respecter des restrictions architecturales et comportementales pour éviter l'utilisation de constructions dangereuses pour le bon fonctionnement de l'application. De manière générale, les constructions compromettant l'analyse du système sont interdites.

Ces restrictions proviennent des différents aspects du système. L'aspect temps-réel impose notamment que l'exécution du système soit prédictible et déterministe. L'aspect embarqué impose une gestion et un contrôle des allocations mémoire rigoureux (par exemple, l'interdiction d'allocation dynamique qui peut être source d'indéterminisme). L'aspect répartition repose bien souvent sur des patrons de conception (usines à objets, aiguillage dynamique) introduisant de l'indéterminisme dans le comportement de l'application qui peut compromettre les analyses temporelles.

Le respect de ces restrictions est important pour la réalisation d'un système critique sûr. Pour notre démarche, nous devons être capables d'exprimer et de vérifier celles-ci sur la représentation du système de manière à renforcer le processus d'analyse et de production et à réduire les différences entre le modèle et le code généré.

1.3.3 Synthèse

Les contraintes que nous venons d'énoncer rendent la résolution des problématiques que nous avons présentées (plus haut) plus difficile. Ainsi, nous devons tenir compte des constructions potentiellement dangereuses pour l'analyse et l'implantation et les interdire dès la modélisation de l'application. Pour ce faire, nous devons être capables d'exprimer et de vérifier des contraintes précises sur le modèle.

Enfin, nous devons être en mesure de déterminer et d'éliminer les éléments de modélisation non pertinents pour l'analyse et la génération.

Dans la suite, nous exposons les objectifs de ce travail de thèse (section 1.4) et l'approche que nous avons définie pour atteindre nos objectifs (section 1.5).

1.4 Objectifs

D'un point de vue général, nous souhaitons renforcer la cohérence entre les modèles d'analyse et le code généré pour fiabiliser le processus de réalisation automatisé des systèmes TR^{2}E critiques. Pour ce faire, notre objectif principal vise à réduire la différence sémantique et comportementale entre les modèles d'analyse et le code généré. Notamment, nous souhaitons que les analyses finales du système porte sur une description architecturale et comportementale des composants applicatifs et intergiciels constituant l'application et représentative de son implantation. Cette même description devra permettre de simplifier le processus de génération automatique du système à partir de transformations modèle-à-code simples.

1.4.1 Niveau d'abstraction pour l'analyse et la génération

Nous devons élaborer un modèle représentatif de l'implantation du système pour l'analyser et limiter les transformations effectuées à partir de ce modèle pour la production du code

© 2012 Gilles LASNIER

source de l'application. Ainsi, plus le modèle et le code généré présenteront des équivalences au niveau sémantique et comportemental, moins il sera nécessaire d'effectuer des transformations et plus la cohérence sera forte.

Notre premier objectif est donc de définir un niveau d'abstraction intermédiaire permettant à la fois la description détaillée de l'architecture de l'application (composants matériels, logiciels, interaction), les informations fonctionnelles des composants (comportement...) et les informations non fonctionnelles (temporelles, structurelles, comportementales...) pour l'analyse.

Par conséquent, nous devons employer un formalisme de description offrant une syntaxe précise et assez détaillée qui permet d'effectuer des analyses sur le modèle et aussi de générer du code automatiquement. Le langage AADL (*Architecture Analaysis & Design Language*) et son annexe comportementale sont les formalismes que nous avons retenus. Pour des raisons que nous exposerons (chapitre 4 et 5), ce couple nous semble être un choix pertinent pour atteindre nos objectifs. Ceci permet de résoudre la problématique 1.2.1 et d'adresser les contraintes énoncées dans 1.3.1 et 1.3.2. L'objectif 1.4.2 (suivant) pourra être réalisé à partir de celui-ci.

1.4.2 Modélisation d'une architecture d'intergiciel critique

Nous souhaitons modéliser et intégrer les ressources intergicielles de manière à effectuer des analyses sur une description complète du système. Dans le cadre d'un système critique, l'intergiciel doit être de petite taille pour satisfaire au caractère embarqué du système. Ainsi, seuls les composants intergiciels requis par l'application doivent être intégrés (pas d'usine à services) et nous devons pouvoir les configurer et les déployer en fonction des besoins et des propriétés de l'application.

Par conséquent, notre second objectif vise la définition d'une bibliothèque de composants intergiciels paramétrables modélisant une architecture et le comportement d'un intergiciel *dédié* aux systèmes critiques. Ceci permet de résoudre la problématique 1.2.2 et d'adresser les contraintes énoncées dans 1.3.2.

1.4.3 Processus de réalisation automatisé

A partir des objectifs 1.4.2 et 1.4.1, nous souhaitons produire un modèle décrivant l'application complète (applicatif et exécutif). Ce modèle intermédiaire effectuera la liaison entre la description architecturale initiale d'un système et son implantation (le code généré). Notre troisième objectif est de trouver un processus assurant la transformation de la description architecturale initiale (applicatif) vers notre modèle complet (applicatif et exécutif), son analyse et la transformation de ce dernier vers le code généré. Ce processus devra être flexible et permettre de faciliter la traçabilité, la maintenance et l'évolution de notre approche. Ceci permet de résoudre les problématiques 1.2.1 et 1.2.3 et d'adresser les contraintes présentées dans 1.3.1 et 1.3.2.

1.5 Approche

Pour satisfaire les objectifs présentés à la section précédente, nous proposons une nouvelle approche pour la réalisation des systèmes critiques basée sur le raffinement d'une description architecturale AADL du système vers une description AADL architecturale et compor-

© 2012 Gilles LASNIER

tementale constituée des composants applicatifs et intergiciels avec une sémantique proche de l'implantation du système. Ce dernier servira de modèle pivot pour l'élaboration de modèles d'analyse (dans un formalisme tiers d'un outil d'analyse), et permettra la production du code source de l'application par une transformation de modèles vers un langage de programmation impératif. Notre approche s'articule autour de quatre contributions.

1.5.1 Profil pour la modélisation architecturale et comportementale

Le langage AADL offre des éléments avec un niveau de détail élevé pour modéliser les systèmes TR^{2}E. Les modèles sont à la fois précis (utilité pour la génération de code) et analysables. Son annexe comportementale permet la spécification explicite du comportement des composants architecturaux du modèle renforçant ainsi les patrons d'analyse et de génération.

Cependant, le langage AADL vise le développement des systèmes du domaine TR^{2}E en général. Pour la réalisation des systèmes critiques, nous devons restreindre son périmètre d'activité pour permettre l'analyse (statique et efficace) du système modélisé et la production d'un code valide, avec un niveau de confiance élevé.

Ainsi, nous proposons la définition d'un sous-ensemble du langage AADL et des éléments de son annexe comportementale, et de contraindre ces composants à partir de restrictions spécifiques pour garantir l'analyse et la réalisation sûre des systèmes critiques. Ce sous-ensemble d'éléments définit notre profil de modélisation pour l'analyse et la génération, et fixe le niveau d'abstraction du modèle complet.

1.5.2 Modélisation de l'intergiciel

Pour chaque application TR^{2}E, nous visons à intégrer les composants intergiciels qui lui sont "dédiés". Le concept d'intergiciel dédié a été introduit par [Zalila *et al.*, 2008] pour la définition de l'intergiciel PolyORB-HI-Ada supportant les composants applicatifs du langage AADL et personnalisés selon les besoins et les propriétés de l'application. Cet intergiciel vise précisément le développement des systèmes TR^{2}E critiques. Ainsi, nous nous basons sur cette implantation et sur le profil de modélisation présenté ci-dessus pour la définition d'une bibliothèque de composants intergiciels paramétrables. Une partie de ces composants indépendants des propriétés de l'application sera pré-existante, l'autre sera générée et configurée à partir de l'analyse des propriétés de l'application. Par conséquent, l'intégration des ressources intergicielles ne se fera plus sous la forme de code mais sera intégrée à la modélisation du système dès les phases d'analyse.

1.5.3 Processus de raffinement incrémental

Notre troisième contribution est la définition d'une chaîne de transformation implantant un raffinement incrémental de la description architecturale initiale de l'application (sa partie applicative) vers un modèle décrivant sa partie applicative et sa partie exécutive, à l'aide de notre profil de modélisation. Notamment, ce processus de raffinement permettra d'éliminer les éléments non concrets pour l'analyse et la génération (mécanisme d'extension..), de transformer les composants applicatifs avec un niveau d'abstraction élevé (sémantique ou comportement implicite...) vers un composant ou un assemblage de composants avec un niveau de détails élevé (niveau d'abstraction réduit), et d'intégrer les composants intergiciels de notre bibliothèque sélectionnés, déployés et configurés.

© 2012 Gilles Lasnier

Le modèle raffiné obtenu servira à la fois pour les analyses et la génération de code et facilitera le développement des outils. L'utilisation des techniques basées sur la transformation de modèles permettra d'expliciter et d'implanter les règles de transformation du processus de raffinement augmentant la modularité et la maintenabilité de l'outil supportant la chaîne de transformation.

1.5.4 Génération de code par transformation de modèles

Notre dernière contribution est un processus de transformation permettant la génération du code source de l'application à partir de notre modèle raffiné. Dans le contexte des systèmes critiques, nous commencerons par la génération vers le langage Ada dont le compilateur peut renforcer plusieurs restrictions que doivent subir ces systèmes (par exemple, en vérifiant les restrictions de son profil Ravenscar [Burns *et al.*, 2004]). L'approche que nous avons adoptée nous permettra d'effectuer la génération de code de manière automatique à partir d'une transformation de modèles simple. En effet, le processus de raffinement (présenté ci-dessus) se sera chargé de raffiner le niveau d'abstraction du modèle initial de manière à maximiser les équivalences sémantiques et comportementales avec les constructions d'un langage de programmation impératif. Notre approche de génération généraliste nous permettra par la suite d'étendre notre approche vers d'autres langages de programmation impératifs comme le langage C.

L'utilisation des techniques de transformation de modèles permettra d'expliciter les règles de traduction, souvent enfouies dans un compilateur traditionnel et rendra le générateur plus flexible. Sa maintenance et son évolution seront ainsi facilitées.

1.6 Organisation du mémoire

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

La partie II est consacrée à l'étude théorique des besoins et des solutions existantes dédiés à la réalisation des systèmes TR^{2}E critiques selon une démarche basée sur l'IDM. Cette partie se divise en trois chapitres. Au cours du chapitre 2, nous étudions les solutions existantes traitant les thématiques de notre travail de thèse. Nous présentons d'abord les processus d'analyse et de production des systèmes critiques existants et nous discutons des avantages et des limites des technologies rencontrées. Puis, nous nous intéressons aux concepts et aux principaux outils de transformation de modèles de manière à déterminer les solutions pour l'élaboration de transformations expressives, maintenables et évolutives. Au cours du chapitre 3, nous nous intéressons plus en détail aux problèmes spécifiques des processus d'analyse et de génération, ce qui nous permettra de développer les objectifs auxquels nos contributions répondent et l'approche que nous proposons. Le chapitre 4 est une présentation du langage de modélisation AADLv2 et de son annexe comportementale, formalismes retenus dans notre approche. Ensuite, nous présentons les restrictions que nous avons définies pour décrire les applications TR^{2}E critiques à l'aide de ces langages.

La partie III, composée de quatre chapitres, présente en détail les contributions qui permettent de mettre en œuvre notre approche. Le chapitre 5 présente notre profil de modélisa-

© 2012 Gilles LASNIER

tion architecturale et comportementale pour l'analyse et la génération. Le chapitre 6 présente notre bibliothèque de composants intergiciels réalisés à partir de l'implantation d'un intergiciel dédié aux systèmes critiques. Le chapitre 7 décrit le processus de raffinement pour la production de notre modèle AADL spécifiant la partie applicative et la partie exécutive de l'application avec une sémantique proche d'un langage de programmation impératif. Enfin, nous présentons, dans le chapitre 8, les règles de transformation pour la génération de code Ada (respectant le profil Ravenscar) à partir des composants AADL.

Au cours de la partie IV, composée de deux chapitres, nous présentons (chapitre 9) la mise en œuvre de notre approche par la réalisation d'une suite d'outils qui automatise l'analyse, le processus de raffinement, la validation et le processus de génération. Puis, le chapitre 10 présente les expérimentations que nous avons effectuées grâce à ces contributions, ainsi que les résultats obtenus.

Enfin, nous concluons l'ensemble de cette étude (chapitre 11) dans la partie V et nous présentons les perspectives possibles pour l'extension de ces travaux de recherche.

© 2012 Gilles LASNIER

© 2012 Gilles LASNIER

Deuxième partie

État de l'Art et Étude Théorique

Chapitre 2

État de l'Art et Positionnement

SOMMAIRE

2.1 Introduction

Les systèmes TR²E retiennent de plus en plus l'attention du fait de leur introduction massive dans les domaines de notre quotidien, que ce soit dans nos loisirs, pour notre sécurité ou dans les transports (baladeurs mp3, géolocalisation mobile, voiture, contrôleur de vitesse d'un train, etc).

L'augmentation de leurs domaines d'application, de la complexité des architectures logicielles proposées, des critères de qualité de service attendue mais surtout du niveau de sécurité et de sûreté de fonctionnement requis pour la fiabilité du système, définit un ensemble

de besoins qui justifient un intérêt tant auprès des chercheurs que des industriels. Ainsi, de nombreux travaux de recherche ont porté sur l'amélioration des méthodes de conception des systèmes TR^{2}E critiques et, plus particulièrement, se sont intéressés à l'automatisation de leur analyse, de leur production et de leur déploiement, ceci afin d'augmenter la productivité des équipes de développement et d'assurer la fiabilité du système, notamment en limitant les interventions humaines lors des étapes *sensibles* du développement logiciel.

L'objectif principal de ces travaux vise la séparation des préoccupations et la décomposition claire des exigences fonctionnelles - *i.e.* ce que réalise le système - et non-fonctionnelles- *i.e.* de quelle manière il le réalise - de l'application. Ainsi, la partie applicative du système est développée indépendamment de la plate-forme cible sur laquelle il s'exécute.

Pour ce faire, des techniques de conception logicielle ont été développées tels les *design patterns*, les profils architecturaux, les preuves de code, les tests définissant la construction d'application déterministe et *correcte-par-construction*. Les intergiciels ont été conçus pour faciliter la mise en œuvre de cette séparation et définissent une couche intermédiaire d'adaptation entre le code applicatif et le code spécifique au support d'exécution.

L'introduction de l'approche par composants a permis d'accroître la décomposition fonctionnelle en offrant une séparation claire entre spécification et implantation des composants. La conception est ainsi facilitée par la construction d'un assemblage de composants préexistants, déployés, configurés et réalisant un sous-ensemble des fonctionnalités du système.

Les descriptions architecturales établies à partir des langages de descriptions d'architectures (ADLs) ont permis d'automatiser les étapes de déploiement et de configuration des composants. Pour améliorer cette solution, les techniques induites par l'*Ingénierie Dirigée par les Modèles* (IDM) ont permis l'automatisation des étapes d'analyse, de la validation du système et de la production du code source des composants.

De nombreuses approches visent à combiner ces solutions pour la production de systèmes TR^{2}E déterministes, *correcte-par-construction* et fiables. De nouveaux processus de conception, d'analyse et d'implantation sont ainsi proposés utilisant des standards, définissant des règles de conception et de développement rigoureuses, proposant la spécification et la vérification de contraintes de réalisation, utilisant des procédures de test exhaustives et plus récemment introduisant de nouvelles méthodes d'analyse basées sur des méthodes formelles.

Toutefois, cette combinaison n'est pas triviale et se heurte à différentes limites que nous abordons dans ce chapitre.

Ce chapitre présente les éléments majeurs requis pour la conception, l'implantation et la validation de systèmes TR^{2}E critiques. Nous présentons et évaluons différentes approches qui ont abordé une ou plusieurs problématiques décrites précédemment. Puis, nous discutons des limites de ces approches vis-à-vis de notre problématique générale. Enfin, nous présentons les grandes lignes de nos travaux de recherche.

2.2 Conception logicielle des systèmes TR^{2}E critiques

Cette section présente différentes techniques pour la conception, l'analyse et l'implantation des architectures logicielles d'un système TR^{2}E critique selon une approche dirigée par les modèles et assistée par ordinateur.

La modélisation d'une application permet de raisonner *à priori* sur celle-ci - *i.e.* avant son implantation. Le "modèle" sert de point d'entrée pour les analyses architecturales et comportementales et la validation du système vis-à-vis de sa spécification. Enfin, il permet la génération

© 2012 Gilles LASNIER

du code source des composants logiciels modélisés et automatise leur configuration et leur déploiement.

2.2.1 Description des systèmes critiques

Différentes approches pour la description architecturale des systèmes distribués ont déjà été présentées dans des travaux antérieurs de notre équipe [Zalila, 2008; Borde, 2009]. Parmi les différentes techniques pour la modélisation de système, nous nous intéressons, dans cet état de l'art, essentiellement à celles dédiées aux systèmes TR²E critiques.

Langages de description d'architecture

Les Langages de Description d'Architecture (ADLs) ont pour objectif de décrire un système sous une forme structurée par assemblage de composants logiciels et matériels ainsi que leurs interactions. Ils s'appuient sur trois concepts fondamentaux [Medvidovic and Taylor, 2000] :

- le composant, qui représente une unité indépendante de calcul, de stockage de données ou d'encapsulation ;
- le connecteur, qui définit explicitement l'interaction et la sémantique de l'interaction entre un ou plusieurs composants ;
- la configuration, qui définit les configurations structurelles (ensemble de composants et de connecteurs d'une architecture propre à une application) et des configurations comportementales (évolution des liens, des communications et des propriétés non fonctionnelles entre composants et connecteurs).

Nous pouvons distinguer différents types d'ADLs en fonction des objectifs qu'ils visent :

- les ADLs génériques, pour la conception des systèmes logiciels en général ;
- les ADLs formels, pour l'analyse des systèmes ;
- les ADLs pour la conception d'un système spécifique à un domaine.

Bien que les ADLs formels (ACME [Garlan *et al.*, 1997], RAPIDE [Luckham *et al.*, 1995], WRIGHT [Allen, 1997] soient dédiés à l'analyse des systèmes, la plupart d'entre eux expriment des limites communes pour la configuration de l'application à partir de leur description et supportent difficilement une démarche de production automatique du code de l'application. Dans le cadre de nos travaux, nous nous intéressons uniquement aux ADLs spécifiques à un domaine comme le langage AADL dans sa première version [SAE Aerospace, 2004].

AADL 1.0. Le langage AADL (Architecture, Analysis and Design Language) [SAE Aerospace, 2004], dans sa première version, a été développé pour décrire l'architecture logicielle et matérielle d'un système TR²E au sein d'une unique représentation - *i.e.* un même modèle. Les motivations principales du langage sont l'analyse du système et la représentation du déploiement de l'architecture logicielle sur la plate-forme matérielle. Les travaux de [Zalila, 2008] étendent le périmètre d'activité du langage AADL en proposant un processus de conception autorisant la production automatique du code source des composants logiciels de l'application à partir de leur description architecturale en AADL 1.0.

Discussion. Le langage AADL dans sa première version fait partie de la catégorie des ADLs et est particulièrement adapté à la conception des systèmes TR²E critiques. Le modèle architectural final ne contient que des composants concrets modélisant des composants tels qu'ils apparaissent physiquement dans le système final. Un mécanisme efficace de propriétés

© 2012 Gilles LASNIER

permet de spécifier pour chaque composant un ensemble d'exigences fonctionnelles ou non fonctionnelles notamment liées au caractère temps-réel, distribué et critique de l'application. Ainsi, il est possible d'automatiser la configuration et le déploiement des composants à partir de l'analyse d'un modèle AADL.

Langages de modélisation spécifique à un domaine

Aujourd'hui, l'ingénierie du logiciel s'oriente vers l'*Ingénierie Dirigiée par les Modèles* (IDM) et le principe du "tout est modèle". L'IDM vise de manière plus radicale que pouvaient l'être les approches de *design-patterns* [Gamma *et al.*, 1995] et des *aspects* [Kiczales and Hilsdale, 2001], à fournir des modèles complémentaires exprimant séparément une préoccupation (point de vue sur le système, étape de modélisation, etc) [Combemale, 2008].

Si la notion de *modèle* est le concept central de l'IDM, celle-ci introduit la méta-modélisation qui consiste à définir un ensemble de règles et de concepts visant la modélisation d'un ensemble de problèmes déterminés voir même spécialisés. L'IDM favorise ainsi la définition de langages de modélisation dédiés à un domaine particulier (*Domain Specific Modeling Languages* - DSML), offrant ainsi aux utilisateurs des concepts propres à leur métier.

L'approche MDA. L'OMG a adopté une approche de modélisation et a défini le MDA (*Model Driven Architecture* [Soley, 2000; OMG, 2003b]) pour promouvoir les bonnes pratiques de modélisation, exploiter les avantages des modèles en terme de pérennité, de productivité, d'interopérabilité et de prendre en compte certaines caractéristiques des plate-formes d'exécution. Pour cela, le MDA s'appuie sur plusieurs standards UML [OMG, 2007c; OMG, 2007d], MOF [OMG, 2006c] et XMI [OMG, 2007b]. L'objectif majeur du MDA est l'élaboration de modèles indépendants des détails techniques des plate-formes d'exécution (*Platform Independent Model* - PIM) et de générer de manière automatique la totalité des modèles de code (*Platform Specific Model* - PSM) par transformation de modèle [Blanc, 2005; Kleppe *et al.*, 2003].

Le profil UML/MARTE. Dans le contexte des systèmes temps-réel, le profil UML/MARTE (*Modeling and Analysis of Real-Time and Embedded systems* [OMG, 2007e]) a été défini et standardisé pour modéliser et analyser les systèmes TR^{2}E à l'aide du langage UML.

Définition 2.1 (Profil UML [OMG, 2007c]) *Un profil est un moyen d'extension d'un méta-modèle permettant de spécialiser une méta-classe pour répondre à des besoins de modélisation spécifiques.*

Selon la définition de la notion de profil UML donnée ci-dessus, MARTE spécialise UML et introduit les concepts de bases issus du domaine des systèmes TR^{2}E. Il favorise l'interopérabilité entre les différents outils utilisés pour la spécification, la conception, la vérification et la génération de code du système. Notamment, MARTE permet la modélisation des propriétés non fonctionnelles propres aux systèmes TR^{2}E. Enfin, MARTE introduit un modèle d'allocation des composants logiciels du système sur les composants matériels de la plate-forme d'exécution tout en respectant les contraintes temporelles et topologiques de l'application.

© 2012 Gilles Lasnier

AADLv2. Récemment, une nouvelle version du langage AADL a été standardisée [SAE Aerospace, 2009b] et de nombreuses améliorations ont ainsi été apportées à savoir : l'introduction de nouveaux composants architecturaux, des précisions sur la sémantique des composants, une syntaxe graphique normalisée, un méta-modèle et diverses annexes (modélisation des données, comportementale...) permettant d'enrichir la modélisation et la qualité des outils associés.

Si le langage AADL 1.0 présenté prédécemment fait partie de la catégorie des ADLs, nous pouvons classifier le langage AADLv2 dans la catégorie des DSMLs considérant les caractéristiques propres à ces langages et les nombreuses améliorations citées ci-dessus et détaillées dans le chapitre 4 de ce manuscrit.

Discussion. Combinés à l'utilisation des techniques de modélisation, les DSMLs permettent d'élaborer des outils de qualité (représentation graphique, méta-modèle, édition de modèles...) qui facilitent particulièrement la représentation visuelle et la compréhension de systèmes complexes. De plus, l'utilisation des techniques liées à la transformation de modèle (présentées section 2.5) amène des perspectives intéressantes en termes d'expressivité, de traçabilité des exigences et d'automatisation.

A la différence des ADLs présentés précédemment, l'un des avantages d'UML/MARTE est la modélisation graphique de l'application TR²E mais celle-ci n'apporte que peu de bénéfices concernant la sémantique des concepts modélisés. Des travaux récents [Salazar, 2010; Vidal *et al.*, 2009] ont montré que le profil MARTE souffre des limites récurrentes du langage UML notamment pour le support d'une démarche de génération. La sémantique des concepts définis en UML et MARTE est trop riche et il est ainsi possible de modéliser un même concept de différentes manières, augmentant la complexité des patrons de génération et rendant la réutilisation des outils assez complexe. Enfin, plus généralement, les approches basées sur UML utilisent des syntaxes différentes pour décrire chacun des aspects d'une application, nécessitant la consolidation de toutes les représentations si le modèle de l'application est modifié ce qui s'avère couteux en terme de développement (ressources et temps).

Les récentes améliorations du langage AADLv2 et de nombreux travaux ont montré les multiples emplois possibles du langage AADL pour la modélisation, l'analyse et la conception de systèmes TR²E critiques. Outre le fait que le langage AADL a été développé précisément pour les systèmes TR²E et non pas adapté, l'avantage majeur du langage est la spécification architecturale et comportementale des composants logiciels et matériels de l'application au sein d'une unique représentation -*ie.e* un unique modèle.

De plus, des travaux ont montré l'intérêt de la communauté UML pour AADL et ont abouti à la définition d'un profil (intégré à MARTE) permettant le passage d'une spécification MARTE vers une description architecturale AADL [Faugere *et al.*, 2007] afin de profiter des nombreux outils basés sur le langage AADL.

Ces éléments nous amènent à penser que le langage AADL est un candidat plus approprié pour la modélisation, l'analyse et la conception de systèmes critiques selon un processus automatisé, ce qui fait l'objet de ce manuscrit. Intéressons-nous maintenant à l'utilisation faite des modèles à des fins d'analyse et de génération du code source du système.

2.2.2 Modèles d'analyses, vérification et validation

Dans notre introduction générale, nous avons vu la nécessité pour un système TR²E critique de satisfaire des exigences fortes en termes de sécurité et de sûreté de fonctionnement.

© 2012 Gilles LASNIER

Différents modèles d'un système sont ainsi réalisés dans le but de couvrir et de permettre la vérification d'un aspect du système. Ainsi, de nombreux techniques et outils ont été développés afin d'établir des modèles d'analyse pour la vérfication et la validation du système vis-à-vis de ces spécifications.

Dans les sections suivantes, nous présentons les analyses les plus communes effectuées dans le contexte des systèmes critiques à savoir l'analyse d'ordonnançabilité et le *model checking*. Ces analyses permettent de conclure sur la faisabilité du système et de vérifier certaines propriétés comportementales. Leur objectif est d'augmenter *la confiance que l'on peut avoir dans ce système*.

Il est important de noter que la vérification du système s'effectue uniquement sur le système modélisé et non sur le système exécuté. Il s'agit d'un complément aux techniques de tests, permettant de vérifier certaines propriétés de manière exhaustive (comportement...) et d'effectuer des vérifications en amont du cycle de développement logiciel.

Ordonnançabilité

L'analyse d'ordonnancement d'un système vise à garantir que l'ensemble des tâches réalisant les fonctions du système finiront toujours leur exécution dans une échéance donnée. Pour ce faire, il est nécessaire de modéliser les tâches s'exécutant sur le processeur et de tenir compte des propriétés fonctionnelles et non fonctionnelles de celles-ci. Ainsi, une tâche doit être caractérisée par son comportement (périodique, sporadique ou apériodique), son pire temps d'exécution (WCET), sa période et/ou son échance (déterminée par son comportement) et une politique d'ordonnancement (RMS, EDF...) pour l'ensemble des tâches.

Divers travaux comme RMS ou EDF [Liu and Layland, 1973] proposent des algorithmes dont l'implantation a permis l'automatisation de l'analyse d'ordonnançabilité d'un système. Ces outils extraient les informations associées aux tâches décrites à l'aide d'un ADL ou d'un DSML.

CHEDDAR [Singhoff *et al.*, 2004] et MAST [Universidad de Cantabria, 2010] sont des outils d'analyse d'ordonnancement dédiés aux systèmes TR^{2}E. Ils permettent aussi de simuler cet ordonnancement.

Model checking

Les techniques de *model checking* consistent à modéliser et à analyser le comportement d'un système afin de valider une propriété donnée. Les propriétés standards généralement exprimées sont l'accessibilité, la sûreté, la vivacité, l'équité et l'absence de blocage. Il est aussi possible de vérifier des propriétés plus spécifiques sous réserve que celles-ci soient exprimables dans une logique compatible avec les algorithmes et les outils de vérification.

Les formalismes les plus utilisés pour représenter et analyser les systèmes TR^{2}E critiques sont les réseaux de Petri colorés (analyse des flots de messages dans le système) ; les réseaux de Petri temporisés [Gorrieri and Siliprandi, 1994] et les automates temporisés [Alur and Dill, 1994] prenant en compte les caractéristiques temporelles de l'application (ordonnancement, dimenssionnement de ressources).

Les outils CPN-AMI [LIP6 - MoVe, 2012], TINA [LAAS, 2012] et UPPAAL [Uppsala University - Aalborg University, 2012] réalisent l'analyse du comportement du système TR^{2}E modélisé respectivement en réseaux de Petri colorés, temporisés et en automates temporisés.

© 2012 Gilles LASNIER

Discussion

Dans le cas d'ensembles de tâches périodiques, les tests de faisabilité sont suffisants pour analyser et valider l'ordonnancement du système. Cependant, ceux-ci s'avèrent limités dans le cadre de la vérification de système TR²E critiques mettant en jeu des mécanismes de synchronisation et de communication complexes (ressources partagées et protégées...).

Les méthodes formelles et les techniques de *model checking* complètent ces analyses pour garantir la faisabilité du système notamment pour la vérification de propriétés de sûreté spécifiques (interblocage, famine...).

Les techniques et les outils que nous venons de présenter permettent de vérifier et de valider l'ordonnancement et le comportement attendu d'un système TR²E critique modélisé. Dans notre contexte, nous n'entendons pas développer des outils d'analyses basés sur ces techniques, mais plutôt, dans le cas, possible réutiliser les outils existants et utilisant notamment le formalisme AADL comme (CHEDDAR pour l'ordonnancement et TINA ou CPN-AMI pour le *model checking* à partir de réseaux de Petri [Renault, 2009]).

Après ces étapes de vérification et de validation du système, nous nous intéressons maintenant à l'implantation du système et plus précisément aux étapes de déploiement et de configuration des systèmes critiques qui, combinés à une démarche de génération de code, autorisent la production automatique du code source de l'application.

2.2.3 Génération de code

La production automatique du code source de l'application est une étape importante de l'automatisation du processus de conception des systèmes TR²E. Outre les gains de productivité, elle augmente la fiabilité (élimination des erreurs introduites manuellement au sein du code, production de code déterministe à partir de patron de génération, etc.) et le niveau de confiance que l'on peut avoir dans le système. De nombreux outils notamment commerciaux tel SCADE [Esterel Technologies, 2012] utilisent ce concept.

Description

Un formalisme source est utilisé pour abstraire les constructions complexes et les difficultés d'implantation dans un langage de programmation (élaboration d'un modèle de code). Le processus de génération transforme alors ces composants abstraits en construction du langage cible (Ada, C, etc). En fonction de l'expressivité du formalisme source et de sa sémantique associée, il est alors possible de produire un squelette (exemple de l'utilisation d'UML) à compléter ou l'implantation complète (exemple de l'utilisation de LUSTRE) de l'application. Les aspects complexes (manipulation de pointeurs, gestion des types de données, etc.) sont ainsi pris en charge par le générateur de code et masqués à l'utilisateur.

Discussion

Si la génération de code possède de nombreuses caractéristiques renforçant l'implantation correcte d'un système (limitation des bugs...) d'un système, elle introduit néanmoins une nouvelle complexité.

La sémantique du formalisme d'entrée doit permettre de décrire efficacement les exigences du système sous peine de nécessiter l'intégration manuelle de code compromettant

© 2012 Gilles LASNIER

la sûreté de l'application. La transposition des problèmatiques d'implantation au modèle requiert une validation du modèle avant de procéder à la génération. De plus, ce modèle doit être analysable afin de vérifier les caractéristiques du système.

La difficulté de certifier l'ensemble des composants logiciels de l'application nécessite l'élaboration d'un processsus rigoureux (standards, documentation, traçabilité, méthodes de tests...) pour assurer un certain niveau de confiance dans le code généré. Enfin, le surplus de code généré à cause de stratégies de génération trop génériques voir inadaptées (nombreux accesseurs, dépendances du code vers des bibliothèques "lourdes"...) et l'intégration de code tiers (utilisateur) sont autant de difficultés à prendre en compte lors de l'élaboration du générateur de code.

2.2.4 Déploiement et configuration automatiques

Pour permettre une démarche de génération de code efficace, il est nécessaire d'automatiser les étapes de déploiement et de configuration des composants logiciels qui constituent l'application. Dans le cadre de ses travaux [Zalila, 2008] a adapté les étapes de déploiement et de configuration décrites par le standard D&C [OMG, 2006b] afin d'accentuer l'aspect statique qui caractérise les systèmes TR^{2}E critiques. Notamment, il donne les définitions suivantes :

Définition 2.2 (Déploiement) *Le déploiement d'une application répartie est la sélection des composants intergiciels requis pour réaliser une sémantique donnée. Le déploiement requiert aussi le placement d'entités qui envoient les messages à travers le réseau à partir des nœuds sources (souches) et d'autres entités qui reçoivent ces messages sur les nœuds de destination (squelettes).*

Définition 2.3 (Configuration) *La configuration d'une application répartie est l'opportunité de paramétrer individuellement chacun des composants intergiciels sélectionnés lors de la phase de déploiement et de les assembler par la suite avec les composants générés automatiquement ainsi que les composants fournis par l'utilisateur.*

Discussion

L'étape du déploiement est une tâche pouvant être source d'erreur. Il est donc nécessaire d'automatiser cette tâche notamment pour tirer bénéfice des descriptions architecturales réalisées lors de la phase de modélisation. Dans le contexte des systèmes critiques, les informations de déploiement doivent être connues à l'initialisation du système pour l'ouverture des canaux de communication. Dans le cadre des systèmes critiques, la phase de configuration est aussi importante. Celle-ci permet de gérer finement et d'allouer uniquement les ressources logicielles nécessaires à l'application et ainsi d'optimiser l'utilisation des ressources matérielles (souvent limitées).

Il existe de nombreux outils pour le déploiement et la configuration des composants logiciels d'un système TR^{2}E (CoSMIC [Gokhale *et al.*, 2003], FractalADL [Consortium, 2012], Ocarina [TELECOM ParisTech, 2012], etc). Dans la section 2.4, nous détaillons ceux visant précisément le contexte des systèmes TR^{2}E critiques.

Nous avons vu, dans cette section, les étapes importantes du processus de conception automatisé des systèmes TR^{2}E critiques. Dans les sections suivantes, nous nous intéressons aux spécificités liées à l'implantation des systèmes critiques et aux processus de conception et d'analyse existants.

© 2012 Gilles Lasnier

2.3 Implantation des systèmes TR²E critiques

Nous pouvons trouver dans la littérature de nombreuses définitions du terme *système critique* [Rushby, 1994], mais intuitivement, nous pouvons qualifier un système TR²E de *critique* si une défaillance, qu'elle soit mécanique, électronique ou logicielle peut provoquer sa perte et engendrer des conséquences importantes voir dramatiques, dont l'impact peut se mesurer en pertes humaines, économiques ou environnementales.

Cette famille de systèmes est généralement présente dans des domaines tels que le spatial, l'aéronautique, le ferroviaire ou le médical nécessitant que le système respecte des exigences fortes en termes de sécurité et de sûreté de fonctionnement [Barnes, 2008].

De nos jours, il est difficile de garantir totalement la sécurité et le bon fonctionnement d'un système dans l'ensemble de ses conditions d'utilisation. Cependant, un certain nombre de méthodologies définissant techniques, règles de conception et outils a été mis en œuvre afin d'éliminer les risques de bugs, de détecter les incohérences et de prévenir des défaillances. Dans la suite, nous présentons certaines de ces méthodologies ou approches utilisées pour la conception et la réalisation des systèmes critiques.

2.3.1 Intergiciel critique

L'intergiciel est un élément cœur dans la conception des systèmes répartis. C'est une couche logicielle *intermédiaire* dont le rôle est d'assurer et de faciliter la communication entre les différents nœuds - *i.e.* applications distribuées - s'exécutant sur une architecture matérielle.

L'intergiciel s'intercale entre le code applicatif et la plate-forme d'exécution. Même si dans la plupart des cas, la plate-forme d'exécution est constituée de l'architecture matérielle et du système d'exploitation, certaines fonctionnalités requises d'un système d'exploitation (concurrence, interaction...) peuvent être implantées par l'intergiciel [Armstrong, 1996]. Par conséquent, celui-ci agit comme une interface en offrant des primitives de plus *haut-niveau*, évitant l'invocation directe des primitives du systèmes d'exploitation ou du matériel améliorant ainsi la portabilité.

De nombreux mécanismes et standards ont été définis afin de normaliser la conception d'applications dans le domaine distribué : de la simple invocation distante de procédures RPC [SUN, 1988], à l'utilisation d'objets répartis CORBA [OMG, 2004], la distribution des données DDS [OMG, 2007a] et même pour l'adaptation de ces mécanismes pour les systèmes temps-réel [OMG, 2005] et embarqués [OMG, 2006a].

Outre la minimisation des coûts de développement (temps, finances), l'intergiciel a pour objectif le renforcement de la "séparation des préoccupations" en satisfaisant l'ensemble des possibilités des exigences non fonctionnelles des systèmes répartis. Ainsi, il devient possible d'adapter la configuration de l'intergiciel en fonction des besoins en ressources intergicielles de l'application distribuée.

Ceci a pose la question fondamentale de savoir si un intergiciel doit satisfaire un caractère "générique" - *i.e.* de répondre à l'ensemble des besoins -, "configurable" - *i.e.* de répondre à un besoin donné - ou "schizophrène" - *i.e.* de répondre simultanément et uniquement à plusieurs besoins.

Dans la suite de cette section, nous présentons brièvement, à travers un exemple de leur implantation, deux stratégies de conception d'intergiciels : "configurable" et "schizophrène"[Quinot, 2003]. Puis, nous discutons de leur utilisation dans le contexte de développement des systèmes critiques.

© 2012 Gilles Lasnier

Intergiciels configurables

Définition 2.4 (Intergiciel configurable [Quinot, 2003]) *Un intergiciel est dit configurable lorsque les composants qu'il intègre peuvent être choisis et leur comportement adapté en fonction des besoins de l'application et des fonctionnalités offertes par l'environnement.*

Dans le contexte des systèmes TR^{2}E critiques, les intergiciels configurables disposent de caractéristiques intéressantes. En effet, la sélection des composants uniquement requis par l'application permet d'optimiser sa taille mémoire et leur personnalisation renforce le respect des exigences.

TAO (The ACE ORB). est un intergiciel développé par l'université de Vanderbilt [Schmidt *et al.*, 1998]. Il s'appuie sur le standard CORBA [OMG, 2004] et a été conçu afin d'assurer la qualité de service de bout-en-bout des systèmes temps-réel répartis en autorisant la configuration de nombreux éléments (caractéristiques des tâches, politique d'ordonnancement, ressources requises pour l'exécution d'une opération, etc). Son architecture extensible autorise l'utilisateur à choisir un contexte (performance, faisabilité...) pour effectuer diverses opérations et assurer le fonctionnement de l'application. Le développement de cet intergiciel a contribué à la définition du standard RT-CORBA [OMG, 2005].

L'utilisation du canevas ACE, reposant sur des patrons de conception classiques des systèmes temps-réel [Schmidt and Cleeland, 2001] permet la configuration des mécanismes de communication (connexions, entités réagissant à la réception d'évènements...) et assure sa portabilité. Enfin, l'utilisation des mécanismes de communication spécifiés au sein du standard CORBA assure par construction l'interopérabilité des applications avec TAO.

Les travaux effectués autour de l'intergiciel TAO [Schmidt *et al.*, 1998; Schmidt and Cleeland, 2001] mettent en évidence l'intérêt de la construction d'un intergiciel dynamiquement configurable par intégration et personnalisation de composants pré-existants réduisant ainsi la complexité et augmentant la maintenabilité des fonctionnalités de base. L'utilisation d'une technologie de répartition reposant sur les patrons de conception simplifie considérablement le développement des applications réparties notamment en terme d'hétérogénéité.

Discussion. Des études montrent des incompatibilités d'utilisation de RT-CORBA pour les systèmes TR^{2}E critiques. En particulier, [Hugues, 2005] révèle la possibilité de modifier dynamiquement la priorité des tâches par l'utilisateur compromettant ainsi l'analysabilité statique d'ordonnancement. De plus, l'architecture orientée objet de TAO limite l'analyse du déterminisme de l'application.

Enfin, la configuration et le déploiement d'une application utilisant TAO s'effectuent de manière dynamique à l'initialisation du système. Si l'utilisation du standard CORBA assure l'interopérabilité, il limite les possibilités d'optimisation de certaines exigences non fonctionnelles comme, par exemple, lors de l'utilisation de mécanismes de communication spécifiques à une application ; ceux-ci ne nécessitant pas tous les mécanismes assurant la compatibilité entre les nœuds (cas des applications distribuées non hétérogènes).

Intergiciels schizophrènes

Définition 2.5 (Intergiciel schizophrène [Quinot, 2003]) *Un intergiciel est dit schizophrène lorsqu'il permet la cohabitation de plusieurs paradigmes de répartition au sein d'une même*

© 2012 Gilles LASNIER

instance et met en place un mécanisme efficace d'interaction entre ces paradigmes (personnalités).

PolyORB. [Quinot, 2003] est un intergiciel développé par AdaCore qui implante l'architecture d'intergiciel définie par [Pautet, 2001]. Cette nouvelle architecture découpe les fonctionnalités de base d'un intergiciel en "services canoniques" assurant l'acheminement correcte d'une requête de l'expéditeur vers le destinataire. Ces services (adressage, liaison, interaction, typage, représentation, protocole, transport, activation et exécution) sont implantés dans un noyau cœur appelé *couche neutre*. Cette implantation est indépendante de tout standard de répartition.

L'architecture de PolyORB est définie en trois couches. Les *personnalités applicatives* correspondent à l'implantation des aspects applicatifs d'un standard de répartition donné (représentation des données, gestions des objets répartis...). Les aspects liés à la communication (protocoles, transport...) sont implantés au sein de *personnalités protocolaires*. *Personnalités applicatives* et *personnalités protocolaires* utilisent et spécialisent les services offerts par la *couche neutre*. Cette architecture "schizophréne" permet ainsi l'utilisation de personnalités différentes et leur coopération au sein d'une même instance d'intergiciel.

Discussion. La configuration de PolyORB s'effectue à l'initialisation du système et s'appuie sur des mécanismes d'aiguillage dynamique (héritage) pour la sélection de l'instance requise pour chaque service. Ceci est principalement dû à l'absence d'un mécanisme pour la configuration statique de l'application à partir d'une analyse fine de ses propriétés.

Pour générer automatiquement et configurer statiquement PolyORB, des travaux ont permis l'utilisation du langage de description d'architecture AADL [Vergnaud, 2006]. Enfin, des travaux plus récents [Zalila *et al.*, 2008] ont raffiné cette architecture pour la définition de l'intergiciel PolyORB-HI répondant aux besoins des systèmes TR²E. Celui-ci vise à fournir une instance d'intergiciel *dédiée* à l'application - *i.e.* seuls les services et les composants intergiciels requis par l'application sont sélectionnés et déployés - avec une empreinte mémoire plus réduite que celle obtenue avec PolyORB.

Conclusion

Nous venons de voir différentes méthodes de conception d'intergiciel pour les systèmes TR²E. L'une de nos contributions vise à proposer une architecture d'intergiciel afin de prendre en compte les composants de l'application complète (applicatif + intergiciel) lors des différentes étapes de vérification et de validation du système modélisé.

Dans le contexte de nos travaux de recherche, nous souhaitons ré-utiliser un intergiciel dédié aux systèmes TR²E. L'intergiciel PolyORB-HI, détaillé dans le chapitre 6 de ce manuscrit, est un candidat approprié car celui-ci répond directement aux spécificités des systèmes critiques. De plus, il supporte les composants logiciels modélisés en AADL, formalisme de description de systèmes critiques que nous avons retenu pour notre approche.

2.3.2 Le profil architectural Ravenscar

Une approche pour l'implantation d'un système critique est la limitation des constructions logicielles pouvant compromettre le respect de certaines exigences. Ainsi, le profil architectural Ravenscar [Burns *et al.*, 2004; Kwon *et al.*, 2002] dédié au langage Ada et Java propose de

© 2012 Gilles Lasnier

restreindre les constructions du langage de programmation utilisé pour l'implantation d'applications monolithiques. Celui-ci n'autorise que les constructions qui garantissent une analyse statique d'ordonnancement, l'absence d'interblocage et une borne temporelle d'inversion de priorités entre les tâches. Cette approche facilite le développement d'entités logicielles déterministes et *correctes-par-construction*. Le respect de l'ensemble de ces restrictions est assuré lors de la phase de compilation.

Description

Concernant le langage Ada, le profil Ravenscar impose que :

- les constructions telles que les *rendez-vous*, les entrées dans les tâches et les entrées multiples d'objets protégés sont interdites car elles rendent complexe l'analyse statique du système (augmentation des mécanismes d'interaction entre les tâches, etc) ;
- les tâches ne doivent jamais finir et toute création de tâche ne peut survenir que lors de la phase d'élaboration (phase d'initialisation de l'application) afin de garantir que l'ensemble de tâches à analyser reste invariant ;
- l'accès aux objets protégés doit être utilisé selon le protocole de plafonnement de priorité PCP (*Priority Ceiling Protol* [Sha *et al.*, 1990]) afin de garantir l'absence d'interblocage et borner l'inversion de priorité ;
- l'utilisation exclusive des tâches périodiques ou sporadiques afin de garantir une analysabilité selon l'analyse par taux d'utilisation du processeur (RMA) ou l'analyse de temps de réponse des tâches (RTA).

Discussion

Le profil Ravenscar vise initialement le développement d'applications monolithiques et les aspects liés à la concurrence. Cependant, dans le cadre de communications asynchrones et de l'utilisation d'un protocole temps-réel entre les nœuds de l'application, celui-ci peut-être étendu pour les systèmes TR^{2}E.

Des restrictions supplémentaires autres que celles sur les aspects de concurrence peuvent aussi être appliquées. L'interdiction de l'utilisation d'instructions d'allocation dynamique de mémoire permet de prévenir des erreurs d'incohérence des pointeurs et d'éliminer des facteurs d'indéterminisme temporel. Cette restriction permet de vérifier statiquement - *i.e.* dès l'étape de compilation du code source - les opérations mémoire, le respect de la taille de la pile, etc. L'interdiction des constructions telles que l'aiguillage dynamique réduit aussi un facteur d'indéterminisme temporel et facilite l'implantation d'outils d'analyse. Enfin, l'interdiction de l'utilisation des nombres à virgules flottantes permet de prévenir d'un certaines erreurs sur les opérations calculatoires et les dépassements mémoire.

Ces restrictions ne sont pas obligatoires mais elles sont régulièrement appliquées au sein de diverses approches et font partie des bonnes recommandations pour la conception et l'analyse des systèmes TR^{2}E critiques. Elles ont, par exemple, été ajoutées au standard Ada 2005 [Working Group, 2005, Annexe H] et sont vérifiées à la compilation (par le compilateur GNAT [Miranda and Schonberg, 2004]). Dans le cadre de nos travaux, nous avons choisi de respecter l'ensemble des restrictions que nous venons de présenter.

Intéressons-nous maintenant aux processus automatisés pour l'analyse et l'implantation de systèmes TR^{2}E critiques existants et s'appuyant sur une démarche IDM.

© 2012 Gilles Lasnier

2.4 Processus d'analyse et d'implantation de systèmes critiques

Dans cette section, nous présentons trois méthodes intégrées pour l'analyse et l'analyse de systèmes critiques. Par méthodes intégrées, nous entendons des approches définissant une méthode, un processus d'ingénierie et fournissant un canevas pour l'analyse et l'implantation (génération, déploiement et configuration) de systèmes critiques.

Ces approches intègrent notamment un ou l'ensemble des éléments présentés dans les sections précédentes. Après leur présentation, nous discutons des avantages et des limites de ces approches. Certaines de ces limitations font l'objet des motivations de ces travaux de recherche et du nouveau processus de conception de systèmes critiques que nous proposons.

2.4.1 HRT-UML/RCM

Description

HRT-UML/RCM (*Hard Real-Time UML for the Ravenscar Computational Model* [Mazzini *et al.*, 2009]) est une méthode intégrée pour le développement d'applications critiques temps-réel dur. Cette méthode a été développée dans le cadre du projet ASSERT [The Assert Project, 2012] et vise à transposer les principes de la méthode HRT-HOOD vers le langage UML selon une approche MDA.

HRT-UML/RCM fournit un canevas pour la conception de systèmes permettant :

- à l'utilisateur de définir des modèles du système indépendants de la plate-forme d'exécution (PIMs) ;
- la production automatique (par transformation) de modèles spécifiques à une plate-forme d'exécution (PSMs) respectant le profil Ravenscar ;
- l'analyse de faisabilité (des contraintes temporelles) et la vérification d'exigences non fonctionnelles sur le système ;
- la production automatique du code source du système à partir des PSMs.

La figure 2.1 décrit le processus de conception proposé par la méthode HRT-UML/RCM.

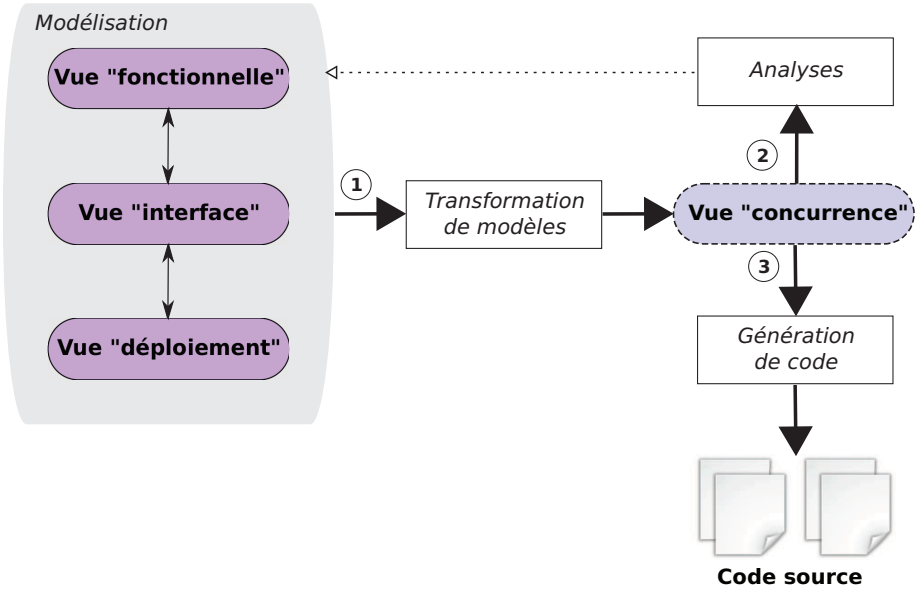


FIGURE 2.1 – Processus de conception de la méthode HRT-UML/RCM

© 2012 Gilles LASNIER

Modélisation

La description architecturale suit les recommandations du standard IEEE 1471 [IEEE, 2000] renforcant la "séparation des préoccupations". Dans ce contexte, HRT-UML/RCM définit trois vues "fonctionnelle", "interface", "déploiement", spécifiées par l'utilisateur et une vue "concurrence", générée à partir des trois précédentes, pour décrire l'architecture logicielle du système.

Les "composants" sont spécifiés à l'aide de diagrammes de classe UML standard. Un méta-modèle définit l'ensemble des classes utilisables. La sémantique associée à une partie des éléments du méta-modèle respecte le modèle de concurrence du profil Ravenscar (RCM [Bordin and Vardanega, 2005]) assurant ainsi un assemblage de composants "correcte-par-construction" et garantissant l'analyse statique de l'application.

La vue "fonctionnelle" capture les informations structurelles et comportementales des différentes opérations réalisées par le système indépendamment des exigences non fonctionelles du domaine TR^{2}E.

La vue "interface" raffine les composants de la vue "fonctionnelle" en y attachant les informations liées à la concurrence, les interfaces (opérations) requises et fournies par les différents composants et leur interaction. Les composants sont instanciés et interconnectés à l'aide de point d'interaction (*PortCluster*). L'ensemble forme une instance de l'application TR^{2}E vérifiable.

La vue "déploiement" spécifie les composants physiques de l'architecture et exprime comment les composants logiciels s'exécutent sur ces derniers. La notion de "partition" y est introduite pour modéliser un nœud logique, encapsulant les instances de composants et assurant leur isolation spatiale et temporelle.

La vue "concurrence", générée à partir des trois vues précédentes (par transformation "prouvée" [Cancila *et al.*, 2010]), décrit l'architecture du système réalisée pour la plate-forme d'exécution spécifiée par la vue de "déploiement". Les composants de la vue "interface" sont ici traduits en des composants respectant la sémantique du RCM.

Analyses

Le PSM produit par la vue "concurrence" est le point d'entrée pour les différentes analyses. Des analyses temporelles et notamment d'ordonnançabilité sont effectuées à l'aide d'une extension (MAST+ [Bordin *et al.*, 2008; Panunzio and Vardanega, 2011]) de l'outil MAST [Universidad de Cantabria, 2010] autorisant notamment l'analyse de tâches sporadiques. La transformation du PSM vers un modèle MAST+ est assurée par le canevas de conception HRT-UML/RCM. Les résultats de l'analyse d'ordonnancement sont automatiquement remontés à l'utilisateur qui peut, le cas échéant, éditer ses modèles et relancer le processus. D'autres analyses sont possibles, utilisant notamment les méthodes formelles et des techniques de *model checking* pour la partie comportementale.

Génération de code

HRT-UML/RCM permet la génération d'une application TR^{2}E critique et sa plate-forme d'exécution dans les langages Ada 2005, Ada 2005/Ravenscar et C pour le standard OSEK/VDX RTOSs [Bordin and Vardanega, 2005]. La génération de code Ada 2005/Ravenscar semble être la plus aboutie, notamment par les choix de conception des auteurs de l'approche et le respect du profil Ravenscar. Elle s'effectue par transformation modèle-à-code implantée en

© 2012 Gilles Lasnier

MOFscript. Les composants RCM modélisés sont utilisés pour paramétrer et instancier des patrons -*i.e.* des paquetages génériques Ada - implantant le comportement des composants.

Discussion

La méthode HRT-UML/RCM est basée sur l'approche MDA et intègre des outils de transformation de modèles. Le choix de contraindre la réalisation du méta-modèle à la sémantique de RCM renforce l'analysabilité statique du système et contraint la modélisation aux seules constructions autorisées par le profil Ravenscar. Enfin, les analyses temporelles effectuées sur le modèle de l'application tiennent compte d'une partie des ressources intergicielles souvent exclues dans les autres approches.

La principale limite de HRT-UML/RCM est l'utilisation du langage UML. En effet, la modélisation d'un système implique le développement de trois vues (fonctionnelle, interface et déploiement) nécessitant une consolidation des différentes représentations du système à chaque modification de l'un des modèles. De plus, si la notation des diagrammes de classes UML est très expressive pour visualiser interface, opérations et propriétés non-fonctionnelles des composants, celle-ci complique fortement la modélisation dans le cas d'un très grand nombre de composants (passage à l'échelle).

Une autre limitation vise les aspects comportementaux qui ne sont pas explicitement modélisés à l'aide d'un formalisme tels les *statecharts* UML. Ceux-ci sont implicitement décrits par la sémantique des composants RCM et enfouis au sein du méta-modèle par construction. L'analyse comportementale requiert alors une transformation (traduction du comportement des composants) plus complexe vers un formalisme d'un outil tiers.

Enfin, le processus de génération de code est basé sur l'instanciation et la configuration de paquetages génériques Ada implantant le comportement réel des composants. Si ce procédé renforce la fiabilité et augmente le niveau de confiance du système généré, ceci peut se faire au détriment de l'optimisation de l'empreinte mémoire.

2.4.2 OCARINA/POLYORB-HI

Description

OCARINA [TELECOM ParisTech, 2012] est un projet réalisé lors de différents travaux de recherche [Vergnaud, 2006; Zalila, 2008; Renault, 2009] de l'équipe S3 de TELECOM ParisTech, s'articulant autour de la conception d'intergiciel, de la production et de l'analyse automatisée de systèmes TR^{2}E critiques.

OCARINA s'appuie sur le langage de description d'architecture standardisé AADL, ses différentes annexes (modélisation des données, génération de code...) et sur le profil architectural Ravenscar. L'intergiciel POLYORB-HI [Zalila, 2008] a été défini et implanté dans différents langages cibles (Ada/Ravenscar, C, et Java/RTSJ) et sert de plate-forme d'exécution pour les composants architecturaux logiciels modélisés en AADL puis générés dans l'un des langages cibles.

La figure 2.2 décrit le processus de conception et d'analyse d'un système critique à partir de sa description architecturale réalisée en AADL. L'architecture du canevas de développement proposée par OCARINA reprend l'architecture classique d'un compilateur (par exemple GNAT).

© 2012 Gilles LASNIER

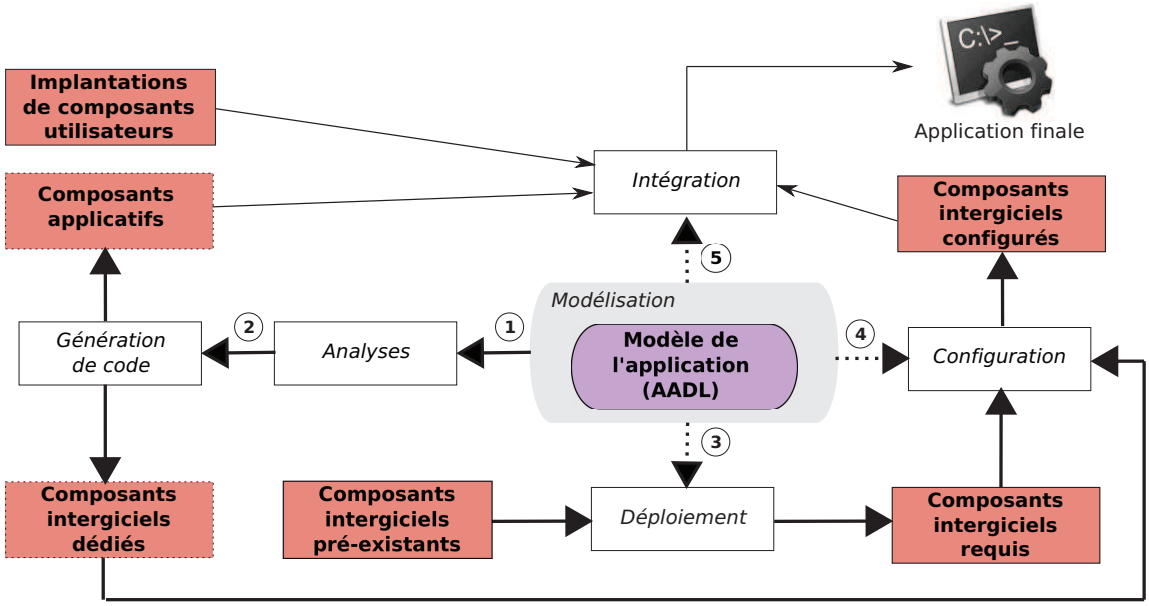


FIGURE 2.2 – Processus de conception de l'approche OCARINA

Modélisation

La modélisation d'un système TR^{2}E critique s'effectue uniquement à l'aide du langage AADL (AADL 1.0 ou AADLv2). L'ensemble des constructions architecturales AADL utilisées par OCARINA est restreint pour la génération de code aux seules constructions autorisées par le profil Ravenscar.

Analyses

Le point d'entrée des analyses par OCARINA est la représentation architecturale finale - *i.e.* complète et valide au sens "légal" AADL - du système. Différentes analyses sont alors possibles. REAL [Gilles, 2008] est un langage de contraintes qui permet de vérifier des propriétés non fonctionnelles sur une description AADL (dimensionnement, définition d'un plafond de priorité, etc). L'analyse d'ordonnancement du système est effectuée à l'aide de l'outil CHEDDAR et un générateur de code permet la transformation du modèle AADL vers les réseaux de Petri pour une analyse comportementale [Renault, 2009].

Génération de code

Une fois le modèle AADL analysé et le système critique validé, OCARINA permet la production automatique du code des nœuds de l'application distribués et des ressources intergicielles (POLYORB-HI) requises par ceux-ci. Les composants intergiciels sont ainsi sélectionnés, déployés, configurés et intégrés de manière automatique aux composants applicatifs pour former l'exécutable final de l'application. Différents générateurs de code sont fournis pour les langages Ada/Ravenscar, C, Java/RTSJ et C/ARINC 653.

© 2012 Gilles LASNIER

Discussion

La principale limitation de l'approche OCARINA vient du fait que l'outil ne supporte que les descriptions architecturales AADL. OCARINA ne disposant pas d'un formalisme pour la spécification des aspects comportementaux (l'annexe comportementale AADL [SAE Aerospace, 2009a] n'étant pas à son niveau de maturité actuel), la spécification des aspects comportementaux s'effectue par l'implantation du comportement au sein de composants opaques (*subprogram*) et explique notamment l'intégration des composants de l'intergiciel POLYORB-HI uniquement lors de la phase de compilation. La non prise en compte des ressources intergicielles (dépendantes de la plate-forme d'exécution) lors des analyses de faisabilité est une limite récurrente des approches basées sur l'IDM. Celle-ci augmente la différence sémantique entre modèles d'analyse et l'implantation réelle du système et complexifie les techniques mises en place (traçabilité...) pour assurer la cohérence entre spécification, analyses et code généré.

Enfin, les transformations vers les modèles d'analyses (réseau de Petri...) et le processus de génération de code à partir de la description architecturale AADL sont implantés en Ada. Ceci complexifie à la fois le développement, la maintenance et l'évolution des générateurs de code comparé à une démarche de génération modèle-à-modèle ou modèle-à-code issue des techniques de transformation de modèles [Brun *et al.*, 2008]. Nous détaillons ces techniques dans les sections suivantes de ce chapitre.

2.4.3 MYCCM-HI

Description

MYCCM-HI [Borde, 2009; Borde *et al.*, 2009] est une méthode de conception dédiée aux systèmes TR^{2}E critiques et adaptatifs - *i.e* reconfigurables -, développée par Thales et TELECOM ParisTech dans le cadre du projet FLEX-EWARE [Flex-eWare Project, 2012]. Cette méthode vise la conception et l'analyse formelle des systèmes TR^{2}E reconfigurables et se base sur le standard Lightweight CCM [OMG, 2003a].

La figure 2.3 décrit le processus de conception de cette méthode. L'architecture générale du canevas de développement MYCCM-HI reprend celle des compilateurs classiques (GNAT, etc).

Modélisation

MYCCM-HI définit son propre langage de description d'architecture COAL (*Component-Oriented Architectecture Language*) pour modéliser le comportement adaptatif d'un système. Celui-ci étend une spécification OMG IDL avec des constructions issues des standards D&C et AADL 1.0. COAL permet de décrire conjointement l'architecture logicielle selon une approche à base de composants "génériques" et les exigences non fonctionnelles spécifiques aux systèmes TR^{2}E sous forme de propriétés.

Les différents comportements possibles du système sont représentés à l'aide de modes de fonctionnement, encapsulés dans des automates analysables.

Analyses

Le point d'entrée des analyses effectuées par le canevas de développement MYCCM-HI est la spécification du système dans le langage COAL. A partir de celle-ci, MYCCM-HI génére

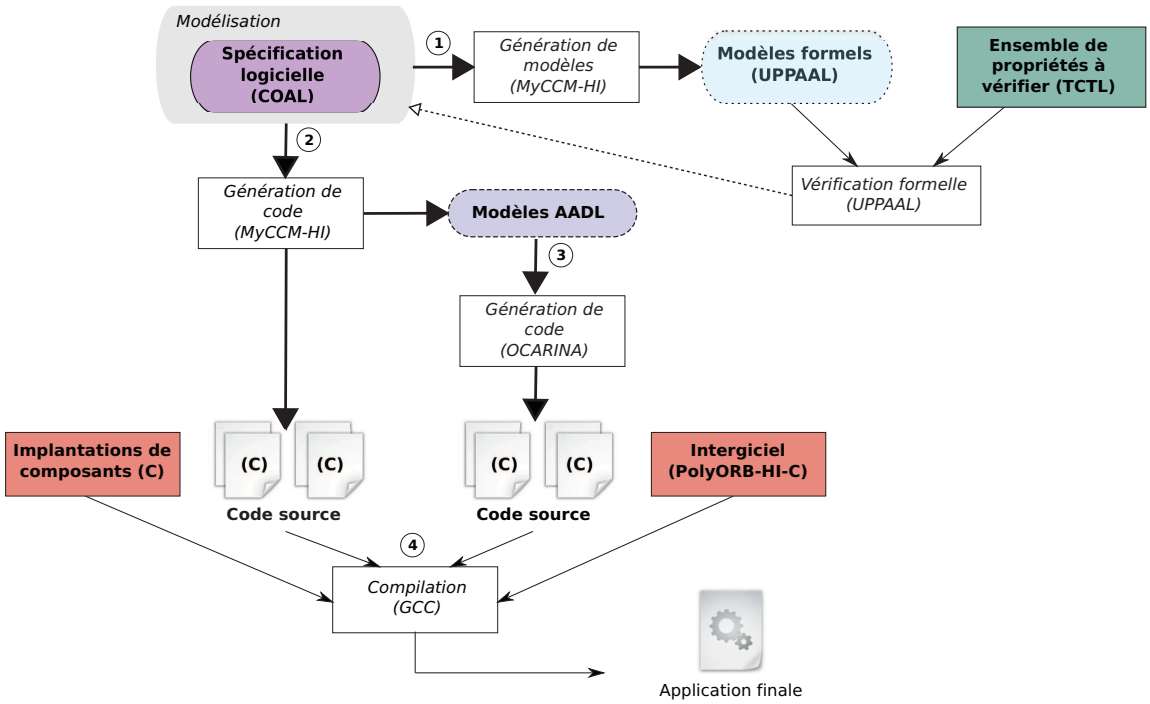


FIGURE 2.3 – Processus de conception de la méthode MyCCM-HI

des modèles formels analysés par l'outil UPPAAL [Uppsala University - Aalborg University, 2012].

Lorsque l'ensemble des propriétés de sûreté requises est validé, un modèle AADL représentant l'architecture logicielle et matérielle est généré à partir de la même spécification COAL. Les outils d'analyse (ordonnancement..) basés sur une représentation AADL et présentés dans le chapitre 4 sont ainsi exploités.

Génération de code

MyCCM-HI propose un processus de production automatique de systèmes critiques et adaptatifs en deux étapes. La première étape est réalisée par un générateur de code C implanté dans MyCCM-HI. Celui-ci génére les différents mécanismes d'adaptation "corrects-par-construction" respectant les contraintes de réalisation des systèmes critiques et requis par le système.

La deuxième partie concerne la génération des activités des tâches et des ressources de communication - *i.e.* des ressources intergicielles - requises par le système et supportant ainsi la partie fonctionnelle générée précédemment. Pour ce faire, MyCCM-HI intègre l'intergiciel PolyORB-HI et utilise le générateur de code C contraint Ravenscar de l'outil Ocarina [TELECOM ParisTech, 2012] pour générer les composants intergiciels.

Discussion

A notre connaissance, la méthode MyCCM-HI est l'une des premières solutions pour la conception et l'analyse de systèmes critiques avec modes. Néanmoins, celle-ci souffre de quelques limitations dues à différents choix de conception.

© 2012 Gilles Lasnier

L'introduction du formalisme COAL et l'utilisation du langage AADL nécessitent l'utilisation d'une transformation dite exogène (voir définition 2.5.1) réalisée et enfouie au sein d'un générateur de code. Ceci complexifie la maintenance de l'outil et nécessite la définition de règles de transformation "correctes" afin de préserver la sémantique des composants modélisés. De plus, l'auteur [Borde, 2009] indique que les améliorations apportées au langage AADLv2 permettent maintenant de se passer de COAL.

Enfin, le processus de production de génération de code qui s'effectue en deux étapes (avec les générateurs MYCCM-HI et OCARINA) complexifie la génération mais aussi la traçabilité des composants requise dans le but de renforcer la cohérence entre modèles d'analyse et code généré.

2.4.4 Conclusion

Dans cette première partie de notre état de l'art, nous nous sommes intéressés aux processus de conception des systèmes TR^{2}E critiques. Certaines de ces solutions tirent bénéfice des techniques issues de l'IDM et notamment de la transformation de modèle pour l'automatisation des étapes d'analyses, de conception et d'implantation des composants logiciels, renforçant ainsi la sécurité et la sûreté de fonctionnement de l'application.

Les techniques de "modélisation" des différents aspects du système (notamment via la séparation des préoccupations) ont permis d'améliorer la productivité et la validation du système très tôt dans le cycle de développement logiciel (avant l'étape d'implantation). Cependant, celles-ci définissent et raisonnent sur des modèles à des niveaux d'abstractions différents introduisant de ce fait un biais sémantique entre modèles d'analyses, modèles de code et l'implantation finale du système. Cette différence de niveaux d'abstraction s'explique en partie, par certaines contraintes des techniques d'analyse (par exemple, les réseaux de Petri nécessitent la définition d'un modèle avec une forte abstraction afin de limiter l'explosion combinatoire) et d'autres résultent de l'inadaptation des outils ou du processus de conception développés.

Ainsi, nous avons identifié trois limites principales :

1. La modélisation implicite des aspects comportementaux et leur utilisation unique lors des étapes d'analyse. Ceci est dû à l'inadaptation du langage de modélisation nécessitant l'intégration ou la traduction vers un formalisme tiers pour expliciter le comportement. L'hétérogénéité des formalismes utilisés complexifie l'interopérabilité.
2. La prise en compte des composants intergiciels uniquement lors de la construction de l'exécutable final. Cette limite découle de la première et de l'incapacité du langage de modélisation à permettre une modélisation simple et efficace des ressources architecturales et comportementales de l'intergiciel. L'intergiciel n'intervient donc pas lors des analyses effectuées sur le modèle du système.
3. Un processus de génération de code inadapté. Dans la plupart des cas, le processus de génération consiste à sélectionner et à configurer un certain nombre de composants logiciels pré-existants. Ceci complexifie les mécanismes mis en œuvre pour assurer la cohérence.

Notre objectif principal vise à renforcer la fiabilité du processus de de réalisation des systèmes critiques et à rendre le code généré du système plus sûr. Pour ce faire, il est nécessaire de renforcer la cohérence entre modèles d'analyse, modèles de code et l'implantation finale du système.

A partir de ces trois limitations, nous proposons un nouveau processus de conception automatisé et détaillé dans le chapitre 3 de ce manuscrit. Ce processus repose sur la définition

d'une chaîne de transformation de modèles assurant le raffinement incrémental d'une description architecturale d'un système modélisé dans le langage AADL. Afin de mieux comprendre le rôle et les différentes étapes de nos transformations, il est nécessaire de s'intéresser aux différents techniques et outils de transformation de modèles.

2.5 Transformation de modèles : enjeux et solutions

L'état de l'art sur la conception et l'analyse des systèmes critiques nous a permis d'identifier deux limitations à savoir : 1) l'absence d'analyse des ressources de l'intergiciel et 2) la différence sémantique entre modèles d'analyse et code généré (limitant la traçabilité et le niveau de confiance du système généré).

L'étude des différents processus de conception de systèmes critiques (voir section 2.4) a montré le rôle central du modèle. Celui-ci se place au cœur du développement logiciel et représente des vues du système à des niveaux d'abstractions variés. Il capture des informations à différentes étapes du cycle de vie (spécification, conception...) et permet leur restitution lors de différentes activités de développement (analyses, génération de code, simulation, etc). La transformation de modèles assure l'évolution de ces représentations du système au sein des différentes étapes du cycle de vie logiciel.

Pour palier aux limites 1) et 2), nous proposons un nouveau processus d'analyse et d'implantation intégrant une étape de "raffinement" incrémental des modèles architecturaux et comportementaux (détaillée dans le chapitre 3) par transformation de modèles. Ce raffinement vise à intégrer les composants intergiciels, à expliciter le comportement des composants logiciels et à réduire le niveau d'abstraction du système critique modélisé tout en préservant la sémantique initiale des composants. Ainsi, nous renforçons la cohérence entre modèles d'analyse et code généré.

Dans cette section, nous présentons des définitions générales autour de la notion de transformation de modèles et nous explorons différents techniques et outils de transformation de modèles disponibles dans le monde de l'IDM.

2.5.1 Principe et concepts généraux

Dans la démarche de l'IDM, la transformation de modèles vise la génération et la manipulation de modèles. Elle trouve son intérêt dans de nombreuses activités du développement logiciel telles que le raffinement, la migration d'un langage, le changement d'espace technologique, la génération de code, etc.

L'OMG définit la transformation de modèles dans le contexte de l'approche MDA comme *"le processus de conversion d'un modèle en un autre modèle appartenant au même système"* [Kleppe *et al.*, 2003]. Mens et al. [Mens and Van Gorp, 2006] définissent la transformation de modèles comme *"la génération d'un ou plusieurs modèles cibles à partir d'un ou plusieurs modèles sources, selon une description de transformation"*. Nous retiendrons cette dernière dans le cadre de ces travaux de recherche.

Définition 2.6 (Transformation de modèles) *Génération d'un ou plusieurs modèles à partir d'un ou plusieurs modèles sources selon une description de transformation.*

© 2012 Gilles LASNIER

Principe de la transformation de modèles

La figure 2.4 décrit le principe d'une transformation de modèles. Celle-ci assure la traduction d'un **modèle source** dans une représentation donnée vers un **modèle cible** dans la même représentation, mettant ainsi à jour les informations du modèle, ou dans une autre représentation, créant le modèle cible à partir d'informations extraites du modèle source.

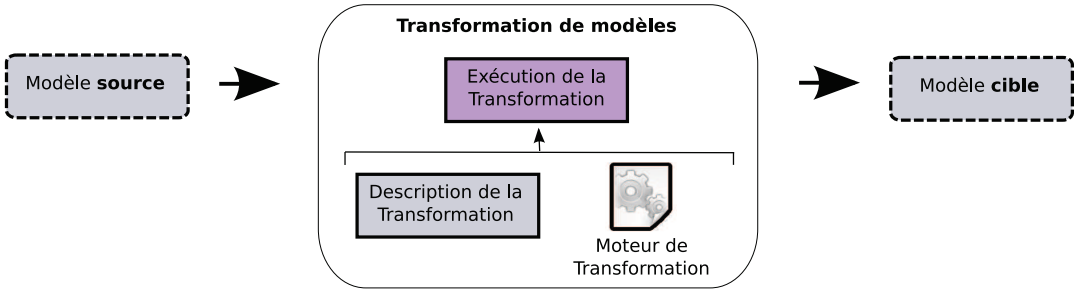


FIGURE 2.4 – Principe de la transformation de modèles

La **description de transformation** exprime comment un ou plusieurs modèles sources sont transformés en un ou plusieurs modèles à l'aide d'un langage de transformation de modèles. Elle définit l'ensemble des **règles de transformation** qui décrivent comment un ou des éléments (ou fragment) du modèle source sont transformés en un ou des éléments du modèle cible. Une règle de transformation contient un patron source et un patron cible. Pour chaque occurence du patron source du modèle source, un patron cible est créé dans le modèle cible.

Le **moteur de transformation** de modèles exécute ou interprète la description de transformation et applique celle-ci pour produire le modèle cible à partir du modèle source. L'automatisation d'une transformation "correctement définie" permet la réutilisation des informations du système et de renforcer la cohérence entre les différents modèles créés, raffinés ou maintenus.

Méta-niveaux d'une transformation de modèles

La figure 2.5 illustre les méta-niveaux de l'architecture d'une transformation de modèles et met en relation les concepts présentés précédemment et ceux issus de la méta-modélisation présentés en introduction (voir chapitre 1).

En particulier, nous observons qu'une description de transformation peut être représentée par un **modèle de transformation** conforme à un méta-modèle. Ainsi, il est possible d'utiliser une transformation de modèles comme modèle source et/ou modèle cible d'une autre transformation. Cette caractéristique offre des perspectives très intéressantes en terme d'évolution et de maintenance. Elle autorise la modification ou l'ajout d'informations (ou d'annotations) à la description de transformation (et par conséquent aux règles qu'elle définit) renforçant, par exemple, les techniques de traçabilité existantes.

Caractérisation des transformations de modèles

Les modèles sources et cibles peuvent être conformes à des méta-modèles identiques ou différents, appartenir à un même niveau d'abstraction (raffinement) ou à des niveaux d'abs-

© 2012 Gilles LASNIER

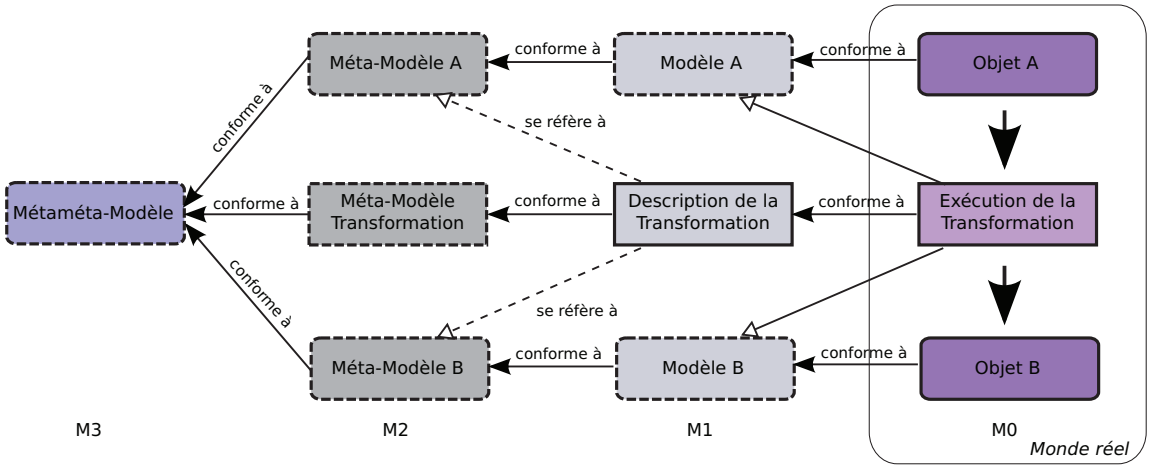


FIGURE 2.5 – Méta-niveaux de l'architecture de la transformation de modèles

traction différents (transformation de PIM vers PSM dans le MDA par exemple). Dans la littérature [Mens and Van Gorp, 2006], on parle alors de transformations *endogènes*, de transformations *exogènes*, de transformations *horizontales* ou de transformations *verticales*. Ainsi, nous pouvons distinguer quatre types de transformation de modèles, 1. et 2. par rapport à la différenciation des méta-modèles et 3. et 4. par rapport à la différenciation du niveau d'abstraction des modèles sources et cibles.

1. Une transformation est dite *endogène* si les modèles sources et cibles sont définis dans le même méta-modèle. Une optimisation, un refactoring ou une simplification sont des exemples de transformations endogènes.
2. Une transformation est dite *exogène* si les modèles sources et cibles sont définis dans des méta-modèles différents. Une synthèse (conversion d'un modèle source vers un niveau d'abstraction moins élevé), une rétro-ingénierie, une translation et une migration (programme Ada vers C) sont des transformations exogènes.
3. Une transformation est dite *horizontale* si les modèles sources et cibles appartiennent au même niveau d'abstraction. Le raffinement ou la translation est un exemple de transformation horizontale.
4. Une transformation est dite *verticale* si les modèles sources et cibles appartienent à des niveaux d'abstraction différents. Une synthèse, une rétro-ingénierie et la génération de code sont des exemples de transformation verticale.

Remarque. Nous pouvons aussi distinguer : la transformation *in-place* qui définit une transformation d'un modèle vers lui-même et assume que la source et la cible sont identiques excepté pour les fragments ciblés par la description de transformation ; et la transformation **higher-order** (HOT) qui définit une transformation de modèles ayant une transformation de modèles comme modèle source ou modèle cible.

Pour finir, les modèles sources et cibles peuvent également appartenir à des espaces technologiques. Un espace technologique est un contexte opérationnel de travail comportant un ensemble de concepts, un corps de connaissances, des outils, des compétences, etc. Il définit l'ensemble des technologies utilisées pour la représentation d'un modèle, les structures

© 2012 Gilles LASNIER

de données, les parties-frontales, les outils nécessaires pour la manipulation des données et les formats de stockage des fichiers. Des exemples connus d'espace technologique pour la transformation de modèles sont EMF [Steinberg *et al.*, 2009], KERMETA [Fleurey, 2006], XML [Bray *et al.*, 2008] et le MDA [Blanc, 2005].

La section suivante présente brièvement les problématiques adressées par la transformation de modèles.

2.5.2 Problématiques adressées

Dans la littérature, nous pouvons trouver plusieurs outils traitant différentes problèmatiques par l'utilisation de la transformation de modèles. Dans cette section, nous présentons un aperçu des problématiques majeures adressées par celle-ci.

Changement du niveau d'abstraction

Selon la définition 1.5 (présentée dans l'introduction), le niveau d'abstraction est une mesure de la quantité d'informations - *i.e.* du niveau de détail - définie pour la description des fragments du modèle. La transformation de modèles permet soit (1) de réduire cette quantité d'informations (augmentation de l'abstraction du modèle), soit d'introduire de nouveaux détails et par conséquent de (2.1) réduire le niveau d'abstraction ou (2.2) de le laisser inchanger. Dans tous les cas (1, 2.1 et 2.2), cette propriété est indépendante des changements du méta-modèle et les méta-modèles source et cible peuvent être les mêmes ou différents.

Une transformation horizontale change l'espace technologique du modèle mais ne change pas le niveau d'abstraction initial du modèle (exemple de la translation). Une transformation verticale change le niveau d'abstraction du modèle. Celui-ci peut être augmenté par une synthèse réduisant le niveau de détail ou diminué par un raffinement ajoutant des détails au modèle source. Un exemple de transformation verticale de *raffinement* est la transformation d'un PIM en PSM [Blanc, 2005].

Changement de méta-modèle

Les transformations exogènes permettent la transformation des concepts d'un méta-modèle A vers des concepts d'un méta-modèle B. Ce type de transformation permet d'exhiber un aspect particulier du système modélisé et de le rendre plus compréhensible que ce soit par l'humain ou à l'aide d'un outil d'analyse spécifique (ex : représentation d'un système en réseau de Petri pour la vérification de propriétés de sûreté).

Changement d'espace technologique

Un espace technologique permet d'établir une représentation du modèle qui peut être stocké soit dans un format de fichier spécifique soit sous la forme de structures de données en mémoire (ex : XML [Bray *et al.*, 2008] ou XMI [OMG, 2007b]). Il fournit alors un ensemble de mécanismes permettant la manipulation de ces données. Ainsi, il est possible que l'espace technologique limite les moteurs de transformation pour diverses raisons telles que la scalabilité, le type de représentation (graphique vs textuelle), la traçabilité, la mise-à-jour incrémentale ou encore la stratégie d'exécution.

Aussi, le changement d'un espace technologique à un autre sous réserve d'une transformation "correctement définie" permet de contourner certaines limites.

© 2012 Gilles LASNIER

Transformation modèle-à-code

La transformation *modèle-à-code* est utilisée pour générer du code dans un langage de programmation (Ada, Java, XML, etc.) à partir d'un modèle source. C'est un cas particulier de transformation *modèle-à-modèle* qui autorise un *mapping* arbitraire d'un élément du modèle source à une syntaxe particulière ou à un artefact du langage de programmation ciblé. Deux approches de transformation sont distinguées. L'approche basée sur le parcours de modèles (visiteurs) permet de traverser la représentation interne du modèle et de générer un code sous forme de texte dans un flux de sortie. L'approche basée sur l'utilisation de *templates* définit des fragments de code cible contenant des *méta-codes* qui assurent l'accès aux informations du modèle source, la sélection de morceaux de code et la réalisation d'expansions de manière itérative.

Cette transformation particulière est une alternative au processus de génération de code basé traditionnellement sur la théorie de la compilation.

Préservation des propriétés du système

Description de transformation avec préservation de la sémantique. Dans le cas des transformations endogènes, il est possible de définir une description de transformation qui préserve la sémantique des fragments du modèle. Ainsi, le modèle cible contient les mêmes informations provenant du modèle source mais celles-ci ne sont plus exprimées ou représentées sous la même forme. Elles sont soit traduites dans un espace technologique différent, soit avec une syntaxe abstraite différente.

Généralement, une description de transformation avec préservation de la sémantique améliore la description des éléments du modèle en affinant la spécification de ces attributs (par exemple, rendre explicite des informations implicites). Cependant, il peut s'avérer difficile d'exprimer un *mapping* qui préserve complètement la sémantique de l'élément ou d'un groupe d'éléments. Ainsi, il est possible de décrire une description de transformation définissant une *approximation* des propriétés du système. Celle-ci préserve alors uniquement les propriétés essentielles du modèle.

Un exemple de transformation avec préservation de la sémantique est l'optimisation du modèle à des fins de simulation ou d'amélioration de performances.

Description de transformation avec préservation du comportement. Une transformation préserve le comportement si les aspects comportementaux (i.e. les contraintes) spécifiés dans le modèle source sont également spécifiés de manière implicite ou explicite dans le modèle cible.

Remarque. Une description de transformation peut préserver le comportement (resp.la sémantique) et/ou la sémantique (resp. le comportement) d'un modèle (ex : la transformation de *modèle-à-code*).

Dans la section suivante, nous présentons brièvement une sélection des techniques et des outils de transformation de modèles des milieux universitaires et industriels.

© 2012 Gilles LASNIER

2.6 Approches et outils de transformation de modèles

La transformation de modèles est l'une des clés du succès de l'IDM. Nous pouvons trouver aujourd'hui de nombreux techniques et outils de transformation de modèles dans les milieux universitaires et industriels. Dans la littérature, ces techniques et outils ont été classés selon différents critères [Czarnecki and Helsen, 2003; Czarnecki and Helsen, 2006; Mens and Van Gorp, 2006; Muller, 2006; Srivastava *et al.*, 2006; Jézéquel, 2008; Biehl, 2010]. Dans le cadre de nos travaux de recherche, nous nous contenterons de brosser un rapide aperçu des approches de transformation de modèles et des outils les plus pertinents pour notre solution. Nous mettons l'accent sur QVT et en particulier l'outil ATL avec lequel nous avons réalisé l'ensemble de nos transformations. Le choix d'ATL s'explique en grande partie par la maturité du langage (syntaxe textuelle précise...) et de ses outils associés et par les contraintes de réalisation (manipulation de méta-modèle Ecore, dévelopement de plug-ins ECLIPSE/RCP) que nous devons satisfaire pour l'implantation d'un prototype évolutif, maintenable et son intégration au canevas de développement OSATE2.

2.6.1 Approches de transformation de modèles

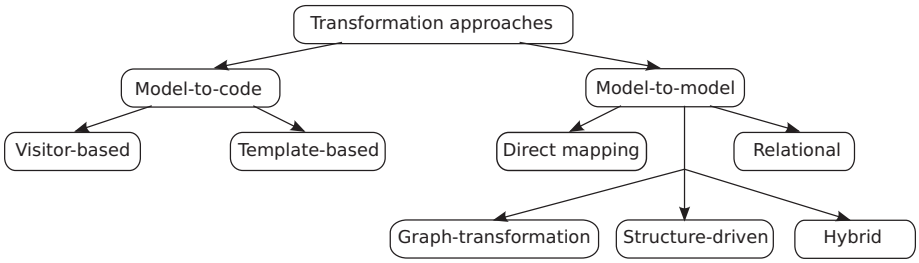


FIGURE 2.6 – Principales approches de transformation de modèles

La figure 2.6 présente les principales approches de transformation de modèles selon les critères de classification de Krzysztof Czarnecki et Simon Helsen [Czarnecki and Helsen, 2003; Czarnecki and Helsen, 2006]. Nous retrouvons les deux catégories présentées dans les sections précédentes : *modèle-à-code* (présenté en section 2.5.2) et *modèle-à-modèle*. Nous nous concentrons ici sur les transformations *modèle-à-modèle*. On distingue cinq principales approches.

1. L'*approche basée sur la manipulation directe de modèles* est généralement implantée sous forme de framework orienté objet (par exemple JMI [Dirckze, 2002]), fournit une représentation interne des modèles et une API pour la manipulation de celle-ci. Les règles de transformation sont définies à l'aide d'un langage de programmation (par exemple Java) par l'utilisateur.
2. L'*approche relationnelle* vise à spécifier, sous forme de contrainte, une relation entre éléments des modèles source et cible. La relation est déclarative et la spécification n'est pas exécutable.
3. L'*approche basée sur les transformations de graphes* repose sur la théorie des graphes. Elle vise la représentation graphique des modèles sous forme de graphes étiquetés et contraints par des règles de cohérence. Les techniques de réécriture de graphes et

© 2012 Gilles LASNIER

de transformation de graphes sont ensuite appliquer. Une description de transformation est un ensemble de règles de réécriture et les règles de transformation peuvent être exprimées de manière déclarative comme dans les approches relationnelles. Nous citons à titre illustratif pour cette catégorie les outils ATOM [Lara and Vangheluwe, 2002], UMLX [Eclipse,], VIATRA [Eclipse Community,] et GREAT [Vanderbilt University, 2012].

4. L'*approche dirigée par la structure* réalise une transformation en deux étapes. La première étape crée la hiérarchie de la structure de la cible et la deuxième étape définit les valeurs des attributs et des références dans la cible. Les règles de transformation sont spécifiées par l'utilisateur. Les stratégies d'ordonnancement et d'application des règles sont définies par l'approche.
5. L'*approche hybride* est une combinaison des approches précédentes. Le langage de transformation de règles combine les approches déclarative et impérative. La spécification et l'implantation d'une transformation sont distinctes et deux types de règles sont définis. Les règles de correspondance déclarent les relations entre éléments du modèle source et du modèle cible. Les règles opérationnelles définissent les règles exécutables (création, modification, suppression, etc.). KERMETA [Fleurey, 2006] et ATL [ATL, 2012] sont des exemples d'approche hybride que nous détaillons dans les sections suivantes.

2.6.2 Outils de transformation de modèles

Intéressons-nous maintenant, brièvement, aux différentes catégories d'outils dédiés à la transformation de modèles. Nous pouvons distinguer cinq catégories dominantes.

1. Les *outils associés aux langages de programmation* sont disponibles sous forme d'APIs dans un langage de programmation classique comme C++ ou Java. Ainsi, Java fournit l'API JMI (*Java Metadata Interface* [Dirckze, 2002]) qui permet l'implantation d'une infrastructure pour gérer la création, l'enregistrement, l'accès, la recherche et l'échange de métadonnées MOF. Si l'utilisateur n'a pas besoin d'apprendre un nouveau langage pour définir les transformations de modèles, les outils de cette catégorie s'avèrent limités dans le cadre de la méta-modélisation et de l'implantation de transformations complexes.
2. Les *outils génériques* tels que XSLT [W3C, 1999] et XQUERY [W3C, 2010] ont profité de la popularité et des nombreux travaux autour d'XML pour atteindre un bon niveau de maturité. S'ils se sont avérés simples et efficaces pour la manipulation des éléments de la syntaxe abstraite, de nombreux travaux ont montré leurs limites en terme de validation et de maintenance et au niveau sémantique des modèles manipulés.
3. Les *outils intégrés aux ateliers de génie logiciel* tel OBJECTEERING [Software, 2012], OPTIMALJ ou FUJABA [Fujaba Core Development Group, 2012] ont pour avantage leur maturité et la qualité de leur intégration dans des ateliers de génie logiciel souvent propriétaires. Cependant, ces outils souffrent aussi de ce dernier avantage car ils sont souvent développés en second plan dans leurs ateliers respectifs et présentent des limites dans le développement (structuration, modularité, réutilisation, maintenance, etc.).
4. Les *outils spécifiques* tel ANDROMDA [AndroMDA, 2012], ATL [ATL, 2012], SMART-QVT [Alizon *et al.*, 2007] ont pour avantage leur simplicité de développement, leur forte expressivité, la maintenance des transformations, l'interopérabilité et la composition des règles de transformation. Bon nombre de ces outils universitaires et industriels de cette catégorie reposent ou s'inspirent du standard QVT de l'OMG que nous présentons dans la section suivante.

© 2012 Gilles LASNIER

5. Les *ateliers (ou outils) de méta-modélisation* reposent sur la programmation par objets pour la construction des modèles de transformations et sur l'IDM pour l'exécution de ces modèles. Une transformation de modèles est ainsi un méta-programme exécutable. Ces outils sont moins nombreux, nous pouvons citer les plus connus KERMETA [Fleurey, 2006], EMF [Steinberg *et al.*, 2009] ou METAEDIT [MetaCase, 2012].

Dans les sous-sections suivantes, nous mettons l'accent sur les ateliers de méta-modélisation EMF et KERMETA, sur le standard QVT et sur l'outil ATL. Une synthèse termine cette section et conclut sur notre choix d'utiliser le couple EMF et ATL pour l'implantation de nos transformations de modèles.

ECLIPSE MODELING FRAMEWORK (EMF/Ecore)

Description. EMF (*Eclipse Modeling Framework* [Steinberg *et al.*, 2009]) est à la fois une plate-forme de modélisation et de génération de code. Son objectif vise à faciliter la construction et le développement rapide d'outils basés sur des modèles structurés et leur intégration au sein de la plate-forme ECLIPSE. De ce fait, l'architecture d'EMF s'articule autour d'un certain nombre de plug-ins. Parmi les plus essentiels, nous pouvons citer : le méta-modèle Ecore, canevas de classes pour la description de modèles EMF et la manipulation des référentiels de modèles ; le canevas de classe *EMF.Edit* pour le développement d'outils d'édition de modèles ; le modèle de génération *GenModel* qui assure la configuration et la personnalisation de la génération de code en Java à partir des éléments décrits au sein d'un modèle EMF (voir figure 2.7). La persistance des instances de modèles est assurée à l'aide de XMI (XML).

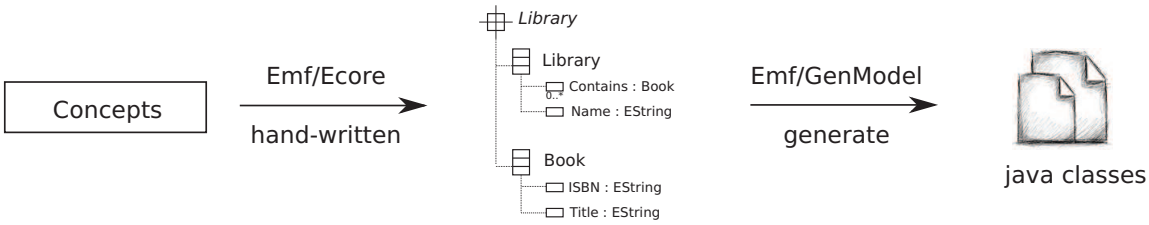


FIGURE 2.7 – Interactions des plug-ins EMF

Enfin, un certain nombre de plug-ins additionnels, développés dans le contexte du *Eclipse Modeling Project* ou par la communauté ECLIPSE en général, s'exécutent au dessus d'EMF et permettent l'interrogation (*Model Query*), la validation (*Validation Framework*) ou la transformation de modèles (EMT), etc.

Discussion. EMF est un atelier de méta-modélisation orienté vers la réalisation de méta-modèles et d'outils dédiés à IDM (représentation, édition, persistance...) à partir de techniques génératives. L'avantage principal de cette technologie est l'importante communauté qui la soutient et les nombreux projets d'IDM venant se greffer au dessus, assurant la maintenance, l'évolution et l'intégration de nombreux standards et outils dédiés à l'IDM. Cependant, son utilisation seule pour effectuer des transformations de modèles reste limitée, nécessitant l'utilisation d'une autre technologie (par exemple ATL que nous détaillons plus bas) pour assurer efficacement les objectifs de transformation de modèles.

© 2012 Gilles LASNIER

KERMETA

Description. KERMETA [Fleurey, 2006; Jézéquel *et al.*, 2011], développé par l'IRISA-INRIA de Rennes [Triskell, 2012], vise à définir un environnement de méta-modélisation qui permet à la fois de spécifier la structure et la sémantique des méta-modèles, d'instancier ces méta-modèles et de réaliser des transformations de modèles. C'est à la fois un langage de méta-modélisation et de programmation impérative. Il offre des supports pour la spécification de contraintes, la spécification comportementale à l'aide d'un langage d'actions et la vérification de modèles.

Une description en KERMETA est comme un programme issu de la fusion d'un ensemble de méta-données (EMOF, Ecore...) et du méta-modèle d'action AS (*Action Semantics* intégré à UML 2.0). KERMETA intervient ainsi à deux niveaux dans l'architecture de méta-modélisation définie par l'OMG (voir figure 2.5). C'est à la fois un langage de niveau *M3* (tous les méta-modèles lui sont conformes) mais également une bibliothèque de base pour la construction de méta-modèles de niveau *M2*.

Un méta-modèle peut être spécifier en EMOF, Ecore ou dans le langage KERMETA. Un support est fourni pour la traduction vers des modèles EMF permettant ainsi d'utiliser l'outillage EMF. Les constructions du langage d'actions permettent de définir des expressions, de naviguer dans les modèles, de créer/modifier des modèles, de parcourir l'arbre syntaxique des méthodes et de modifier la sémantique opérationnelle, etc. Les modèles et les méta-modèles sont enregistrés dans des référentiels. Le moteur de transformation associé au langage lit le modèle source à partir du référentiel, exécute la transformation et écrit le modèle cible dans un référentiel conformément aux spécifications de la transformation.

Enfin, KERMETA supporte la gestion des erreurs et la programmation par aspect mais ne fournit aucun support pour la traçabilité, la multi-directionnalité et la transformation de modèles incrémentale.

Discussions. L'idée principale de KERMETA est de fournir un noyau "commun" pour la spécification de la structure et de la sémantique des méta-modèles afin de palier aux limitations de cohérence et à l'interopérabilité liées à l'hétérogénéité des technologies utilisées (MOF, EMOF ou Ecore pour l'écriture des méta-modèles, OCL ou AS *Action Semantics* pour l'expression de contraintes, Java ou QVT pour l'écriture de transformations, etc.). Si cette idée présente un réel avantage du fait que l'utilisateur n'a pas besoin d'apprendre de multiples technologies, il s'avère que, bien souvent, pour des fonctionnalités manquantes ou des besoins d'interopérabilité, l'utilisateur soit amené à exporter ces modèles vers une autre technologie tel par exemple EMF.

Avec KERMETA, les modèles et les méta-modèles ont besoin d'être explicitement chargés en mémoire et sauvegardés dans un référentiel. Les éléments cibles ont aussi besoin d'être explicitement instanciés et ajoutés au modèle cible. Le contrôle de la stratégie d'application et d'exécution des règles de transformation doit être explicitement spécifié. A la différence de certains outils de transformation de modèles (ATL, SmartQVT, etc.), ces points nécessitent un effort et l'écriture de code supplémentaire dans le langage KERMETA par l'utilisateur.

Query/View/Transformation (QVT)

Description. QVT (*Query/View/Transformation*) est un langage hybride de transformation de modèles standardisé par l'OMG [OMG, 2009]. Il s'aligne sur l'approche MDA [Blanc, 2005] et utilise le langage MOF (*Meta Object Facility* [OMG, 2006c]) pour décrire la syntaxe abstraite

© 2012 Gilles Lasnier

de son méta-modèle. Il offre un langage de requêtes s'appuyant sur le langage OCL [OMG, 2010]. QVT définit une syntaxe textuelle concrète et un méta-modèle basé sur XMI pour créer des représentations de modèles de transformation (voir définition 2.5.1). Il autorise l'intégration et l'invocation d'implantation (du code) "externe" à l'intérieur d'une transformation à l'aide d'un mécanisme de *Black Box*. La traçabilité est un objectif du standard QVT et est supportée de manière automatique.

Langages et structure. La dernière version du standard QVT [OMG, 2009] définit trois langages (et méta-modèles) pour des transformations modèle-à-modèle. Les langages QVT Relational et QVT Core sont des langages déclaratifs et ont des niveaux d'abstraction différents. Le langage QVT Operational est un langage impératif.

- QVT Relational permet de spécifier des relations entre des modèles du MOF et repose sur l'utilisation de patrons d'objets. Une transformation est définie à l'aide d'un ensemble de patrons qui sont instanciés pour la spécification de nouveaux éléments, mis en relation avec des éléments et/ou utilisés pour effectuer des modifications dans le modèle. Une syntaxe textuelle et une syntaxe graphique simple sont fournies. Le langage offre des mécanismes pour la création et la suppression automatiques d'objets, l'expression des relations entre modèles, l'identification des éléments cibles et la gestion automatique implicite des informations de traçabilité entre éléments des différents modèles d'une transformation. L'utilisateur n'a donc pas ces derniers points à gérer. Le langage supporte les transformations bidirectionnelles et le sens de la transformation doit être spécifié à l'exécution.
  Ce langage de relations permet de vérifier et de renforcer la cohérence lors de la modification du modèle cible, de synchroniser les deux modèles et autorise les transformations *in-place*. Le langage OCL permet d'exprimer des requêtes pour l'élaboration de patrons complexes. Enfin, la sémantique est définie par un *mapping* avec le langage QVT Core.
- QVT Core est un langage de transformation de modèles technique de bas niveau. C'est une extension minimale de EMOF et OCL qui sert de fondation pour le langage de relations. Il est défini par une syntaxe textuelle.
- QVT Operational définit la partie impérative de QVT et étend les deux langages déclaratifs précédents en ajoutant des constructions impératives et des constuctions OCL. Il définit une syntaxe concrète plus familière aux utilisateurs des langages de programmation impératifs et des mécanismes de gestion automatique de traces.

Remarque. Un langage d'implantation opaque d'opérations MOF est proposé et définit une *Black Box* pour étendre et invoquer des fonctionnalités de transformation implantées dans un langage supportant le MOF, d'implanter une partie des transformations de manière opaque ou d'utiliser des librairies spécifiques à un domaine (dans le cas de contraintes non exprimables en OCL). Ainsi, une transformation peut être spécifiée dans l'un des langages déclaratifs et implantée de manière impérative dans le langage QVT Operational ou par le mécanisme de *Black Box*.

Outillages. Le standard QVT de l'OMG est une référence pour les outils spécifiques à la transformation de modèles. Ainsi, il existe de nombreuses implantations des langages du standard QVT (ou s'en inspirant fortement tel ATL) commerciales ou libres. Nous présentons brièvement les outils SMARTQVT et MODELMORF.

© 2012 Gilles LASNIER

SMARTQVT [Alizon *et al.*, 2007] est une implantation du standard QVT Operational basée sur EMF. La description de transformation est compilée en code Java. L'outil supporte le mécanisme de *Black Box*, la traçabilité, le contrôle de la stratégie et des paramètres des règles. La transformation incrémentale et la multi-directionnalité ne sont pas supportées.

MODELMORF [Services, 2012] est une implantation du langage standard QVT Relational. Il supporte la multi-directionnalité et une même règle de transformation peut être utilisée pour *mapper* des éléments dans les deux directions. Les transformations *in-place* sont supportées de même que la transformation incrémentale du modèle cible ce qui implique que, lors d'un changement du modèle source, seule la partie changée du modèle est transformée. De plus, il offre un support et des fonctionnalités pour créer ses propres traces. Enfin, la réflexion et la transformation incrémentale du modèle source ne sont pas supportées.

Discussion. QVT a été proposé par l'OMG dans l'objectif de normaliser la définition de transformations de modèles et d'assurer l'interopérabilité des outils de transformation de modèles. Si l'initiative est très intéressante, celle-ci se heurte aujourd'hui à une définition très large de la notion de conformité d'un outil au standard QVT. Ainsi, de nombreux outils tel MODELMORF ou SMARTQVT n'implantent qu'une partie de la norme et ne répondent pas forcément aux attentes en termes de réutilisation et de portabilité des transformations. De plus, le caractère non standard des *Black Box* complexifie fortement la réalisation de ces derniers points. Enfin, la spécification QVT concerne uniquement la définition de transformations de modèle-à-modèle et la normalisation des transformations modèle-à-code est en cours d'étude.

ATLAS TRANSFORMATION LANGUAGE (ATL)

Description. ATL (*ATLAS Transformation Language* [ATL, 2012]) est un langage de transformation modèle-à-modèle hybride - *i.e.* autorisant des constructions déclaratives et impératives - développé par OBEO et supporté par la communauté ECLIPSE. Il s'inspire en grande partie du standard QVT [Jouault *et al.*, 2006]. La structure déclarative est la plus utilisée, elle permet de définir une implantation de transformation plus simple et plus claire. Les constructions impératives visent des descriptions de transformations plus complexes.

ATL supporte les transformations endogènes et exogènes et autorise les descriptions de transformation prenant en entrée (resp. en sortie) un ou plusieurs modèles sources (resp. cibles). Une description de transformation est compilée (en ASM) puis exécutée par le moteur de transformation d'ATL [Jouault *et al.*, 2008]. Dans le mode initial, les transformations sont unidirectionnelles (sources vers cibles), l'ordre d'exécution des règles est déterminé automatiquement et le contrôle de la stratégie d'application des règles n'est pas explicitement permis. Néanmoins, celui-ci peut être influencé par filtrage des patrons du modèle source. La transformation incrémentale de modèles n'est pas supportée - *i.e.* un modèle source complet est lu et un modèle source complet est créé - mais certains travaux autour d'ATL ont montré la possibilité d'obtenir un résultat similaire [Jouault and Tisi, 2010]. ATL supporte la traçabilité. Le *refining mode* d'ATL autorise des transformations *in-place* en limitant l'utilisation et la combinaison des constructions et autorise la suppression des éléments dans le modèle cible.

Enfin, des travaux ont montré la possibilité d'utiliser ATL à des fins de vérification de validation de modèles par l'expression de contraintes sous la forme de requêtes OCL [Bézivin and Jouault, 2006] .

© 2012 Gilles LASNIER

Structure d'une description de transformation. L'exemple de code ATL 2.1 illustre une description de transformation ATL. Celle-ci est composée de règles décrivant comment créer et initialiser les éléments du modèle cible. ATL définit un modèle MOF pour sa syntaxe abstraite et possède une syntaxe concrète textuelle. Un sous-ensemble du langage OCL est supporté et permet d'exprimer, sous forme d'expressions, des requêtes (*helper*) pour accéder aux éléments d'un modèle, naviguer entre les éléments du modèle, appeler des opérations, extraire des informations et exprimer des contraintes ou des gardes sur les patrons.

Une règle déclarative (*Matched Rule)* est constituée d'un nom, d'un ensemble de patrons sources (*InPattern*) désignant des éléments des modèles sources et d'un ensemble de patrons cibles (*OutPattern*) représentant les éléments créés des modèles cibles. Deux constructions impératives sont utilisables : des règles impératives (*Called Rule*), exprimées dans la même syntaxe que les règles déclaratives et appelées explicitement ; un bloc d'instructions impératives (*ActionBlock)* attaché aux deux types de règles dont la syntaxe peut-être du code (par exemple Java).

Exemple 2.1 – Exemple de description de transformation ATL

```
-- @path Families=/Families2Persons/Families.ecore
-- @path Persons=/Families2Persons/Persons.ecore

module Families2Persons;
create OUT : Persons from IN : Families;

helper context Families!Member def: isFemale() : Boolean =
  if not self.familyMother.oclIsUndefined() then
    true
  else
    if not self.familyDaughter.oclIsUndefined() then
      true
  else
      false
  endif
endif;

rule Member2Male {
  from    s : Families!Member (not s.isFemale())
  to      t : Persons!Male (fullName <- s.firstName + ' ' + s.familyName)
}

rule Member2Female {
  from    s : Families!Member (s.isFemale())
  to      t : Persons!Female (fullName <- s.firstName + ' ' + s.familyName)
}
```

Outillages. Les outils de transformation liés à ATL sont intégrés sous forme de plug-in ADT (ATL *Development Tool*) pour la plate-forme de développement ECLIPSE et peuvent gérer les modèles basés sur EMF. Les modèles basés sur des profils UML construits à partir d'EMF sont aussi supportés.

Exécution. ATL supporte deux modes d'exécution. Dans le mode standard, les éléments sont uniquement créés lorsque les patrons sources définis dans les règles déclaratives sont reconnus, puis le système instancie les éléments des patrons cibles. Une fois l'instanciation réalisée, un lien de traçabilité est créé associant chaque élément du modèle source (reconnu) à un élément du modèle cible. Ces liens de traçabilité sont ensuite évalués et permettent de déterminer les propriétés des éléments instanciés.

Dans le mode par raffinement (*refining mode*), tous les éléments du modèle source non reconnus par un patron source sont recopiés automatiquement dans le modèle cible, réduisant les efforts et favorisant ainsi les transformations d'une toute petite partie du modèle.

© 2012 Gilles LASNIER

Discussion. L'atout principal d'ATL réside dans l'expressivité de ces transformations (syntaxe précise et facilement compréhensible), sa compatibilité avec EMF (méta-modèles Ecore) et dans sa facilité d'utilisation (que nous verrons en détail dans le chapitre 8 de ce manuscrit). Son intégration comme projet au sein de la plate-forme ECLIPSE, le support de la communauté ECLIPSE et celui de la société OBEO (qui offre un support industriel) sont autant d'atouts considérables qui ont permis de structurer, de développer et de maintenir le langage et son outil. Le support de la réflexion permet d'utiliser des descriptions de transformation ATL comme modèle source et/ou cible d'une transformation et offre des facilités pour la mise en œuvre des mécanismes d'annotation, de traçabilité, d'évolution et de maintenance efficaces.

Néanmoins, le mode par raffinement proposé supporte un ensemble très faible des constructions du langage et limite fortement l'utilisation de règles impératives. La structure des descriptions de transformation (exemple 2.1) contraint le chainage transformations de modèles et notament, la *superimposition* des descriptions de transformation (permettant la surchage des règles de transformation) a une configuration statique des entrées/sorties des transformations. Enfin, les transformations unidirectionnelles limitent les possiblités de synchronisation entre les modèles sources et cibles.

2.6.3 Conclusion et justification du choix du langage de transformation

Dans cette seconde partie de notre état de l'art, nous avons présenté le principe, les concepts généraux et les problèmes adressés par la transformation de modèles. A travers une "brève" étude des différents approches et outils de transformation de modèles, nous avons présenté les principales caractéristiques attendues et celles supportées par des outils pertinents ayant un bon niveau de maturité.

Notre objectif est de définir des transformations de modèles qui permettent de décrire et d'automatiser un processus de raffinement incrémental d'un système critique modélisé en AADL. L'automatisation de l'ensemble du processus de raffinement nécessite la définition de transformations qui agissent uniquement sur la structure des composant AADL et non sur l'arbre syntaxique de ses méthodes - *i.e* sa représentation technologique interne. De plus, nous visons l'intégration de notre prototype de raffinement par transformation de modèles au sein de l'outil OSATE2, canevas de développement et plate-forme fédératrice pour l'intégration des projets autour du langage AADL, reposant sur ECLIPSE. Cet outil fournit notamment le méta-modèle du langage AADL (en Ecore) réalisé à l'aide des technologies UML 2.0 et EMF. En outre, par contraintes de coûts, temporelle et humaine, nous ne souhaitons pas redévelopper un langage ou un outil de transformation mais réutiliser une technologie libre offrant des perspectives d'interopérabilité avec l'outil OSATE2, de maintenance aisée et d'un support s'inscrivant sur le long terme.

Tous ces éléments nous amènent à éviter la complexité du développement d'un prototype à l'aide des ateliers de méta-modélisation tel KERMETA. Les caractéristiques et les fonctionnalités d'ATL répondent à nos critères, nous permettent la réutilisation du méta-modèle AADL et simplifient l'intégration de notre prototype au sein de l'outil OSATE2 (ces deux outils sont construits au dessus de la plate-forme ECLIPSE). Enfin, le langage ATL nous permet aussi d'assurer notre dernier objectif de production automatisée du code source du système.

© 2012 Gilles LASNIER

2.7 Synthèse

Ce chapitre, divisé en deux parties, nous a permis de traiter de l'état de l'art de deux domaines différents mais en relation, à savoir : l'ingénierie des systèmes critiques et la transformation de modèles.

Dans la première partie, nous avons présenté les différentes *briques* requises (intergiciel critique, approches par composants, validation, déploiement et configuration...) et leur rôle pour la conception et l'analyse de systèmes TR^{2}E critiques. Nous avons vu, lors de l'étude de différentes méthodes intégrées pour la conception de systèmes critiques, le rôle central du "modèle" et son implication au sein du processus de conception. Ce dernier est supporté par un ensemble de méthodes, de standards et technologies définis dans le spectre large de l'*Ingénierie Dirigée par les Modèles* (IDM).

Si l'apport de l'IDM a été conséquent à tous niveaux du cycle de développement logiciel (spécifications et traçabilité des exigences, automatisation de tâches manuelles critiques...), les différentes abstractions pertinentes du système introduisent un biais important entre les analyses des exigences fonctionnelles, non-fonctionnelles et du comportement réalisées *à-priori* sur le système et vis-à-vis de son implantation physique. Ceci se traduit par une différence sémantique voir comportementale plus ou moins importante dûe à la différence de niveau d'abstraction défini pour la modélisation des composants architecturaux et comportementaux utilisés lors des analyses de faisabilité et de performance et celui requis pour leur traduction dans un langage de programmation traditionnel (comme Ada ou C).

Ainsi, nous nous sommes intéressés à différentes approches basées sur la modélisation des systèmes critiques afin de déterminer les limites introduisant ce biais. Ces limitations sont rappelées et détaillées dans le chapitre 3 de ce manuscrit.

La transformation de modèles est une composante fondamentale de l'IDM permettant la manipulation de modèles. A travers un bref état de l'art, nous avons présenté les concepts généraux et les problématiques adressées par ce domaine. Ainsi, nous avons retenu des éléments pertinents à intégrer au sein de notre approche en réponse aux objectifs et aux limitations que nous avons présentés respectivement en introduction et dans la première partie de ce chapitre. Dans le chapitre suivant, nous détaillons le rôle et l'intervention de ces éléments dans notre solution.

© 2012 Gilles LASNIER

© 2012 Gilles LASNIER

Chapitre 3

Problématique et Approche Proposée

SOMMAIRE

3.1 Introduction

Notre problématique générale vise à améliorer la fiabilité du processus d'analyse et d'implantation des systèmes TR²E. Notamment, nous visons la réduction des différences sémantiques entre les modèles d'analyse et le code généré afin de rendre plus cohérent les résultats d'analyse vis-à-vis de l'implantation du système. Dans notre étude préliminaire (chapitre 2) nous avons présenté différents techniques, méthodes et outils mis en œuvre pour la conception, l'analyse et l'implantation de systèmes critiques selon une approche dirigée par les modèles et automatisée.

Cette étude a permis la mise en lumière de certaines limitations nous amenant à proposer, définir et implanter un nouveau processus d'analyse et d'implantation dédié aux systèmes TR²E critiques. Certains aspects pertinents des solutions présentées dans la section 2.4 ont été repris et améliorés au sein de notre approche. Nous avons retenu le formalisme AADLv2

(détaillé chapitre 4) comme langage pour la description architecturale et comportementale des composants logiciels et matériels du système. Ce langage combiné aux techniques de transformation de modèles issues de l'IDM et présentées dans la section 2.5 constitue l'épine dorsale de notre solution.

Ce chapitre est organisé de la façon suivante. Premièrement, nous rappelons et détaillons les limitations identifiées lors de notre étude préliminaire (section 3.2). Ceci nous permet de présenter les enjeux et les solutions apportées par notre proposition (section 3.3). Puis, nous présentons, de manière synthétique, nos différentes contributions pour la mise en œuvre de notre solution (section 3.4). Enfin, nous synthétisons les objectifs et le rôle de chacune de nos contributions en réponse à notre problématique (section 3.5).

3.2 Problématique détaillée et limitations

L'utilisation des modèles et des techniques issues de l'IDM accroît la fiabilité du processus d'ingénierie des systèmes (validation des spécifications, production automatique de code source, etc). Notamment, le pouvoir d'abstraction des modèles permet la définition de représentation pertinente de l'application à des niveaux d'abstraction variés et leur utilisation au sein des différentes étapes du développement logiciel. Cependant, l'utilisation de méthodes et d'outils inadaptés au domaine des systèmes TR^{2}E critiques a entraîné la multiplication de ces niveaux d'abstraction et a complexifié les techniques visant à assurer la cohérence entre ces différentes représentations. A présent, nous nous confrontons à une différence sémantique conséquente entre les modèles d'analyse, les modèles de code et l'implantation du système. De ce fait, les processus et les outils actuels employés expriment leur limite face à cette nouvelle problématique.

Dans cette section, nous rappelons le processus classique de réalisation de systèmes TR^{2}E et l'implication de différents modèles. Puis, nous détaillons les limites découlant de ce processus et des technologies utilisées.

3.2.1 Processus de réalisation de systèmes TR^{2}E

La figure 3.1 détaille le processus classique de réalisation d'un système TR^{2}E selon une approche basée sur l'IDM. Le point d'entrée est la spécification de l'architecture des composants de l'application à l'aide d'un formalisme de description. A partir de cette spécification initiale, un ou plusieurs modèles d'analyse sont élaborés à des niveaux d'abstraction différents pour assurer l'analyse d'un aspect du système (comportement, ordonnancement, flots de données, etc). Les différents résultats produits lors de ces analyses permettent alors de conclure sur la faisabilité du système avant son implantation et valident les spécifications faites en amont du cycle de développement.

Après cette étape de validation, certaines informations architecturales sont extraites pour l'élaboration de modèles de code autorisant le pilotage des différentes étapes de sélection, de configuration et de déploiement des composants nécessaires pour la production d'une partie ou de l'ensemble du code de l'application dans un langage de programmation cible (Ada, C, etc). Cette dernière étape procède aussi à l'intégration des composants intergiciels requis par l'application et des composants applicatifs spécifiés par l'utilisateur.

© 2012 Gilles LASNIER

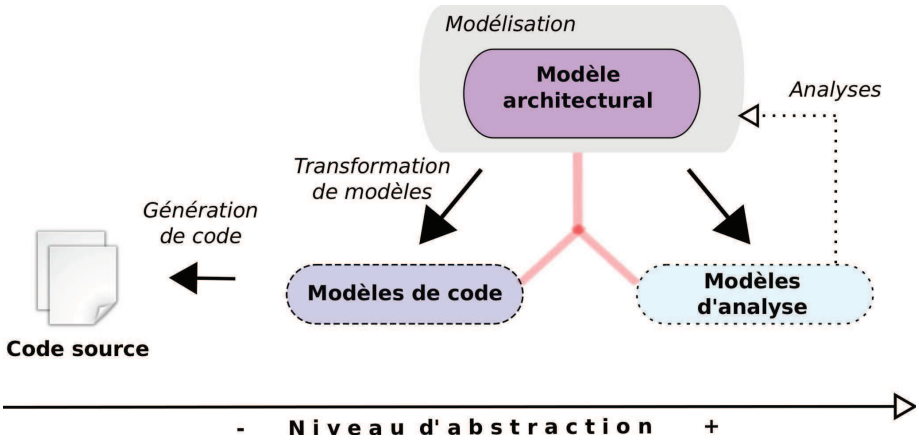


FIGURE 3.1 – Processus classique de conception de systèmes TR^{2}E critiques

3.2.2 Limite 1 : Cohérence des différents modèles

Modèles d'analyse et modèles de code

L'analyse du processus de réalisation de systèmes critiques présenté dans la section précédente (figure 3.1) illustre l'utilisation de transformation de modèles à différents niveaux. Une première transformation exogène est réalisée pour l'élaboration des modèles d'analyse nécessitant parfois un haut niveau d'abstraction afin de limiter la complexité des techniques et outils d'analyse (exemple des méthodes formelles section 2.2.2). Une deuxième transformation exogène ou endogène verticale génère un modèle de code avec un faible niveau d'abstraction autorisant la spécification d'informations relatives à la plate-forme d'exécution et assurant la sélection, la configuration et le déploiement des composants logiciels. Une dernière transformation exogène verticale assure la traduction d'une partie des composants vers les artefacts d'un langage de programmation.

Si ces transformations ont en commun le même modèle source, elles sont d'autant plus complexes qu'elles impliquent des formalismes différents et émettent des hypothèses sur les propriétés des systèmes. Ainsi, chaque transformation peut être réalisée selon des hypothèses différentes, ceci complexifiant la cohérence entre modèles d'analyse et modèles de code mais aussi la mise en œuvre de techniques de traçabilité.

Modèles et implantation

Les processus de conception, étudiés lors de notre état de l'art (section 2.4), proposent une phase de génération de code pilotée par l'analyse de la description architecturale du système critique. Lors de cette phase, de nombreuses constructions sont ajoutées pour le support et l'exécution des composants applicatifs du système. Ces éléments ont un impact non négligeable sur la gestion des ressources et sur le comportement du système. S'ils sont pris en compte lors de différentes analyses du code de l'application (analyses à la compilation ou à l'exécution, tests, etc.), ce n'est pas le cas lors des analyses effectuées sur les modèles, en amont du cycle de développement, relatives à la validation des spécifications. Ceci révèle ce que nous qualifions par manque de cohérence entre le système modélisé et validé et son implantation - *i.e.* le code généré.

© 2012 Gilles LASNIER

La non prise en compte de ces constructions au sein des modèles d'analyse s'explique par leur étroite dépendance vis-à-vis des choix d'implantation portant sur la plate-forme d'exécution, les mécanismes de communication et le langage de programmation. En effet, l'intégration de ces composants dès les premières phases d'analyse nécessite l'élaboration de descriptions architecturales et comportementales des composants du système avec un faible niveau d'abstraction (niveau de détail élevé). Ceci pourrait rendre l'extraction des informations pertinentes par les outils d'analyse plus difficile et à terme de complexifier ces analyses. Enfin, les formalismes de description architecturale utilisés pour la génération de code ne permettent de spécifier explicitement qu'une partie de ces informations et ne tiennent pas compte du comportement de l'application.

Proposition

Les éléments que nous venons de présenter plus haut illustrent la difficulté actuelle d'assurer la cohérence entre les différents modèles et l'implantation du système. Nous observons que le formalisme de description utilisé et le processus de conception proposé sont limités voir inadaptés aux contraintes des systèmes TR^{2}E critiques.

La solution que nous proposons vise à fournir un unique modèle de code analysable. Ce modèle est réalisable à l'aide du langage AADLv2 et de ses récentes avancées. Pour cela, nous identifions ces constructions additionnelles et nous les intégrons, par transformation de modèles verticale (avec préservation de la sémantique), aux descriptions architecturale et comportementale tout en assurant l'analysabilité du système et le respect des spécifications initiales. La mise en évidence explicite de ces composants et de leur comportement nous permettrait alors de diminuer les différences sémantiques entre modèles et code généré et, par conséquent, de renforcer la cohérence entre les résultats des analyses sur modèles et les vérifications effectuées sur le code généré.

La préservation de l'analysabilité du système est notamment assurée via la définition de contraintes de modélisation (patrons de modélisation, restrictions des composants architecturaux, etc.) conformes aux recommandations du profil architectural Ravenscar et aux restrictions classiques sur les systèmes critiques (présentées section 2.3.2). Ces restrictions font l'objet de la définition du langage intermédiaire AADL-HI Ravenscar (un sous-ensemble d'AADL et de son annexe comportementale) présenté dans la section 3.4.2 de ce chapitre et détaillé dans le chapitre 5.

3.2.3 Limite 2 : Prise en compte des aspects comportementaux

Limitations des ADLs et des DSMLs usuels

Lors de l'étude des processus d'analyse et d'implantation de systèmes critiques (section 2.4), nous avons présenté le rôle et l'utilisation de spécifications comportementales pour valider le comportement attendu du système. Les formalismes de descriptions architecturales ne permettent pas d'expliciter concrètement le comportement d'un composant et son analyse (par exemple, le comportement est implanté dans un formalisme externe et est attaché au composant à l'aide d'un mécanisme de type boite noire). Ainsi, nous devons recourir à l'utilisation d'un formalisme tiers (basé sur les automates, les graphes, les flots, etc) pour expliciter et analyser les différents aspects comportementaux. Dans un processus de réalisation automatisé, les langages de description d'architecture actuels ne sont pas conçus pour expliciter ces aspects et imposent alors à l'utilisateur d'établir des règles de transformation complexes et

© 2012 Gilles LASNIER

d'effectuer des hypothèses fortes sur l'architecture pour traduire le comportement d'un composant. Ceci introduit de nouveaux facteurs limitants comme la difficulté de représentation et l'assurance de la cohérence entre la description architecturale et la spécification comportementale du système. Ces derniers points empêchent notamment la réutilisation de ces spécifications comportementales pour la phase de génération de code.

Proposition

L'utilisation des descriptions comportementales et leur mise en relation avec les composants architecturaux pour les étapes d'analyse et de génération de code renforceraient la cohérence entre modèles et implantations. Notre solution vise à fournir des descriptions comportementales explicites et incluses au sein même de la description architecturale. Le langage AADL et son annexe comportementale autorisent la spécification explicite du comportement des composants (matériel ou logiciel) sous la forme d'automates attachés aux composants architecturaux. De plus, les actions spécifiées au sein d'un automate se refèrent directement à ces composants architecturaux. Cette combinaison renforce la cohérence entre les représentations architecturales et comportementales mais aussi simplifie la traduction des spécifications comportementales vers un formalisme dédié à l'analyse comportementale.

Notre implication au sein du comité de rédaction AADL nous a permis d'introduire certaines constructions dans le langage d'action de ces automates facilitant la manipulation d'éléments architecturaux tels les données et les sous-programmes. Ces constructions proches de celles d'un langage de programmation impératif simplifient le processus de génération code en autorisant dans la plupart des cas un *mapping* simple (voir sémantiquement équivalent) des éléments architecturaux modélisés vers les artefacts du langage de programmation cible. Nous présentons, dans le chapitre 4, l'ensemble de ces contributions.

3.2.4 Limite 3 : Analyse partielle du support d'exécution

Les limitations précédentes ont indirectement soulevé une autre limite de nombreuses approches pour la réalisation de systèmes critiques. Les approches notamment basées sur l'outil OCARINA révèlent l'intégration des composants intergiciels relatifs au support et à l'exécution des composants applicatifs uniquement lors de la phase de production du code source.

Outre l'incohérence engendrée entre modèle et code généré, cette limite révèle que les analyses effectuées sur les modèles ne tiennent pas compte des ressources des composants de l'intergiciel requis par l'application TR^{2}E critique. Or, l'impact de ces ressources sur la qualité de service, l'ordonnancement, le pire temps d'exécution ou encore l'empreinte mémoire n'est pas négligeable allant même à rendre le système irréalisable ou défaillant.

La prise en compte de ces composants dans la mesure du possible lors de différentes analyses effectuées sur les modèles est un gain conséquent dans le processus de développement logiciel (validation, prototypage, etc.), ceci moyennant tout de même un effort et un coût supplémentaire lors des différentes étapes de modélisation et de réalisation d'outils.

Proposition

La solution que nous proposons (présentée figure 3.2) est en adéquation avec celles présentées pour les limites 1 et 2. Les améliorations récentes du langage AADLv2 en terme de modélisation de composants architecturaux (matériels et logiciels) et des aspects comportementaux permettent, maintenant, la modélisation des élements cœurs d'un exécutif pour les

composants applicatifs AADL - *i.e.* l'intergiciel. Par le biais d'un processus incrémental, il est alors possible de *raffiner* la spécification architecturale et comportementale (implicite) initiale de l'utilisateur pour réduire son niveau d'abstraction et autoriser l'intégration des composants intergiciels. Les analyses s'effectueront alors sur un modèle de l'application critique *complet*, intégrant l'applicatif et l'exécutif (intergiciel) AADL et non plus uniquement sur la partie applicative du système.

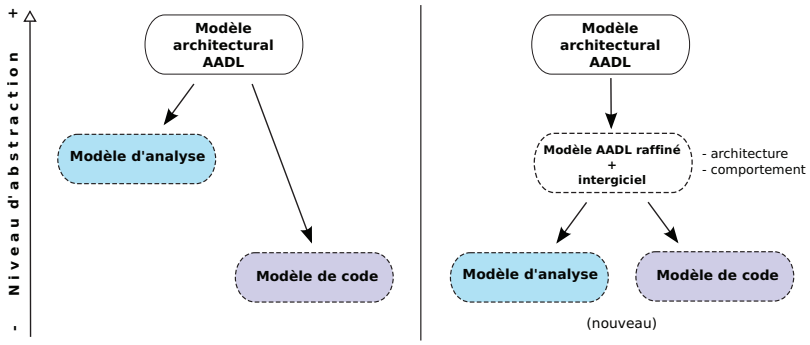


FIGURE 3.2 – Processus de raffinement du modèle AADL

3.3 Présentation de l'approche proposée

Dans les sections précédentes, nous avons identifié certaines limites récurrentes pour la réalisation et la validation de systèmes critiques. Les différentes propositions pour palier ces limites nous amènent à la définition d'un nouveau processus de conception et d'analyse de systèmes TR^{2}E critiques. Dans cette section, nous présentons et détaillons les grandes étapes de notre solution.

3.3.1 Architecture du processus d'analyse et de production

La figure 3.3 décrit l'architecture générale de notre processus d'analyse et de production automatique de systèmes critiques. Nous distinguons deux phases. Une première phase vise à raffiner, par transformations verticales, le niveau d'abstraction de la description architecturale AADL du système jusqu'à l'obtention d'un modèle de code analysable complet (applicatif + intergiciel). Puis, l'implantation du système est réalisée par transformation horizontale renforçant ainsi la cohérence entre modèles et code généré et facilitant la mise en oeuvre de techniques de traçabilité.

Notre processus se divise en 4 étapes :

1. Spécification des composants architecturaux du système à l'aide du langage AADL et validation du modèle AADL à l'aide des outils AADL existants.
2. Raffinement incrémental de la description architecturale initiale par transformations verticales préservant et/ou précisant la sémantique et le comportement initial du système. Intégration des composants intergiciels.
3. Validation du modèle de code analysable obtenu - *i.e.* le modèle raffiné.
4. Production automatique du code source de l'application par transformation de modèles horizontale.

Nous détaillons ces différentes étapes dans les sous-sections suivantes.

© 2012 Gilles LASNIER

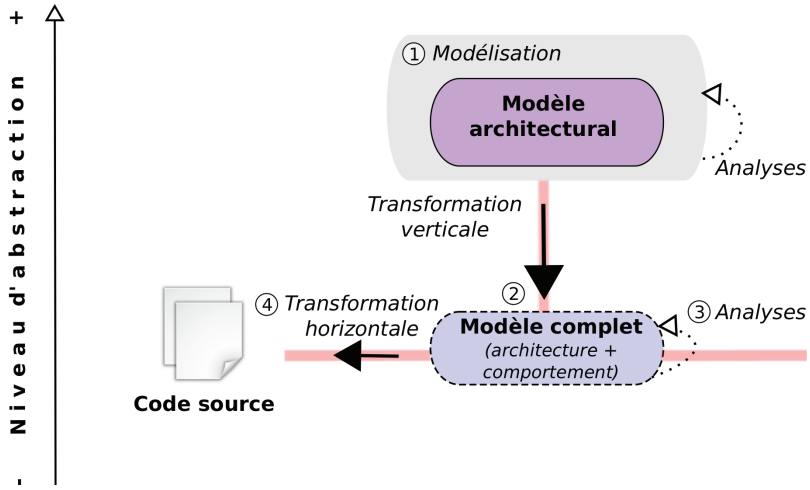


FIGURE 3.3 – Nouveau processus de conception de systèmes TR^{2}E critiques

3.3.2 Spécification et validation de modèles AADL

Spécification AADL

Le point d'entrée de notre approche est une description architecturale de l'application spécifiée par l'utilisateur. Celle-ci doit suivre les recommandations du standard AADLv2. L'annexe de modélisation des données est requise pour la description des types de données et l'annexe comportementale peut être utilisée pour décrire le comportement des composants. La description architecturale considérée est la modélisation *finale* du système vue par l'utilisateur - *i.e. ne comprenant que des composants concrets dont la spécification est complète et tels qu'ils seront physiquement déployés*.

Validation de la description initiale

A partir de cette spécification finale, une première étape de validation est réalisée. Dans un premier temps, nous vérifions la syntaxe et la sémantique des composants à l'aide de l'outil OSATE2. Puis, nous analysons l'ordonnancement de la partie applicative du système à l'aide de l'outil CHEDDAR [Singhoff *et al.*, 2004]. Enfin, nous vérifions les restrictions architecturales que nous avons définies selon le profil Ravenscar et les recommandations d'usage pour la conception des systèmes critiques (présentées chapitre 4). A l'issue de ces analyses, si une anomalie est détectée, celle-ci est retournée à l'utilisateur pour qu'il puisse modifier sa description initiale. Dans le cas contraire, nous pouvons procéder au raffinement incrémental du modèle.

3.3.3 Raffinement incrémental de la spécification

La seconde étape de notre processus est le raffinement incrémental de la description architecturale initiale. Les transformations incrémentales endogènes nous assurent la modification d'une partie ciblée des composants du système et nous évitent la réécriture du modèle complet. La réalisation de transformations verticales visent à réduire le niveau d'abstraction des composants AADL ayant une forte abstraction vers un ou plusieurs composants dont la sémantique est plus proche des constructions d'un langage de programmation impératif. Lors de

© 2012 Gilles LASNIER

ces transformations verticales, la sémantique et le comportement des composants sont préservés et/ou précisés. La réduction du niveau d'abstraction nous permet alors l'intégration des composants intergiciels requis par l'application. L'issue de cette étape est l'élaboration de notre modèle de code analysable complet - *i.e. comprenant composants applicatifs et intergiciels*.

Dans le chapitre 5, nous discutons du niveau d'abstraction requis pour notre modèle de code analysable. La définition des composants intergiciels fait l'objet du chapitre 6. Enfin, les différentes transformations du processus de raffinement sont explicitées dans le chapitre 7.

3.3.4 Validation du modèle complet

La troisième étape est une nouvelle phase de validation portant, cette fois-ci, sur le modèle de code complet analysable obtenu à l'étape précédente. Cette étape nous assure que le modèle AADL architectural et comportemental obtenu reste *légal* syntaxiquement et sémantiquement vis-à-vis du standard.

Lors de cette étape, nous menons des analyses additionnelles sur l'ordonnancement du système complet intégrant à présent les ressources intergicielles. Ces résultats sont, ensuite, mis en comparaison avec ceux issus de la première étape de validation (décrite plus haut, sans les composants intergiciels et sans le raffinement). Ainsi, nous pouvons nous assurer de la faisabilité du système et du respect des exigences initiales. La prise en compte des composants intergiciels permet, dans certains cas, d'affiner les résultats d'analyse (ordonnancement, comportement, etc.) ou de détecter des anomalies de conception.

Remarque. L'introduction d'analyses additionnelles (estimation du pire temps d'exécution, dimensionnement mémoire) sont réalisables à ce niveau de modélisation. En effet, le raffinement permet d'obtenir l'ensemble des types et des transformations de données et les différents chemins d'exécution possible des composants logiciels threads et sous-programmes.

3.3.5 Production du système final

La dernière étape de notre approche est la production automatisée du code source de l'application à partir du modèle de code complet. La réécriture d'une partie des composants et l'intégration des aspects comportementaux réalisées à l'aide de l'annexe comportementale, réalisées par notre processus de raffinement, simplifient notre processus de génération. Chaque élément du modèle de code est ainsi traduit en une ou plusieurs constructions du langage Ada sémantiquement équivalentes (expression d'un *mapping* 1-1 entre un composant et un élément de langage). Le processus de génération est implanté à l'aide de transformations horizontales exogènes dont l'avantage, vis-à-vis des techniques traditionnelles, est l'expression de règles de transformation facilitant la traçabilité, la compréhension, l'évolution et la maintenance du générateur. Le chapitre 8 présente l'ensemble des règles de transformation pour la production automatique du code source des composants de l'application.

A la fin de cette étape, nous obtenons l'exécutable final prêt à être déployé sur sa plateforme cible. Une nouvelle phase d'analyse est menée notamment sur la vérification des restrictions Ravenscar à l'aide du compilateur GNAT. Des analyses additionnelles sur l'empreinte mémoire de l'exécutable final ou l'estimation du WCET (à l'aide de l'outil BOUND-T) peuvent être réalisées et mises en comparaison avec les analyses effectuées sur les modèles, validant ainsi les spécifications initiales de l'utilisateur.

La section suivante synthétise nos différentes contributions pour la mise en oeuvre de notre processus de conception et d'analyse de systèmes critiques.

© 2012 Gilles LASNIER

3.4 Identification des contributions

Cette section présente, de manière synthétique, nos différentes contributions (standard, méthodologie, bibliothèques de composants, outils, etc.) réalisées dans le cadre de ces travaux de recherche.

3.4.1 Contributions à l'annexe comportementale (AADL-BA)

Notre équipe, évoluant au sein de la communauté AADL, participe activement au développement de plusieurs annexes AADL dont l'annexe comportementale (pour sa dernière version). Ainsi, nous avons participé aux différentes phases (rédaction, validation, publication) du standard et nous avons pu suivre la maturation de ce dernier.

Dans le cadre de notre approche, l'annexe comportementale nous permet de définir des patrons comportementaux analysables et de décrire les aspects comportementaux des composants de l'intergiciel supportant les composants applicatifs AADL. Ainsi, ces spécifications renforcent et/ou précisent la sémantique des composants du système distribué notamment en explicitant la concurrence, les entités actives pour la gestion de la communication, etc. De plus, les constructions de l'annexe (langage d'action, langage d'expression) expriment une sémantique proche des artefacts d'un langage de programmation nous autorisant une réutilisation simple et efficace des spécifications comportementales au sein du processus de génération.

Pour permettre une telle démarche, notre première contribution a été la proposition et la précision de certaines constructions de l'annexe comportementale portant essentiellement sur les langages d'action et d'expression (présentées chapitre 4, section 4.3.6) . Ainsi, nous avons autorisé et/ou amélioré les supports pour :

- la manipulation des données et l'utilisation des données dans les appels de sous-programmes ;
- la manipulation des données décrites à l'aide d'une propriété AADL (valeur d'énumération, champs d'une structure) ;
- le support des séquences d'appel de sous-programmes ;
- le support des prototypes.

La combinaison de ces langages offre un ensemble de constructions impératives d'un langage de programmation (boucles, manipulation de variables, etc.) (voir chapitre 4) essentiel à la définition de nos patrons de génération de code détaillés dans le chapitre 8.

Notre deuxième contribution a été l'implantation du standard AADL-BA au sein de la plateforme OSATE2. Le compilateur développé nous assure la validation des spécifications comportementales élaborées dans le cadre de notre approche. De plus, il nous a notamment permis de révéler des incohérences de l'annexe qui ont été discutées et corrigées par les acteurs de la communauté AADL lors de différents comités. Il s'agit de précision sémantique sur la construction *timeout* et son utilisation, de précision sémantique sur les éléments pour le verrouillage d'une ressource (modélisation d'une section critique) et du support des tableaux en général.

Intéressons-nous maintenant aux restrictions des composants architecturaux et comportementaux pour la définition d'un sous-ensemble du langage AADL analysable.

3.4.2 Le profil AADL-HI Ravenscar, un langage intermédiaire

Nos objectifs en réponse aux limites 1 et 3 (section 3.2) visent à réduire le niveau d'abstraction d'une description architecturale AADL pour obtenir un modèle de code analysable et

© 2012 Gilles LASNIER

complet avec une sémantique de composants la plus proche possible d'un langage de programmation. Ce modèle de code doit à la fois contenir des informations (fonctionnelles et non fonctionnelles) pour permettre la vérification du système par les outils d'analyse et permettre la production automatique du code source des composants.

Le langage AADL introduit des mécanismes et des constructions (composants abstraits, héritage, raffinement, prototypes, détaillés dans le chapitre suivant) ajoutés afin de faciliter la description pour l'utilisateur mais nécessitant une interprétation ou une transformation complexifiant l'extraction d'informations par les outils d'analyse ou les générateurs de code.

Pour mener à terme nos objectifs, il est nécessaire d'éliminer ces constructions par réécriture d'une partie des composants et de garantir l'analysabilité de l'ensemble des composants du modèle final. Par conséquent, nous avons défini le profil [1] AADL-HI Ravenscar selon les critères suivants :

C1. Restreindre l'ensemble des composants architecturaux AADL et leur utilisation aux seules constructions concrètes pour l'analyse et la génération de code.

C2. Restreindre l'ensemble des constructions comportementales aux seules constructions autorisant l'analyse et une démarche de génération.

C3. Contraindre les composants retenus selon les recommandations du profil Ravenscar et les restrictions additionnelles pour la conception de systèmes critiques afin de garantir l'analysabilité statique du système.

Le profil AADL-HI Ravenscar définit le niveau d'abstraction, les composants, les restrictions et les patrons de modélisation utilisés pour l'élaboration de notre modèle de code analysable. Celui-ci est présenté dans le chapitre 5 de ce manuscrit. Dans la section suivante, nous présentons le modèle d'intergiciel que nous avons élaboré pour compléter la modélisation du système critique.

3.4.3 Le modèle d'intergiciel POLYORB-HI-AADL

Notre troisième contribution est la définition d'une bibliothèque de composants intergiciels AADL (POLYORB-HI-AADL) analysables, supportant les composants logiciels (applicatifs) AADL. Pour ce faire, nous avons repris le concept "d'instance d'intergiciel dédié" introduit dans l'approche OCARINA et nous nous sommes basés sur l'architecture de l'intergiciel POLYORB-HI. Les composants ont été modélisés selon le profil AADL-HI Ravenscar. Le modèle d'intergiciel que nous proposons permet à la fois la prise en compte des composants intergiciels au sein des analyses (en réponse à la limite 3) mais aussi lors de notre processus de génération de code, réduisant ainsi le saut sémantique entre modèle et code généré (en réponse à la limite 1). L'architecture de la bibliothèque de composants intergiciels est détaillée dans le chapitre 6 de ce manuscrit.

3.4.4 Chaîne de transformation automatisée

Notre étude préliminaire a permis de mettre en lumière l'importance de l'automatisation de différentes tâches du processus d'analyse et de production de systèmes critiques. Ainsi, notre dernière contribution est la réalisation d'un prototype implantant notre nouveau processus selon des techniques basées la transformation de modèles AADL.

1. La section "Profile and Extensions" du standard AADL autorise la restriction des constructions du langage pour la définition d'un profil.

© 2012 Gilles LASNIER

L'utilisation des techniques de transformation de modèles nous a permis d'exprimer et d'expliciter les différentes transformations de notre processus. L'élaboration des transformations sous la forme d'un assemblage de règles amène une plus grande flexibilité et facilite la maintenance et l'évolution de notre outil.

Ainsi, ce prototype assure le raffinement incrémental, l'analyse et la validation du modèle AADL et la production du code source de l'application dans le langage Ada/Ravenscar. Les étapes de sélection, configuration et déploiement des composants AADL sont aussi prises en charge par notre prototype.

Les différentes transformations évoquées dans ce chapitre, détaillées dans les chapitres 7 et 8, ont été implantées à l'aide du langage ATL. L'automatisation de la chaîne de transformation a été réalisée à l'aide de l'outil ADT supportant le langage ATL. La mise en œuvre expérimentale de notre solution est détaillée dans le chapitre 9.

3.5 Synthèse

Notre objectif principal vise à améliorer la fiabilité du processus automatisé de réalisation des systèmes TR^{2}E critiques. Dans ce chapitre, nous avons clarifié les problèmes observés dans le chapitre 2 et identifié les limites des processus d'analyse et de production actuels à savoir :

1. le manque de cohérence des différentes représentations du système (modèles et implantation) ;
2. la prise en compte des aspects comportementaux dans les étapes d'analyse et dans le processus de génération ;
3. l'analyse partielle du support d'exécution (ressources intergicielles non prises en compte).

La figure 3.4 décrit le nouveau processus d'ingénierie que nous proposons pour résoudre ces problèmes. Notre approche s'articule autour de plusieurs contributions scientifiques qui visent :

1. la définition d'un profil architectural et comportemental de modélisation de composants dédié à l'analyse et à la génération de code ;
2. la définition d'une bibliothèque de composants intergiciels (à partir du profil précédent) analysables statiquement ;
3. une méthode de raffinement incrémental de modèles assurant la réduction du niveau d'abstraction (préservant la sémantique et le comportement) des composants, l'intégration des composants du support d'exécution et permettant l'élaboration d'un modèle de code analysable complet (applicatif + intergiciel) du système critique ;
4. la production du code source de l'application à partir du modèle de code (selon un *mapping* 1-1), renforcant ainsi la cohérence et la mise en œuvre de techniques de traçabilité ;
5. l'implantation d'un prototype implantant la méthode de raffinement et automatisant les tâches de transformation, d'analyse et de production du code source.

Ce processus nous a permis de résoudre l'ensemble des problèmes que nous avons présenté. La suite de ce manuscrit est consacrée à la description détaillée des contributions scientifiques de nos travaux de recherche. Chacun de points présentés fait l'objet d'un chapitre particulier.

© 2012 Gilles LASNIER

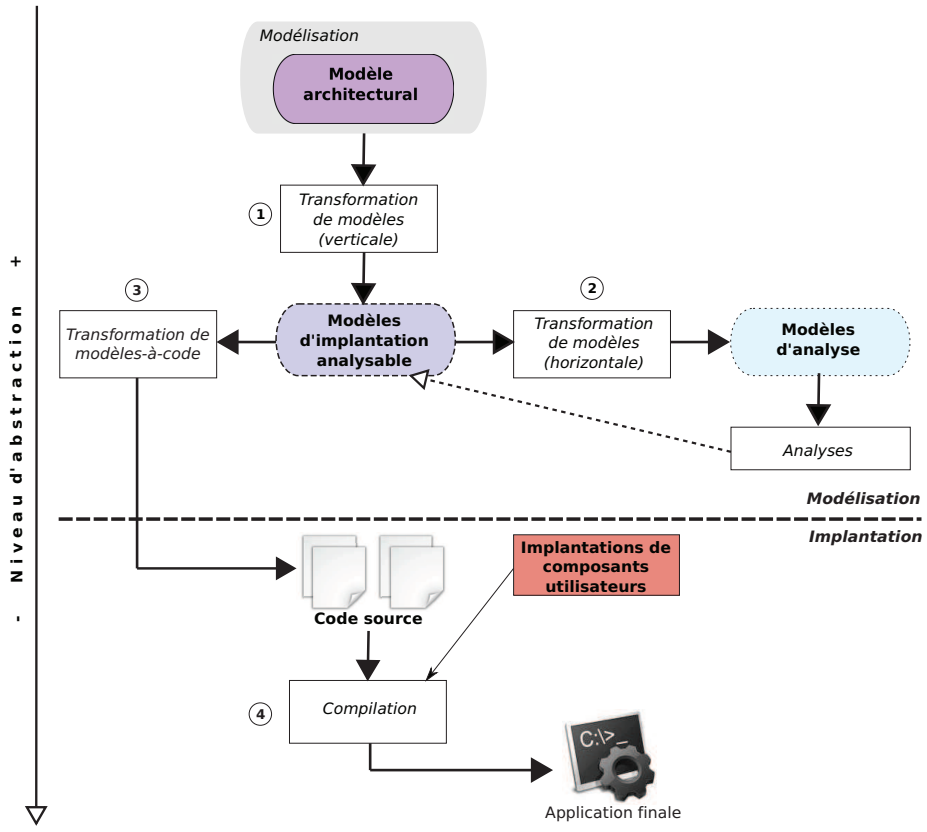


FIGURE 3.4 – Nouveau processus pour l'analyse et la production automatisée de systèmes TR^{2}E critiques

© 2012 Gilles LASNIER

Chapitre 4

Description Architecturale et Comportementale à l'aide du langage AADLv2

SOMMAIRE

4.1 Introduction

Dans les chapitres précédents, nous avons présenté l'utilisation du langage AADL pour la modélisation, la configuration et le déploiement des composants architecturaux des systèmes TR²E critiques. Dans le chapitre 3, nous avons justifié le choix de l'utilisation du langage

AADLv2 et de ses annexes qui introduisent de nombreux éléments et améliorations permettant une description plus fine des composants architecturaux et de leur comportement.

AADL (*Architecture Analysis and Design Language*) est un langage de modélisation d'architecture qui tire ses origines du langage METAH [Vestal, 2000] développé pour les systèmes avioniques et spatiaux. Destiné lui aussi à l'avionique, la richesse d'expression et les nombreuses extensions du langage AADL ont étendu son activité au domaine des systèmes TR^2E logiciels et matériels. La version 2.0 du standard [SAE Aerospace, 2009b] a été publiée en janvier 2009 et une version 2.1 est en cours de rédaction.

Dans ce chapitre, nous proposons une introduction au langage AADLv2 et à son annexe comportementale (*Behavior Annex* [SAE Aerospace, 2009a]), dont un sous-ensemble d'éléments constitue notre épine dorsale pour la description logicielle architecturale et comportementale d'une application répartie avec une sémantique très proche de son implantation physique.

Ce chapitre s'organise de la manière suivante. La section 4.2 présente les éléments du langage cœur (composants matériels, logiciels, connecteurs, propriétés, etc). La section 4.3 est une introduction aux différents langages définis par l'annexe comportementale et présente nos contributions pour le développement et l'écriture de ce standard. La section 4.4 présente les avantages et les restrictions pour la modélisation des systèmes critiques. Enfin, la section 4.6 conclut ce chapitre.

Remarque. Une partie des illustrations de ce chapitre est issue de la description architecturale de notre cas d'étude détaillé dans le chapitre 10.

4.2 Eléments du langage cœur

AADLv2 permet de décrire les systèmes TR^2E par un assemblage de composants matériels et logiciels connectés. Les propriétés AADL complètent la modélisation du système en autorisant l'attachement d'exigences fonctionnelles (fonction de calcul...) ou non fonctionnelles (contraintes temporelles, informations de déploiement...). Les annexes étendent le pouvoir d'expression du langage en définissant des règles, des recommandations, des patrons de modélisation ou des langages différents (ou externes) pour couvrir différents aspects du système que ce soit à des fins de représentation, d'analyse, d'implantation, etc.

Intéressons-nous à l'élément majeur d'une description architecturale AADL : le composant.

4.2.1 Composants

La première version du langage [SAE Aerospace, 2004] présentait le langage AADL comme un langage classique de description d'architecture (voir la classification [Medvidovic and Taylor, 2000]) construit sur les trois concepts fondamentaux : composants, connecteurs, connexions. Le langage AADLv2 apporte différentes améliorations (un méta-modèle, représentation graphique standardisée, profils, extensions...) caractéristiques des langages de modélisation spécifique à un domaine.

AADLv2 définit des composants matériels et logiciels du système spécifiés par une interface (**component_type**) décrivant les éléments d'interface pour la connexion inter-composants et une implantation (**component_implementation**) spécifiant la structure interne du composant. Un agrégat (ou composition) de composants peut décrire un composant établissant ainsi une hiérarchie composant $\rightarrow$ sous-composants structurant la description architecturale. AADL

© 2012 Gilles LASNIER

propose deux catégories principales de composants : les composants matériels et logiciels et des composants spécifiques pour la composition du système (**system**) ou pour les premières étapes de la modélisation (**abstract**, voir 4.2.7).

Composants matériels, logiciels et système

AADL définit les composants matériels suivants :
- le **device** représente tous les types de périphériques (clavier, souris, capteur, etc.) sous la forme d'une boite noire exprimant ses interfaces mais aucun détail sur son implantation ;
- le **processor** modélise un contrôleur, un processeur ou un système d'exploitation minimal (ordonnanceur, pilotes, etc.) pouvant accéder à des ressources matérielles (périphériques, bus, etc.) ;
- le **virtual processor** spécifie un cœur, un ordonnanceur, une machine virtuelle ou une partition permettant de représenter la structure interne d'un processeur physique ;
- le **bus** modélise les accès entre les processeurs, les périphériques et les mémoires. L'association des connexions ou d'accesseurs permet de décrire le flot de contrôle ou la stratégie de déploiement d'une application distribuée ;
- le **virtual bus** modélise la structure interne d'un bus (qualité de service, protocole, etc.) ;
- le composant **memory** modélise une mémoire physique quelconque (ROM, RAM, disque dur, etc.) pour le stockage de données.

AADL définit les composants logiciels suivants :
- le composant **data** représente des types de données, des instances de données ou de variables partagées ;
- le **process** modélise l'espace d'adressage pour l'exécution des tâches (comme un processus LINUX) ;
- le **thread** modélise un fil d'exécution qui constituent la partie applicative de l'application ;
- le **subprogram** modélise le code (C ou Ada) exécuté par une tâche et dont le comportement est spécifié à l'aide de propriétés, d'annexes ou de séquences d'appels à d'autres sous-programmes.

Enfin, le composant **system** décrit l'assemblage final (ou une configuration) des différents composants logiciels et matériels de l'application distribuée. Il structure et hiérarchise l'assemblage de composants, c'est en quelque sorte la racine du modèle.

Composition

Le standard AADL autorise l'agrégation de composants et la définition de sous-composants selon des règles sémantiques précises. Le tableau 4.1 présente les relations légales entre composants et sous-composants découlant de la logique de composition de l'application. Par exemple, un composant de donnée ne peut pas contenir un processus ou un processeur ; cela ne correspond à aucun assemblage réel.

4.2.2 Eléments d'interfaces et connexions

Les éléments d'interfaces des composants (spécifiés au sein d'un **component_type**) modélisent les flôts de contrôles ou de données. Ils décrivent les mécanismes d'interaction pour l'échange et/ou la synchronisation de données entre les éléments de l'architecture. La communication est alors établie à l'aide des connexions reliant deux éléments d'interfaces. AADL

© 2012 Gilles LASNIER

Composant	Sous-composants
abstract	data, subprogram, subprogram group, thread, thread group, process, processor, virtual processor, memory, bus, virtual bus, device, system, abstract
data	data, subprogram, abstract,
subprogram	data, abstract
subprogram group	subprogram group, subprogram, data, abstract
thread	data, subprogram group, subprogram, abstract
thread group	data, subprogram group, subprogram, thread group, thread, abstract
process	data, subprogram group, subprogram, thread group, thread, abstract
processor	virtual processor, memory, bus, virtual bus, abstract
virtual processor	virtual processor, virtual bus, abstract
memory	memory, bus, abstract
bus	virtual bus, abstract
virtual bus	virtual bus, abstract
device	bus, virtual bus, data, abstract
system	data, subprogram, subprogram group, process, processor, virtual processor memory, bus, virtual bus, device, system, abstract

TABLE 4.1 – Composition légale des composants AADL

définit deux catégories concrètes d'éléments d'interfaces, le **port** et le **parameter**. Une troisième catégorie, les **features**, sert lors la phase préliminaire de la modélisation et est destinée à être raffinée vers l'une des catégories concrètes.

Les ports de type **data** et les paramètres de sous-programmes modélisent l'échange d'une donnée, les ports de type **event** (événements) modélisent l'émission et la réception de signaux. Un sens (**in**, **out**, **in out**) caractérise la nature de l'élément (entrée/sortie).

Le listing 4.1 illustre l'utilisation des éléments d'interfaces et des connexions. Le processus `Transmitter.Impl` contient un thread `Transmitter_Thread.Impl` qui exécute le sous-programme `Transmit`. Le port en entrée du processus est connecté à celui du thread, lui-même connecté au paramètre d'entrée du sous-programme. Le paramètre de sortie du sous-programme est connecté au port en sortie du thread, lui-même connecté au port en sortie du processus.

Les accesseurs de composants (**provides** ou **requires access**) permettent de spécifier les accès aux sous-composants ou les relations entre les composants de données partagées et les sous-programmes pour leur manipulation. Le listing 4.2 décrit le patron de modélisation d'une variable partagée et les sous-programmes pour sa manipulation. Le thread `Producer` qui exécute le sous-programme `Update` déclare requérir l'accès à la donnée partagée `Shared_Data`. Le composant processus spécifiant le thread `Producer` effectue la connexion entre la donnée partagée et le thread.

4.2.3 Propriétés et Annexes

Le langage AADL offre deux mécanismes pour enrichir et étendre les capacités d'expression ou de modélisation du langage cœur : les annexes et les propriétés.

© 2012 Gilles LASNIER

Exemple 4.1 – Connexions entre composants AADL

```
data Message
end Message ;


subprogram Transmit
features
  Input : in parameter Message ;
  Output : out parameter Message ;
end Transmit ;


thread Transmitter_Thread
features
  Input : in event data port Message ;
  Output : out event data port Message ;
end Transmitter Thread ;

thread implementation Transmitter_Thread.Impl
calls
Mycall : { Do_Transmit : subprogram Transmit ; } ;
connections
  p1 : parameter Input -> Do Transmit.Input ;
  p2 : parameter Do Transmit.Output -> Output ;
end Transmitter_Thread.Impl ;


process Transmitter_Process
features
  Input : in event data port Message ;
  Output : out event data port Message ;
end Transmitter_Process ;

process implementation Transmitter_Process.Impl
subcomponents
  Transmitter : thread Transmitter_Thread.Impl ;
connections
  c1 : port Input -> Transmitter.Input ;
  c2 : port Transmitter.Output -> Output ;
end Transmitter_Process.Impl ;
```

Exemple 4.2 – Accesseurs et patron de modélisation des variables partagées en AADL

```
data Shared_Data
features
  Read   : provides subprogram access Read ;
  Update : provides subprogram access Update ;
properties
  Concurrency_Control_Protocol => Priority_Ceiling ;
  Deployment::Priority => 240;
end Shared_Data;


subprogram Read
 features
  Shared_Data : requires data access Shared_Data;
end Read;


subprogram Update
  features
   Shared_Data : requires data access Shared_Data;
end Update;


thread Producer
features
  Shared_Data : requires data access Shared_Data;
properties
  Dispatch_Protocol                => Periodic;
  Period                           => 1000 ms;
  Compute_Execution_time           => 0 ms .. 600 ms;
  Deadline                         => 1000 ms;
  Deployment::Priority => 11;
end Producer;

thread implementation Producer.Impl
calls call_seq : { Do_Update : subprogram Update; };
connections
  Access_Data : data access Shared_Data -> Do_Update.Shared_Data;
end Producer.Impl;


thread Consumer
features
  Shared_Data : requires data access Shared_Data;
  Dispatch_Port : in event port Base_Types::Integer;
properties
  Dispatch_Protocol                => Sporadic;
  Period                           => 600 ms;
  Compute_Execution_time           => 0 ms .. 100 ms;
  Deadline                         => 600 ms;
  Deployment::Priority => 11;
end Consumer;

thread implementation Consumer.Impl
calls call_seq : { Do_Read : subprogram Read; };
connections
  Access_Data : data access Shared_Data -> Do_Read.Shared_Data;
end Consumer.Impl;


process Simple_Producer_Consumer
end Simple_Producer_Consumer;

process implementation Simple_Producer_Consumer.Impl
subcomponents
  Producer    : thread Producer.Impl;
  Consumer    : thread Consumer.Impl;
  The_Shared_Data : data Shared_Data;
connections
  Cnx1 : data access The_Shared_Data -> Producer.Shared_Data;
  Cnx2 : data access The_Shared_Data -> Consumer.Shared_Data;
end Simple_Producer_Consumer.Impl;
```

© 2012 Gilles LASNIER

Annexes

Les annexes permettent d'intégrer à tout composant du modèle d'une application des éléments d'un langage tiers. Ceci permet à la fois d'enrichir la description, de décrire un aspect manquant du système et d'interfacer un modèle AADL avec des outils tiers exploitant la spécification architecturale. Aujourd'hui, plusieurs annexes ont été développées et sont standardisées ou en cours de standardisation. Parmi les plus connues, nous pouvons citer :

- l'annexe de modélisation des données (*Data Model Annex* [SAE Aerospace, 2009d]), qui décrit les patrons de modélisation pour les types de données basiques (nous présentons les différents patrons de modélisation de cette annexe dans le chapitre 8) ;
- l'annexe comportementale (*Behavior Annex* [SAE Aerospace, 2009a]), qui décrit le comportement des composants. Celle-ci est détaillée dans la section 4.3 de ce chapitre ;
- l'annexe de modélisation des erreurs (*Error Model Annex* [SAE Aerospace, 2011]), qui spécifie les erreurs, leur propagation et leur gestion au sein des composants ;
- l'annexe REAL (*Requirements Enforcement Analysis Language* [Gilles, 2008]), qui définit un langage de vérification de contraintes sur les composants AADL.

Le listing 4.4 décrit le comportement d'un composant thread spécifié à l'aide de l'annexe comportementale.

Propriétés

Les propriétés AADL autorisent la spécification d'informations (ou d'annotations) sur un élément AADL (composants, connexions, etc.). Il s'agit d'attributs spécifiant des caractéristiques ou des contraintes s'appliquant aux éléments de l'architecture : contraintes temporelles (pire temps d'exécution, période...), information de déploiement (adresse IP, port de communication), caractéristiques réseaux (bande passante d'un bus...). Le standard AADL définit un ensemble de propriétés standards et offre la possibilité de définir des propriétés spécifiques à l'aide des **property sets**. Le thread `Producer` décrit dans le listing 4.2 (présenté plus haut), spécifie des propriétés temporelles sur son type, sa période, son pire temps d'exécution, etc.

4.2.4 Flots et systèmes adaptatifs avec modes

Le langage AADL introduit aussi des contructions pour la spécification des flux de données et des différentes configurations possibles d'un système adaptatif avec modes.

Les flots AADL permettent de préciser explicitement les flots de données et d'exécution dans un modèle. Cette fonctionnalité vient compléter les analyses de flots de données possibles à partir des éléments d'interfaces des composants, de leur connexion et de leur association au composant bus. Ainsi, les flots AADL permettent de vérifier la cohérence de la topologie de l'application et facilite l'utilisation d'outils d'analyse de flots (les flots sont déjà spécifiés, il n'est pas utile de les déduire de la topologie de l'application).

Les modes AADL permettent l'expression des différentes configurations d'un système. Leur spécification au sein des composants autorise le contrôle des ressources matérielles, logicielles mais aussi des valeurs des propriétés, de l'activation des connexions voir même des sous-composants (par exemple, un système multi-tâche s'exécutant en mode dégradé mono-tâche). La transition d'un mode à un autre s'effectue alors dynamiquement à l'exécution de l'application par la réception d'événements sur les ports de type événement en entrée.

© 2012 Gilles LASNIER

Dans la suite de ce manuscrit, nous ne considérons pas les flots AADL bien que ceux-ci pourraient s'intégrer dans notre processus. De même, nous ne considérons pas la notion de mode. Des travaux au sein de notre équipe ([Borde, 2009]) ont montré que l'analyse et l'implantation des systèmes reconfigurables avec modes nécessitent un processus d'ingénierie spécifique à cette famille de systèmes.

4.2.5 Instanciation d'un modèle AADL

Dans une description architecturale AADL, la plupart des composants peuvent contenir des sous-composants permettant ainsi de définir une description hiérarchique du système. Le composant système est le composant de plus haut niveau contenant toutes les instances des autres composants. Le standard AADL définit la phase d'instanciation d'un modèle AADL, on parle alors de la transformation d'un modèle AADL déclaratif vers un modèle d'instance. Le modèle d'instance donne une vue hiérarchique de la description architecturale AADL à partir d'un nœud racine, le composant système.

L'instanciation d'un système vise à résoudre les références et les valeurs des propriétés des différentes instances de composants comme les calculs et l'héritage des valeurs de propriétés (obtenues par extension ou raffinement de composants), la résolution des prototypes, la résolution bout-en-bout des chemins des connexions des éléments d'interfaces, etc.

4.2.6 Services de l'exécutif AADL

Le standard AADL donne des recommandations sur le comportement et les services que doit fournir un exécutif AADL *i.e.* une runtime supportant les composants logiciels AADL (threads, sous-programmes, etc.). Notamment, le standard spécifie un certain nombre d'interfaces relatives aux services qu'un exécutif AADL doit offrir pour assurer l'émission et la réception des données à travers les composants threads. Nous discuterons plus précisément du rôle de ces services dans le chapitre 6 de ce manuscrit.

4.2.7 Eléments pour la modélisation

AADLv2 introduit différents éléments, présentés dans le listing 4.3, pour structurer les déclarations (paquetages...) et faciliter la description architecturale d'une application répartie (réutilisation, extension, etc.) :

- les **packages** AADL organisent les déclarations des composants permettant entre autres la séparation de la partie matérielle et logicielle et la définition de bibliothèques de composants. Un mécanisme d'importation (**with**) autorise le référencement des composants d'un paquetage ;
- le composant **abstract** permet de modéliser un composant sans connaître son déploiement final. Des mécanismes d'extension et de raffinement de composants permettent plus tard, dans la modélisation, de raffiner un composant abstrait en un composant matériel ou logiciel concret. Ces mêmes mécanismes d'extension et de raffinement permettent la réutilisation des composants ;
- les **prototypes** autorisent la définition de composants génériques qui seront instanciés et configurés plus tard dans la modélisation. L'instanciation et la configuration des prototypes s'effectuent à l'aide du mécanisme d'extension et des **prototype bindings** permettant de typer ou de résoudre (dans le cas des sous-programmes prototypes) le composant prototypé.

© 2012 Gilles LASNIER

Exemple 4.3 – Utilisation des paquetages, des extensions et des prototypes AADL

```
package Compute_Library public

with Base_Types;

subprogram Generic_Spg
  prototypes
    dt      : data;
  features
    input   : in  parameter dt;
    output  : out parameter dt;
end Generic_Spg;

data My_Float extends Base_Types::Float end My_Float;

subprogram Compute_Float extends Generic_Spg (dt => data My_Float)
end Compute_Float;

end Compute_Library;
```

Cette section a présenté les différents composants pour la spécification architecturale d'une application à l'aide du langage AADLv2. Dans la section suivante, nous nous intéressons à la spécification des aspects comportementaux des composants du système.

4.3 Introduction à l'annexe comportementale

L'annexe comportementale AADL [SAE Aerospace, 2009a] a été développée dans l'objectif de préciser et d'affiner les aspects comportementaux implicites issus de la description architecturale AADL du système. Elle propose à travers la spécification d'un automate à états/transitions et de différents langages (actions, expressions...) de décrire le comportement interne des composants à travers leur interface de communication et un ensemble d'actions précisant l'émission, la réception et la manipulation de données. Cette annexe vise l'ensemble des composants architecturaux du langage AADLv2. Cependant, l'accent est logiquement mis sur les principaux composants logiciels traduisant le comportement du système : les composants threads et les sous-programmes.

Cette section présente une introduction à l'annexe comportementale AADL et nous permet de cerner le périmètre d'activité et de raffinement proposé par celle-ci. Notamment, nous pensons que la nouvelle version de cette annexe ([SAE Aerospace, 2009a]) a un pouvoir d'expressivité important autorisant jusqu'à la modélisation complète de l'exécutif AADL sous-jacent partiellement décrit par le standard AADLv2.

4.3.1 Spécification comportementale, structure d'automate et langages

Une description comportementale AADL est spécifiée à l'aide de cinq langages définissant des concepts comportementaux avec une syntaxe, un ensemble de règles de nommage, de règles légales, de règles sémantiques et de règles assurant la cohérence de la spécification comportementale vis-à-vis de la description architecturale.

L'exemple de code AADL 4.4 illustre la combinaison de ces langages pour établir la description comportementale de l'exécution du thread sporadique **Consumer** présenté dans l'exemple 4.2. On y retrouve différents types d'états respectant la sémantique de l'automate d'exécution du thread défini par le standard AADL et détaillé dans le chapitre 5. Notamment, l'état **complete** modélise l'état de suspension et de reprise d'activité du composant.

Une **transition** représente un changement de l'état courant (source) à un état de destination. Le déclenchement d'une transition s'effectue quand une condition spécifique est évaluée

© 2012 Gilles LASNIER

Exemple 4.4 – Description comportementale du thread sporadique `Consumer`

```
thread Consumer.impl
calls call_seq : { Do_Read : subprogram Read; };
connections
 Access_Data : data access Shared_Data -> Do_Read.Shared_Data;

annex behavior_specification {**
 variables
  lastValue : Base_Types::Integer;
 states
  stInit     : initial state;
  stDispatch : complete final state;
 transitions
  On_Init : stInit -[ ]-> stDispatch { lastValue := 0 };

  On_Dispatch : stDispatch -[on dispatch Dispatch_Port]-> stDispatch {
     lastValue := lastValue + Dispatch_Port;

     if (lastValue mod 2 = 0)
       Do_Read!(lastValue);
     end if;
 }
 **};
end Consumer.impl;
```

à vraie. Lorsqu'une transition est déclenchée, les actions attachées à celle-ci sont exécutées (voir le bloc {...} attaché à la transition **On_Dispatch** de l'exemple 4.4).

Les **conditions** (*behavior condition*) sont exprimées à l'aide d'un second langage et affinent l'utilisation des ports et des appels de sous-programmes AADL. Ces deux constructions permettent de spécifier le déclenchement d'activité d'une tâche. Dans notre exemple, le port **Dispatch_Port** défini dans l'interface du composant AADL (voir exemple 4.2, colonne droite) est utilisé comme condition pour le déclenchement de l'activité du composant.

Un troisième langage permet de spécifier les interactions du composant avec ses sous-composants ou avec d'autres composants à travers l'utilisation de ses éléments d'interfaces (données partagées, ports, appels de sous-programmes, etc.).

Un langage d'actions permet quant à lui de décrire les actions exécutées lors du déclenchement d'une transition. Celui-ci utilise le langage d'interaction pour définir les actions entre les composants (dans notre exemple, appel du sous-programme **Do_Read !(...)**).

Enfin, un langage d'expression basé sur le langage d'expression Ada permet la manipulation des données AADL (sous-composants de données, variables d'automates, donnée reçue sur le port, valeur de paramètre...) à travers la spécification d'expression logique, relationelle ou arithmétique.

Nous venons de présenter la structure globale d'un automate comportemental défini par l'annexe. Dans la suite de cette section, nous nous intéressons aux automates particuliers des composants threads et sous-programmes ainsi qu'aux langages d'interactions, d'actions et d'expressions s'apparentant à des constructions classiques d'un langage de programmation impérative.

4.3.2 Spécificités des automates des threads et des sous-programmes

L'annexe comportementale définit des restrictions pour la spécification des automates des threads et des sous-programmes.

Ainsi, l'automate comportemental d'un composant thread doit décrire un unique état initial représentant l'état de la tâche avant son initialisation, un ou plusieurs états de suspension (`complete`), zéro ou plusieurs états d'exécution (modélisant un chemin d'exécution des actions associées aux transitions) et un ou plusieurs états de finalisation. Cette structure d'automate vise à assurer la cohérence et le respect de la sémantique de l'automate d'exécution

© 2012 Gilles LASNIER

du thread défini par le langage cœur et détaillé dans le chapitre 5. Le langage de spécification des conditions pour le déclenchement d'activité d'un composant est restreint au composant thread. Celui-ci spécifie les différentes conditions possibles à savoir la réception d'un événement sur un port, l'appel d'un sous-programme fourni par le composant ou le déclenchement d'un `timeout`.

L'automate d'un composant sous-programme doit, quant à lui, décrire un unique état initial modélisant le point de départ de l'appel du sous-programme, zéro ou plusieurs états intermédiaires d'exécution (chemin d'exécution) et uniquement un état final représentant la terminaison de l'appel (de l'exécution) du sous-programme. Les états de suspension sont strictement interdits par l'annexe afin d'éviter les risques de suspension d'une tâche au milieu de l'exécution d'un sous-programme.

4.3.3 Interactions entre composants

L'annexe comportementale propose un langage d'interaction pour expliciter les interactions et la communication entre les différents composants architecturaux. Ce langage définit notamment :

- le comportement d'un port et de sa file d'attente lors de l'arrivée d'une donnée (gel de la donnée reçue, ordre de délivraison...) ;
- des constructions spécifiques pour la manipulation des données reçues sur un port (lecture d'une donnée, nombre de messages reçus, affectation d'une donnée à un port, envoi d'une donnée à partir d'un port, etc.) ;
- le comportement des données reçues à travers les paramètres de sous-programme ;
- des constructions spécifiques pour la réception, la consultation, l'affectation des données à travers les paramètres de sous-programmes ;
- des constructions spécifiques pour modéliser l'accès aux données partagées (modélisation d'une section critique...).

La précision du comportement des ports est basée sur l'utilisation de constructions de l'annexe et l'interprétation de propriétés AADL. Une syntaxe spécifique est définie pour les constructions autorisant la manipulation des ports, des paramètres de sous-programmes et des sous-programmes modélisant une section critique.

4.3.4 Eléments du langage d'actions

Le langage d'action de l'annexe comportementale définit les différentes actions qui sont exécutées lors du déclenchement d'une transition *i.e.* lorsque la condition ou la garde de la transition est évaluée à vraie.

Ce langage utilise les constructions spécifiées par le langage d'interactions et introduit de nouvelles constructions équivalentes aux constructions classiques d'un langage de programmation impératif comme Ada. On y retrouve :

- la structure conditionnelle `if ... then ... elsif ... else` ;
- les boucles de contrôles `for ... in`, `forall ... in`, `while ...` et `do ... until` ;
- l'opération d'affectation de valeur à une variable, opérateur `:=` ;
- l'opération d'appel de sous-programme `<nom_spg>!(param1,param2)` permettant l'appel d'un sous-programme référencé dans un paquetage AADL ou à l'aide d'un accesseur de sous-programme spécifié dans l'interface du composant ;
- les éléments de syntaxe pour l'accès aux sous-composants de données ou aux composants de données spécifiés à l'aide d'accesseurs ;

© 2012 Gilles LASNIER

– deux opérations pré-définies `computation(min,max)` et `delay(min,max)` exprimant respectivement l'utilisation du processeur et un temps de suspension compris entre durée minimale et maximale.

Des éléments de syntaxe et des règles sémantiques régissent l'utilisation de ces constructions. De plus, l'annexe comportementale introduit la possibilité de spécifier des séquences ordonnées d'actions ou un ensemble d'actions parallélisées. Dans la sous-section suivante, nous donnons un exemple d'utilisation du langage d'actions.

4.3.5 Eléments du langage d'expressions

Le langage d'expression proposé par l'annexe comportementale est basé sur le langage d'expression du langage de programmation Ada. On y retrouve les éléments de syntaxe et les constructions spécifiant des expressions logiques, des expressions relationnelles ou des expressions arithmétiques. Les éléments du langage d'expression s'appliquent uniquement aux composants AADL spécifiant des données, à savoir les ports, les paramètres de sous-programmes, les instances de données (sous-composants de données) et les données partagées (référencées à l'aide des accesseurs de données dans l'interface du composant).

Exemple 4.5 – Description comportementale du sous-programme `Read`

```
1  subprogram Read
2    features
3      Stream : in out parameter Message_Type.Impl;
4      Item : out parameter PolyORB_HI_Streams::Stream_Element_Array;
5      Last : out parameter PolYORB_HI_Streams::Stream_Element_Offset;
6  end Read;
7
8  subprogram implementation Read.Impl
9    subcomponents
10     L1 : data PolyORB_HI_Streams::Stream_Element_Count
11               { Data_Model::Initial_Value => ("Item'Length");
12                 Access_Right => read_only; };
13     L2 : data PolyORB_HI_Streams::Stream_Element_Count
14               { Data_Model::Initial_Value => ("Length (Stream)"); };
15     Tmp_L : data PolyORB_HI_Streams::Stream_Element_Count;
16     Item_First : data PolyORB_HI_Streams::Stream_Element_Offset
17               { Data_Model::Initial_Value => ("Item'First"); };
18   annex behavior_specification {**
19   states
20     s0 : initial final state;
21
22   transitions
23     t0_0 : s0 -[ ]-> s0 {
24       if (L1 < L2)
25         L2 := L1
26       end if;
27
28       Tmp_L := L2 - 1;
29
30       for ( J : PolyORB_HI_Streams::Stream_Element in 0 .. Tmp_L )
31         {
32          Item[Item_First + J] := Stream.Content[Stream.First + J]
33         };
34
35       Last := Item_First + L2 - 1;
36       Stream.First := Stream.First + L2
37     }
38   **};
39 end Read.Impl;
```

L'exemple 4.5 illustre quelques éléments des langages d'interactions, d'actions et d'expressions combinés pour la spécification du comportement d'un sous-programme AADL. Le comportement décrit est l'extraction des données dans un tampon de communication (**Stream** de type **Message_Type.Impl**). On y retrouve l'utilisation des sous-composants de données du sous-programme. L'utilisation des propriétés de l'annexe de modélisation des données nous permet de spécifier les valeurs d'initialisation de ces variables locales (exemple du sous-composant de données **L1**, l.10).

© 2012 Gilles LASNIER

Exemple 4.6 – Types énumérés et structurés en AADL

```
data Enum_Type
properties
  Data_Model::Data_Representation => Enum;
  Data_Model::Enumerators => ("InS_K", "WoM_K");
end Enum_Type;

data Struct_Type
properties
  Data_Model::Data_Representation => Struct;
end Struct_Type;

data implementation Struct_Type.impl
subcomponents
  Enum    : data Enum_Type;
  Value   : data Base_Types::Integer;
end Struct_Type.impl;
```

```
subprogram Check_Element
features
  Enum    : in out parameter Enum_Type;
  Output :    out parameter Struct_Type.impl;
end Check_Element;

subprogram implementation Check_Element.Impl
annex behavior_specification {**
  states       stExec : initial final state;
  transitions  tExec : stExec -[]-> stExec {
                 if (Enum = Enum_Type.InS_K)
                       Output.Enum  := Enum_Type.InS_K;
                       Output.Value := Output.Value + 1
                 end if
               }
 **};
end Check_Element.Impl;
```

4.3.6 Contributions à l'annexe comportementale

Les travaux menés durant cette thèse sur la modélisation de composants AADL nous ont amenés à contribuer et à proposer des évolutions à l'annexe comportementale. Nous présentons dans cette section les évolutions majeures que nous avons introduites.

Support pour les types énumérés, structurés et union

Le standard AADL recommande la spécification des types de données à l'aide de l'annexe de modélisation. Celle-ci définit l'ensemble de propriétés *Data_Model* contenant différentes propriétés pour spécifier la nature d'une donnée à l'aide des composants de données. Parmi celles-ci, nous retrouvons le patron de modélisation des types énumérés présentés dans l'exemple de code AADL 4.6.

Le support des types énumérés est manquant dans l'annexe comportementale. En effet, la syntaxe actuelle ne permet pas d'effectuer des comparaisons entre deux instances d'une énumération. Pour palier ce manque, nous avons introduit la syntaxe et les règles légales pour l'accès aux littéraux des énumérations au sein de l'annexe comportementale. La description comportementale du composant **Check_Element.Impl** illustre celle-ci (exemple 4.6).

Comme pour les types énumérés, les types structurés et les types union ne sont pas supportés par l'annexe comportementale. Une syntaxe et des règles légales pour le support de ces types ont été ajoutées sur le standard de l'annexe. La syntaxe pour la manipulation des sous-composants (des champs) d'un type structuré est décrite dans l'exemple 4.6.

Support pour la définition de bibliothèques de sous-programmes génériques

L'annexe comportementale autorise la manipulation des composants prototypes pour la définition de composants génériques à étendre et paramétrer lors de l'élaboration d'un modèle AADL. La spécification actuelle est incomplète car elle ne porte que sur les composants de données. La spécification d'un prototype de type sous-programme ou d'une séquence d'appel de sous-programme est impossible. Nous avons proposé et ajouté ces éléments dans l'annexe.

Cette dernière modification autorise la modélisation de bibliothèques de composants sous-programmes génériques configurables lors des différentes évolutions du modèle AADL. Dans le chapitre 6 et 7 nous illustrons l'utilisation d'une telle bibliothèque.

© 2012 Gilles LASNIER

4.4 Avantages et restrictions pour les systèmes TR^{2}E critiques

4.4.1 Avantages pour l'analyse et l'implantation

Dans notre étude préliminaire 2, nous avons exposé différents travaux proposant des processus d'ingénierie autorisant la modélisation, l'analyse et la production automatisée (génération, déploiement et configuration) des systèmes TR^{2}E à partir d'une description architecturale des composants d'une application dans le langage AADL.

Les composants matériels et logiciels proposés par le langage cœur permettent la spécification des entités concrètes d'une application. Le mécanisme des propriétés s'avère efficace pour l'intégration d'exigences fonctionnelles et non fonctionnelles (contraintes temporelles, informations de déploiement...) permettant d'exploiter le modèle AADL pour de nombreuses analyses. Enfin, l'intégration de spécifications comportementales au sein même de la description architecturale est un avantage conséquent pour expliciter le comportement des composants mais aussi pour assurer la cohérence de ces descriptions comportementales avec la spécification architecturale de l'application.

En outre, [Zalila *et al.*, 2008] expose, dans ses travaux, un processus d'implantation automatisée d'une application TR^{2}E critique piloté uniquement par l'analyse d'une description architecturale AADL. Nous observons tout de même que ces différentes approches n'intègrent aucun élément de son annexe comportementale.

De plus, le langage AADL vise la famille des systèmes TR^{2}E au sens large et autorise des constructions qui ne sont pas conformes aux exigences des systèmes critiques. Ces éléments sont présentés dans la sous-section suivante.

4.4.2 Restrictions architecturales et comportementales

Dans cette section, nous restreignons l'utilisation des éléments du langage AADL aux constructions ne compromettant pas l'analyse statique et l'implantation d'un système critique fiable. Dans le cadre de la production d'un système critique, nous souhaitons respecter les restrictions du profil Ravenscar et celles liées aux caractères des systèmes critiques présentées dans la section 2.3.2 de notre état de l'art (chapitre 2). De plus, afin de faciliter l'automatisation des étapes d'analyse et de génération nous supprimons les éléments de modélisation non concrets pour la réalisation de ces tâches.

Restriction des composants architecturaux

Dans le chapitre précédent, nous avons défini des critères pour restreindre l'utilisation du langage AADL. Le critère C1 porte sur la restriction de l'utilisation des composants abstraits, des prototypes, des ports de communication et sur les patrons de modélisation des types de données.

- *Les composants abstraits et les prototypes* sont utilisés au début de la modélisation avant l'identification précise des composants logiciels et matériels. Ils sont raffinés par héritage, extension ou *bindings* lors de l'évolution de la modélisation. A la fin de cette étape, la réécriture du composant concret final permet d'éliminer l'ensemble de ces constructions non pertinentes pour l'analyse et la génération.
- *Les ports de communication* définissent les connecteurs et les interactions entre les composants et les propriétés AADL qui y sont attachées, et spécifient leur comportement. L'utilisation de patrons comportementaux nous permet d'expliciter ce comporte-

© 2012 Gilles LASNIER

ment ainsi que les éléments du support d'exécution (intergiciel) mis en jeu. Cette réécriture réduit la différence sémantique entre le composant modélisé et son implantation physique.
– La modélisation des données (types, constantes et variables) est très permissive à l'aide du langage AADL. L'utilisation de l'annexe de modélisation des données limite ces patrons de modélisation et renforce la génération de code.

Restriction des constructions comportementales

Pour satisfaire le critère C2, nous avons développé des patrons comportementaux pour les différents types de tâches supportés dans notre approche. Ces patrons ont été élaborés dans le but de préserver l'analysabilité du modèle AADL et de permettre la production du code source du composant à partir de sa spécification comportementale. Ces patrons sont présentés dans le chapitre 5.

Restrictions pour l'analyse statique

Pour satisfaire le critère C3 et ainsi assurer l'analysabilité statique du système modélisé, nous avons contraint l'ensemble de nos patrons de modélisation (architecturaux et comportementaux) selon les recommandations du profil Ravenscar. Pour s'assurer de la conformité avec ce dernier, nous avons défini **trois** restrictions sur la description architecturale (RA1, RA2, RA3) et **deux** (RC1 et RC2) sur la spécification comportementale - *i.e.* les automates :

RA1. Chaque processeur doit être modélisé avec un attribut spécifiant un ordonnancement à priorité fixe (par exemple l'ordonnancement de type *FIFO within priorities*) et un attribut spécifiant que l'ordonnanceur est préemptif.

RA2. Chaque tâche du système doit être cyclique - *i.e. périodique ou sporadique*.

RA3. Chaque objet protégé du système doit être modélisé avec un attribut spécifiant la politique de plafonnement de priorité (PCP).

RC1. Chaque tâche doit posséder au plus un point de suspension.

RC2. Chaque tâche ne doit pas terminer son exécution - *i.e. infinie*.

L'interdiction de la création dynamique de tâches imposée par le profil Ravenscar est assurée par la description architecturale des entités actives de l'application (les composants threads) et par notre processus de production automatique. Ainsi, grâce au modèle AADL, nous pouvons déterminons le nombre exact de tâches à créer avant l'implantation du système.

Nous souhaitons aussi vérifier un certain nombre de restrictions supplémentaires inhérentes aux systèmes critiques. Ainsi, il s'agit de garantir :

1. La présence d'éléments d'interfaces non connectés, compromettant la validation du système et conduisant à la génération de code inutile ;
2. L'utilisation des types de données de taille non bornée. Ceci permet l'évaluation de la taille exacte de toutes les données et l'allocation statique de la mémoire requise.
3. L'utilisation d'une borne temporelle spécifiant le temps minimal entre deux déclenchements d'activités des threads sporadiques.

© 2012 Gilles LASNIER

L'ensemble des restrictions que nous venons de présenter sont vérifiés sur la modélisation AADL à différentes étapes de notre processus de développement. Nous notons que la même approche a été adoptée par OCARINA mais concerne uniquement la partie architecturale de l'application.

Dans les chapitres 7 et 8 nous présentons les différents patrons de modélisation pour l'analyse et la génération de code mettant en œuvre les restrictions que nous avons définies. Nous illustrons ces différents patrons par des exemples concrets de composants AADL.

4.5 Outillages AADL existants

Pour terminer notre introduction sur le langage AADL, nous nous intéressons et présentons dans le chapitre 5 une rapide description des principaux outils de modélisation, d'analyse et de génération basés sur le langage AADL. Nous devons noter que ces outils supportent notamment zéro, une ou plusieurs annexes du langage AADL. Pour compléter le panorama des outils existants, il est possible de consulter le site officiel AADL[2] et l'article suivant [Hugues, 2011].

4.6 Synthèse

Ce chapitre a proposé une introduction aux éléments du langage AADLv2 et aux éléments de son annexe comportementale. Le langage AADL est le formalisme pivot que nous avons retenu pour atteindre les différents objectifs de ces travaux de thèse. Notamment, nous savons que celui-ci dispose des capacités pour la spécification, la configuration et le déploiement des composants d'une application TR^{2}E.

Son annexe comportementale nous permet d'affiner la spécification du comportement des composants architecturaux notamment pour les composants threads et sous-programmes. L'utilisation des constructions s'apparentant à un langage de programmation impératif nous offre des perspectives pertinentes pour la réutilisation de ces descriptions dans un processus de production automatique du code source des composants.

Nous avons décrit l'ensemble des éléments architecturaux et comportementaux spécifiés par le standard du langage cœur dans sa version 2.0 [SAE Aerospace, 2009b] et de la *draft* v2.12 de son annexe comportementale [SAE Aerospace, 2009a]. Durant notre travail de thèse, cette dernière a poursuivi son évolution et nous sommes à l'origine de certaines améliorations proposées par notre équipe de recherche. Ces contributions nous ont positionnés comme contributeur pour l'annexe comportementale et nous nous sommes vus confier la tâche de la rédaction des évolutions et des modifications de l'annexe comportementale [TELECOM ParisTech, 2011].

Après avoir constaté le caractère généraliste du langage cœur et de son annexe, nous avons défini un ensemble de restrictions architecturales, comportementales et non fonctionnelles pour garantir l'analysabilité statique des composants modélisés et faciliter l'implantation des outils d'analyse et de génération.

Dans la suite de ce manuscrit, nous détaillons ce sous-ensemble d'éléments pour la définition d'un profil AADL autorisant l'élaboration de patrons de génération de code préservant l'analyse. Ce profil servira de base pour le processus de raffinement, d'analyse et d'implantation automatisé que nous proposons et fait l'objet du chapitre suivant.

2. http://www.aadl.info

© 2012 Gilles LASNIER

© 2012 Gilles LASNIER

Troisième partie

Détail des contributions

Chapitre 5

Niveau d'Abstraction Pour l'Analyse et la Génération de Code

SOMMAIRE

5.1 Introduction

Dans le chapitre 3, nous avons exposé les motivations du raffinement incrémental d'un modèle architectural AADL initial (spécifié par l'utilisateur) pour l'obtention d'un modèle de code complet et analysable. Ce modèle de code est une représentation des composants applicatifs et intergiciels de l'application avec une sémantique proche de son implantation. Le processus de raffinement incrémental constitue le cœur de notre approche et est contruit sous la forme d'une chaîne de transformations endogènes verticales assurant la réduction du niveau d'abstraction de la description initiale (présenté chapitre 7).

Le modèle architectural et comportemental obtenu doit être utilisé comme point d'entrée par nos différents outils et, dans le plus grand nombre de cas possibles, par les outils d'analyse existants de la communauté AADL. Il doit notamment permettre d'affiner les analyses de faisabilité (ordonnancement, comportement, etc.) par la prise en compte des ressources intergicielles. Enfin, ce modèle doit autoriser l'expression d'un *mapping* simple vers les contructions classiques des langages de programmation impératifs (par exemple Ada ou C).

Notre étude préliminaire (chapitre 2) a révélé l'utilisation de représentation du système critique à des niveaux d'abstraction différents par les outils d'analyse et de génération. Se pose

alors la question du niveau d'abstraction requis par notre modèle de code analysable. Selon la définition 1.5 (chapitre 2), celui-ci dépend du niveau de détail exprimé pour la description des composants. Un niveau d'abstraction trop bas facilite la génération de code à partir de la description du composant, mais peut parasiter le modèle et complexifier les transformations vers les formalismes utilisés par les outils d'analyse. Un niveau d'abstraction trop élevé ne permettrait pas de décrire les composants intergiciels dépendants de la plate-forme d'exécution et rendrait difficile une démarche automatique de production. Par conséquent, nous devons déterminer un niveau d'abstraction intermédiaire qui limiterait les tranformations pour l'élaboration des modèles d'analyse et autoriserait l'expression des informations architecturales et comportementales nécessaires pour décrire les composants applicatifs et intergiciels de l'application.

Le langage AADL et son annexe comportementale (présenté chapitre 4) dispose de caractéristiques intéressantes pour la réalisation de cet objectif. L'étude des éléments architecturaux du langage AADL (composant, port, propriétés...) impliqués dans les modèles d'analyse et utilisés par les outils de génération de code détermine un sous-ensemble du langage AADL pertinent. L'intégration des descriptions comportementales à ce sous-ensemble précise le comportement des composants et contribue à réduire le niveau d'abstraction pour permettre la modélisation des composants intergiciels. Ces éléments combinés aux restrictions que nous avons définies à partir des recommandations du profil Ravenscar et pour la réalisation des systèmes critiques (voir chapitre 4) constituent un profil de modélisation (AADL-HI Ravenscar) définissant un niveau d'abstraction intermédiaire pour l'analyse et la génération. Nous élaborons nos patrons de génération (achitecturaux et comportementaux) préservant l'analyse à partir de ce profil.

Ce chapitre est organisé de la manière suivante. La section 5.2 (respectivement 5.3) synthétise les différents composants AADL impliqués par les modèles d'analyse (resp. la génération). La section 5.4 présente les patrons de modélisation comportementale que nous avons définis pour les composants AADL retenus pour notre approche. La section 5.5 synthétise nos résultats pour la définition du profil AADL-HI Ravenscar.

5.2 Composants requis pour l'analyse

Rappel. La vérification d'un système vise à s'assurer que celui-ci est correctement construit et sa validation vise à garantir le respect de ses spécifications. Validation et vérification s'effectuent au travers de différentes analyses reposant sur une abstraction structurée et complète modélisant un aspect particulier du système. Ces aspects sont liés à la satisfaction de contraintes relatives à la spécificité du système.

Le langage AADL permet d'exprimer des contraintes fonctionnelles et non fonctionnelles sous une forme condensée et structurée au sein d'une représentation unique. Il sert de point d'entrée pour procéder à l'analyse du système ou de langage pivot pour la production d'un modèle d'analyse dans un formalisme tiers. Dans cette section, nous rappelons les différentes contraintes inhérentes aux systèmes TR^{2}E critiques. Puis, nous nous intéressons aux principaux outils d'analyse basés sur AADL et plus particulièrement aux composants AADL et à leurs attributs (propriétés...) qu'ils utilisent. Ces éléments nous permettent de déterminer la granularité du composant AADL et le niveau d'abstraction requis pour différentes analyses.

© 2012 Gilles LASNIER

5.2.1 Contraintes spécifiques aux systèmes critiques

Afin de déterminer les composants AADL les plus pertinents requis par les outils d'analyse, nous proposons une classification simple des contraintes fonctionnelles et non fonctionnelles à satisfaire dans le contexte des systèmes TR^{2}E critiques. Cette classification nous permet de distinguer les caractéristiques des différentes suites d'outils supportant une modélisation AADL. Ainsi, nous distinguons six familles de contraintes :

1. les contraintes sur la cohérence de la modélisation : conformité de la description architecturale, hiérarchie entre composants, sémantique des composants ;
2. les contraintes structurelles : informations de déploiement, préparation de l'application en vue de la distribuer (critère distribué de l'application) ;
3. les contraintes sur la consommation des ressources matérielles et logicielles : utilisation processeur, bande passante, empreinte mémoire (critère embarqué de l'application) ;
4. les contraintes temporelles : respect d'échéances temporelles, ordonnançabilité (critère temps-réel de l'application) ;
5. les contraintes de criticité : intégrité des données, sûreté de fonctionnement, sécurité, tolérances aux pannes, déterminisme ;
6. les contraintes comportementales : validation du comportement vis-à-vis des spécifications de l'utilisateur, équité, interblocage, vivacité, etc.

Dans la sous-section suivante, nous présentons les principaux outils d'analyse visant la vérification d'une ou plusieurs de ces contraintes.

5.2.2 Vérification et validation de modèles AADL

Le tableau 5.1 synthétise les principaux outils et les familles de contraintes qu'ils vérifient. Pour certaines plates-formes d'intégration (par exemple TASTE ou AADL INSPECTOR intégrant l'outil CHEDDAR), nous ne considérons que les fonctionnalités supplémentaires offertes ou intégrées.

	OSATE2	OCARINA	CHEDDAR	TASTE	TINA	AADL INSPECTOR
Cohérence	x	x		x		x
Structurelles		x				
Ressources	x	x	x			x
Temporelles			x	x		x
Criticité	x	x				
Comportement					x	x

TABLE 5.1 – Contraintes et outils d'analyses AADL

OSATE2

OSATE2 [SEI/CMU, 2011] est une plate-forme dédiée à l'analyse et à l'intégration d'outils basés sur les modèles AADL. La partie frontale permet de vérifier la cohérence de la description (syntaxe et sémantique des composants). Un certain nombre de plugins d'analyse assurent aussi la vérification de propriétés de sécurité et de sûreté, du flot d'exécution des données, de l'allocation et de la consommation mémoire, de l'intégrité et de la cohérence des données, de la consommation d'énergie et de la propagation d'erreurs.

© 2012 Gilles LASNIER

Tous les composants AADL sont supportés. Les analyses sont effectuées à partir d'un modèle déclaratif AADL ou d'un modèle d'*instance* généré par OSATE2. Celui-ci décrit une instance du système modélisé où les liens d'extension, de raffinement, de protypage et certains calculs sur les valeurs de propriétés sont résolus.

OCARINA

OCARINA [TELECOM ParisTech, 2012] propose une partie frontale assurant l'analyse de la cohérence d'une description architecturale et des informations structurelles (déploiement, répartition) pour la génération du système. Différentes analyses à travers l'implantation du langage de contraintes REAL et l'exportation de modèles vers le formalisme des réseaux de Petri ou vers l'outil CHEDDAR sont possibles :

- l'annexe AADL-REAL [Gilles, 2008] permet la vérification de contraintes et le calcul d'expressions complexes sur les modèles architecturaux AADL. Elle supporte un sous-ensemble de composants (matériels et logiciels) et leurs propriétés associées. Il permet la vérifiation de la cohérence du dimensionnement d'une application (projet MOSIC), des spécifications (respect des patrons de modélisation, respect des restrictions du profil AADL/Ravenscar) et des exigences de sécurité et sûreté au sein de la plate-forme POK.
- OCARINA/RDP est une partie dorsale permettant la transformation des composants architecturaux AADL vers les réseaux de Petri. Ainsi, nous pouvons vérifier le comportement d'un système par l'analyse de contraintes telles l'absence d'interblocage, la famine, la vivacité ou l'équité dans une application répartie (multi-tâches) contenant des ressources partagées. Le *mapping* des composants AADL vers les constructions des réseaux de Petri est décrit dans [Renault, 2009] (chapitre 4, section 4.1.1) et nécessite les composants suivants : sous-programmes, threads, processeurs, variable partagée, port. Les propriétés AADL standards pour ces composants sont aussi requises. Enfin, les composants systèmes et processus sont utilisés pour structurer la spécification et apportent des informations hiérarchiques sous la forme d'espace de nommage permettant la traçabilité des résultats de la vérification et la détection de composants fautifs.

CHEDDAR

CHEDDAR permet la vérification de performances et de l'ordonnançabilité d'un système modélisé en AADL. Dans [Singhoff *et al.*, 2005], les auteurs décrivent quels sont les éléments du langage AADL pris en compte pour l'analyse d'ordonnancement, à savoir : les composants de données (variable partagée), les accesseurs aux composants de données, les threads, les processus, les processeurs et les composants systèmes. CHEDDAR utilise certaines propriétés AADL standards mais aussi un ensemble de propriétés AADL (pour les composants de données, threads, processeurs) spécifiques à l'analyse d'ordonnancement. Enfin, CHEDDAR permet à l'aide de la spécification des ports de threads d'effectuer une analyse mémoire sur la taille et la capacité des tampons pour les communications entre threads.

TASTE (MAST)

TASTE [Perrotin *et al.*, 2011] est une plate-forme d'intégration assurant l'interopérabilité entre différents outils pour le développement et l'analyse de systèmes TR^{2}E. Ainsi, TASTE intégre de nombreux outils tels OCARINA (pour la génération), CHEDDAR, MAST, MARZHIN (pour l'analyse), etc. Nous nous intéressons, ici, uniquement à l'intégration de l'outil MAST

© 2012 Gilles LASNIER

pour l'analyse de l'ordonnancement du système. MAST nécessite une transformation des éléments architecturaux du langage AADL vers les éléments du méta-modèle MAST. Le *mapping* de AADL vers MAST est exprimé dans [Perrotin *et al.*, 2011] (chapitre 16, section 5). Ainsi, les composants processeur, thread, sous-programme, bus, périphérique et les composants de données sont utilisés. Un sous-ensemble des propriétés standards AADL associées à ces composants est requis.

AADL-FIACRE-TINA

L'outil TINA permet l'analyse comportementale de réseaux de Petri et de systèmes temporisés. Dans [Berthomieu *et al.*, 2010], les auteurs proposent une transformation de modèles AADL vers FIACRE pour exprimer le comportement, les aspects temporels et la sémantique d'exécution du système puis une transformation de modèles FIACRE vers le formalisme pris en compte par TINA. Les auteurs décrivent les composants architecturaux et les éléments de l'annexe comportementale AADL transformés. L'annexe comportementale utilisée ici est une ancienne version de celle présentée chapitre 4, basée sur le langage AADL 1.0 et n'intégrant pas de langage d'expression aussi détaillé que la nouvelle. Ainsi, les composants inclus sont les composants threads, les ports de threads, les composants de données (variable partagée), les accesseurs aux composants de données, les modes et les constructions de l'annexe comportementale relatives aux threads. Un sous-ensemble de propriétés AADL standards dédiées à ces composants est aussi employé.

AADL INSPECTOR (MARZHIN)

AADL INSPECTOR [Ellidiss Technologies, 2012] est un outil d'analyse extensible basé sur l'expression et la vérification de règles sur les modèles AADL. Nous nous intéressons ici à la simulation du système avec MARZHIN. Pour ce faire, l'outil fournit une transformation de modèle AADL vers le formalisme d'entrée de MARZHIN. Le sous-ensemble AADL supporté concerne les composants processeur, thread, les ports d'événements, les données (variable partagée), les accesseurs de données, les appels de sous-programmes et les constructions de l'annexe comportementale pour les threads et les sous-programmes. Un sous-ensemble des propriétés AADL standards spécifiques à ces composants est aussi utilisé pour la transformation.

5.2.3 Granularité des composants et niveau d'abstraction

Le tableau 5.2 synthétise l'ensemble des composants AADL et leur rôle au sein des outils d'analyse que nous venons de présenter. Nous constatons que l'ensemble des composants concrets (logiciels et matériels) sont utilisés. Un sous-ensemble d'attributs spécifiques est retenu en fonction du type d'analyse effectué. Ainsi, ce sous-ensemble du langage AADL observé définit le niveau de détails requis pour la description des composants AADL pour l'analyse des systèmes du domaine TR^2E. Ceci nous montre le pouvoir d'expression du langage. Un niveau de détail similaire doit pouvoir être exprimé par le niveau d'abstraction intermédiaire de notre modèle de code complet.

Rappel. Le périmètre d'étude (présenté en introduction de ce mémoire) que nous avons défini pour notre approche ne concerne pas les systèmes adaptatifs et/ou tolérants aux pannes. Ainsi, pour la suite, nous ne tiendrons pas compte des constructions pour les modes et celles de l'annexe de modélisation des erreurs.

© 2012 Gilles LASNIER

Composants AADL	Rôle	Utilité pour l'analyse
Système	Hiérarchie	Nommage - traçabilité
Processus	Hiérarchie - stockage ressources	Nommage - traçabilité
Thread	Envoi/réception/traitement messages	Ordonnancement - comportement
Sous-programme	Unité de calcul	WCET - ordonnancement - comportement
Donnée	Ressource	Typages - mémoires - flots
Processeur	Ressource d'exécution	Ordonnancement - énergie
Mémoire	Ressource	Ordonnancement - Mémoire
Bus	Canal de communication	Bande passante
Ports	Interconnexion de composants	Mémoire - flots
Flots	Flots d'exécution	Flots
Modes	Reconfiguration	Tolérance aux pannes
Annexes	Intégration langages externes	Comportement - erreurs
Propriétés	Attributs non fonctionnels	*tous types*

TABLE 5.2 – Synthèse des composants AADL requis pour l'analyse

Dans la section suivante, nous nous intéressons à présent aux éléments du langage AADL utilisés pour la production du code source des composants applicatifs du système.

5.3 Composants requis pour la génération de code

Les travaux de [Zalila, 2008] et l'annexe de génération de code AADL [SAE Aerospace, 2009c] ont montré les avantages (simplicité vis-à-vis des autres formalismes de description plus généralistes) de la production automatique de code source d'un système critique à partir de sa description architecturale AADL. Pour permettre la transformation des composants logiciels vers les artefacts d'un langage de programmation, la description du système doit exprimer des informations détaillées sur la structure externe et interne (interface, interaction inter-composants, séquence d'appels de sous-programmes, variables, initialisation des données, etc.), sur la configuration et le déploiement du composant.

L'annexe de génération de code AADL propose des patrons de génération pour les composants architecturaux vers les langages de programmation Ada et C. Aucune directive n'est cependant exprimée concernant les constructions de l'annexe comportementale dont nous présentons les bénéfices dans la section 5.4 de ce chapitre. Dans le contexte de nos travaux, nous nous intéressons, dans un premier temps, à la génération de code source vers le langage Ada respectant le profil Ravenscar de façon à pouvoir garantir la production d'un code analysable et correct-par-construction. La généralisation de notre approche pour les langages de programmation impératifs (par exemple, le langage C/POSIX) est discutée comme perspective future dans la conclusion de ce manuscrit (chapitre 11).

Pour réduire la distance sémantique entre modèles d'analyse et l'implantation produite, il est nécessaire de pouvoir exprimer explicitement les liens entre les composants de modélisation et les contructions relatives à leur implantation. Dans notre approche, nous visons la définition d'un modèle intermédiaire, appelé modèle de code avec un niveau d'abstraction proche de celui d'un langage de programmation - *i.e* avec des composants dont la sémantique et le comportement sont équivalents voir très proches des constructions du langage de programmation. Pour ce faire, nous devons déterminer le sous-ensemble des composants logiciels présentant cette dernière caractéristique, puis nous intéresser aux composants essentiels pour

© 2012 Gilles LASNIER

la génération, ayant une forte sémantique et nécessitant une transformation. Nous nous appuierons aussi sur les constructions offertes par l'annexe comportementale pour préciser le comportement des composants de ce sous-ensemble. Ces éléments permettront d'exprimer la granularité de la description des composants de notre modèle de code qui permettent une démarche de production automatisée.

5.3.1 Composants concrets pour la génération

Historiquement, le langage AADL est basé sur le langage de description d'architecture METAH et a été enrichi afin de faciliter la modélisation et l'analyse des systèmes TR^{2}E. Différentes contributions sont notamment issues des retours d'expérience et d'utilisation des acteurs de la communauté AADL sur les langages de description d'architectures (FRACTAL ou ACME), la modélisation UML et les langages de programmation pour système embarqué tel Ada, particulièrement connu pour son emploi dans l'implantation de systèmes critiques. Ainsi, de nombreux éléments du langage AADL ont pour origine certains mécanismes ou artefacts du langage Ada (par exemple, paquetage et visibilité de composants, tâches, génériques, etc). Le tableau 5.3 présente les composants AADL ayant une équivalence sémantique (voir aussi structurelle) directe avec ceux du langage Ada.

Composants AADL	**Eléments du langage Ada**	**Fonction**
Paquetage	Paquetage	Nommage - encapsulation - visibilité
Thread	Tâche	Fil d'exécution
Sous-programme	Sous-programme	Unité de calcul - comportement
Données	Variables	Typages - stockage de données
Données partagées - Accès donnée et sous-programme	Objets protégés	Typages - stockage de données - concurrences
Prototypes	Génériques	APIs génériques & configurables
Propriétés Annexes	Pragma, commentaires, annotations (ex : SPARK)	Configuration - traçabilité - vérification formelle

TABLE 5.3 – Equivalences entre composants AADL et éléments du langage Ada

Des travaux menés par notre équipe ont montré que ce sous-ensemble était sastisfaisant pour la génération de systèmes critiques dans les langages Ada/Ravenscar et C/POSIX [Zalila *et al.*, 2008; Lasnier *et al.*, 2009]. Notamment, nous appliquons les restrictions architecturales et comportementales (définies chapitre 3, section 4.4.2) sur nos patrons de génération pour garantir à la génération la conformité du code Ada produit vers le profil Ravenscar.

Rappel. Le profil Ravenscar limite l'usage du langage Ada aux seules constructions qui garantissent l'analyse statique d'ordonnancement, l'absence d'interblocage et une borne temporelle d'inversion de priorités entre les tâches.

Cas particulier des ports AADL pour la communication inter-thread

Les ports et les connexions du langage AADL expriment les liens de communication et les données échangées entre les composants. Ils facilitent la compréhension, l'analyse et l'implantation d'outils d'analyse (par exemple, analyse de flots d'exécution, des canaux de

© 2012 Gilles LASNIER

communication inter-processus, etc). L'inconvénient est qu'un tel composant n'a pas d'équivalent sémantique explicite et/ou unique dans un langage de programmation. En fonction de la sémantique, des propriétés et des politiques de gestion des files d'attente des ports spécifiés, ce dernier peut se traduire par une interruption, un signal, un message, un tampon de communication ou une structure de données plus complexe nécessitant une ou plusieurs transformations par le compilateur pour son implantation.

Dans notre contexte, l'emploi du profil Ravenscar limite fortement les communications entre les tâches Ada. En effet, seuls les objets protégés à une seule entrée sont autorisés et l'attente de plusieurs sources d'événements simultanément pour une tâche est interdite. Ainsi, transformer les ports et les connexions AADL pour implanter la communication des tâches en Ada/Ravenscar consiste à élaborer un mécanisme de communication qui permet à la fois d'effectuer une seule attente mais aussi de recevoir plusieurs événements à partir d'une spécification autorisant plusieurs événements sources (les ports de l'interface du thread) et plusieurs files d'attente (chaque port AADL dispose de sa propre file d'attente).

Pour ce faire, [Zalila *et al.*, 2008][chapitre 6, section 6.4.2] propose une implantation des ports en Ada (intégrée à l'intergiciel PolyORB-HI), appelée *Global Queue* (file d'attente globale de messages) dont l'architecture est basée sur une table constituée d'une concaténation de tableaux circulaires. Chacun de ces tableaux circulaires représente une file d'attente d'un port pour une tâche donnée. Cette architecture a été implantée sous la forme d'un objet protégé manipulé à l'aide de procédures et de fonctions dont l'exécution est déterministe (constant ou en *O(N)* où *N* est la taille de la donnée spécifiée par le port). Des structures de données supplémentaires complètent l'architecture de la file globale et implante les différents comportements des ports d'un thread (politique de gestion des files, priorité de livraison...) spécifiés à l'aide des propriétés du standard AADL.

Composants pour la modélisation de la *Global Queue*

Pour renforcer la cohérence entre modèle d'analyse et code généré, nous avons choisi de modéliser l'implantation de la *Global Queue* décrite ci-dessus. Notre processus de raffinement sera chargé de transformer les ports et les connexions vers un assemblage de composants architecturaux et comportementaux modélisant la structure et le comportement de la file globale. Les éléments produits seront intégrés de manière automatique aux interfaces des threads de la description initiale.

Les équivalences entre les composants de données partagées AADL et les objets protégés Ada (voir tableau 5.3 précédent) nous permettent de modéliser l'architecture de la file globale. Nous utilisons les constructions des langages d'action et d'expression de l'annexe comportementale pour modéliser le comportement des procédures et des fonctions pour la manipulation de la file. Les composants architecturaux et comportementaux utilisés sont détaillés dans le tableau 5.4. Nous précisons aussi le rôle de chaque élément. Enfin, les règles de transformation pour la production d'une file d'attente globale de messages sont présentées dans le chapitre 7.

Ainsi, la transformation des ports d'un composant vers une *Global Queue* préserve le comportement spécifié par l'utilisateur, autorise l'analyse (par construction respectant le profil Ravenscar) et réduit la sémantique entre le modèle du système et son implantation.

© 2012 Gilles Lasnier

Composants AADL	Rôle	Constructions Ada
Donnée partagée	Structure globale	Objet protégé
Accès aux composants de données	Exclusion mutuelle sur l'accès à l'objet	Sémantique de l'objet protégé
Propriétés AADL : `Concurrency_Protocol => Priority_Ceiling` `Priority => <integer>`	Protocole seuil plafond et priorité (garantie l'inversion de priorité)	Pragma `Ravenscar` et pragma `Priority` de l'objet protégé
Sous-programmes	Comportement et manipulation des données	Procédures/Fonctions
Accès aux sous-programmes et propriété AADL `Access_Right`	Exclusion mutuelle sur l'objet	Sémantique de l'objet protégé
Annexe de modélisation des données	Structures de données	Typages et initialisation des données
Annexe comportementale (sous-programme)	Comportement des routines	Implantation des procédures

TABLE 5.4 – Composants AADL pour la modélisation de la *Global Queue*

5.3.2 Granularité des composants architecturaux pour la génération

Le tableau 5.5 regroupe les observations effectuées dans les sous-sections précédentes et présentées dans les tableaux 5.4 et 5.3. Il présente l'ensemble des composants architecturaux AADL logiciels impliqués pour la génération de code vers les contructions du langage Ada/Ravenscar.

Composants AADL	Eléments du langage Ada	Fonction
Paquetage	Paquetage	Nommage - encapsulation - visibilité
Système	-	Nommage - hiérarchie
Processus	-	Nommage
Thread	Tâche	Fil d'exécution
Sous-programme	Sous-programme	Unité de calcul - comportement
Données	Variables	Typages - Stockage de données
Données partagées - Accès donnée et sous-programme	Objets protégés	Typages - stockage de données - concurrences
Prototypes	Génériques	APIs génériques & configurables
Propriétés Annexes	Pragma, commentaires, annotations (ex : SPARK)	Configuration - traçabilité - vérification formelle
Annexe de modélisation des données	Types de données et valeurs initiales	Typages et initialisation de données

TABLE 5.5 – Composants AADL architecturaux pour la génération de code vers le langage Ada

Nous y ajoutons arbitrairement les composants `systèmes` et `processus`. Ces éléments n'ont pas d'équivalent concret vers une construction du langage Ada. Cependant, le composant processus ajoute des informations de nommage qui nous permettent d'identifier les composants applicatifs et intergiciels produits. Le composant système permet de hiérarchiser la description architecturale et de déterminer uniquement les instances de composants qui constituent l'application finale. Il sert par conséquent à piloter le processus de génération de

code et nous assure de ne pas générer des composants applicatifs et intergiciels ne faisant pas partie de l'application finale et pouvant être source d'erreurs lors de son déploiement.

Ce sous-ensemble du langage AADL définit le niveau de détail que doit exprimer la description architecturale du système pour permettre la production automatique de l'application. Les constructions de l'annexe comportementale, que nous présentons dans la section suivante, permettent de compléter ce sous-ensemble pour préciser le comportement des composants à travers la spécification d'action et d'expression proche des constructions des langages de programmation impératifs.

5.4 Niveau de raffinement et spécifications comportementales

Rappel. L'annexe comportementale (présentée chapitre 4) permet, à travers la définition d'automates à états/transitions et l'utilisation de langages d'interaction, d'action et d'expression, de spécifier le comportement de tout type de composant AADL. Ce formalisme est ainsi utilisé comme langage pivot pour traduire le comportement des composants vers des formalismes tiers usités par les outils de *model checking* ou de simulation. Les langages d'action et d'expression proposent des éléments de langage (boucles, affectations de valeur, appels de sous-programmes, etc.) ayant la même sémantique que ceux issus d'un langage de programmation impératif classique.

Dans cette section, nous présentons nos patrons de modélisation comportementale définis pour l'analyse et la génération de code. Nous nous intéressons uniquement aux composants logiciels qui traduisent le comportement de l'application modélisée à savoir les sous-programmes et les threads. L'utilisation de spécifications comportementales au sein de notre approche permet de raffiner certains composants de la description architecturale initiale, de modéliser et d'intégrer les ressources intergicielles - *i.e.* l'exécutif AADL. Dans les sous-sections suivantes, nous présentons la structure et les éléments de langage de l'annexe sélectionnés. Puis, nous discutons des restrictions définies sur ces automates pour garantir l'analyse statique du système modélisé et des bénéfices des langages de l'annexe pour la génération de code.

5.4.1 Description comportementale des composants threads

Analyse du cycle de vie du thread

Le thread est un composant majeur d'une description architecturale car il est le seul à traduire implicitement les fontionnalités de l'application (implicite au sens où il spécifie une séquence d'appels de composants sous-programmes). Une section entière du standard AADL [SAE Aerospace, 2009b][section 5.4] est consacrée à la description du cycle de vie du composant thread et précise sa sémantique. La figure 5.1 présente l'automate du cycle de vie spécifié par le standard [SAE Aerospace, 2009b][section 5.4.1] :

1. L'état de départ d'un *thread* est "arrêté" (*halted*).
2. Lorsque celui-ci est chargé, il effectue une unique fois une phase d'initialisation.
3. Le *thread* est associé à un *mode* du système. S'il n'appartient pas à la configuration initiale du système (*initial mode*), il attend l'activation de son mode.

© 2012 Gilles LASNIER

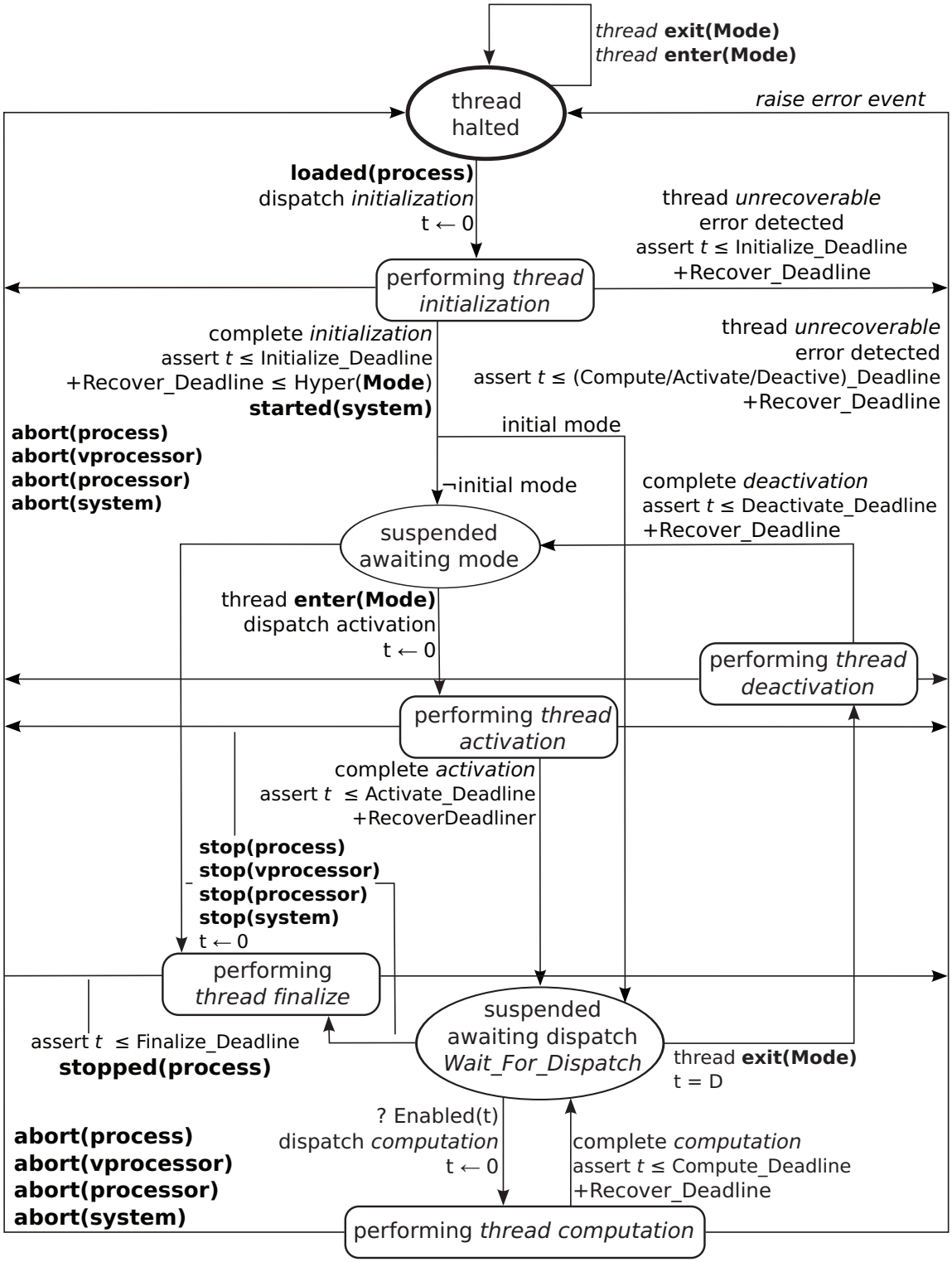


FIGURE 5.1 – Automate du cycle de vie du composant thread (fil d'exécution)

© 2012 Gilles LASNIER

4. Il passe dans un état suspension (*awaiting dispatch*) en attente du déclenchement de son exécution.
5. Si les conditions spécifiées par la fonction *Enabled* sont validées, le *thread* est déclenché (*dispatch*), passe dans un état où il effectue son exécution et réalise toutes les actions qui y sont associées.
6. Une fois son exécution terminée, il retourne dans l'état de suspension en attente du prochain déclenchement de son exécution.

Dans le cas d'une erreur (faute du composant, échéance manquée) ou sur la réception d'une commande d'arrêt (*stop*, *abort*, *exit*), le thread retourne dans l'état "arrêté".

Patron de modélisation comportementale du thread

L'exemple de code AADL 5.1 présente la description architecturale du thread **External_Event_Source** de l'utilisateur et l'automate comportemental généré conformément à notre patron. Sur l'exemple (partie haute), nous pouvons observer la spécification des propriétés temporelles (non fonctionnelles) du thread (période de 5000 millisecondes, échéance de 100 millisecondes...).

Exemple 5.1 – Description architecturale et comportementale d'un thread périodique

```
1   -- user declarative part

3   thread External_Event_Source
      properties
        Activate_Entrypoint_Source_Text    => "Event_Source.Init";
        Compute_Entrypoint_Source_Text     => "Event_Source.New_External_Event";
        Recover_Entrypoint_Source_Text     => "Event_Source.Recover";
8       Dispatch_Protocol                  => Periodic;
        Period                             => 5000 ms;
        Deadline                           => 100 ms;
        Compute_Execution_Time             => 0 ms .. 10 ms;
    end External_Event_Source;
13
    -- generated part

    thread implementation External_Event_Source.impl
      subcomponents
18      Error : data PolyORB_HI_Errors::Error_Kind;

      annex behavior_specification {**
        states
             stInit : initial state;
23       stDispatch : initial complete state;

        transitions
          tInit : stInit -[]-> stDispatch { External_Event_Source_Entrypoint! };
                                          -- call activate entrypoint
28
          tDispatch : stDispatch -[on dispatch sysDispatch]-> stDispatch {
            Event_Source::New_External_Event!;
            -- call compute entrypoint

33          PolyORB_HI_Generated_InS_AADL_Runtime::Send_Output!(Error);

            if (Error /= PolyORB_HI_Errors::Error_Kind.Error_None)
                  Event_Source::Recover_Entrypoint!
                -- call recover entrypoint
38          end if
          };
      **};
    end External_Event_Source.impl;
```

L'automate généré (partie basse de l'exemple) contient deux états : `stInit` définit l'état `initial` du processus avant la phase d'initialisation et `stDispatch` représente son état de suspension (`complete`). La transition de l'état initial à l'état de suspension correspond à la phase d'initialisation du composant. Lors de celle-ci, le sous-programme `External_Event_Source_Activate_Entrypoint` spécifié par l'utilisateur est exécuté une seule fois. Ensuite,

© 2012 Gilles LASNIER

l'automate passe dans une boucle infinie (rebouclement sur l'état `stDispatch`) et effectue l'exécution du sous-programme `External_Event_Source_Compute_Entrypoint` à chaque déclenchement périodique du composant. Ce sous-programme, spécifié par l'utilisateur, encapsule l'implantation du comportement applicatif du thread - *i.e* sa fonction de travail.

Discussion à propos de l'automate réduit

L'automate que nous proposons est volontairement réduit pour respecter les restrictions que nous avons définies (chapitre 3, section 4.4.2). Analysons sa structure :

- *Etats :* l'automate n'est composé que de deux états : l'état initial `stInit` et l'état de suspension `stDispatch`.
- *Transitions* : les transitions `tInit` et `tDispatch` expriment les changements d'état du thread. La première correspond à la phase d'initialisation du thread et déclenche l'exécution du sous-programme spécifié par la propriété `Activate_Entrypoint`. A l'aide de la spécification comportementale, nous rendons l'appel de ce sous-programme explicite.
- *Déclenchement :* Le déclenchement du thread est modélisé par la clause `on dispatch sysDispatch` conforme au standard. Le port spécifique `sysDipatch`[3] est ajouté à l'interface du composant et modélise l'événement déclencheur de l'exécution du composant.
- *Fonction :* L'exécution de la fonction de travail du thread, partie applicative spécifiée par l'utilisateur à l'aide de la propriété `Compute_Entrypoint`, est modélisée explicitement par un appel de sous-programme à chaque exécution du thread.

La même structure pour l'automate du thread sporadique a été définie. Seule change la sémantique du déclenchement de son exécution. Les automates réduits que nous proposons pour modéliser le comportement des composants threads restent conformes au cycle de vie décrit par le standard AADL (voir figure 5.4.1).

Remarques. Les états d'exécution définis par l'annexe comportementale (chapitre 4) n'interviennent pas dans l'automate du thread (l.21-23, exemple 5.1). L'expression des actions attachées à la transition de l'état `stDispatch` (l.30-38) suffit à décrire l'exécution du thread lors de son déclenchement.

Eléments d'analyse pour la vérification

La description architecturale initiale du composant thread présentée dans l'exemple 5.1 (partie haute) spécifie les propriétés non fonctionnelles du thread (par exemple, les propriétés temporelles priorité, échéance, WCET, etc.) utilisées par les outils d'analyses (analyse d'ordonnancement...). Nous avons contraint la structure de notre automate pour qu'ils respectent les restrictions définies pour la réalisation des systèmes critiques et les recommandations du profil Ravenscar (section 4.4.2). Ainsi, par construction, notre automate garantit l'analysabilité statique de l'application. Pour nous assurer de cette propriété, nous pouvons à présent vérifier les restrictions comportementales portant sur l'unique point de suspension d'une tâche (restriction RC1, issue du profil Ravenscar) modélisée par l'absence d'état `final` dans l'automate, et l'exécution infinie d'une tâche (restriction RC2, issue du profil Ravenscar) modélisée par la transition `tDispatch` ayant pour source et destination l'état de suspension `stDispatch` (boucle).

3. Le port `sysDispatch` est l'unique port intervenant dans notre modélisation, il modélise l'événement du système pour le réveil d'une tâche.

© 2012 Gilles LASNIER

La restriction RC2 pouvait auparavant être vérifiée sur la description architecturale (restriction RA2 que nous avions définie) par l'interprétation de la propriété `Dispatch_Protocol` renseignant si le thread était de type sporadique ou périodique. Cependant, la spécification du comportement du thread à l'aide d'une simple propriété (`Compute_Entrypoint`) ne permettait pas de vérifier l'adéquation des éléments spécifiés par ces propriétés. Par conséquent, l'automate que nous proposons permet d'affiner l'analyse.

Le chapitre 9 présente comment nous exprimons ces restrictions architecturales et comportementales à l'aide d'un langage d'expression de contraintes pour la vérification de notre modèle de code complet.

Eléments pour la génération de code

Exemple 5.2 – Thread périodique généré dans le langage Ada

```
1  -- (specification)
   package External_Event_Source_Task is
     Dispatch_Offset : constant Ada.Real_Time.Time_Span := Ada.Real_Time.To_Time_Span(0.0);
4    Task_Period     : constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(700.0);
     Task_Deadline   : constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(300.0);
     Task_Priority   : constant System.Any_Priority := 2;
     Task_Stack_Size : constant Natural := 100000;

9    task type External_Event_Source_Impl is
       pragma Priority      (Task_Priority);
       pragma Storage_Size (Task_Stack_Size);
     end External_Event_Source_Impl;
   end External_Event_Source_Task;
14
   -- (body)
   package body External_Event_Source_Task is

     Next_Deadline_Val : Time;
19
   task body External_Event_Source_Impl is
      Next_Start : Ada.Real_Time.Time;
      Error : PolyORB_HI_Errors;
   begin
24
      External_Event_Source_Entrypoint;                                    -- call the initialize entrypoint

      Suspend_Until_True (Task_Suspension_Objects (Entity));               -- wait for the system initialization

29    delay until (System_Startup_Time + Dispatch_Offset);                 -- wait startup time
      Next_Start        := System_Startup_Time + Dispatch_Offset + Task_Period;   -- compute next period
      Next_Deadline_Val := System_Startup_Time + Dispatch_Offset + Task_Deadline; -- compute next deadline

      loop    -- main loop
34       Event_Source.New_External_Event;                                  -- call the compute entrypoint

         Error := PolyORB_HI_Generated_InS_AADL_Runtime.Send_Output;       -- call the AADL runtime service

         if Error /= PolyORB_HI_Errors.Error_None then
39          Event_Source.Recover;
         end if;

         delay until Next_Start;                                           -- wait next start
         Next_Start        := Next_Start + Task_Period;                    -- compute next start time
44       Next_Deadline_Val := Ada.Real_Time.Clock + Task_Deadline;         -- compute next deadline
      end loop;
   end External_Event_Source_Impl;

   end External_Event_Source_Task;
```

L'exemple de code 5.2 montre le code Ada généré à partir de la spécification architecturale et comportementale du thread périodique de l'exemple AADL 5.1. La spécification architecturale du composant permet de générer la structure complète d'une tâche Ada périodique. Les propriétés `Period`, `Deadline` et `Priority` du modèle AADL (l.9-11, exemple 5.1) permettent la génération des constantes pour la configuration de la tâche (l.4-7, exemple 5.2). La structure de l'automate réduit (états, transitions, conditions de déclenchement) nous permet de générer le comportement de la tâche (initialisation, calcul de la prochaine période, etc.). Enfin, la

© 2012 Gilles LASNIER

partie fonctionnelle (applicative) de la tâche (l.34-40, exemple 5.2) est assurée par l'appel aux sous-programmes spécifiés par l'utilisateur à l'aide de propriétés AADL (l.5-7, exemple 5.1) et explicités dans l'automate généré (l.25 et 26-38, exemple 5.1). Ces observations montrent la mise en relation simple des éléments de notre automate avec les constructions du langage Ada.

Les règles de transformation d'un composant thread vers une tâche Ada sont présentées dans le chapitre 8. La sous-section présente la description comportementale des composants sous-programmes AADL et leur apport pour l'analyse et la génération de code.

5.4.2 Description comportementale des composants sous-programmes

Modélisation de l'automate d'un sous-programme

L'exemple de code AADL 5.3 montre la description architecturale et comportementale du sous-programme `New_External_Event`. Sa fonction consiste à déterminer et à envoyer une valeur d'interruption (un entier) selon une condition arbitraire.

La description architecturale du composant spécifie les paramètres d'entrée et de sortie, leur type et leur direction (l.6-9). Aucun sous-composant de donnée (modélisant par exemple une variable locale) n'est déclaré dans l'implantation du sous programme. La fonction de calcul modélisant le comportement du sous-programme est quant à elle décrite uniquement à l'aide de l'automate comportemental (l.22-28).

La structure d'automate que nous proposons est réduite et ne contient qu'un état (`stExec` l.17) et qu'une transition (`tExec` l.20) modélisant l'exécution du sous-programme. Ainsi, à chaque exécution du sous-programme, les actions (l.22-28) attachées à la transition sont réalisées.

Exemple 5.3 – Description architecturale et comportementale d'un sous-programme

```
1  -- Calculate an abitrary criterion for the sending
2  -- condition of external interrupts.

   subprogram New_External_Event
     features
       Interrupt_Count : in out parameter Interrupt_Counter;
7      Criterion       : in out parameter Base_Types::Natural;
       Divisors        : in     parameter Divisors_Array;
       Divisor         : in out parameter Divisor_Type;
     properties
       Compute_Execution_Time => 0 ms .. 5 ms;
12 end New_External_Event;

   subprogram implementation New_External_Event.impl
   annex behavior_specification {**
    states
17    stExec : initial final state;

    transitions
     tExec : stExec -[ ]-> stExec
      {
22      if (Criterion mod Divisors[Divisor] = 0)
          Divisor := Divisor + 1;
          Interrupt_Count := Interrupt_Count + 1;
        end if;

27      -- Evaluate the sending condition
        Criterion := Criterion + 1;
      }
   **};
   end New_External_Event.impl;
```

© 2012 Gilles LASNIER

Discussion autour de l'automate

L'automate comportemental décrit dans l'exemple 5.3 est conforme à la syntaxe et à la sémantique définies par l'annexe AADL. La fonction calculatoire pour déterminer si la valeur d'interruption doit être incrémentée est spécifiée à l'aide des langages d'**interaction** (accès aux paramètres d'entrée/sortie), d'**action** (structure conditionnelle) et d'expression (expressions arithmétiques et logiques) définis par l'annexe comportementale.

Eléments d'analyse pour la vérification

L'objectif initial de l'annexe comportementale est de fournir un prototype réaliste du comportement du sous-programme pour les outils d'analyse. La description architecturale permet l'étude de la structure interne du sous-programme - *i.e.* des sous-composants modélisant les variables d'instances de données utilisées par le sous-programme. Cette étude permet notamment d'analyser l'espace mémoire requis par celui-ci. La propriété `Compute_Execution_Time` spécifie le WCET du sous-programme, information utilisée pour les analyses temporelles. Ainsi, il est possible d'estimer (de manière pessimiste) le pire temps d'exécution d'une tâche en sommant les WCET des sous-programmes spécifiés dans sa séquence d'appels.

La description comportementale permet d'affiner ces analyses en explicitant les interactions, les opérations d'affectation, les tests conditionnels et les instructions de boucle lors de l'exécution du programme. Ainsi, il est possible d'exprimer de manière moins pessimiste le calcul du pire temps d'exécution.

De plus, nous présentons dans [Lasnier *et al.*, 2010] le raffinement du niveau d'abstraction d'une description architecturale à l'aide de l'annexe comportementale. Notamment, nous détaillons la modélisation d'une section critique par l'utilisation des primitives de prise/relâchement de verrous au sein d'un sous-programme. Ainsi, le niveau d'abstraction initial induit de la description architecturale (appels de sous-programme au sein d'un thread) se retrouve réduit par l'utilisation de primitives spécifiques au sein de l'automate comportemental du sous-programme, affinant la spécification d'une section critique et l'estimation plus précise du temps passé dans celle-ci. Ce raffinement rend moins pessimiste l'estimation du pire temps d'exécution du sous-programme et, par conséquent, celui de la tâche qui l'exécute.

Eléments pour la génération de code

L'annexe comportementale AADL n'est pas vouée à se substituer à un langage de programmation, mais l'étude des éléments de langage qu'elle propose (voir chapitre 4) révèle que certains éléments (constructions) ont une sémantique, voire une syntaxe, équivalente à certaines constructions d'un langage de programmation impératif comme C ou Ada. Dans le tableau 5.6, nous présentons les équivalences entre les éléments de langage de l'annexe comportementale et ceux du langage Ada.

Le langage d'*interaction* permet précisément de modéliser les appels de sous-programmes avec leurs paramètres respectifs. Le langage d'*action* définit quant à lui les structures conditionnelles (`if` ... `elsif` ... `else`) et les boucles de contrôle (`for`, `forall`, `while`, `do` ... `until`) classiques. Enfin, le langage d'*expression* est directement basé sur celui du langage Ada et permet de spécifier des expressions logiques, relationnelles et arithmétiques.

Enfin, l'intégration de l'annexe au sein de la description architecturale du composant permet la manipulation explicite des composants sous-programmes, des paramètres d'interface

© 2012 Gilles LASNIER

Eléments de l'annexe	Eléments du langage Ada	Fonction
Action de communication `spg!(p1,p2)`	Appels de fonctions/procédures	Exécution de routines
Action d'assignement `:=`	Affectation de valeurs	Affectation d'une valeur à une variable
`if ( ... ) ... elsif ( ... ) ... else ... end if`	`if ... then ... elsif ... then ... else ... end if`	Structures conditionnnelles
`for ( ... in ... ) { ... }`	`for ... in ... range ... loop ... end loop`	Compteurs de boucles
`forall ( ... in ... ) { ... }`	`for ... in ...'Range loop ... end loop`	Compteurs de boucles
`while ( ... ) { ... }`	`while ... loop ... end loop`	Boucle avec condition d'arrêt "au début"
`do ... until ( ... )`	`loop ... exit ... when ... end loop`	Boucle avec condition d'arrêt "à la fin"
Paramètres	Paramètres	Paramètres de routines
Composants et sous-composants de données (donnée partagée)	Variables, objet protégé	Manipulation de variables ou d'objet protégé
Langage d'expression	Langage d'expression	Expression calculatoire

TABLE 5.6 – Equivalences entre annexe comportementale et langage Ada

et des composants de données au sein même de l'automate comportemental. Cela renforce la cohérence entre spécifications architecturale et comportementale.

Contributions. Lors de l'étude des premières versions de l'annexe et de son implantation, nous avons relevé des manques et des incohérences concernant le support de certains éléments architecturaux (tableaux, types de données, sous-programmes pour la manipulation des objets protégés...) et de certains mécanismes pour l'utilisation de composants génériques paramétrables (présentés chapitre 4). Nous avons proposé des corrections et des modifications pour améliorer le support de ces éléments, leur sémantique et certaines règles de l'annexe (légale, consistance, nommage, etc.) lors de nos interventions dans le processus de rédaction et de validation de l'annexe ou lors des comités AADL. Certaines de ces contributions ont aussi été présentées à la communauté scientifique lors de conférences internationales [Lasnier *et al.*, 2010; Lasnier *et al.*, 2011a].

La structure des automates que nous avons définie pour les composants thread et sous-programme et les éléments de langage présentés dans le tableau 5.6 constituent le sous-ensemble des éléments de l'annexe comportementale que nous utilisons pour la définition de nos patrons de modélisation pour l'analyse et la génération de code. Ces éléments ont été sélectionnés pour leurs sémantiques équivalentes à des constructions du langage Ada ou parce qu'ils autorisent des transformations simples vers une ou un ensemble de contructions du langage Ada. Ce sous-ensemble d'éléments de l'annexe comportementale nous permet de compléter les descriptions architecturales des composants thread et sous-programme traduisant le comportement de l'application TR^{2}E.

© 2012 Gilles LASNIER

5.5 Synthèse : le profil AADL-HI Ravenscar

Dans ce chapitre, nous avons présenté les éléments architecturaux du langage AADL (composants, propriétés...) exploités par les outils d'analyse (voir tableau 5.2) et ceux impliqués pour la production automatisée du code source de l'application (voir tableau 5.5). Nous avons notamment expliqué comment remplacer les ports AADL et les connexions modélisant la communication entre les composants threads par un assemblage de composants architecturaux et comportementaux exprimant les mêmes propriétés, le même comportement et respectant les différentes restrictions que nous avons définies pour la réalisation des systèmes critiques et le respect du profil Ravenscar (garantissant l'analysabilité statique de l'application).

Ainsi, l'abstraction particulière des ports AADL est raffinée (par une transformation verticale avec préservation des propriétés) en un composant de donnée partagée et des composants sous-programmes modélisant une file d'attente globale de message et les routines pour sa manipulation. La modélisation du comportement de cette file et de ses routines est basée sur une implantation dans le langage Ada (respectant le profil Ravenscar) intégrée à l'intergiciel PolyORB-HI-Ada (un exécutif AADL) supportant les différentes configurations et propriétés des ports AADL. Dans le chapitre 7, nous détaillons comment est réalisée cette transformation par notre processus de raffinement.

Dans la dernière partie de ce chapitre (section 5.4), nous avons détaillé les éléments d'annexe comportementale qui nous ont permis de définir des patrons de modélisation comportementale (des automates) pour les composants thread et sous-programmes. Nous avons montré comment ces patrons préservent ou affinent l'analyse du système et autorisent une démarche de génération de code simple.

La combinaison des éléments architecturaux AADL, des éléments de langage de l'annexe comportementale et des restrictions que nous avons définies pour garantir l'analysabilité et une implantation du système correcte-par-construction (restrictions Ravenscar, restrictions pour la réalisation des systèmes...) définissent un profil de modélisation qui fixe le niveau d'abstraction de notre modèle de code complet analysable, que nous appelons dans la suite de ce manuscrit : le profil AADL-HI Ravenscar. Dans le chapitre 7, nous détaillons comment raffiner les composants architecturaux de la description initiale vers les composants supportés par notre profil.

Dans le chapitre suivant, nous présentons l'utilisation de ce profil pour la définition d'une bibliothèque de composants (architecturaux et comportementaux) modélisant les ressources intergicielles requises par un système TR^{2}E critique.

© 2012 Gilles Lasnier

Chapitre 6

Bibliothèque de Composants Intergiciels Dédiés aux Systèmes TR²E Critiques

SOMMAIRE

6.1 Introduction

Dans notre état de l'art et dans le chapitre 3, nous avons présenté les motivations de l'intégration des ressources intergicielles dès l'étape de modélisation afin d'autoriser la vérification et la validation du système critique complet (applicatif et exécutif) et de réduire la distance sémantique entre le modèle analysé et l'implantation finale. Lors de notre étude sur les différents intergiciels pour les systèmes TR²E, nous avons retenu l'architecture en services de l'intergiciel POLYORB-HI [Zalila *et al.*, 2008] dédié aux systèmes critiques. Celui-ci dispose des caractéristiques suivantes :

1. C'est un exécutif supportant les composants logiciels AADL, il fournit et assure les mécanismes pour la communication locale et distante des tâches applicatives (composant thread).

2. Il optimise l'utilisation des ressources intergicielles à partir de l'analyse de la description architecturale AADL du système, seuls les services requis par l'application sont ainsi sélectionnés, configurés et déployés (faible empreinte mémoire).

3. Son implantation en Ada est contrainte selon les recommandations du profil architectural Ravenscar (analyse statique du système).

4. Il s'intégre dans un processus de production automatisée du code source de l'application et s'interface avec les composants logiciels fournis par l'utilisateur.

Ce chapitre présente notre contribution sur la définition d'une bibliothèque de composants AADL architecturaux et comportementaux modélisant les ressources intergicielles requises (et leur utilisation) par un système critique. Ces composants sont modélisé à l'aide de notre profil AADL-HI Ravenscar présenté dans le chapitre précédent. Dans la section 6.2, nous rappelons brièvement le découpage en services de l'architecture de PolyORB-HI. La section 6.3 présente notre étude sur l'identification et l'intégration des services intergiciels au sein d'une description architecturale AADL et les choix de modélisation arbitraires se référant au support des composants logiciels AADL. Enfin, la section 6.4 présente l'architecture finale et les principaux composants intergiciels modélisés à partir de l'implantation Ada de PolyORB-HI.

6.2 L'intergiciel PolyORB-HI

Dans cette section, nous présentons l'architecture en services et leur rôle au sein d'une instance d'intergiciel dédiée à une application TR²E.

6.2.1 Intergiciel dédié et architecture

PolyORB-HI repose sur le concept d'intergiciel dédié. Une instance d'intergiciel dédiée à une application critique contient uniquement les entités requises par celle-ci. Plus l'architecture de l'intergiciel est flexible et modulaire, plus la configuration et la personnalisation des services et de leurs constituants sont fines. L'architecture en services de PolyORB-HI est issue du raffinement des services canoniques (adressage, transport, exécution, etc) proposé par [Kordon and Pautet, 2005]. Ce raffinement est effectué en fonction de leur degré de personnalisation (faible ou fort).

Ainsi, PolyORB-HI est divisé en deux parties. Le *noyau minimal* regroupe les services dont l'implantation est indépendante de l'application répartie (très faible personnalisation). Ces services en nombre limité sont sélectionnés (si requis) et configurés le cas échant.

La seconde partie est un ensemble de services fortement personnalisables qui viennent se greffer sur le noyau minimal. Les composants de ces derniers sont produits automatiquement et personnalisés à partir des caractéristiques extraites par le biais d'une analyse poussée de la description architecturale de l'application répartie en AADL. Le tableau 6.1 présente ce découpage en services et leur localisation. Dans les sous-sections suivantes, nous décrivons ces services afin de mieux comprendre leur rôle et leur intégration avec les composants applicatifs. Ceci nous permettra par la suite de déterminer les services explicites ou implicites spécifiés au sein d'une description architecturale AADL et les services manquants et de proposer une architecture pour notre bibliothèque de composants intergiciels.

© 2012 Gilles Lasnier

Services intergiciels	Personnalisation	Localisation
Adressage	forte	*généré*
Liaison	forte	*généré*
Activation	forte	*généré*
Typage	forte	*généré*
Parallélisme	faible	noyau minimal
Exécution	forte	*généré*
Représentation élémentaire	faible	noyau minimal
Représentation avancée	forte	*généré*
Interrogation	faible	noyau minimal
Interaction	forte	*généré*
Protocole	faible	noyau minimal
Couche basse de transport	faible	noyau minimal
Couche haute de transport	forte	*généré*

TABLE 6.1 – Localisation des services de l'intergiciel POLYORB-HI

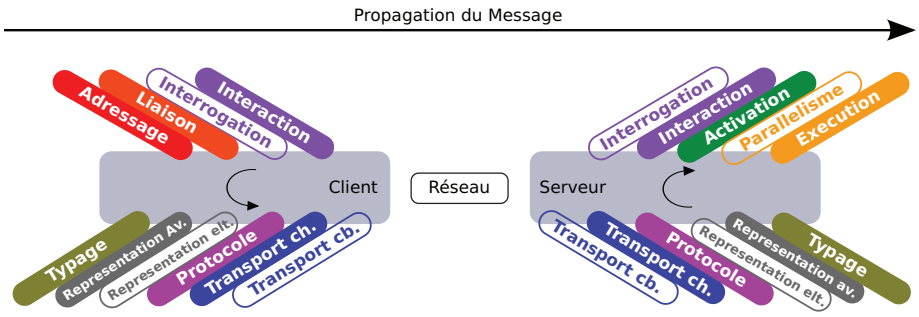


FIGURE 6.1 – Services de l'intergiciel POLYORB-HI

© 2012 Gilles LASNIER

6.2.2 Services du noyau minimal

La figure 6.1 présente les différents services intergiciels de l'architecture de PolyORB-HI et leur ordre d'exécution. Dans cette section, nous présentons succinctement les services faiblement personnalisables du noyau minimal et leur rôle.

Parallélisme

Ce service permet l'instanciation des archétypes décrivant le comportement des différents types de tâche supportés par l'intergiciel. Ainsi, pour chaque tâche spécifiée au sein d'un nœud de l'application répartie, un archétype correspondant à sa catégorie et au langage de programmation cible est automatiquement instancié.

Interrogation

Ce service implante les mécanismes basiques d'échanges de données ou d'événements entre les différentes tâches des nœuds de l'application TR²E. Ainsi, il réalise l'envoi de données d'une tâche vers une autre et le déclenchement d'activité des tâches à événements (sporadique). Son implantation décrit le comportement des routines d'envoi et de réception ainsi que les structures de données utilisées. Le service d'interaction (présenté section 6.2.3) instancie ces mécanismes et implante le comportement désiré pour chaque tâche en fonction des propriétés et de la topologie déduite de l'application.

Représentation élémentaire

Un service de représentation assure et gère la transmission correcte des données à travers le réseau entre une entité émettrice et une entité réceptrice. Il gère le caractère hétérogène ou homogène d'une plate-forme en controlant l'emballage et le déballage des données transitant dans les canaux de communication. Pour les types de données élémentaires - *i.e.* prédéfinis (entier, booléen, etc) -, ces mécanismes sont indépendants de l'application répartie. Seule l'analyse des communications entre les nœuds de l'application amène une configuration particulière des techniques d'emballage et de déballage, comme, par exemple, la copie mémoire des données dans un tampon de communication dans le cas de plates-formes homogènes.

L'implantation de ces mécanismes pour les types de données prédéfinis supportés par l'intergiciel forme le service de représentation élémentaire. Ils seront utilisés par le service de représentation avancée pour la composition de mécanismes d'emballage et de déballage pour les types de données plus complexes spécifiés par l'utilisateur.

Protocole

Le service de protocole assure la transmission d'un message entre deux entités. Il utilise, pour cela, le tampon de communication construit par le service de représentation et ajoute toutes les informations requises pour son routage vers l'entité destinataire. Il s'occupe aussi de la gestion du mode de communication sélectionné par le service de liaison.

Couche basse de transport

Le service de transport assure et gère l'utilisation des canaux de communication (ouverture/fermeture) utilisés pour l'échange de messages entre les nœuds. Son implantation dépend

© 2012 Gilles Lasnier

fortement du système d'exploitation ou de l'environnement d'exécution de l'application ainsi que des composants matériels relatifs au transport de données (bus de communication Ethernet, etc).

Le service de couche basse de transport contient les mécanismes d'accès direct au médium de communication entre deux nœuds. Il fournit les routines nécessaires pour l'échange d'information entre l'application et le pilote du périphérique de communication.

Si son implantation est indépendante des propriétés de l'application, son utilisation dépend fortement de la nature même de l'application. Dans le cas d'une application monolithique ce service n'est pas requis. Sa sélection est gérée par le service de couche haute de transport après analyse de l'application.

6.2.3 Services fortement personnalisables

Dans cette section nous décrivons brièvement les services fortement personnalisables de l'intergiciel POLYORB-HI (illustrés figure 6.1) :

Couche haute de transport

Le service de couche haute de transport fait le lien entre le service de protocole et la couche basse de transport requise. Cette partie fortement personnalisable sélectionne le service de couche basse de transport selon les informations extraites du service d'adressage et dépend exclusivement des caractéristiques de l'application répartie. Ce service implante trois routines. `Init` initialise l'ensemble des couches basses de transport requises par le nœud. `Send` invoque l'entité destinataire du message à l'aide de la couche basse de transport dans le cas d'une communication distante ou à l'aide de la routine `Deliver` (ci-dessous) dans le cas d'une communication locale. `Deliver` délivre le message reçu à la couche de protocole. Cette routine est invoquée soit par la couche basse de transport dans le cas d'une communication distante soit par la routine `Send` (décrite ci-dessus) dans le cas d'une communication locale.

Représentation avancée

Le service de représentation avancée est la partie personnalisable du service de représentation (voir représentation élémentaire, section 6.2.2). Il implante les routines d'emballage et déballage de données spécifiques pour la gestion des types complexes définis par l'utilisateur (par exemple, structures de données, type discriminant Ada, etc). Il s'appuie sur le service de représentation élémentaire.

Typage

Le service de typage gère les types de données requis et fournis à l'application répartie (spécifiés par l'utilisateur). Combiné avec le service de représentation, celui-ci permet l'emballage et le déballage des types de données complexes à partir des tampons de communication. Il assure ainsi la transmission correcte des données entre les différents nœuds de l'application. L'implantation de ce service dépend des propriétés de l'application.

POLYORB-HI utilise le langage AADL et son annexe de modélisation des données pour la spécification des types de données basiques et/ou complexes. Une vérification des restrictions

définies sur ces types est effectuée dans le but de garantir leur conformité vis-à-vis des systèmes critiques. Enfin, un générateur de code produit, à partir de ces composants modélisés les types de données correspondants dans le langage de programmation cible.

Adressage

Le service d'adressage gère les références - *i.e.* les adresses - des entités extérieures à l'intergiciel. Ce service permet de retrouver sans ambiguïté l'entité traitant les requêtes et permet aux entités extérieures de se connecter et d'envoyer des messages à celle-ci. La production des adresses dépend fortement du nombre, de la topologie et du type d'interaction entre les nœuds d'une application répartie. L'implantation de ce service est simplifiée par le caractère statique de l'application TR²E où la configuration des tâches et les canaux de communication sont connus et gérés lors de la phase d'initialisation de l'application.

Ainsi, l'implantation de ce service peut se réduire pour chaque entité à une table permettant de faire la correspondance entre entité destinataire connectée et la ressource de transport associée (par exemple, une socket) - *i.e.* la table de nommage.

Liaison

Ce service assure la construction des ressources de communication pour l'échange d'informations entre entités locales et/ou distantes. Il définit le mode de communication (passage par message, appel de procédure ou objet distant) et sélectionne les instances des services d'interaction, de représentation, de protocole et de transport requis dans le cas d'une communication distante. Ce service s'exécute au-dessus du service d'adressage et utilise les références produites par celui-ci.

L'implantation de ce service est facilitée part le caractère statique de l'application. La topologie de l'application permet la connaissance définitive du type de communication entre deux entités (locale ou distante) et ainsi la sélection des instances pertinentes des services du noyau minimal (couche basse de transport, représentation élémentaire, etc).

Interaction

Le service d'interaction gère les modes de communication entre les nœuds de l'application répartie. L'implantation de ce service décrit l'invocation des routines correspondantes au mode d'interaction choisi. Par exemple, si le mode est synchrone, le service d'interaction bloque l'entité émettrice jusqu'à l'obtention d'une réponse de la part de l'entité réceptrice. Dans le contexte d'un mode asynchrone, l'entité émettrice est libérée immédiatement après l'envoi de la requête.

PolyORB-HI raffine cette définition du service d'interaction en fonction de la personnalisation des composants. Ainsi, la partie faiblement personnalisable est implantée par le service d'interrogation (section 6.2.2) et la partie fortement personnalisable par le service d'interaction (même nom). Celui-ci réalise l'instanciation des archétypes des différents types d'interaction déduits des interfaces des composants logiciels (tâches et sous-programmes) décrits dans l'application répartie.

© 2012 Gilles Lasnier

Activation

Ce service gère l'activation de l'entité requise pour le traitement d'une requête à la livraison d'un message. Il crée, selon la nécessité et gére des ressources temporaires afin d'assurer le bon fonctionnement du traitement. Il n'est requis que par les entités qui peuvent recevoir des messages.

La personnalisation de ce service consiste à déterminer la tâche à activer lors de la réception d'un message par analyse de l'application répartie. L'implantation de ce service inclut l'acheminement des messages entre la couche protocolaire et la couche d'interaction ce qui permet de délivrer correctement le message à la tâche de réception.

Exécution

Ce service s'exécute au-dessus du service d'activation et utilise le service de parallélisme (section 6.2.2). L'entité sélectionnée et activée par le service d'activation est ici utilisée pour effectuer le traitement requis lors de la réception d'un message. Le service d'exécution gère alors l'allocation, la création et la destruction des tâches selon la politique de service choisie (mono-tâche, tâche par session, tâche par requête, etc). Son implantation existe exclusivement sur les entités réceptrices de messages.

A partir de l'analyse du nombre de tâches, de leur caractéristiques (type, priorité, taille de pile, etc) et de l'architecture de l'application, ce service crée statiquement l'ensemble des tâches requises par l'application. Cet ensemble de tâches comprend à la fois celles définies par l'utilisateur mais aussi les tâches supplémentaires (tâche de réception...) induites par les différentes couches du service de transport sélectionné.

6.3 Identification des services et choix de modélisation

Dans cette section, nous identifions les différentes informations implicites, explicites ou manquantes relatives aux ressources intergicielles, existantes au sein d'une description architecturale AADL. Ces éléments détermineront l'ensemble de composants intergiciels à modéliser. Puis, nous étudions brièvement les contraintes de modélisation issues des choix d'implantation relatifs à un exécutif supportant les composants logiciels AADL en nous basant sur POLYORB-HI.

6.3.1 Description architecturale AADL et services intergiciels

Pour déterminer les ressources intergicielles à modéliser et comment les intégrer à la description architecturale du système nous présentons ici notre cas d'étude Ravenscar (présenté dans le chapitre 10) sous sa représentation graphique, figure 6.2. Il s'agit d'un système de gestion de charges de travail (le nœud `Workload_Manager`) pouvant recevoir et traiter des interruptions (envoyées par le nœud `Interruption_Simulator`).

Nous y avons ajouté des annotations informelles traduisant les informations spécifiées par les propriétés AADL attachées à une partie des composants. Un code de couleur permet de distinguer les composants relatifs aux services intergiciels décrits dans la section précédente (présentés dans la figure 6.1).

Le service de transport est décrit par l'utilisation d'un composant matériel *bus* et d'une propriété AADL qui spécifie le protocole de transport (dans notre cas le bus SPACEWIRE). Un composant périphérique (`device`) spécificie le pilote (couche basse de transport) associé

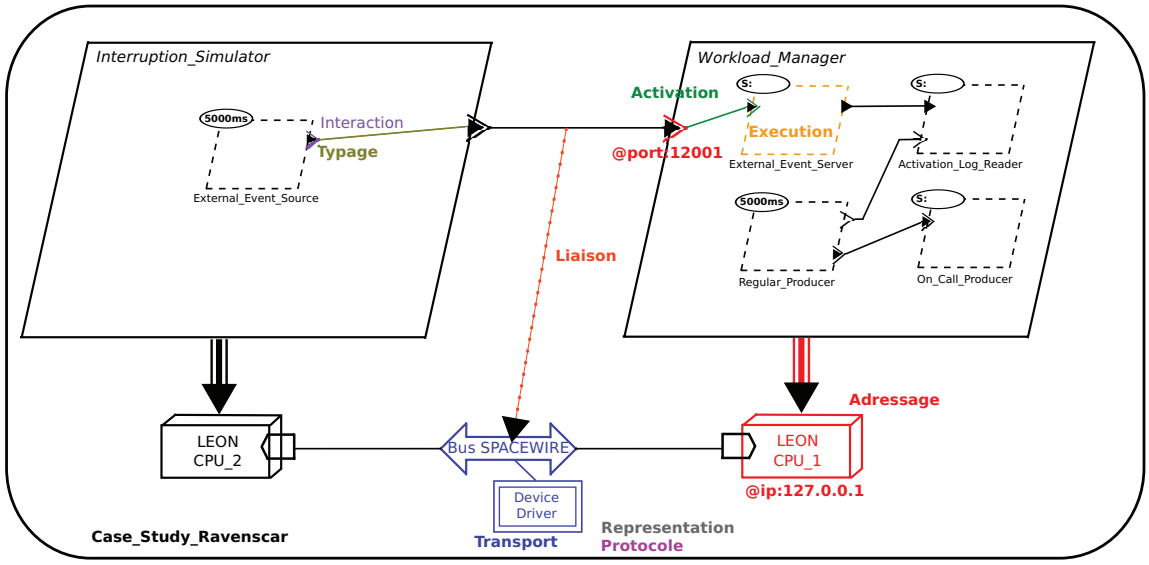


FIGURE 6.2 – Identification des services sur la description architecturale AADL

au bus et attaché à l'aide d'une propriété. Le service de typage est explicité par les types de données associées aux élements d'interfaces - *i.e* les ports des composants threads et processus. Le service de liaison n'est pas explicite, il est déduit de l'analyse des connexions entre les processus et leur *mapping* à un bus de transport. Le service d'adressage est déduit des informations spécifiées sur les composants processeurs (par exemple l'`IP`) et sur les ports des composants processus (par exemple le numéro de port pour la communication). Le service d'activation peut être déduit des connexions entre les ports de processus et les ports des threads qu'ils contiennent. Le service d'exécution est implicitement déduit de la sémantique des composants threads. Enfin, les services de représentation, de protocole, de parallélisme et d'interrogation ne sont pas représentés par la description.

Discussion

Cette rapide analyse de la description architecturale d'un système TR²E en AADL confirme qu'un certain nombre d'informations explicites ou implicites sur les ressources intergicielles requises sont présentes. Nous avons aussi pu déterminer quelles étaient les ressources manquantes et constater le manque d'information comportementale sur l'utilisation de l'ensemble de ces ressources.

Notre bibliothèque de composants AADL (présentée section 6.4) vise à expliciter l'architecture (constantes, types de données, données) et le comportement (interaction, mécanismes d'envoi et de réception de messages, etc) de ces ressources intergicielles à partir de la spécification initiale de l'utilisateur et d'un ensemble de composants préexistants. Pour la modélisation de ces composants, nous réutilisons le profil AADL-HI Ravenscar (détaillé chapitre 5, détermine le niveau d'abstraction des composants) et nous nous basons sur l'implantation des composants de l'intergiciel POLYORB-HI pour l'élaboration de spécifications comportementales, renforçant ainsi la cohérence entre le modèle et l'implantation.

© 2012 Gilles LASNIER

6.3.2 Gestion locale de la communication

L'implantation des mécanismes pour la gestion de la communication entre les tâches de l'application nécessite d'effectuer certains choix de conception et d'implantation. Le standard AADL définit de nombreux composants logiciels (données, données partagées, sous-programmes, tâches, ports) avec une sémantique précise et permettant une description fine des composants applicatifs du système. Cependant, celui-ci ne fournit que très peu de directives ou de recommandations sur la sémantique de l'exécutif AADL - *i.e.* l'intégration et l'exécution des composants applicatifs sur les composants intergiciels. Notamment, comme nous l'avons expliqué dans le chapitre 4, le standard AADL définit les interfaces d'un ensemble de routines pour accéder en écriture ou en lecture aux interfaces des tâches pour l'envoi et la réception de données et événements.

Ainsi, POLYORB-HI effectue des choix de conception pour le support et l'exécution des tâches spécifiées par l'utilisateur. De plus, ces choix sont contraints par les recommandations du profil Ravenscar. Notamment, la communication entre les tâches des nœuds est implantée selon un protocole de communication asynchrone basé sur l'échange de messages (encapsulant la donnée initiale). Pour chaque tâche, les ports AADL sont implantés comme une unique file d'attente globale de messages (un objet protégé Ada) pour assurer l'envoi et la réception correcte des informations à travers l'intergiciel.

Autre choix de conception : la couche basse de transport doit déclarer au moins une tâche de réception par nœud récepteur et assurer la délivrance des messages reçus en invoquant les routines de la couche haute de transport qui aiguillera ceux-ci vers la file d'attente de la tâche correspondante.

Discussion

Les ressources créées par ces différents services ont un impact direct sur l'analyse d'ordonnancement du système et justifie l'intégration de ces composants à la description architecturale initiale.

Dans le cadre de notre bibliothèque de composants intergiciels, nous modéliserons le service de protocole et le service d'interraction conformément à l'implantation de POLYORB-HI. Les couches basses de transport feront l'objet d'une modélisation partielle à l'aide de composants opaques en raison de leurs dépendances trop spécifiques à la runtime supportant l'intergiciel (dans notre cas la runtime Ada). Cette approche assure la prise en compte de ces composants lors des analyses effectuées sur le modèle.

6.4 La bibliothèque de composants POLYORB-HI-AADL

Cette section présente la modélisation des services intergiciels décrits dans les sections précédentes. La spécification des composants a été réalisée à l'aide de notre profil AADL-HI Ravenscar (présenté chapitre 3) nous garantissant l'analyse et la production automatique du code source du système à partir de ces composants. L'ensemble de ces composants constituent notre bibliothèque POLYORB-HI-AADL que nous détaillons dans la suite.

6.4.1 Architecture générale

La figure 6.3 présente l'architecture générale de notre bibliothèque de composants intergiciels et décrit les interactions entre composants (préexistants et générés). Chaque service

© 2012 Gilles LASNIER

est modélisé sous la forme d'un ou plusieurs paquetages AADL encapsulant les composants intergiciels qu'ils fournissent. Une partie de ces composants est produit automatiquement à partir de l'analyse de la description architecturale initiale du système. L'autre partie est définie sous la forme de composants génériques - *i.e.* spécifiés à l'aide de *prototypes*. Une partie des composants générés fournit les informations requises pour l'instanciation et la configuration des composants génériques.

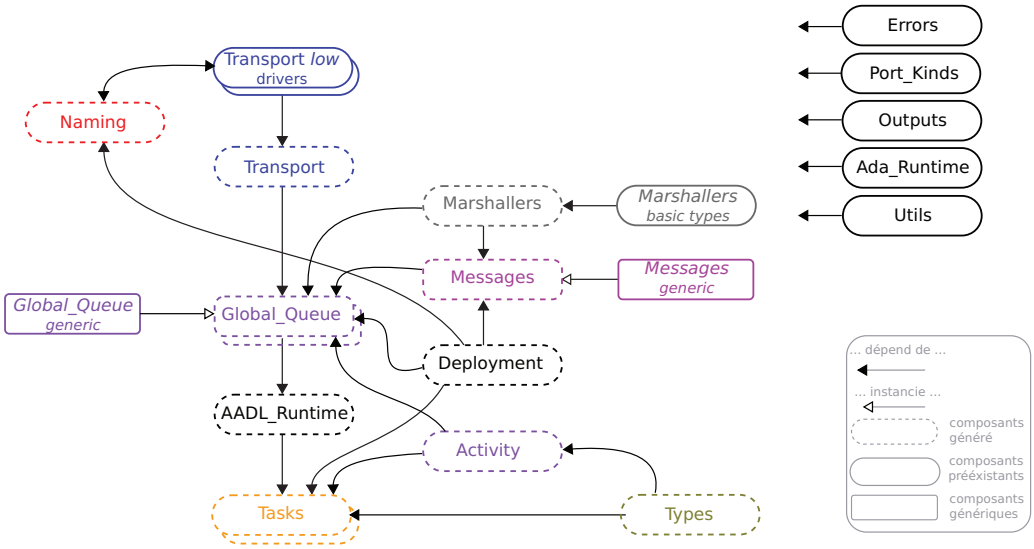


FIGURE 6.3 – Architecture de la bibliothèque POLYORB-HI-AADL

Les sous-sections suivantes décrivent les différents paquetage AADL de notre bibliothèque et les ressources intergicielles qu'ils contiennent. Le détail des composants générés et la manière dont ils sont produits sont présentés dans le chapitre 7 sous la forme de descriptions de transformation intégrées à notre processus de rafinement incrémental de modèles AADL.

6.4.2 Composants intergiciels générés

Deployment

Ce paquetage modélisent les composants nécessaires pour le déploiement des nœuds de l'application répartie sous une représentation équivalente aux constructions d'un langage de programmation. Ces composants sont générés exclusivement à partir de l'analyse de la topologie de l'application. Ils définissent des constantes et des énumérations utilisées pour la configuration de nombreux services intergiciels.

Naming

Ce paquetage modélise les services d'adressage et de liaison. Il définit les composants constituant les tables de nommage requises pour chaque service de transport utilisé par un nœud de l'application répartie. Ces composants sont générés à partir de l'analyse des connexions et des propriétés de déploiement (`Deployment::Location` et `Deployment::Port_Number`) spécifiés sur les composants processeurs et processus. Le composant modélisant la table de nommage est utilisé pour configurer la couche basse de transport sélectionnée.

© 2012 Gilles LASNIER

Transport

Ce paquetage modélise les routines (`Init`, `Send`, `Deliver`) définies par la couche haute de transport et permettant de relier la couche basse de transport sélectionnée au service de protocole.

Remarque. La visibilité de ce service permet de tracer précisément le chemin d'exécution complet lors de l'émission ou de la réception d'un message. Ces informations sont utiles pour estimer le pire temps d'exécution d'une tâche à l'aide de la séquence d'appel de sous-programmes déclarée par celle-ci.

Messages

Le paquetage `Messages` correspond au service de protocole. Il définit le tampon de communication globale `Message_Type` configuré en fonction de l'analyse de la taille maximale des données échangées entre les nœuds de l'application (utilisation du composant `Max_Payload_Size` généré à cet effet pour garantir son allocation statique et le respect de nos restrictions sur les systèmes critiques).

Le composant `Message_Type` généré est utilisé pour instancier et configurer les composants génériques du paquetage `Messages_Generic` (présenté dans l'annexe A) explicitant le protocole utilisé pour l'encapsulation d'une donnée dans un message et la construction de l'en-tête pour son routage correcte (voir section 6.4.3).

Remarque. Ces composants sont à la fois utiles pour l'analyse mémoire et l'estimation du pire temps d'exécution.

Marshallers

Les paquetages `Marshallers` et `Marshallers_Basic_Types` modélisent respectivement les services de représentation élémentaire et avancée. Les mécanismes d'emballage et de déballage des données dans un message y sont décrits à l'aide de composants sous-programmes et d'éléments de langage de l'annexe comportemententale. Ces routines sont définies pour l'ensemble des types de données transférables dans un message. Les routines pour emballer les données temporelles et celles pour emballer les énumérations correpondantes aux ports sources ou destinations d'un message sont aussi incluses dans ce paquetage.

Remarque. Ces paquetages définissent des composants qui permettent d'affiner l'analyse de flots de données (représentation des différentes transformations de données et flux d'exécution), l'analyse de mémoire (modélisation des différents types de données utilisés) et l'estimation du pire temps d'exécution d'une tâche dans le cas de communication distante.

Activity et Global_Queue

Ces deux paquetages réunifient le service d'interrogation et d'interaction. En effet, le paquetage `Activity` spécifie les composants de données représentant les interfaces d'une tâche applicative. Le paquetage `Global_Queue` spécifie les composants de la file d'attente globale de messages et ses différents interrogateurs assurant sa gestion et son utilisation (spécification du comportement) de manière sûre et déterministe.

© 2012 Gilles LASNIER

Un paquetage `Activity` est généré pour chaque tâche applicative du système. Les interfaces qu'il définit sont utilisées pour instancier et configurer les composants génériques du paquetage `Global_Queue_Generic` (voir section 6.4.3).

Remarque. La modélisation de cette file d'attente de messages est d'autant plus nécessaire qu'elle donne une visibilité claire et précise des mécanismes et des données mises en jeu pour l'échange de messages entre les tâches. Ceux-ci sont alors pris en compte par les outils d'analyse et contribue à réduire la distance sémantique entre le modèle et l'implantation du système.

AADL_Runtime

Dans le chapitre 4, nous avons expliqué que le standard AADLv2 définit des interfaces pour l'envoi et la réception de données des tâches. Ainsi, AADL standardise certaines interactions avec la runtime - *i.e.* l'intergiciel - en fournissant des interfaces modélisées par des composants sous-programmes (`Send_Ouput`, `Put_Value`, etc.).

Nous avons choisi d'intégrer ces composants sous la forme d'un service (`AADL_Runtime`) à notre architecture d'intergiciel pour les motivations suivantes. Premièrement, nous nous devons de rester conforme aux exigences du standard AADL notamment dans le but de simplifier la compréhension pour les utilisateurs et de garantir l'interopérabilité de nos modèles avec les outils de la communauté AADL.

Deuxièmement, ces interfaces standardisées amènent une modularité supplémentaire à notre exécutif AADL et permet de séparer clairement la partie applicative et la partie exécutive (intergiciel) au sein de la modélisation. Ceci renforce les possibilités de réutilisation, de maintenance et d'évolution de notre bibliothèque de composants en facilitant notamment le changement d'une partie de l'exécutif AADL.

Types

Ce paquetage modélise le service de typage. Il centralise au même endroit les différents composants spécifiant les types de données d'un nœud définies par l'utilisateur. La centralisation de ces données facilite leur réutilisation au sein des différents services intergiciels.

Tasks

Ce service réunifie les services de parallélisme et d'exécution. Pour chaque tâche applicative spécifiée par l'utilisateur, nous générons un paquetage `Task`. Celui spécifie le composant thread final utilisé dans notre modèle de code analysable. Nous reprenons la description architecturale de l'utilisateur et nous y ajoutons le patron comportemental correspondant au type de la tâche analysé. Le composant thread obtenu décrit l'architecture et le comportement de la tâche telle qu'elle sera implantée dans le langage de programmation cible.

6.4.3 Composants intergiciels préexistants

Paquetage Messages_Generic

Le paquetage `Messages_Generic` (voir annexe A) modélise les composants explicitant le comportement du service de protocole. Ces composants correspondent aux routines néces-

© 2012 Gilles LASNIER

saires pour lire et écrire les messages qui sont échangés entre les entités d'une application. La structure de ce paquetage est inspirée de celle du paquetage `PolyORB_HI.Messages` de l'intergiciel PolyORB-HI garantissant le respect de restrictions spécifiques aux systèmes critiques (taille de message borné et configuré statiquement, pas d'allocation de mémoire dynamique, pire cas des opérations de lecture et d'écriture sur les messages qui se font en un temps proportionnel à la taille de la donnée lue ou écrite, voir [Zalila *et al.*, 2008][chapitre 6]).

Ainsi, ces routines dépendent du composant `Message_Type` modélisant le message de taille borné (voir sous-section précédente). L'utilisation de *prototypes* nous permet de modéliser ces composants de manière générique, de les instancier et de les configurer[4] plus tard dans la modélisation.

Paquetage Global_Queue_Generic

Rappel. Dans PolyORB-HI, la file d'attente globale de messages d'une tâche est une concaténation de tableaux circulaires correspondant chacun à la file d'attente d'un des ports en entrée d'une tâche. Cette file implante le comportement des ports conformément à la sémantique AADL. Elle est définie à l'aide d'un objet protégé Ada qui encapsule l'ensemble des tableaux, des données et des routines nécessaires pour sa manipulation. Un certain nombre de routines assure l'envoi et la réception de données de manière déterministe et transparente à l'utilisateur. La structure de cette file répond aux recommandations du profil et est implantée dans le paquetage `PolyORB_HI.Thread_Interrogators` [Zalila *et al.*, 2008].

Le paquetage `Global_Queue_Generic` (voir annexe A) modélise cette file d'attente sous la forme d'un composant de donnée partagée AADL. Les routines nécessaires à sa manipulation sont modélisées à l'aide de composants sous-programmes et reliées à la file à l'aide des accesseurs de sous-programmes. L'ensemble des composants et la structure interne de la file ont été modélisés à l'aide de *prototypes* permettant la définition de composants génériques. Ainsi, pour chaque tâche applicative du système, les composants de déploiement générés (voir sous-section précédente) et l'analyse des connexions entre les tâches et les nœuds du système permettent l'instanciation et la configuration de ces composants au sein du paquetage `Global_Queue`.

Paquetage des couches basses de transport

Un service de couche basse de transport est fortement dépendant de l'environnement d'exécution de l'application répartie. La description complète du service impliquerait alors la modélisation d'un grand nombre de composants issus de cet environnement (par exemple, dans notre cas la runtime Ada) nécessitant un effort conséquent pour chaque environnement d'exécution supporté et dont la prise en compte par les outils d'analyses est quasi-nulle (complexité d'extraction d'information, granularité trop fine du composant ou composant non pertinent).

Cependant, certains composants (tâche de réception...) s'avèrent nécessaires pour une analyse plus fine de l'ordonnancement du système. Par conséquent, nous proposons la définition de paquetage de composants AADL modélisant uniquement les interfaces des primitives offertes par ce service. L'emploi des sous-programmes opaques, des composants thread et de propriétés AADL adéquates (`Compute_Execution_Time`...) nous permet de fournir une des-

4. Pour simplifier la compréhension nous traduisons le *binding* des protypes comme une instanciation.

cription architecturale de la couche basse de transport utilisable par les outils d'analyse et d'intégrer les pièces de code correspondant lors de la génération du système.

Paquetages additionnels

Nous avons modéliser un certain nombre de paquetages AADL supplémentaires fournissant des composants utilisés par les différents services intergiciels. Il s'agit de :

- `Errors`, qui modélise sous la forme d'un composant d'énumération les différentes valeurs d'erreurs des sous-programmes ;
- `Port_Kinds`, qui définit les différents types de ports sous la forme d'une énumération et des primitives pour la vérification des types de ports (représentation plus proche de l'implantation) ;
- `Ada_Runtime`, qui modélise de manière opaque certains types de données et routines de la runtime Ada (permet de compléter les descriptions de certains composants).
- `Utils`, qui fournit des routines opaques pour la transformation de données.

Ces paquetages nous permettent de compléter la modélisation de nos composants intergiciels afin de rester *légal* vis-à-vis du standard AADL (règles syntaxique, cohérence, etc.).

6.5 Synthèse

Dans ce chapitre, nous avons présenté notre bibliothèque de composants intergiciels permettant la modélisation et l'analyse complètes du système à partir de sa description AADL. Ces composants reposent sur une architecture modulaire et flexible (découpage en services) décrivant un intergiciel supportant les composants applicatifs AADL et dédié aux systèmes critiques (sélection des composants uniquement requis par l'application, etc). L'ensemble de ces composants a été modélisé à l'aide de notre profil AADL-HI Ravenscar.

Une partie de ces composants intergiciels est générée et configurée à partir d'une analyse poussée de la description architecturale initiale du système (modèle utilisateur). Les composants restants sont définis sous la forme de paquetage de composants AADL (architecture + comportement) génériques - *i.e.* basés sur l'utilisation de *prototypes*. Une partie des composants intergiciels générés (composants pour le déploiement, etc) est utilisée pour l'instanciation et la configuration de ces paquetages.

L'architecture que nous proposons est un raffinement de l'architecture de l'intergiciel PolyORB-HI. Les spécifications comportementales des composants ont été réalisées en fonction des implantations des constructions de cet intergiciel. Ceci renforce la cohérence entre le modèle et l'implantation finale du système.

Le chapitre suivant explique la manière dont ces composants sont produits, instanciés, configurés et intégrés automatiquement à la description initiale du système à l'aide d'un processus de raffinement incrémental basé sur la transformation de modèles AADL. L'issue du processus de raffinement est l'obtention de notre modèle de code complet (applicatif et exécutif) nécessaire à l'analyse et à la production automatique du sytème critique.

© 2012 Gilles Lasnier

Chapitre 7

Processus de Raffinement Incrémental dédié aux Systèmes Critiques

SOMMAIRE

7.1 Introduction

Dans le chapitre précédent, nous avons défini une représentation d'un intergiciel supportant l'exécution des composants applicatifs AADL. L'architecture en services de cet intergiciel a été spécifiée à l'aide de notre profil AADL-HI Ravenscar (défini chapitre 5) restreignant l'utilisation du langage AADL (composants, éléments d'interfaces, attributs, etc) et de son annexe

comportementale aux seules constructions préservant l'analyse et autorisant une démarche de génération de code. Chaque service s'exprime sous la forme d'un paquetage AADL déclarant les ressources requises par l'application - *i.e.* des composants architecturaux et comportementaux.

L'intégration de ces composants au sein de la description architecturale initiale de l'application est assurée par un processus de raffinement incrémental basé sur des transformations de modèles. Celui-ci mène deux objectifs :

1. La transformation des composants architecturaux du modèle initial vers les constructions de notre profil AADL-HI Ravenscar.
2. La génération, la configuration et l'intégration des composants intergiciels selon les besoins de l'application répartie.

L'issue de ce processus est la production d'un modèle de code complet autorisant l'analyse de la partie applicative et de la partie exécutive du système critique. Enfin, le niveau d'abstraction de ce modèle autorise la production automatique du code source des composants applicatifs et intergiciels du système à l'aide de traductions simples des éléments architecturaux et comportementaux vers des constructions sémantiquement équivalentes d'un langage de programmation impératif comme Ada.

Ce chapitre est organisé de la manière suivante. La section 7.2 présente les vérifications et la validation du modèle de l'utilisateur qui servent de point d'entrée pour le raffinement. La section 7.3 détaille les descriptions de transformation du processus de raffinement. La section 7.4 présente les vérifications supplémentaires pour la validation du modèle de code obtenu à l'étape précédente. La section 7.5 conclut ce chapitre.

7.2 Validation du modèle AADL utilisateur

La figure 7.1 illustre la première étape de notre processus de raffinement. Celle-ci consiste à valider le modèle de l'utilisateur qui sera utilisé comme point d'entrée par notre processus. Celui-ci doit fournir une description complète de l'assemblage des composants architecturaux, des connexions et des informations pour le déploiement des composants logiciels sur la plateforme matérielle cible.

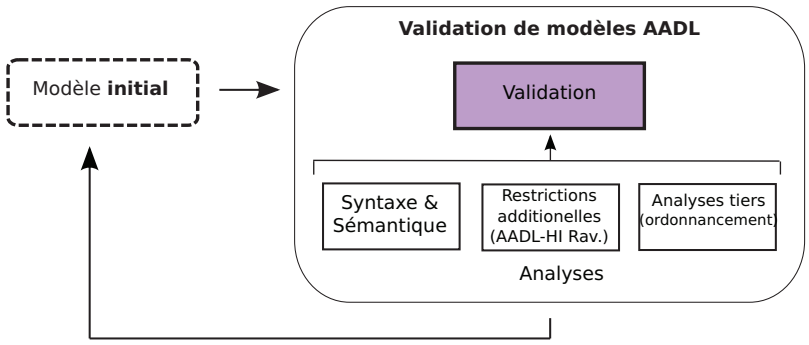


FIGURE 7.1 – Schéma de la phase de validation du modèle AADL utilisateur

Pour garantir le déroulement correct de notre processus de raffinement, nous devons nous assurer de :

1. La légalité au sens AADL du modèle initial (syntaxe et sémantique).

© 2012 Gilles LASNIER

2. La compatibilité des composants avec notre profil AADL-HI Ravenscar.
3. La présence des informations de déploiement qui seront utilisées pour la sélection et la configuration des composants intergiciels.
4. L'ordonnançabilité du système avant le raffinement (faisabilité).

La vérification de ces éléments garantit la présence de toutes les informations requises comme préconditions pour nos règles de tranformation dédiées au raffinement des composants architecturaux et à la génération des composants intergiciels.

7.2.1 Analyses syntaxique et sémantique

Lors de ces analyses, nous vérifions que le modèle AADL est conforme aux spécifications du standard AADLv2. Il s'agit de la vérification des règles syntaxiques et sémantiques du standard. Les règles sémantiques assurent que la spécification architecturale de l'application est un système réalisable dont l'assemblage des composants est cohérent. Il s'agit notamment, dans notre cas, de nous assurer que :

1. Chaque modèle doit contenir au moins un composant système (composant `system`) décrivant la hiérarchie des composants spécifiés (pour l'instanciation AADL, voir 4).
2. Dans le cadre d'une application répartie multi-processeurs, chaque système doit contenir au moins un bus de communication reliant les processeurs.
3. Chaque processus doit contenir au moins un fil d'exécution (composant `thread`).

Ces règles peuvent être vérifiées automatiquement à l'aide d'un compilateur dédié au langage AADL (par exemple, OCARINA) ou à l'aide d'un langage de spécification de contraintes (REAL ou OCL).

7.2.2 Analyses supplémentaires

Les analyses supplémentaires concernent la vérification de la spécification correcte des composants de données, des informations de déploiement et du respect des restrictions Ravenscar. Ces analyses sont effectuées en plus de celles réalisées par un outil d'édition de modèles AADL. Le tableau 7.1 récapitule l'ensemble des propriétés AADL impliquées et leur rôle.

Types de données

Cette vérification vise à satisfaire les contraintes liées aux systèmes critiques sur les données définies dans le chapitre 3. Il s'agit de s'assurer de la spécification de la taille mémoire, de la politique d'accès (lecture et/ou écriture), de la taille bornée des tableaux pour les composants de données.

Dans le cas où l'une de ces propriétés est manquante, nous l'indiquons à l'utilisateur, qui peut choisir de modifier ou non son modèle. Si aucune modification n'est apportée, alors nous procédons à l'ajout des propriétés manquantes avec une valeur par défaut lors du raffinement.

Restrictions architecturales Ravenscar

Nous nous assurons que les restrictions architecturales du profil Ravenscar (RA1, RA2 et RA3, chapitre 3 section 3.4.2) sont satisfaites. Si une restriction est invalidée à cause d'une

Composants	**Propriétés**	**Rôle dans la modélisation**
Data	`Source_Data_Size`	Spécifie la taille mémoire requise pour une instance de la donnée dans le système
	`Access_Right`	Spécifie la politique d'accès à la donnée, les différentes politiques sont : *lecture*, *écriture*, *lecture/écriture* et *par méthode*
	`Data_Model::Dimension`	Spécifie la dimension de la donnée type tableau modélisé
	`Concurrency_Control_Protocol`	Protocole de concurrence pour l'accès à l'objet protégé, sa valeur doit être `Priority_Ceiling`
	`Deployment::Priority`	Spécifie la priorité associée à l'objet protégé
Thread	`Dispatch_Protocol`	Spécifie le type de déclenchement (périodique, sporadique, ...)
	`Period`	Spécifie la période
	`Deadline`	Spécifie l'échéance
	`Priority`	Spécifie la priorité
	`Compute_Execution_Time`	Spécifie le pire temps d'exécution du thread
	`Compute_Entrypoint`	Spécifie la fonction de calcul du thread (sous-programme)
Process	`Deployment::Location`	Spécifie l'adresse `IP` (cas BSD socket)
	`Deployment::Port_Number`	Spécifie le numéro du port pour la communication (cas BSD socket)
Processor	`Execution_Platform`	Spécifie la plate-forme d'exécution cible
	`Scheduling_Protocol`	Spécifie le protocole d'ordonnancement autorisé
	`Preemptive_Scheduler`	Spécifie que l'ordonnanceur du processeur peut préempter les tâches
Bus	`Deployment::Transport_API`	Spécifie la couche basse de transport utilisée pour les communications distantes
	`Deployment::Protocol`	Spécifie le protocole utilisé pour échanger les données à travers les bus

TABLE 7.1 – Attributs AADL requis pour les analyses supplémentaires

© 2012 Gilles LASNIER

propriété manquante sur un composant, nous avertissons l'utilisateur qu'une propriété avec une valeur par défaut est ajoutée.

Analyse des informations de déploiement

Dans le chapitre précédent, nous avons expliqué l'utilité de la spécification des informations de déploiement sur la description architecturale intiale pour la configuration des services intergiciels (tables de routage des nœuds, sélection de la couche basse de transport, etc). Il s'agit essentiellement des propriétés AADL de l'ensemble de propriétés `Deployment` définies à cet effet. Ainsi, nous nous assurons que :

- chaque processus dispose des informations sur sa localisation (`IP`, `PID`...) et sur ses ressources matérielles de communication (numéro de port, etc.) ;
- chaque processeur spécifie la propriété identifiant la plate-forme sur laquelle le processus (ou nœud) s'exécute ;
- chaque bus spécifie les propriétés relatives à la sélection de la couche basse de transport et du protocole.

Ces propriétés sont rappelées dans le tableau 7.1. Dans le cas où l'une de ces propriétés est manquante, celle-ci sera générée avec une valeur conventionnelle lors du processus de raffinement.

Analyse de l'ordonnancement du système

Une première analyse d'ordonnancement est menée pour valider la faisabilité du système sans la prise en compte des ressources de l'exécutif AADL. Ceci nous permet de nous assurer que nous disposons de toutes les informations non fonctionnelles (période, échéance, etc.) sur les tâches applicatives du système nécessaires pour le raffinement et l'intégration de nos spécifications comportementales sur les composants threads.

De plus, les résultats obtenus pourront être comparés avec ceux produits par l'analyse du modèle complet obtenu à la fin du processus de raffinement. Nous pourrons ainsi étudier l'impact de l'intégration des ressources intergicielles au sein du système.

La seconde étape de notre processus est une suite de transformation incrémentale du modèle initial pour le raffinement des composants vers les constructions autorisées par notre profil AADL-HI Ravenscar et la production, la configuration et l'intégration des composants intergiciels. La section suivante décrit les différentes descriptions de transformation constituant ce processus.

7.3 Processus de raffinement

Nous avons présenté, dans le chapitre 5, les patrons comportementaux, pour l'analyse et la génération, explicitant le comportement des tâches applicatives à partir de leur description architecturale initiale. Dans le chapitre 6 nous avons détaillé les différents composants constituant les services intergiciels requis pour le support et l'exécution de ces tâches. Les patrons et les composants modélisés ont été réalisés à l'aide de notre profil AADL-HI Ravenscar.

Nous présentons, dans cette section, l'intégration de ces différents éléments au sein de la description initiale de l'application. Ce raffinement est réalisé par une chaîne de transfor-

© 2012 Gilles LASNIER

mations verticales sur une modélisation AADL (endogène). Nous détaillons par la suite les objectifs et les composants raffinés ou produits pour chaque transformation menée.

7.3.1 Principe et application des transformations

La figure 7.2 décrit la chaîne de transformation assurant la phase de raffinement. L'intégration de chaque service intergiciel est réalisée par une transformation distincte. Nous ajoutons une transformation supplémentaire à la fin de la chaîne pour assurer la construction du système AADL final.

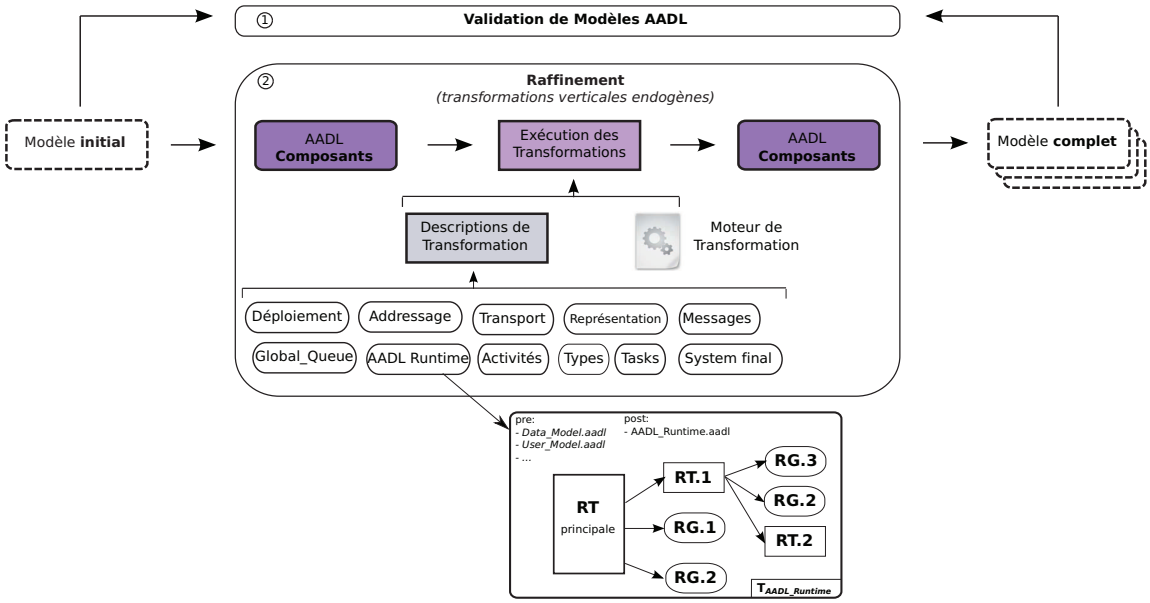


FIGURE 7.2 – Principe et application de la chaîne de transformation

Structure d'une transformation et types de règle

Les transformations sont structurées de la façon suivante (voir encadré bas figure 7.2). Chaque description spécifie des dépendances (modèle(s) source(s) et interrogateurs de modèle), des préconditions sur les dépendances (des contraintes sur le composant racine du modèle source), le modèle cible qui contiendra les composants produits et les règles de transformation exécutées. Nous distinguons trois types de règles :

1. Une règle principale (`RP`) encapsule la stratégie d'application des règles de transformation et de génération.
2. Une règle de transformation (`RT`) produit un composant ou un ensemble de composants à partir des informations extraites du modèle.
3. Une règle de génération (`RG`) produit un composant en fonction de paramètres spécifiques.

Une règle `RT` peut faire appel à des règles `RT` et `RG`. Une règle `RG` ne peut faire appel qu'aux règles `RG`. Enfin, `RT` et `RG` peuvent être utilisées par plusieurs descriptions de transformation (généricité). Le résultat de chaque transformation de la chaîne est la production d'un paquetage AADL légal et valide au sens AADL.

© 2012 Gilles LASNIER

Dans les sections à venir, nous rappelons les objectifs d'une transformation et nous discutons de la préservation des propriétés (sémantique et comportement). Puis, nous donnons la règle principale et nous présentons, dans un tableau ordonné, l'ensemble des règles de transformation et de génération spécifiées par la transformation.

Exécution des transformations

La figure 7.2 illustre le principe et l'application du processus de raffinement comprenant onze transformations. Toutes ces transformations sont obligatoires excepté les transformations `Transport`, `Naming` et `Tasks` qui dépendent du caractère de l'application (communication distante...) ou de l'utilisation initiale de description comportementale pour les composants threads compatible avec les restrictions de notre profil AADL-HI Ravenscar.

7.3.2 Transformation Déploiement ($T^{deployment}$)

Objectifs

Cette transformation explicite les informations pour le déploiement des nœuds de l'application TR^2E. Elles sont déduites de l'analyse de la topologie du système. En particulier, un nœud doit connaître les autres nœuds auxquels il est connecté. Une tâche (entité) du nœud doit connaître les différentes tâches avec lesquelles elle communique et les éléments d'interfaces avec qui elles échangent des informations. Enfin, nous voulons que l'accès aux informations citées ci-dessus soit déterministe.

Pour expliciter ces informations de déploiement, nous construisons des énumérations et des tables permettant l'association des éléments d'interfaces, des tâches et des nœuds. Celles-ci sont explicitées dans le tableau 7.2. Une partie de ces énumérations et de ces tables permettent de compléter l'initialisation des couches basses de transport en ouvrant le nombre correct de canaux de communication vers chaque nœud distant.

Préservation des propriétés

Le résultat de la transformation $T^{deployment}$ est un paquetage de composants AADL permettant la manipulation directe (sous forme de tableaux) et déterministe des informations sur la topologie de l'application distribuée. Il n'y a donc aucune perte de propriété sémantique ou comportementale. Ces composants tableaux AADL s'utilisent comme ceux d'un langage de programmation classique. Lors de cette transformation, il y a donc une réduction du niveau d'abstraction du modèle initial.

Composants produits par la transformation

Les composants générés par cette transformation sont détaillés dans le tableau 7.2. Les règles pour la génération de ces composants en Ada sont décrites dans le chapitre 8.

Règle principale

L'algorithme 1 décrit la règle principale s'appliquant à chaque nœud (composant `process`) du système (l.1). Le paquetage contenant les composants produits est créé à l'aide de la règle de génération `Create_Package` (l.2).

© 2012 Gilles LASNIER

Paquetage de composants AADL pour le déploiement ($T^{deployment}$)		
Rôle	**Modélisation AADL**	**Règles**

Enumération représentant les nœuds de l'application connectés au nœud analysé, y compris lui-même.	`Node_Type` : composant `data`
	Propriétés `Data_Model` associées :
	`Data_Representation => Enum`
	`Enumerators => <`*list_connected_nodes*`>`
	`Representation => <`*representation_clause*`>`
		RT.1

Enumération représentant les tâches de l'application connectées (à travers les autres nœuds) au nœud analysé, y compris les siennes (utilisé par RG.1).	`Entity_Type` : composant `data`
	Propriétés `Data_Model` associées :
	`Data_Representation => Enum`
	`Enumerators => <`*list_connected_entities*`>`
	`Representation => <`*representation_clause*`>`
		RT.2

Enumération représentant les ports des tâches de l'application connectés (à travers les tâches des autres nœuds) au nœud analysé, y compris les siens (utilisé par RG.2).	`Port_Type` : composant `data`
	Propriétés `Data_Model` associées :
	`Data_Representation => Enum`
	`Enumerators => <`*list_connected_ports*`>`
	`Representation => <`*representation_clause*`>`
		RT.3

Table définissant l'association entre énumérateurs de tâches et énumérateurs de nœuds.	`Entity_Table` : composant `data`
	Propriétés `Data_Model` associées :
	`Data_Representation => Array`
	`Dimension => <Entity_Type>`
	`Base_Type => <Node_Type>`
	`Initial_Value => <`*list_connected_entities*`>`
	`Representation => <`*list_connected_nodes*`>`
		RG.1

Table définissant l'association entre énumérateurs de ports et énumérateurs de tâches qui les contiennent.	`Port_Table` : composant `data`
	Propriétés `Data_Model` associées :
	`Data_Representation => Array`
	`Dimension => <Port_Type>`
	`Base_Type => <Entity_Type>`
	`Initial_Value => <`*list_connected_ports*`>`
	`Representation => <`*list_connected_entities*`>`
		RG.2

TABLE 7.2 – Composants et règles issus de la transformation $T^{deployment}$

© 2012 Gilles LASNIER

Le modèle de l'utilisateur et l'ensemble de propriétés `Data_Model` sont référencés comme dépendances (l.4). Conformément à notre profil AADL-HI Ravenscar, nous utilisons l'annexe de modélisation des données pour la spécification complète de nos différents types ou instances de données.

Enfin, l'exécution d'une transformation nécessite un ensemble de règles de transformation et de génération (l.5). Celui-ci doit être ordonné car une partie des composants produits est réutilisée par d'autres règles (par exemple, le type de donnée `Entity_Type` produit par la règle RT.2 et réutilisé par la règle RG.1).

Algorithme 1: Règle principale `RP` de $T^{deployment}$

```
1 foreach subcomponent process PROC of system SYS do
2 |   Deployment ← Create_Package(PROC.name, "_Deployment")
3 |   PreConditions<> ← <SYS, PROC, Deployment>
4 |   Dependencies<> ← <User_Model, Data_Model>
5 |   Execute(PreConditions<>, Dependencies<>, Deployment, rules=[table 7.2])
6 end
```

Règles de transformation et de génération

Nous décrivons ici uniquement les algorithmes des règles `RT.1` et `RG.1`, les autres règles de la transformation $T^{deployment}$ sont définies dans la même logique.

La règle `RT.1` (algorithme 2) génère l'énumération `Node_Type` représentant l'ensemble des nœuds de l'application connectés au nœud analysé. L'interrogateur de modèle `Get_Connected_Nodes` (l.1) construit la liste des nœuds connectés au nœud analysé à partir de l'analyse des connexions du modèle.

Nous avons défini des règles de génération pour l'ensemble des types de propriétés AADL et pour l'ensemble des composants spécifiés lors de nos transformations. Les propriétés de l'annexe de modélisation `Data_Representation`, `Enumerators`, `Representation` requises par notre profil AADL-HI Ravenscar sont produites aux lignes 2, 3 et 4 et spécifient le type de la donnée (énumération), les différentes valeurs d'énumérateurs et les clauses de représentation pour la distinction de ces derniers. Enfin, la règle de génération `Gen_Data_Type` est utilisée pour générer le composant `Node_Type` avec les propriétés qui lui sont associées.

Algorithme 2: Règle de transformation `RT.1` - Composant `data` Node_Type

```
1 Connected_Nodes<> ← Get_Connected_Nodes(SYS, PROC)
2 Data_Representation ← Gen_Data_Representation("Enum")
3 Enumerators ← Gen_Enumerators(Connected_Nodes<>)
4 Representation ←
  Gen_Representation(Get_Clause_Representation(Connected_Nodes <>))
5 Property_Association<> ← <Data_Representation, Enumerators, Representation>
6 Node_Type ← Gen_Data_Type("Node_Type", Property_Association <>)
```

La règle `RG.1` (algorithme 3) génère le tableau `Entity_Table` qui effectue l'association entre les énumérateurs des tâches et les énumérateurs des nœuds. Celle-ci utilise notamment les composants générés `Entity_Type` (l.4, issu de `RT.2` tableau 7.2) et `Node_Type` (l.5, issu de `RT.1` tableau 7.2) pour la dimension et le typage des éléments. L'interrogateur `Get_Element`

© 2012 Gilles LASNIER

(I.4) permet de récupérer un composant dans le paquetage cible ou dans les paquetages AADL spécifiés dans les dépendances de la transformation.

Algorithme 3: Règle de génération `RG.1 - Entity_Table`

```
1  Connected_Nodes<> ← Get_Connected_Nodes(SYS, PROC)
2  Connected_Entities<> ← Get_Connected_Threads(SYS, PROC)
3  Data_Representation ← Gen_Data_Representation("Array")
4  Dimension ← Get_Element(Deployment, Entity_Type)
5  Base_Type ← Gen_Base_Type(Get_Element(Deployment, Node_Type))
6  Initial_Value ←
   Gen_Representation(Get_Clause_Representation(Connected_Entities<>))
7  Representation ← Gen_Representation(Connected_Nodes<>)
8  Property_Association<> ←
9  <Data_Representatation, Dimension, Base_Type, Initial_Value, Representation>
10 Entity_Table ← Gen_Data_Type("Entity_Table", Property_Association<>)
```

7.3.3 Transformation Adressage (T^{naming})

Objectifs

La description de transformation T^{naming} assure l'intégration du service d'adressage. Elle permet la génération des tables statiques de nommage pour chaque service de couche basse de transport utilisé par les nœuds de l'application répartie.

Préservation des propriétés

La table de nommage d'un nœud centralise toutes les informations nécessaires pour la création des canaux de communication requis par l'application répartie. Elle sert notamment à la configuration du service de couche basse de transport. La transformation T^{naming} que nous proposons ne crée aucune information supplémentaire excepté dans le cas où aucune information n'est spécifiée sur la description architecturale du nœud. Des valeurs conventionnelles par exemple `127.0.0.1` pour l'adresse et *6667* pour le port sont utilisées dans ce cas. Aucune modification sémantique ou architecturale n'est apportée sur les composants assurant ainsi la préservation des propriétés. Cette transformation contribue simplement à expliciter les structures de données pour les services d'adressage et de liaison et, par conséquent, à réduire le niveau d'abstraction du modèle initial.

Composants produits par la transformation

Le tableau 7.3 récapitule les différents composants générés par la transformation T^{naming}. Chaque table de nommage possède une entrée `Naming_Entry` qui renseigne les informations de déploiement déterminées à partir de l'analyse des propriétés des nœuds du modèle initial (ports ou canaux de communication, adresses réseaux, etc.).

© 2012 Gilles LASNIER

Paquetage de composants AADL du service d'adressage (T^{naming})		
Rôle	**Modélisation AADL**	**Règles**
Type de donnée représentant le port de destination du nœud connecté au nœud courant.	`Node_Port` : composant `data` (entier) Propriété `Data_Model` associée : `Data_Representation => Integer`	RG.3
Type de donnée représentant l'adresse de destination du nœud connecté au nœud courant.	`Node_Addr` : composant `data` (string) Propriétés `Data_Model` associées : `Data_Representation => String` `Dimension => (16)` (arbitraire)	RG.4

Structure de donnée représentant une entrée (port et adresse) d'un nœud dans la table de nommage.	`Naming_Entry` : composant `data` (structure)
	Propriétés `Data_Model` associées :
	`Data_Representation => Struct`
	`Base_Type => <Node_Port, Node_Addr>`
	`Element_Names => <`*"thePort", "location"*`>`
		RG.5

Pour chaque nœud connecté, générer l'entrée correspondante de la table de nommage.	`XXX_Naming_Entry` : un composant `data`
	Propriétés `Data_Model` associées :
	`Base_Type => <Naming_Entry>`
	`Element_Names => <`*"Location", "thePort"*`>`
	`Initial_Value =>`
	<@location, @port_number>
	Propriété standard associée :
	`Access_Right => read_only`
		RT.4

Type de donnée tableau représentant la table de nommage.	`Naming_Table_Type` : composant `data` (tableau)
	Propriétés `Data_Model` associées :
	`Data_Representation => Array`
	`Base_Type => Naming_Entry`
	`Dimension => Deployment::Node_Type`
		RG.6

Table de nommage des nœuds connectés au nœud courant, y compris lui-même (utilisation des entrées générées par `RT.4`).	`Naming_Table` : composant `data`
	Propriétés `Data_Model` associées :
	`Base_Type => Naming_Table_Type`
	`Element_Names => <`*list_connected_nodes*`>`
	`Initial_Value =>`
	<*list_generated_naming_entry*>
	Propriété standard associée :
	`Access_Right => read_only`
		RG.7

TABLE 7.3 – Composants et règles issus de la transformation T^{naming}

© 2012 Gilles LASNIER

Règle principale

L'algorithme 4 décrit la règle principale de la transformation. Notons, dans les dépendances, la spécification du paquetage AADL `Base_Types` modélisant les types de données basiques (entier, booléen, etc.) et le paquetage `Deployment` généré à la transformation précédente. Les composants de déploiement produits à la transformation précédente permettent de configurer une partie des composants du service d'adressage.

Algorithme 4: Règle principale `RP` de T^{naming}

```
1 foreach subcomponent process PROC of system SYS do
2 |   Naming ← Create_Package(PROC, "_Naming")
3 |   PreConditions<> ← <SYS, PROC, Naming>
4 |   Dependencies<> ← <User_Model, Data_Model, Base_Types, Deployment>
5 |   Execute(PreConditions<>, Dependencies<>, Naming, rules=[table 7.3])
6 end
```

Règles de transformation et de génération

Le composant `Naming_Table_Type` (voir 7.3, `RG.6`) modélise le type tableau représentant la table de nommage. Les entrées de la table de nommage sont des données structurées de type `Naming_Entry` (généré par `RG.5` tableau 7.2) caractérisé par le nom du nœud, l'adresse et la ressource de communication à utiliser pour atteindre le nœud distant (récupérés par des interrogateurs spécifiques). Sa dimension est déterminée par la taille du composant d'énumeration `Node_Type` généré par la transformation $T^{deployment}$ (voir `RT.1` tableau 7.2).

Exemple 7.1 – Instance de la table de nommage du nœud `Workload_Manager`

```
data Naming_Table
  properties
    Data_Model::Base_Type      => (classifier (PolyORB_HI_Generated_InS_Naming::Naming_Table_Type));
    Data_Model::Element_Names => ("PolyORB_HI_Generated_InS_Deployment.InS_K",
                                  "PolyORB_HI_Generated_InS_Deployment.WoM_K");
    Data_Model::Initial_Value => ("InS_K_Naming_Entry",
                                  "WoM_K_Naming_Entry");
    Access_Right => read_only;
end Naming_Table;
```

L'exemple de code AADL 7.1 décrit le composant produit par la règle de génération `RG.7` (algorithme 5). Celui-ci modélise l'instance de la table de nommage pour un nœud de l'application. L'interrogateur `Get_Connected_Nodes` (l.2) récupère la liste des nœuds connectés au nœud donné. A partir de celle-ci, nous analysons, pour chaque nœud, les propriétés de déploiement relatives au port de communication (`Port_Number`) et à l'adresse (`Location_Value`) du nœud. Ces informations sont récupérées à l'aide des interrogateurs `Get_Deployment_Port_Number_Property` et `Get_Deployment_Location_Property`. Dans le cas où la propriété ciblée n'est pas spécifiée, alors une valeur par défaut est attribuée au nœud analysé.

7.3.4 Transformation Protocole ($T^{messages}$)

Objectifs

La transformation $T^{messages}$ assure l'intégration du service de protocole. Elle génère le tampon de communication global (le type message) encapsulant les données utilisateurs transi-

© 2012 Gilles LASNIER

Algorithme 5: Règle de génération `RG.7 - Naming_Table`

```
1  Naming_Entries<> ← ∅
2  Connected_Nodes<> ← Get_Connected_Nodes(SYS, PROC)
3  Base_Type ← Gen_Base_Type(Naming_Table_Type)
4  Element_Names ← Gen_Element_Names(Connected_Nodes<>)
5  foreach process P of Connected_Nodes<> do
6  |   Port_Number ← Get_Deployment_Port_Number_Property(P)
7  |   Location_Value ← Get_Deployment_Location_Property(P)
8  |   Naming_Entries<>.add(Gen_Naming_Entry(P, Location_Value, Port_Number))
9  end
10 Initial_Value ← Gen_Initial_Value(Naming_Entries<>.name())
11 Access_Right ← Gen_Access_Right("read_only")
12 Property_Association<> ←
13 <Base_Type, Element_Names, Initial_Value, Access_Right>
14 Naming_Table ← Gen_Data_Type("Naming_Table", Property_Association<>)
```

tant entre les nœuds. Celui-ci est alloué statiquement conformément aux restrictions sur les systèmes critiques et dépend de la taille maximale des données échangées.

Préservation des propriétés

Le service de protocole que nous avons modélisé définit un type message utilisé dans plusieurs envois et réceptions sans impliquer une allocation dynamique de mémoire. Le comportement des routines pour la lecture et l'écriture est modélisé. Elles s'exécutent dans un temps proportionnel à la taille de la donnée lue ou écrite dans le pire cas.

Lors de cette transformation, les données de l'utilisateur ne sont en aucun cas modifiées, elles sont encapsulées dans ce message. L'intégrité des données est ainsi préservée. En revanche, nous transformons le type de donnée échangée à travers les canaux de communication. Cette transformation est courante lors de l'implantation d'un tel protocole dans un langage de programmation. Ainsi, en intégrant les composants du service de protocole, nous rendons explicite ce mécanisme et nous pourrons en tenir compte plus tard lors des analyses mémoires (dépassement de la consommation mémoire autorisée), du pire temps d'exécution mais aussi pour la génération de code et la traçabilité des composants. Par conséquent, cette transformation contribut à réduire le niveau d'abstraction du modèle initial.

Composants produits par la transformation

Les composants de données et sous-programmes du service de protocole sont modélisés à l'aide de composants génériques (utilisant les prototypes AADL) dans le paquetage `PolyORB_HI_Messages`. La transformation $T^{messages}$ sélectionne les composants du paquetage et configure les prototypes à l'aide du type message généré à partir des informations extraites du modèle initial.

Les différents composants générés sont présentés dans le tableau 7.4. On y retrouve le tampon global de communication `Message_Type` (`RG.7`) et les opérations élémentaires pour sa manipulation (`RG.8`, écriture, lecture, etc.).

© 2012 Gilles LASNIER

Paquetage de composants AADL du service de protocole T^{messages}		
Rôle	**Modélisation AADL**	**Règles**
Taille maximale du plus grand type de donnée utilisé dans le nœud analysé, utilisé pour l'allocation statique des tampons de communications.	`Max_Payload_Size` : propriété AADL constante `Property_Value` => *<max_communication_data_size>*	RT.5
Taille du tampon global de communication déterminée à partir des constantes `Max_Payload_Size` et `Header_Size`.	`PDU_Size` : propriété AADL constante `Property_Value` => *<PDU_Size = Header_Size + (Max_Payload_Size / 8) + 1>*	RT.6
Tampon de communication global, structure de donnée représentant les messages pour la communication entre les nœuds locaux et distants de l'application.	`Message_Type` : composant `data` 1. Sélection du composant paramétrable `Message_Type` du paquetage `PolyORB_HI_Messages` 2. Configuration du composant à l'aide de propriété constante `_PDU_Size` (`RT.5` ci-dessus)	RG.8
Routines pour la manipulation du tampon de communication global.	Sélection et configuration des sous-programmes du paquetage `PolyORB_HI_Messages` pour la manipulation du composant `Message_Type` : - lecture, écriture, encapsulation, vidage mémoire	RG.9

TABLE 7.4 – Composants et règles issus de la transformation T^{messages}

Règle principale

La règle principale de la transformation T^{messages} est décrite par l'algorithme 6. Parmi les dépendances, nous remarquons l'utilisation des paquetages `PolyORB_HI_Messages` et `PolyORB_HI_Streams`. Ce dernier est une modélisation des constructions du paquetage `Ada.Streams` utilisées pour l'implantation d'une couche basse de transport en Ada et adaptées pour respecter nos restrictions.

Algorithme 6: Règle principale `RP` de T^{messages}

```
1 foreach subcomponent process PROC of system SYS do
2     Messages ← Create_Package(PROC, "Messages")
3     PreConditions<> ← <SYS, PROC, Messages>
4     Dependencies<> ← <User_Model, PolyORB_HI_Messages, PO_HI_Streams>
5     Execute(PreConditions<>, Dependencies<>, Messages, rules=[table 7.4])
6 end
```

Règles de transformation et de génération

L'algorithme 7 de la règle `RG.8` (tableau 7.4) illustre le mécanisme de **sélection** et de **configuration** du composant `Message_Type`. L'exemple de code AADL 7.2 est le résultat produit par cette règle.

La sélection du composant et la configuration du prototype qu'il spécifie s'effectue en trois étapes. Premièrement, nous sélectionnons le composant concret **PDU_Size** destiné à remplacer le **prototype** (l.4), puis nous construisons le mécanisme **prototype_binding** spécifiant la

© 2012 Gilles LASNIER

Algorithme 7: Règle de génération `RG.8` - Instanciation `Message_Type`

```
1  Message_Type_Extension ← Get_Element(PolyORB_HI_Messages, "Message_Type")
2  Message_Type ← Gen_Data_Type("Message_Type", Message_Type_Extension)
     - Build the final data type 'Message_Type'
3
4  PDU_Size ← Get_Element(Messages_Properties, PDU_Size)
5  contentType ← Gen_Data_Type("Content_Type", PDU_Size)
     - Build the final component which replaces the prototype
6
7  contentPrototype ← Get_Element(PolyORB_HI_Messages, "contentP")
8  content ← Gen_Component_Prototype_Binding (contentPrototype, contentType)
     - Build the prototype_binding to attach final component (instanciation)
9
10 Message_Type_Impl_Extension ←
11 Get_Element(PolyORB_HI_Messages, "Message_Type.Impl")
12 Message_Type ←
13 Gen_Data_Implementation("Message_Type.Impl", Message_Type_Extension, content)
     - Build the final data implementation 'Message_Type.Impl'
```

Exemple 7.2 – Sélection et configuration du composant `Message_Type`

```
-- paquetage "PolyORB_HI_Generated_<Process>_Messages"
data Message_Type
  properties
    Data_Model::Data_Representation => Struct;
end Message_Type;

data implementation Message_Type.Impl
  prototypes
    contentP : data;
  subcomponents
    Content : data PolyORB_HI_Streams::Stream_Element_Array[contentP];
    First : data PolyORB_HI_Streams::Stream_Element_Count
                { Data_Model::Initial_Value => ("1"); };
    Last : data PolyORB_HI_Streams::Stream_Element_Count
                { Data_Model::Initial_Value => ("0"); };
end Message_Type.Impl;


-- paquetage "Messages_Properties" produit par la transformation Tmessage
Message_Length_Size : constant aadlinteger => 2;

Header_Size : constant aadlinteger => 4;
-- header_Size = Message_Length_Size + 2;

Max_Payload_Size : constant aadlinteger => 112;
-- value generated from user_model

Pdu_Size : constant aadlinteger => 19;
-- pdu_Size = Header_Size + (Max_Payload_Size / 8) + 1


-- paquetage "Messages" produit par la transformation Tmessage
data Message_Type extends PolyORB_HI_Messages::Message_Type
end Message_Type;

data implementation Message_Type.Impl extends PolyORB_HI_Messages::Message_Type.Impl
(contentP => PolyORB_HI_Generated_WoM_Messages_Properties::PDU_Size)
end Message_Type.Impl;
```

© 2012 Gilles LASNIER

configuration (l.5-8). Ce dernier permet l'identification de l'élément à paramétrer (paramètres de sous-programme, sous-composants...) et son association avec le composant concret. Enfin, nous récupérons le composant générique à étendre et à paramétrer (l.11) à partir duquel sera généré le composant final **Message_Type** (l.13).

Le même mécanisme est utilisé pour les composants de données ou des sous-programmes AADL paramétrables des paquetages `PolyORB_HI_Messages` et `PolyORB_HI_Gqueue`. Une phase de nettoyage du modèle raffiné est effectuée à la fin de la chaîne de transformation. Elle permet l'élimination des prototypes et des `prototype_bindings`, conformément à nos restrictions, par recopie des éléments du composant générique dans le composant final.

7.3.5 Transformation Typage (T^{types})

Objectifs

Nous introduisons brièvement la transformation T^{types} intégrant le service de typage. Cette transformation vise à rassembler dans un seul paquetage les types de données utilisés pour un nœud donné. Ce service permet la simplification de la localisation et de l'utilisation des données dans le code de l'utilisateur et par l'exécutif AADL (service d'interaction, service de représentation...).

Préservation des propriétés

Cette transformation est triviale. Elle effectue une copie des composants de données spécifiés pour un nœud. Les seules modifications apportées à la spécification initiale portent sur la vérification de la cohérence de la taille d'une donnée vis-à-vis de son type et l'ajout d'une valeur initiale par défaut si celle-ci est manquante. L'exemple de code AADL 7.3 présente le service de typage produit pour le nœud `Workload_Manager` de notre cas d'étude.

Exemple 7.3 – Modélisation du service de typage du nœud `Workload_Manager`

```
package PolyORB_HI_Generated_WoM_Types
public

with Data_Model;

  data Interrupt_Counter
    properties
      Data_Model::Data_Representation => Integer;
      Data_Model::Initial_Value => ("0");
  end Interrupt_Counter;

  data Workload
    properties
      Data_Model::Data_Representation => Integer;
      Data_Model::Initial_Value => ("0");
  end Workload;

end PolyORB_HI_Generated_WoM_Types;
```

7.3.6 Transformation Interaction ($T^{activity}$)

Objectifs

La transformation $T^{activity}$ assure l'intégration du service d'interaction. Cette transformation génère, à partir de l'analyse des éléments d'interface d'une tâche d'un nœud donné, les structures de données modélisant l'interface de la tâche avec une sémantique proche d'un langage de programmation.

© 2012 Gilles LASNIER

Préservation des propriétés

Lors de cette transformation, l'interface d'un composant thread modélisée dans la description initiale à l'aide des `features` (ports de données, d'événements...) est remplacée par une interface équivalente basée sur la spécification de composants de données. Cette représentation à l'avantage de définir une sémantique proche des types complexes (`record`...) d'un langage de programmation impératif. Cette transformation contribue à réduire le niveau d'abstraction du modèle initial.

Composants produits par la transformation

Le tableau 7.5 présente les règles de transformation produisant les composants de données modélisant l'interface d'une tâche. La règle RT.10 produit un composant de donnée de type énuméré spécifiant les identifants des ports de la tâche. La règle RT.11 produit un composant de donnée de type union modélisant l'interface du thread. L'énumération produit par la règle RT.10 est réutilisée pour la spécification du type union.

L'interface d'une tâche est ainsi représentée comme une structure de donnée paramétrée par l'identifiant du port et contenant, le cas échéant, un champ représentant la donnée envoyée ou reçue par ce port. L'exemple de code AADL 7.4 illustre les composants produits par les règles `RT.10` et `RT.11` sur le nœud `Workload_Manager` de notre cas d'étude.

Exemple 7.4 – Interface du thread `Activation_Log_Reader` du nœud `Workload_Manager`

```
-- rule RT.10
data Ravenscar_Activation_Log_Reader_Port_Type
  properties
    Data_Model::Data_Representation => Enum;
    Data_Model::Enumerators => ("External_Interrupt_Depository", "Signal");
    Data_Model::Representation => ("1", "2");
end Ravenscar_Activation_Log_Reader_Port_Type;


-- rule RT.11
data Ravenscar_Activation_Log_Reader_Interface
  properties
    Data_Model::Data_Representation => Union;
end Ravenscar_Activation_Log_Reader_Interface;

data implementation Ravenscar_Activation_Log_Reader_Interface.Impl
  subcomponents
    Port_T : data Ravenscar_Activation_Log_Reader_Port_Type
        {Data_Model::Initial_Value => ("Ravenscar_Activation_Log_Reader_Port_Type'First");};
    External_Interrupt_Depository_Data : data PolyORB_HI_Generated_WoM_Types::Interrupt_Counter;
    Signal_Data : data;
end Ravenscar_Activation_Log_Reader_Interface.Impl;
```

Règle principale

L'algorithme 8 décrit la règle principale de la transformation $T^{activity}$. Celle-ci dépend du service de typage généré par la transformation précédente. Cette règle s'applique pour chaque composant thread spécifié comme sous-composant d'un processus (nœud) du système.

7.3.7 Transformation Représentation ($T^{marshallers}$)

Objectifs

La transformation $T^{marshallers}$ assure l'intégration du service de représentation. Elle génère l'ensemble des routines nécessaires pour assurer la cohérence des données lors des envois et des réceptions à travers le réseau. Ainsi, pour chaque nœud de l'application, nous générons

Paquetage de composants AADL du service d'interaction ($T^{activity}$)		
Rôle	**Modélisation AADL**	**Règles**
Enumération représentant les identifiants des ports du thread analysé.	<thread_name>_Port_Type : composant data Propriétés Data_Model associées : Data_Representation => Enum Enumerators => <*list_ports_classifier*> Representation => <*representation_clause*>	RT.10
Interface du thread analysé.	<thread_name>_Interface : composant data union - sous-composant Port_T : typé par le composant généré par la règle RT.10 - sous-composants correspondant à la liste des types de données des ports du thread Propriétés Data_Model associées : Data_Representation => Union	RT.11

TABLE 7.5 – Composants et règles issus de la transformation $T^{activity}$

Algorithme 8: Règle principale RP de $T^{activity}$

```
1 foreach subcomponent thread TH of process PROC do
2     Activity ← Create_Package(TH.name, "_Activity")
3     PreConditions<> ← <TH, PROC, Activity>
4     Dependencies<> ← <User_Model, Data_Model, Types>
5     Execute(PreConditions<>, Dependencies<>, Activity, rules=[table 7.5])
6 end
```

les routines d'emballage (Marshall_XXX) et de déballage (Unmarshall_XXX) des types de données impliquées.

Préservation des propriétés

Le service de représentation que nous avons modélisé assure l'encodage et le décodage des informations pour la cohérence et le routage des données d'un nœud à un autre. Nous avons modélisé le comportement des routines produites pour l'emballage et le déballage d'un type de donnée dans un message. Un emballage est réalisé de manière simple par copie mémoire. Dans le cas d'une application répartie, cela nécessite que les nœuds appartiennent à des plates-formes homogènes. Le temps d'exécution de ces routines dans le pire des cas est en *O(N)* où *N* est la taille de la donnée.

Comme pour la transformation $T^{messages}$ nous n'effectuons aucune modification du contenu des données mais nous changeons la représentation de la donnée. Les routines de ce service peuvent augmenter le pire temps d'exécution d'une tâche spécifiée sur la description initiale. L'intégration de ces composants permet maintenant de prendre en compte cette information.

Cette transformation n'altère donc pas la sémantique et le comportement de l'application mais elle précise les opérations de l'exécutif AADL dans le cas de l'envoi ou de la réception d'une donnée.

© 2012 Gilles LASNIER

Composants produits par la transformation

Le tableau 7.6 récapitule les règles de transformation et les routines générées pour chaque nœud (composant processus) du système. Nous analysons les types de données et générons les routines correspondantes. Des routines d'emballage et de déballage pour les interfaces des tâches générées par la transformation $T^{activity}$ (service d'interaction) sont aussi produites lors de cette transformation. L'exemple de code 7.5 illustre la description architecturale et comportementale d'une routine d'emballage produite lors de cette transformation (règle `RG.10`).

Paquetage de composants AADL du service de représentation ($T^{marshallers}$)		
Rôle	**Modélisation AADL**	**Règles**

Sérialisation et désérialisation des types de données échangées entre les nœuds connectés et le nœud courant.	`Marshaller_XXX_Interface` : composant `subprogram`
	`Unmarshaller_XXX_Interface` : composant `subprogram`
	- génération de sous-programmes (+comportement)
	- spécification du paramètre d'entrée des sous-programmes avec le type de donnée analysé
		RG.10

Sérialisation et désérialisation des interfaces des tâches générées par la transformation de données échangées entre les nœuds connectés et le nœud courant.	`Marshaller_XXX` : composant `subprogram`
	`Unmarshaller_XXX` : composant `subprogram`
	- génération de sous-programmes (+comportement)
	- spécification du paramètre d'entrée des sous-programmes avec le type de donnée analysé
		RG.11

Sérialisation et désérialisation du type de donnée représentant l'énumération des ports des threads du nœud analysé.	`Marshaller_Port_Type` : composant `subprogram`
	`Unmarshaller_Port_Type` : composant `subprogram`
	- génération de sous-programmes (+comportement)
	- spécification du paramètre d'entrée des sous-programmes avec le type de donnée `Port_Type`
	(généré par $T^{deployment}$, `RT.3`)
		RG.12

Sérialisation et désérialisation du type de donnée représentant les messages échangés dans les canaux de communication du nœud analysé.	`Marshaller_Message_Type` : composant `subprogram`
	`Unmarshaller_Message_Type` : composant `subprogram`
	- génération de sous-programmes (+comportement)
	- spécification du paramètre d'entrée des sous-programmes avec le type de donnée `Message_Type`
	(généré par $T^{messages}$, `RG.8`)
		RG.13

TABLE 7.6 – Composants et règles isssus de la transformation $T^{marshallers}$

Règle principale

L'algorithme 9 décrit la règle principale de la transformation. Le paquetage `Ada_Runtime` est utilisé pour résoudre des dépendances vers des types de données et des sous-programmes spécifiques de la runtime Ada. Nous l'utilisons ici pour l'usage de la routine `Unchecked_Conversion` qui effectue une simple copie mémoire pour les types de données complexes.

© 2012 Gilles LASNIER

Exemple 7.5 – Routine d'emballage de l'interface du thread `External_Event_Source`

```
subprogram Marshall_Ravenscar_External_Event_Source_Interface
 features
  A_Data : in parameter
   PolyORB_HI_Generated_InS_External_Event_Source_Activity::
                                    Ravenscar_External_Event_Source_Interface;
  Message : in out parameter PolyORB_HI_Generated_InS_Messages::Message_Type.Impl;
end Marshall_Ravenscar_External_Event_Source_Interface;

subprogram implementation Marshall_Ravenscar_External_Event_Source_Interface.Impl
annex behavior_specification {**
states
 s0 : initial final state;

transitions
 t0_0 : s0 -[ ]-> s0 {
  if (A_Data.Port_T = PolyORB_HI_Generated_InS_External_Event_Source_Activity::External_Interrupt)
        PolyORB_HI_Generated_InS_Marshallers::Marshall_Interrupt_Counter!
             (A_Data.External_Interrupt_Data, Message);
  end if
   };
 **};
end Marshall_Ravenscar_External_Event_Source_Interface.Impl;
```

Algorithme 9: Règle principale `RP` de $T^{marshallers}$

```
1 foreach subcomponent process PROC of system SYS do
2     if Require_Network_Communication(PROC) then
3         Marshallers ← Create_Package(PROC.name, "Marshallers")
4         PreConditions<> ← <SYS, PROC, Marshallers>
5         Dependencies<> ← <User_Model, Types, Activity, Ada_Runtime>
          Execute(PreConditions<>, Dependencies<>, Marshallers, rules=[table 7.6])
6     end
7 end
```

7.3.8 Transformation Transport ($T^{transport}$)

Objectifs

Cette transformation est réalisée dans le cas d'une communication distante entre deux nœuds du système. Elle explicite le service de transport spécifié dans la description initiale. Elle assure la génération des composants modélisant la couche haute du service de transport et l'intégration des composants fournis pas la couche basse de transport sélectionnée.

Remarque. La couche basse de transport est spécifiée dans la description initiale à l'aide d'un composant `device` accédant au composant matériel bus. Les sous-composants du `device` modélisent les composants du protocole de transport. La propriété `Actual_Connection_Binding` spécifie le bus utilisé pour la communication entre les éléments d'interface des nœuds. Enfin, la propriété `Transport_API` associée au bus nous indique la couche basse de transport spécifiée et nous permet de déterminer la transformation de transport correspondante.

Préservation des propriétés

La transformation $T^{transport}$ est un raffinement de la spécification architecturale initiale. Son objectif est d'intégrer complètement les composants modélisant le pilote du protocole de transport associés aux canaux de communication. Ainsi, pour chaque nœud de l'application, nous rendons explicites les ressources enfouies au sein du pilote du protocole (par exemple, la tâche de réception).

© 2012 Gilles LASNIER

Lors de cette transformation, nous n'observons aucun ajout d'information mais un raffinement de la description initiale des nœuds de l'application distribuée. La sémantique et le comportement d'un nœud ne sont pas modifiés. Cependant, le modèle AADL raffiné obtenu reflète un déploiement des ressources de transport plus cohérent. Les propriétés spécifiées au sein du modèle initial sont conservées.

Composants produits par la transformation

Le tableau 7.7 décrit l'ensemble des règles et des composants produits par la transformation. Pour chaque tâche émettrice (resp. réceptrice), nous générons un composant sous-programme assurant l'envoi (resp. la livraison) du message (`RG.14` et `RG.15`). Les routines de haut-niveau `Send` et `Deliver` (`RG.16` et `RG.17`) invoquent ces sous-programmes lors de l'émission ou de la réception d'une donnée par le nœud (traitée par la tâche de réception du nœud dans le second cas).

La règle `RT.7` assure l'intégration de la couche basse de transport spécifiée au sein de la description architecturale pour un nœud donné. L'analyse du composant matériel `bus` et du composant `device` (le pilote associé) nous permet de récupérer les différents composants modélisés de la couche basse de transport. Les routines pour l'initialisation, l'envoi et la réception des données à travers la couche basse sont intégrées au paquetage `Transport`.

Le composant thread modélisant la tâche de réception est généré dans le paquetage `Transport_Receiver_Task`. Sa priorité et sa période sont déterminées à partir de l'analyse des propriétés initiales de l'application. Notamment, cette tâche est arbitrairement plus prioritaire que les tâches applicatives du nœud considéré et sa période dépend de la plus petite période des tâches émettrices connectées au nœud considéré. Ce composant sera intégré par la transformation T^{system_final} (décrite plus bas) comme sous-composant d'un nœud (composant processus) recevant des données distantes.

Règle principale

Algorithme 10: Règle principale `RP` de $T^{transport}$

```
1 foreach subcomponent process PROC of system SYS do
2     Transport ← Create_Package(PROC.name, "Transport")
3     PreConditions<> ← <SYS, PROC, Transport>
4     Dependencies<> ←
5     <User_Model, Base_Types, Data_Model, TCP_Protocol..., Polyorb_HI_Streams,
      Deployment, Naming, Messages, Marshallers>
6     Execute(PreConditions<>, Dependencies<>, Transport, rules=[table 7.7])
7 end
```

L'algorithme 10 décrit la règle principale de la transformation $T^{transport}$. Les couches basses de transport supportées (`TCP_IP`...) sont spécifiées comme dépendances à la transformation. Les composants produits par les transformations déploiement, adressage, message et représentation sont ici réutilisés.

© 2012 Gilles LASNIER

Paquetage de composants AADL du service de Transport ($T^{transport}$)		
Rôle	**Modélisation AADL**	**Règles**
Routine pour la livraison locale du message reçu dans la file d'attente de la tâche considérée.	`XXX_Deliver` : composant `subprogram` - Utilise le composant `Message_Type` généré par $T^{messages}$ (règle `RG.8`) - Utilise les composants `Port_Type` et `Entity_Type` générés par $T^{deployment}$ (règles `RT.2` et `RT.3`) - Utilise le composant `Unmarshall_XXX_Interface` généré par $T^{marshallers}$ (règle `RG.10`) - Génération de la spécification comportementale adéquate	RG.14
Routine pour l'envoi du message aux destinataires de la tâche considérée.	`XXX_Send` : composant `subprogram` - Utilise le composant `Entity_Type` généré par $T^{deployment}$ (règles `RT.2`) - Invoque la routine d'envoi de la couche basse de transport - Génération de la spécification comportementale adéquate	RG.15
Routine (haut-niveau) traitant les émissions de messages aux nœuds connectés au nœud courant.	`Send` : composant `subprogram` - Utilise le composant `Message_Type` généré par $T^{messages}$ (règle `RG.8`) - Utilise le composant `Entity_Type` généré par $T^{deployment}$ (règles `RT.2`) - Invoque la routine `XXX_Send` (ci-dessus `RG.15`) - Génération de la spécification comportementale adéquate	RG.16
Routine (haut-niveau) traitant la livraison locale des messages reçus dans les files d'attente des entités	`Deliver` : composant `subprogram` - Utilise le composant `Message_Type` généré par $T^{messages}$ (règle `RG.8`) - Utilise les composants `Port_Type` et `Entity_Type` générés par $T^{deployment}$ (règles `RT.2` et `RT.3`) - Invoque la routine `XXX_Deliver` (`RG.14`) correspondante à l'entité destinataire - Génération de la spécification comportementale adéquate	RG.17
Intégration des composants de la couche basse de transport sélectionnée (`TCP_IP`...).	- `Initialize` : composant `subprogram` pour l'initialisation de la couche basse, utilise la table de nommage générée par T^{naming} (règle `RG.7`) - `Initialize_Receiver` : composant `subprogram` pour l'initialisation de la tâche de réception - `Receive` : composant `subprogram` pour la réception d'un message (bas-niveau) - `Send` : composant `subprogram` pour l'envoi d'un message (bas-niveau) - `Receiver_Thread` : sous-composant `thread` pour l'intégration de la tâche de réception au sein du nœud analysé	RT.7

TABLE 7.7 – Composants et règles issus de la transformation $T^{transport}$

© 2012 Gilles LASNIER

7.3.9 Transformation Interrogation (T^{global_queue})

Objectifs

La tranformation T^{global_queue} assure l'intégration du service d'interrogation de notre intergiciel. Ce service raffine les mécanismes d'envoi et de réception des données entre les tâches. Lors de cette transformation, nous remplaçons les composants ports dans les interfaces des tâches par une structure modélisant une file d'attente globale de messages. La configuration de cette file est assurée par l'analyse des interfaces des tâches applicatives (des ports) de la description initiale.

Nous avons modélisé la file d'attente globale de messages et le comportement des routines pour sa manipulation à l'aide de composants AADL paramétrables. Ces composants sont spécifiés dans le paquetage `PolyORB_HI_Gqueue` présenté dans l'annexe A.2. Cette transformation s'applique pour chaque tâche (appelée aussi *entité* pour respecter les conventions de nommage) appartenant au nœud analysé - *i.e.* pour tout sous-composant `thread` d'un composant `processus`.

Préservation des propriétés

La modélisation des composants et du comportement de la file d'attente globale de messages est basée sur son implantation dans l'intergiciel POLYORB-HI-Ada (objet protégé Ada). Notre file d'attente est modélisée à l'aide d'un composant `data` partagé. Les informations déduites des élements d'interface de la tâche permettent la génération des données et des tableaux circulaires spécifiant la structure interne de la file. Grâce à notre profil AADL-HI Ravenscar, nous avons réussi à modéliser le même comportement décrit par son implantation. De plus, les composants modélisés respectent les recommandations du profil Ravenscar.

Les routines modélisées qui permettent d'effectuer des actions à partir de la file s'exécutent de manière déterministe (constante pour la majorité et, au pire cas, en *O(N)* où *N* est la taille de la donnée traitée [Zalila *et al.*, 2008][section 6.4]). Enfin, de nombreux cas d'utilisation et l'implication de POLYORB-HI-Ada dans de nombreux projets ont permis de tester et d'éprouver cette implantation. La reproduction d'un code identique nous assure par conséquent la fiabilité de cette structure.

T^{global_queue} transforme ainsi le mécanisme d'envoi et de réception entre tâches basé sur la sémantique des `ports` par un mécanisme assurant le même comportement et conservant les mêmes propriétés architecturales (configurées à partir de l'analyse du modèle initial). De plus, les composants produits par cette transformation autorisent une traduction vers des constructions du langage de programmation Ada ayant une sémantique équivalente (données, sous-programmes et objet protégé). Cette transformation précise à la fois les ressources de l'exécutif AADL pour la communication des tâches et introduit des constructions préservant le déterminisme et l'analyse statique. Elle contribue à réduire le niveau d'abstraction du modèle initial et la différence sémantique entre le modèle et le code généré.

Composants produits par la transformation

Le tableau 7.8 présente l'ensemble des règles de tranformation et des composants générés par la transformation. Les composants spécifiés dans le paquetage générique `PolyORB_HI_Gqueue` sont sélectionnés et configurés de la même manière que ceux du paquetage `PolyORB_HI_Messages` (voir $T^{messages}$ plus haut).

© 2012 Gilles LASNIER

Paquetage de composants AADL du service d'Interrogation (T^{global_queue})		
Rôle	**Modélisation AADL**	
Types de données pour la génération et la configuration des tableaux circulaires constituant la file d'attente de messages globale (`Global_Queue`) pour chaque composant `thread`.	Types de données : composants type `data` – `Port_Type` : énumération représentant les `ports` du `thread` analysé – `Integer_Array` : type tableau utilisé pour la spécification des tailles des files de messages et des priorités des `ports` du `thread` – `Port_Kind_Array` : type tableau pour la spécification de la catégorie du `port` (`event`, `event data`, `data`) – `Overflow_Protocol_Array` : type tableau pour la spécification du protocole de dépassement de capacité d'une file d'un `port` – `Thread_Interface_Type` : type *record* pour la spécification de l'interface du `thread` représentant les types de données des `ports` du `thread`	RG.18
Instances de données constituant la file d'attente de messages globale pour chaque composant `thread`.	Données : composants type `data` – `Current_Entity` : identifiant représentant le `thread` analysé – `Thread_Port_Kinds` : tableau représentant la catégorie et l'orientation des `ports` du `thread` – `Thread_Fifo_Sizes` : tableau représentant la taille des files de messages des `ports` du `thread` – `Thread_Fifo_Offsets` : tableau représentant la position des files d'attente – `Thread_Overflow_Protocols` : tableau représentant le protocole de dépassement de capacité spécifié pour un `port` donné – `Urgencies` : tableau représentant la priorité associé à un `port` – `Global_Data_Queue_Size` : donnée représentant la somme de toutes les files d'attente de messages des `ports` du `thread` analysé – `N_Destinations` : tableau représentant le nombre de destinataires pour chaque `port` – `Destinations` : tableau représentant les destinataires de chaque `port` du `thread`	RG.19

Instanciation de la file d'attente globale (`Global_Queue`) de messages pour chaque composant `thread`.	`<thread_name>_Global_Queue` : composant `data`
	- Utilise `Global_Queue` du paquetage `PolyORB_HI_Gqueue` (patron d'objet partagé)
	Propriétés AADL associées :
	`Concurrency_Control_Protocol => Priority_Ceiling`
	`Priority =>` <*max_system_priority*>
		RT.8

Instanciation des routines pour la manipulation de la file d'attente globale	Instanciation des composants `subprograms` du paquetage `PolyORB_HI_Gqueue` : (`Send_Output`, `Receive_Input`...)	RT.9

TABLE 7.8 – Composants et règles de transformation T^{global_queue}

© 2012 Gilles LASNIER

Règle principale

L'algorithme 11 décrit la règle principale (RP) de la transformation T^{global_queue}. Celle-ci nécessite l'utilisation d'un certain nombre d'ensembles de propriétés standards AADL. Elle s'applique pour tout composant thread spécifié au sein d'un composant processus PROC du système.

Algorithme 11: Règle principale de la transformation T^{global_queue}

```
1 foreach subcomponent thread TH of process PROC do
2     Gqueue ← Create_Package(PROC, TH, "Gqueue")
3     PreConditions<> ← <PROC, TH, Gqueue>
4     Dependencies<> ←
5     <User_Model, Data_Model, AADL_Project, Memory_Properties, Thread_Properties,
      Deployment_Properties, PolyORB_HI_Port_Kinds,
6     PolyORB_HI_Gqueue, Deployment>
7     Execute(PreConditions<>, Dependencies<>, Gqueue, T^global_queue[table 7.8])
8 end
```

Règles de transformation et de génération

L'algorithme 12 illustre un fragment de la règle RG.19 pour la génération des instances de données utilisées par la structure de la file d'attente. L'interrogateur spécifique Get_Port_Information (l.1, détaillé dans l'annexe B) construit une liste structurée des spécifications des différents ports d'une tâche. Cette liste structurée récupère l'ensemble des informations d'un port à savoir : son nom, son parent, sa direction, sa catégorie, son type, son sous-programme de calcul associé et différentes propriétés (taille de la file d'attente, politique de gestion des messages de la file, etc.). Il suffit d'interroger cette liste structurée par la suite pour obtenir les informations pertinentes pour la génération des différentes instances de données et leur configuration (par exemple l.4 et l.5).

Algorithme 12: Fragment de règle RG.19 - Génération des données de la file

```
1  Port_Information<> ← Get_Port_Information(TH)
2  ...
   - Generate Thread_Fifo_Sizes
3  Base_Type ← Gen_Base_Type(Integer_Array)
4  Element_Names ← Gen_Element_Names(Get_Port_Name(Port_Information<>))
5  Representation ← Gen_Representation(Get_Port_Queue_Size(Port_Information<>))
6  Access_Right ← Gen_Access_Right("read_only")
7  Property_Association<> ←
8  <Base_Type, Element_Names, Representation, Access_Right>
9  Thread_Fifo_Sizes ← Gen_Data_Type("Thread_Fifo_Sizes", Property_Association<>)
10 ...
```

© 2012 Gilles LASNIER

7.3.10 Transformation AADL_Runtime ($T^{aadl\_runtime}$)

Objectifs

L'objectif de la transformation $T^{aadl\_runtime}$ est l'intégration des sous-programmes définissant les interactions des tâches, pour la communication, avec un exécutif AADL (un intergiciel), conformément aux spécifications du standard AADLv2. Les sous-programmes modélisés assurent les opérations d'envoi et de réception des données d'une tâche à une autre de manière transparente à l'utilisateur.

Préservation des propriétés

La transformation $T^{aadl\_runtime}$ explicite les interactions entre les tâches applicatives et l'exécutif AADL conformément aux recommandations du standard AADL. Ces interactions s'expriment sous la forme de composants sous-programmes manipulant un paramètre unique modélisant un type de donnée paramétrable (`AADL_Context`). Ce composant facilite l'invocation des mécanismes d'interaction dans le code utilisateur.

Ainsi, le code utilisateur n'invoque pas les routines de chaque file d'attente globale de messages générées pour les tâches du nœud. Le type de donnée `AADL_Context` permet d'utiliser une seule structure de donnée pour l'ensemble des tâches applicatives du nœud. Le comportement des sous-programmes modélisés par cette transformation se charge d'invoquer les files d'attente et les types de données correspondants.

Cette transformation est conforme aux recommandations du standard AADL. Elle raffine la description initiale mais ne modifie pas la sémantique du modèle.

Composants produits par la transformation

Le tableau 7.9 récapitule l'ensemble des composants produits par la transformation $T^{aadl\_runtime}$.

Comme pour la transformation $T^{activity}$ et le composant modélisant l'interface d'une tâche, nous modélisons ici le type de donnée paramétrable `AADL_Context` à l'aide d'un composant AADL de donnée de type union. L'énumération `Entity_Type` (générée par la transformation $T^{deployment}$) identifiant les tâches d'un nœud sert à paramétrer le composant. Les interfaces des tâches (composants générés par $T^{activity}$, voir plus haut) servent à paramétrer les champs représentant la donnée envoyée ou reçue. Dans notre cas, la donnée est une interface d'une tâche du nœud transportant elle-même la donnée envoyée ou reçue sur un port de la tâche.

L'exemple de code AADL 7.6 décrit le composant `AADL_Context` produit par la règle `RG.20` pour le nœud `Workload_Manager` de notre cas d'étude.

Enfin, nous générons uniquement les sous-programmes requis par la tâche. Pour ce faire, nous analysons si la tâche est émettrice, réceptrice ou émettrice/réceptrice. Dans le cas où la tâche ne fait qu'émettre des données, seule la règle `RG.21` (tableau 7.9) sera exécutée. Elle génère le sous-programme `Put_Value` assurant le placement de la donnée dans la file d'attente globale de messages et le sous-programme `Send_Output` modélisant l'opération d'envoi de cette donnée depuis la file d'attente.

© 2012 Gilles LASNIER

Exemple 7.6 – Composant paramétrable `AADL_Context` du nœud `Workload_Manager`

```
data AADL_Context
  properties
    Data_Model::Data_Representation => Union;
end AADL_Context;

data implementation AADL_Context.Impl
  subcomponents
    Discriminant               : data PolyORB_HI_Generated_WoM_Deployment::Entity_Type;
    Regular_Producer_Itf       : data PolyORB_HI_Generated_WoM_Regular_Producer_Activity::
                                                          Ravenscar_Regular_Producer_Interface;
    On_Call_Producer_Itf       : data PolyORB_HI_Generated_WoM_On_Call_Producer_Activity::
                                                          Ravenscar_On_Call_Producer_Interface;
    External_Event_Server_Itf  : data PolyORB_HI_Generated_WoM_External_Event_Server_Activity::
                                                          Ravenscar_External_Event_Server_Interface;
    Activation_Log_Reader_Itf  : data PolyORB_HI_Generated_WoM_Activation_Log_Reader_Activity::
                                                          Ravenscar_Activation_Log_Reader_Interface;
end AADL_Context.Impl;
```

Paquetage de composants AADL du service AADL_Runtime ($T^{aadl\_runtime}$)		
Rôle	**Modélisation AADL**	**Règles**
Structure générique pour l'envoi ou la réception d'une donnée sur un nœud.	`AADL_Context` : composant `data` union Sous-composants : composant `data` .nom => énumération des entités du nœud . type => interfaces générées par $T^{activity}$ (règle `RT.11`) Propriétés `Data_Model` associées : `Data_Representation` => `Union`	RG.20
Génération des routines de la runtime AADL pour l'émission de données. `Put_Value` place une donnée dans une file d'attente globale de messages. `Send_Output` envoie une donnée à partir de la file.	`Put_Value` : composant `subprogram` - plus spécification comportementale `Send_Output` : composant `subprogram` - plus spécification comportementale	RG.21

Génération des routines de la runtime AADL pour la réception de données. `Get_Count` renvoie le nombre de messages reçus et placés dans une file d'attente global de messages. `Get_Value` recupère une donnée placée dans la file d'attente. `Next_Value` enlève la donnée de la file d'attente.	`Get_Count` : composant `subprogram`
	- plus spécification comportementale
	`Get_Value` : composant `subprogram`
	- plus spécification comportementale
	`Next_Value` : composant `subprogram`
	- plus spécification comportementale
		RG.22

TABLE 7.9 – Composants et règles issus de la transformation $T^{aadl\_runtime}$

© 2012 Gilles LASNIER

Règle principale

L'algorithme 13 décrit la règle principale de la transformation $T^{aadl\_runtime}$. Celle-ci spécifie dans ses dépendances l'ensemble des paquetages `Deployment`, `Activity` et `Gqueue` produits pour chaque tâche du nœud.

Algorithme 13: Règles principale `RP` de $T^{aadl\_runtime}$

```
1 foreach subcomponent process PROC of system SYS do
2     if IS_SENDING(PROC) or IS_RECEIVING(PROC) then
3         AADLRuntime ← Create_Package(PROC.name, "_AADL_Runtime")
4         PreConditions<> ← <SYS, PROC, AADL_Runtime>
5         Dependencies<> ← <User_Model, Base_Types, Data_Model, PolyORB_HI_Errors,
6         Deployment, Activity, Gqueue>
7         Execute(PreConditions<>, Dependencies<>, AADL_Runtime, rules=[table 7.9])
8     end
9 end
```

7.3.11 Transformation Exécution (T^{tasks})

Objectifs

La transformation T^{tasks} raffine les composants `threads` du modèle initial, représentant les tâches des nœuds de l'application distribuée. Ce raffinement s'effectue conformément aux patrons de modélisation des tâches périodique et sporadique que nous avons décrits dans la section 5.1 lors de notre discussion sur la réduction du niveau d'abstraction du modèle AADL.

Pour rappel, ces patrons contribuent à améliorer la description et l'analysabilité d'une tâche en rendant explicite, à travers la spécification d'un automate comportemental, son cycle de vie, les différents appels aux sous-programmes (initialisation, charge de travail, reprise) décrivant son comportement fonctionnel, les différents appels aux services de l'intergiciel PolyORB-HI-AADL et ses interactions avec sa file d'attente globale de messages (produite par T^{global_queue}) pour l'envoi et la réception de données.

La structure de cet automate a été définie en respectant les recommandations du profil Ravenscar (exécution infinie...) nous permettant ainsi de limiter les transformations à effectuer lors de la phase de génération de code.

Préservation des propriétés

Nous avons expliqué dans les chapitres 3 et 4 que de nombreuses informations comportementales étaient attachées à la description architecturale d'un composant `thread` sous la forme d'une propriété AADL (`Compute_Entrypoint, Initialize_Entrypoint`, protocole de déclenchement, etc.). La génération de l'automate comportemental conformément aux recommandations de notre profil AADL-HI Ravenscar nous permet d'expliciter ces informations sous la forme de composants concrets (`data`, `subprogram`) mais aussi les interactions du composant avec les différentes ressources intergicielles.

Nous conservons ainsi l'ensemble des propriétés spécifiées dans la description architecturale sous une autre forme et nous renforçons celle-ci à l'aide d'une description comportementale précisant la sémantique des opérations de la tâche. Cette spécification est cohérente avec l'implantation physique de la tâche. Les seules informations ajoutées au comportement sont

© 2012 Gilles Lasnier

les interactions avec l'exécutif AADL telles que le standard AADL les spécifie. Par conséquent, cette transformation contribue à réduire le niveau d'abstraction de la description architecturale initiale des tâches applicatives du système. Elle permettra à la fois une analyse plus fine (séquence et ordre d'appels des sous-programmes complets, intégration et utilisation précise des ressources) et facilitera la phase de génération de code.

Composants produits par la transformation

Le tableau 7.10 récapitule les différentes règles et les composants générés par T^{tasks} à partir de l'analyse de la description architecturale et des propriétés attachées aux composants `threads` d'un nœud donné.

Règle principale

L'algorithme 14 présente la règle principale de la transformation T^{tasks}. Nous générons un paquetage pour chaque composant `thread` du nœud analysé. Il n'est pas utile de spécifier les paquetages modélisant les interfaces des tâches et les paquetages `Gqueue` générés pour chaque tâche, l'utilisation des composants du paquetage `AADL_Runtime` offre les mécanismes et les types de données nécessaires pour l'envoi et la réception de données.

Algorithme 14: Règle principale `RP` de T^{tasks}

```
1 foreach subcomponent thread TH of process PROC do
2     Task ← Create_Package(TH.name, "_Task")
3     PreConditions<> ← <TH, PROC, Task>
4     Dependencies<> ← <User_Model, Base_Types, Data_Model, Programming_Properties,
5     PolyORB_HI_Errors, AADL_Runtime, Deployment>
6     ForEach(PreConditions<>, Dependencies<>, Task, rules=[tableau 7.10])
7 end
```

Règles de transformation et de génération

La règle `RT.15` décrit la génération d'une spécification comportementale pour un sous-composant thread du nœud conformément au profil AADL-HI Ravenscar. Elle est détaillée dans l'algorithme 15.

La règle de génération `Gen_Communication_Action` (l.5) produit l'appel au sous-programme d'initialisation généré par la règle `RT.13` (issue de la propriété initiale `Initialize_Entrypoint`). Cette règle est aussi appelée par la règle `Gen_Behavior_Actions_From_Ports` (l.9) pour la génération des appels aux différents sous-programmes relatifs à la fonction de calcul et de reprise de la tâche. Cette dernière génère aussi les appels aux routines de la runtime AADL en fonction des caractéristiques de l'interface du `thread` (`ports` en émission et/ou réception).

La condition du déclenchement d'activité est générée à l'aide de la règle `Gen_Dispatch_Condition` (l.10). Celle-ci produit l'unique port (`sysDispatch`, l.1) autorisé par notre processus de raffinement pour rester conforme au standard AADL. Ce port modélise l'événement système déclencheur du réveil d'une tâche périodique, ou l'événement reçue par une tâche sporadique pour son activation.

© 2012 Gilles LASNIER

Paquetage de composants AADL pour le service d'exécution (T^{tasks})		
Rôle	**Modélisation AADL**	**Règles**

Expansion des propriétés temporelles et structurelles sous la forme de données.	`Priority_<thread_name>` : sous-composant `data` spécifié par la propriété `Priority`
	`Period_<thread_name>` : sous-composant `data` spécifié par la propriété `Period`
	`Deadline_<thread_name>` : sous-composant `data` spécifié par la propriété `Deadline`
	`Source_Stack_Size_<thread_name>` : sous-composant `data` spécifié par la propriété `Source_Stack_Size`
		RT.12

Expansion des propriétés comportementales sous la forme de sous-programmes.	`Initialize_<thread_name>` : sous-composant `subprogram` spécifié par la propriété `Initialize_Entrypoint`
	`Activate_<thread_name>` : sous-composant `subprogram` spécifié par la propriété `Activate_Entrypoint`
	`Compute_<thread_name>` : sous-composant `subprogram` spécifié par la propriété `Compute_Entrypoint`
		RT.13

Génération du composant `thread` raffiné.	`<thread_name>_Task` : composant `thread` - recopie des propriétés initiales - intégration de la file d'attente globale de messages du thread analysé - génération du `port sysDispatch` pour le déclenchement d'activité - intégration des sous-programmes générés par la règle `RT.12` comme sous-composants - intégration du sous-composant `AADL_Context` généré par la $T^{aadl\_runtime}$ (règle `RG.20`)	RT.13
Génération de la séquence d'appels de sous-programme globale.	`call_seq` : composant `subprogram_call_sequence` - liste d'appel des sous-programmes dans l'ordre - intégration au composant généré par `RT.13`	RT.14
Génération de la spécification comportementale à partir du thread analysé.	`Behavior_Specification` : composant `annex_subclause` - utilisation du patron comportemental correspondant - intégration des appels aux routines d'activation et de recouvrement - intégration des appels aux routines du service AADL_Runtime - intégration de l'appel à la fonction de calcul	RT.15

TABLE 7.10 – Composants et règles issus de la transformation T^{tasks}

© 2012 Gilles LASNIER

Algorithme 15: Règle RT.15 - Spécification comportementale du thread

input : thread subcomponent $TH \in$ process subcomponent $PROC$
output : thread annex subclause $Behavior\_Specification$ of TH

```
1  sysDispatch ← Gen_Behavior_Identifier("sysDispatch")
2  stInit ← Gen_Behavior_State("stInit", Initial=true, Complete=false, Final=false)
3  stDispatch ← Gen_Behavior_State("stDispatch", Initial=false, Complete=true, Final=false)
4  actions<> ← ∅
5  Call_Init ← Gen_Communication_Action(Get_Initialize_Entrypoint(TH))
6  actions<>.add(Call_Init)
7  tInit ← Gen_Behavior_Transition(Src=stInit, Dst=stDispatch, actions<>)
8  actions<> ← ∅
9  actions<>.add(Gen_Behavior_Actions_From_Ports(TH))
10 Dispatch_Condition ← Gen_Dispatch_Condition(sysDispatch)
11 tDispatch ← Gen_Behavior_Transition(Src=stDispatch, Dst=stDispatch, actions<>,
   Dispatch_Condition)
12 States<> ← <stInit, stDispatch>
13 Transitions<> ← <tInit, tDispatch>
14 Behavior_Specification ← Gen_Behavior_State(TH, States<>, Transitions<>)
```

7.3.12 Transformation Système (T^{system_final})

Objectifs

La transformation T^{system_final} assure la construction du composant `system` final qui intègre et connecte les nœuds (processus) raffinés, les différentes tâches applicatives raffinées, les files d'attente globale de messages et la tâche de réception de la couche basse de transport (si nécessaire). Le résultat de cette transformation est notre modèle complet.

Préservation des propriétés

Pour chaque nœud (composant `process`) du système, nous intégrons ses tâches raffinées, produites par la transformation T^{tasks}, et leur file d'attente globale de messages (composants `data` partagés), produite par la transformation T^{global_queue}. Si une tâche de réception a été produite (transformation $T^{transport}$) pour ce nœud, elle est aussi intégrée. Ces composants processus raffinés sont ensuite intégrés au système final. Cette dernière transformation produit le modèle AADL complet de l'application contenant les composants applicatifs raffinés, les composants intergiciels et les différentes connexions.

C'est à partir de ce modèle final et de ce composant système que seront à présent effectuées les analyses Ravenscar et d'ordonnancement.

Par construction et grâce à notre processus de raffinement, nous sommes sûrs que l'intégration de ces composants pilotée par l'analyse du composant `system` de la description initiale n'engendre pas de modification sémantique et comportementale et produit un modèle légal au sens AADL et cohérent. Sur ce dernier point, nous effectuons une nouvelle étape d'analyse pour la validation de notre modèle de code complet. Cette étape fait l'objet de la section suivante.

© 2012 Gilles LASNIER

Composants produits par la tranformation

Le tableau 7.11 décrit les règles de transformation assurant le raffinement des composants processus et le raffinement du composant système, à partir de l'analyse de la description architecturale initiale.

Paquetage de composants AADL - composant `system` (T^{system_final})		
Rôle	**Modélisation AADL**	**Règles**
Ré-écriture des composants `process` (nœuds), intégration complète des tâches raffinées, des files d'attente globale de messages respectives et de la tâche de réception (le cas échéant).	ahr_<node_name> : composant `process` - intégration des composants `thread` générés par T^{tasks} (règle `RT.13`) - intégration des composants `data Global_Queue` générés par T^{global_queue} (règle `RT.8`) - intégration du composant `thread`, modélisant la tâche de réception, généré par $T^{transport}$ (règle `RT.7`)	RT.16
Ré-écriture du composant `system` final intégrant les nœuds raffinés (produits ci-dessus).	ahr_<system_name> : composant `system` (nœuds) - intégration des composants `process` produits par la règle `RT.16`	RT.17

TABLE 7.11 – Composants et règles issus de la transformation T^{system_final}

Règles de tranformation

L'algorithme 16 décrit la règle `RP` de la transformation. Celle-ci s'applique sur le composant système du modèle initial. L'analyse de ce composant et de ses sous-composants processus nous permet de diriger cette transformation et de sélectionner les composants produits par les transformations T^{tasks}, T^{global_queue} et $T^{transport}$. Les paquetages de composants AADL produits par ces transformations (`Tasks`, `Gqueue` et `Transport`) sont spécifiés dans les dépendances.

Algorithme 16: Règle principale `RP` de T^{system_final}

```
1 foreach system implementation SYS do
2     AHR_System ← Create_Package(SYS.name, "AHR_System")
3     PreConditions<> ← <SYS, AHR_System>
4     Dependencies<> ← <User_Model, Deployment, Tasks, Gqueue, Transport>
5     Execute(PreConditions<>, Dependencies<>, AHR_System, rules=[table 7.11])
6 end
```

7.4 Validation du modèle AADL complet

La troisième étape de notre processus est la validation du modèle complet de l'application produit par les transformations de notre processus de raffinement. Comme lors de la première validation, nous effectuons une analyse de la syntaxe et de la sémantique de la description pour valider sa conformité au standard AADLv2 (sémantique, cohérence, légalité...). Puis, nous reconduisons les analyses supplémentaires.

© 2012 Gilles LASNIER

7.4.1 Analyse des restrictions Ravenscar

L'analyse des restrictions Ravenscar définies par notre profil AADL-HI Ravenscar (chapitre 3, section 3.4.2) s'effectue cette fois-ci sur les composants applicatifs et les composants intergiciels constituant l'application complète.

Les spécifications comportementales attachées à l'ensemble des composants architecturaux peuvent être analysées. Ainsi, nous pouvons désormais vérifier la restriction RC1 (un unique point de suspension par tâche) et RC2 (pas d'état final spécifié dans l'automate d'un thread) sur les automates des tâches applicatives du système.

7.4.2 Analyse d'ordonnancement

L'analyse d'ordonnancement prend aussi en compte les ressources intergicielles. Nous pouvons vérifier si le système est toujours ordonnançable. Dans le cas contraire, la comparaison des résultats obtenus avec ceux de la première analyse peut nous permettre de déduire l'origine du problème et les composants fautifs (hypothèses d'ordonnancement, gestion des sections critiques sur les ressources partagées...).

Notamment, notre processus de raffinement préserve les hypothèses pour l'analyse d'ordonnançabilité de l'ensemble des tâches applicatives du système. Cependant, l'intégration des composants intergiciels peut influer sur l'ordonnancement du système en augmentant le pire temps d'exécution de ces tâches (interactions avec l'intergiciel), ou encore, la tâche ajoutée pour la réception des messages sur un nœud peut compromettre l'ordonnancement. L'utilisateur peut alors vérifier ses spécifications initiales et intervenir.

A la fin de cette étape de notre processus, nous sommes assurés d'avoir un modèle de code complet spécifiant l'ensemble des informations architecturales et comportementales requises pour la génération du code source de l'application.

7.5 Synthèse

Ce chapitre a présenté notre processus de raffinement et d'analyse d'applications TR^{2}E critiques. Celui-ci repose sur une modélisation contrainte des composants applicatifs et intergiciels s'appuyant sur notre profil AADL-HI Ravenscar. Les étapes du processus présentées ont été élaborées afin de garantir les objectifs suivants :

1. Réduire le niveau d'abstraction du modèle initial pour la prise en compte des ressources d'un exécutif supportant les composants applicatifs (logiciels) AADL, lors des analyses réalisées sur le modèle. Ces ressources intergicielles sont modélisées par un assemblage de composants architecturaux et comportementaux définissant des constructions déterministes et analysables statiquement.
2. Réduire les différences sémantiques entre modèle analysé et code généré par raffinement de la description des composants et par l'intégration des composants intergiciels. La sémantique exprimée par les composants du modèle raffiné est très proche des constructions d'un langage de programmation impératif comme Ada ou C.

Nous avons explicité l'ensemble des transformations assurant l'intégration des composants intergiciels et le raffinement des composants architecturaux existants. Celles-ci impliquent la modification ou la génération de composants AADL à partir d'un modèle AADL (transformation

© 2012 Gilles LASNIER

endogène). L'expression de ces transformations en terme de composants AADL est peut -être une tâche fastidieuse, mais elle permet de faciliter la maintenance et l'adaptation de ces règles. Notamment, l'utilisateur peut intervenir, par la suite, sur les composants du modèle raffiné.

Le résultat de ces transformations est un modèle AADL légal de l'application fidèle à une implantation du système critique. Dans le chapitre suivant, nous présentons les règles de transformation pour la génération du code source de ces composants vers les constructions d'un langage de programmation impératif : le langage Ada.

© 2012 Gilles LASNIER

Chapitre 8

Règles de Transformation pour la Production Automatique de Systèmes TR^{2}E Critiques

SOMMAIRE

8.1 Introduction

Dans le chapitre précédent, nous avons présenté la première phase de notre processus automatique pour l'analyse et la production des applications TR^{2}E critiques. Cette phase mène, par transformation de modèles verticale, le raffinement de la description architecturale AADL de l'application vers un modèle complet de l'application (applicatif et exécutif) analysable et sémantiquement proche de son implantation.

Dans ce chapitre, nous détaillons les transformations des composants de ce modèle vers les constructions du langage de programmation Ada respectant les recommandations de son profil Ravenscar. Les transformations simples que nous présentons montrent le bien-fondé de notre processus de raffinement. Le *mapping* AADL vers Ada que nous exprimons est structuré

de manière à permettre sa réutilisation dans un autre contexte que notre approche, à faciliter son évolution (modifications introduites dans les langages AADL ou Ada) et sa maintenance (expressivité des règles de transformation). Enfin, notre *mapping* autorise aussi la production de composants AADL à partir du sous-ensemble des constructions Ada que nous supportons.

Ce chapitre est organisé de la manière suivante. La section 8.2 présente les règles de transformation des composants de données AADL. La section 8.3 détaille les transformations des composants sous-programmes vers les procédures et fonctions Ada. La section 8.4 décrit le *mapping* et les règles pour la génération des tâches Ada à partir de la description architecturale et comportementale des composants threads. La section 8.5 détaille la génération de code à partir des éléments de langage spécifiés au sein des descriptions comportementales. La section 8.6 présente la génération des fichiers supports pour l'intégration des composants utilisateurs et la compilation du code généré. La section 8.7 conclut ce chapitre.

Simplification et structure des transformations

Pour faciliter la compréhension et l'implantation de nos règles de transformation, nous nous appuyons sur les éléments suivants :

1. Nous avons défini une bibliothèque d'interrogateurs des éléments du modèle AADL qui permet l'extraction des informations (structurelles, comportementales, etc.) à partir des composants, des propriétés ou des descriptions comportementales. Ils sont à la fois utilisés dans les préconditions et dans le corps de nos règles de transformation.
2. Pour des raisons de maintenance et d'évolution, nous avons choisi de respecter la syntaxe abstraite du langage Ada définissant les nœuds d'un arbre syntaxique abstrait (AST). Les règles de transformation et les constructions Ada (respectant le profil Ravenscar) produites sont ainsi nommées conformément à la dénomination de la BNF du langage présentée dans l'annexe C de ce manuscrit.
3. Notre processus de raffinement a pris en charge différentes vérifications sur la spécification des composants, notamment la présence des propriétés essentielles pour la définition d'un type ou d'une instance de données, la vérification de la cohérence de la taille mémoire spécifiée vis-à-vis du type, etc.
4. La dernière phase de notre processus de raffinement a éliminé les éléments de modélisation non concrets (extension, prototypes...). Ainsi, la spécification d'un composant contient toutes les propriétés qui lui sont associées (par exemple, il n'y a plus d'héritage).
5. Une routine spécifique (`Map_Ada_Name`) permet de résoudre les problèmes issus des différentes conventions de nommage entre le langage AADL et le langage Ada (identifiant, nommage des paquetages, etc.). Cette routine est appelée pour l'analyse et la modification (le cas échéant) de tout identifiant AADL rencontré.

8.2 Transformation des composants de données

Cette section présente les règles de transformation assurant la traduction des composants de données AADL vers les types de données, les constantes et les variables globales du langage de programmation Ada. Pour illustrer concrètement la simplicité de nos transformations, nous avons choisi de présenter les règles de transformation vers le langage Ada. Cependant, les patrons de modélisation pour la génération que nous présentons sont généralistes.

© 2012 Gilles LASNIER

Ils peuvent ainsi être réutilisés pour l'expression de règles de transformation vers un autre langage de programmation impératif comme le langage C.

L'annexe de modélisation des données du langage AADL (présentée 4) définit les patrons de modélisation des types de données classiques. Ceux-ci reposent sur un ensemble de propriétés AADL spécifiques (l'ensemble de propriétés `Data_Model`). Nous avons complété ces patrons lors de notre processus de raffinement par les propriétés : `Source_Data_Size` (propriété issue du standard) exprimant la taille mémoire maximale de la donnée ; `Access_Right` (propriété issue du standard) exprimant si la donnée est accessible en lecture, en écriture ou en lecture/écriture ; `Initial_Value` (propriété issue de l'annexe) spécifiant une valeur par défaut pour la donnée modélisée.

Ces informations sont utilisées respectivement pour allouer la quantité de mémoire requise (allocation statique...), pour déterminer les types de données, les constantes et les variables, pour l'initialisation du système ou en cas de pannes (liées au caractère asynchrone des communications utilisées par l'intergiciel).

Logiquement, ces propriétés permettent de différencier les composants de données AADL et de déterminer la construction Ada équivalente (types de données ou instances, entier signé ou booléen, etc.). Les interrogateurs produits pour chacune de ces propriétés sont, par conséquent, utilisés comme préconditions par nos règles de transformation. Nous distinguons deux catégories de transformation : la transformation des types de données et la transformation des instances de données (constantes et variables).

Identification d'une transformation d'un type de donnée

Dans le cas des types de données le patron source (le patron d'identification d'un composant) d'une règle de transformation est basé sur une déclaration d'un composant interface ou implantation `data` spécifiant la propriété `Data_Model::Data_Representation` (que nous détaillons plus bas). Dans le cas d'un composant de donnée partagée et/ou avec accesseurs, les accesseurs définis dans la déclaration de l'interface du composant et la propriété spécifiant le protocole sélectionné pour la gestion de la concurrence (`Concurrency_Control_Protocol`) sont ajoutés aux préconditions du patron source.

Identification d'une transformation d'une instance de donnée

Dans le cas des instances de données, le patron d'identification de la transformation est le suivant :

- un composant interface ou implantation `data` ;
- la propriété `Data_Model::Base_Type` spécifiant le type de l'instance ;
- la propriété `Access_Right` spécifiant si l'instance est accessible en lecture ou en écriture. Cette dernière nous permet de déterminer s'il s'agit d'une variable ou d'une constante.

Enfin, la propriété `Data_Model::Initial_Value` définit une valeur pour l'initialisation de l'instance.

Informations structurelles

Un composant de donnée AADL est caractérisé par son identifiant, sa catégorie, ses propriétés spécifiques (dans le cas d'un `data`, l'ensemble de propriétés `Data_Model`) et, dans le

cas d'un composant implantation, sa structure interne - *i.e.* ses sous-composants. Ces informations sont utilisées pour générer la construction Ada conformément à la spécification AADL. Nous utilisons la propriété `Source_Data_Size` pour déterminer précisément le type de donnée Ada équivalent (entier sur 32 bits, entier sur 64 bits, etc.). D'autres propriétés de l'annexe de modélisation des données permettent de déterminer la précision (nombres à virgule flottante), la signature de la donnée, etc.

Dans les sections suivantes, nous explicitons nos règles de transformation des composants `data` AADL vers les types de données du langage Ada.

8.2.1 Types basiques

Cette sous-section présente les règles de transformation pour les types basiques du langage de programmation Ada. La propriété `Data_Model::Data_Representation` spécifie le type de donnée : booléen, entier, flottant, caractère et chaîne de caractères.

Type booléen

La règle décrite par l'algorithme 17 assure la transformation d'un composant de donnée modélisant un type booléen (exemple 8.1). La sémantique du composant est équivalente à celle du type booléen Ada. Nous avons développé pour chaque type spécifié par la propriété `Data_Representation => <data_type>` un interrogateur AADL `is<data_type>()`. Le patron source de la règle (l.1) utilise cet interrogateur comme précondition pour la transformation.

Conformément à la syntaxe abstraite Ada, la génération d'un type booléen correspond à l'**instanciation** du type dérivé **Boolean** de la bibliothèque **Standard**. Le nœud `Full_Type_Declaration` (voir annexe C) est la déclaration complète du type de donnée (l.4). La propriété `Data_Model::Initial_Value` spécifie la valeur par défaut pour le type généré. Les exemples 8.1 et 8.2 représentent respectivement la source et la cible produites par la transformation.

Algorithme 17: Transformation d'un composant `data` en type booléen Ada

```
1  foreach component data D of AADL package AadlPackage | D.isBoolean() do
2      id ← Map_Ada_Name(D.name)
3      typeDef ← Gen_Derived_Type_Definition('Boolean', 'Standard')
4      adaType ← Gen_Full_Type_Declaration(id, typeDef)
5      idDefaultValue ← Map_Ada_Name(D.name, '_Default_Value')
6      initialValue ← D.get_Initial_Value()
7      constant ← true
8      defaultValue ← Gen_Default_Value(idDefaultValue, initialValue, constant,
       adaType)
9      AdaPackage.add(adaType, defaultValue)
10     AdaPackage.add_Withed_Packages('Standard')
11 end
```


© 2012 Gilles LASNIER

Exemple 8.1 – Type booléen AADL

```
data A_Boolean
 properties
  Data_Model::Data_Representation => Boolean;
  Data_Model::Initial_Value => ("false");
end A_Boolean;
```

Exemple 8.2 – Type booléen Ada

```
type A_Boolean is
  new Standard.Boolean;

A_Boolean_Default_Value : constant AADL_Boolean := false;
.
```

Exemple 8.3 – Type entier AADL

```
data My_Integer
 properties
  Data_Model::Data_Representation => Integer;
end My_Integer;

data Integer_32
 properties
  Data_Model::Data_Representation => Integer;
  Data_Model::Number_Representation => Signed;
  Source_Data_Size => 4 Bytes;
end Integer_32;
```

Exemple 8.4 – Type entier Ada

```
type My_Integer is
  new Standard.Integer

type Integer32 is
  new Interfaces.Integer_32;

for Integer32'Size use 32;

Integer32_Default_Value : Integer32
  := 0;
.
```

Type entier

L'algorithme 18 décrit la règle de transformation pour la génération d'un type entier en Ada. L'exemple 8.3 décrit la modélisation de différents types entiers en AADL (non signé, etc.).

Algorithme 18: Transformation d'un composant `data` en type entier Ada

```
1  foreach component data D of AADL package AadlPackage | D.isInteger() do
2      dataSize ← D.get_Data_Size()
3      dataSign ← D.get_Number_Representation()
4      dataType ← D.get_Type_Of_Ada_Integer(dataSize, dataSign)
5      id ← Map_Ada_Name(D.name)
6      typeDef ←
7      Gen_Ada_Numeric_Type(dataType, dataSize, dataSign)
8      adaType ← Gen_Full_Type_Declaration(id, typeDef)
9      idDefaultValue ← Map_Ada_Name(D.name, '_Default_Value')
10     initialValue ← D.get_Initial_Value()
11     constant ← true
12     defaultValue ← Gen_Default_Value(idDefaultValue, initialValue, constant,
       adaType)
13     AdaPackage.add(adaType, defaultValue)
14 end
```

L'interrogateur `Get_Type_Of_Ada_Integer` (l.4, algorithme 18) détermine si le type entier modélisé fait référence à un type de donnée Ada de la bibliothèque **Standard** ou **Interfaces**. Le résultat retourné par cet interrogateur est utilisé par la règle de génération `Gen_Ada_Numeric_Type` (l.7) appellée dans le cas des transformations des entiers et des flottants. Celle-ci assure la création du type Ada (**Integer**, **Integer_32**, **Inte_64**, etc.) en fonction de sa taille. L'exemple 8.4 illustre le code Ada produit par cette transformation à partir des patrons de modélisation AADL de l'exemple 8.3.

© 2012 Gilles LASNIER

Exemple 8.5 – Type flottant AADL

```
data Float_64
 properties
  Data_Model::Data_Representation => Float;
  Data_Model::IEEE754_Precision => Double;
  Source_Data_Size => 8 Bytes;
end Float_64;
.
```

Exemple 8.6 – Type flottant Ada

```
type Float64 is
  new Interfaces.Float64;

for Float64'Size use 64;

Float64_Default_Value : Float64
  := 0.0;
```

Exemple 8.7 – Type caractère AADL

```
data My_Character
 properties
  Data_Model::Data_Representation => Character;
end My_Character;
.
```

Exemple 8.8 – Type caractère Ada

```
type My_Character is
  new Standard.Character;

My_Character_Default_Value : constant My_Character
  := ' ';
```

Type flottant

La transformation d'un composant `data` modélisant un flottant (voir exemple 8.5) en un type Ada est décrite par l'algorithme 19. Cette règle repose sur le même principe que la règle précédente (algorithme 18) pour la génération d'un type entier. L'interrogateur `Get_Type_Of_Ada_Float` détermine précisément quel type flottant en Ada doit être généré. Les exemples 8.5 et 8.6 illustrent respectivement la source et la cible produites par cette transformation.

Algorithme 19: Transformation d'un composant `data` en type flottant Ada

foreach *component data* D *of AADL package* $AadlPackage$ | D.`isFloat()` **do**
  $dataSize \leftarrow D$.`get_Data_Size()`
  $dataPrec \leftarrow D$.`get_IEEE754_Precision()`
  $dataType \leftarrow D$.`get_Type_Of_Ada_Float`$(dataSize, dataPrec)$
  $id \leftarrow$ `Map_Ada_Name`$(D.name)$
  $typeDef \leftarrow$
  `Gen_Ada_Numeric_Type`$(dataType, dataSize, dataSign)$
  $adaType \leftarrow$ `Gen_Full_Type_Declaration`$(id, typeDef)$
  $idDefaultValue \leftarrow$ `Map_Ada_Name`$(D.name, '\_Default\_Value')$
  $initialValue \leftarrow D$.`get_Initial_Value()`
  $constant \leftarrow true$
  $defaultValue \leftarrow$ `Gen_Default_Value`$(idDefaultValue, initialValue, constant, adaType)$
  $AdaPackage$.`add`$(adaType, defaultValue)$
end

Type caractère

La règle 20 décrit la génération d'un type caractère en Ada à partir du patron de modélisation décrit dans l'exemple 8.7. Le résultat de cette transformation est illustré par l'exemple 8.8.

Type chaîne de caractères

Les patrons de modélisation (définis par l'annexe de modélisation des données) des chaînes de caractères de tailles bornées ou non bornées sont présentés dans l'exemple 8.9. La trans-

© 2012 Gilles LASNIER

Algorithme 20: Transformation d'un composant `data` en type caractère Ada

```
1  foreach component data D of AADL package AadlPackage | D.isCharacter() do
2      id ← Map_Ada_Name(D.name)
3      typeDef ← Gen_Derived_Type_Definition('Character', 'Standard')
4      adaType ← Gen_Full_Type_Declaration(id, typeDef)
5      idDefaultValue ← Map_Ada_Name(D.name, '_Default_Value')
6      initialValue ← D.get_Initial_Value()
7      constant ← true
8      defaultValue ← Gen_Default_Value(idDefaultValue, initialValue, constant,
       adaType)
9      AdaPackage.add(adaType, defaultValue)
10     AdaPackage.add_Withed_Packages('Standard')
11 end
```

Exemple 8.9 – Type string AADL

```
data My_Strings
 properties
  Data_Model::Data_Representation => String;
end My_Strings;

data Bounded_Strings
 properties
  Data_Model::Data_Representation => String;
  Data_Model::Dimension => (10);
end Bounded_Strings;
```

.

Exemple 8.10 – Type string Ada

```
package My_Strings_Pkg is
  new Ada.Strings.Bounded.Generic_Bounded_Length (64);

subtype My_Strings is
  My_Strings_Pkg.Bounded_String;

My_Strings_Default_Value : constant My_Strings
  := My_Strings_Pkg.Null_Bounded_String;

package Bounded_Strings_Pkg is
  new Ada.Strings.Bounded.Generic_Bounded_Length (10);

subtype Bounded_Strings is
  Bounded_Strings_Pkg.Bounded_String;

Bounded_Strings_Default_Value : constant Bounded_Strings
  := Bounded_Strings_Pkg.Null_Bounded_String;
```

formation de ces composants vers un type chaîne de caractères (*string*) Ada est décrite par l'algorithme 21.

Nous utilisons la propriété `Data_Model::Dimension` pour déterminer la dimension de la chaîne de caractères. Si celle-ci n'est pas spécifiée, l'interrogateur `Get_Dimension` renvoie par défaut la valeur **64** (valeur arbitraire).

La transformation du composant `data` en une chaîne de caractères Ada s'effectue en deux étapes. Dans un premier temps, nous instancions le paquetage générique **Ada.String.Bounded**. Celui fournit un environnement pour la manipulation de chaînes de caractères bornées. Il permet notamment de déterminer la taille de la chaîne au niveau de la runtime Ada et non pas lors de la compilation, ce qui préserve l'analysabilité statique. La seconde étape est la génération du type correspondant à la chaîne de caractères modélisée. Celui-ci est défini en Ada comme un sous-type du type de donnée de taille bornée configuré lors de l'instanciation du générique. L'exemple 8.10 illustre le résultat de cette transformation.

Si une valeur par défaut n'est pas définie par l'utilisateur, nous utilisons la valeur par défaut **Null_Bounded_String** définie par le paquetage générique instancié.

8.2.2 Types complexes

Nous présentons, dans cette sous-section, la génération des types de données complexes du langage Ada. Il s'agit des types énumérés, des *record* et des tableaux.

© 2012 Gilles LASNIER

Algorithme 21: Transformation d'un composant `data` en type chaîne de caractères Ada

```
1  foreach component data D of AADL package AadlPackage | D.isString() do
2      strDimension ← D.get_Dimension()
3      idStrPackage ← Map_Ada_Name(D.name, '_Pkg')
4      strGenericPackage ← Gen_Package_Instantiation('Generic_Bounded_Length',
       'Ada_Strings_Bounded', strDimension)
5      id ← Map_Ada_Name(D.name)
6      typeDef ← Gen_Derived_Type_Definition('Bounded_String', strGenericPackage)
7      adaType ← Gen_Full_Type_Declaration(id, typeDef)
8      idDefaultValue ← Map_Ada_Name(D.name, '_Default_Value')
9      initialValue ← D.get_Initial_Value()
10     constant ← true
11     defaultValue ← Gen_Default_Value(idDefaultValue, initialValue, constant,
       adaType)
12     AdaPackage.add(strGenericPackage, adaType, defaultValue)
13     AdaPackage.add_Withed_Packages('Ada_Strings_Bounded')
14 end
```

Type énuméré

L'algorithme 22 décrit la règle de transformation d'un composant `data` en un type énuméré Ada. Les attributs requis pour le patron de modélisation d'un type énuméré en AADL sont illustrés dans l'exemple 8.11. Les listes d'énumérateurs et des clauses de représentation de ces énumérateurs sont construites à l'aide des interrogateurs `Get_Enumerators` et `Get_Representation`.

Algorithme 22: Transformation d'un composant `data` en type énuméré

```
1  foreach component data D of AADL package AadlPackage | D.isEnum() and
   D.has_Enumerators() and D.has_Representation() do
2      enumerators ← D.get_Enumerators()
3      enumRepr ← D.get_Representation()
4      id ← Map_Ada_Name(D.name)
5      typeDef ← Gen_Enumeration_Type_Definition(enumerators)
6      adaType ← Gen_Full_Type_Declaration(id, typeDef)
7      enumReprClause ← Gen_Enumeration_Reprensentation_Clause(id, enumerators,
       enumRepr)
8      AdaPackage.add(adaType, enumReprClause)
9  end
```

La génération d'une énumération correspond à la génération d'un nœud de type `Enumeration_Type_Definition` (voir la BNF du langage Ada, annexe C) spécifiant la définition du type de l'objet à constuire et d'un nœud (de type `Full_Type_Declaration` correspondant à la déclaration du type). Ce dernier effectue l'association entre l'identifiant - *i.e.* le nom du type - et la définition du type.

La clause de représentation d'un énumérateur - *i.e* un littéral - définit la représentation interne de celui-ci. Cette représentation interne est utilisée pour la distinction des littéraux au niveau runtime. Pour chaque énumérateur, une clause de représentation a été spécifiée en ac-

© 2012 Gilles LASNIER

Exemple 8.11 – Type énuméré AADL

```
data An_Enum
 properties
  Data_Model::Data_Representation => Enum;
  Data_Model::Element_Names => ("foo", "bar");
  Data_Model::Representation => ("00", "11");
end An_Enum;
.
```

Exemple 8.12 – Type énuméré Ada

```
type An_Enum is
  (foo,
   bar);

for An_Enum use
  (foo => 00,
   bar => 11);
```

cord avec notre patron de modélisation d'un type énuméré AADL. La règle `Gen_Enumeration_Representation_Clause` génère les différentes clauses de représentation en fonction des valeurs spécifiées par l'utilisateur à l'aide de la propriété `Data_Model::Representation` (voir exemple 8.11). L'exemple 8.12 illustre le résultat de la transformation du composant AADL modélisant un type énuméré présenté dans l'exemple 8.11.

Type structuré

L'équivalent du type structuré AADL est le type `record` Ada. L'annexe de modélisation des données permet deux patrons de modélisation possibles présentés dans l'exemple 8.13. Le premier patron (voir patron (1), 8.13) utilise la spécification des sous-composants AADL permettant la manipulation directe des éléments du type structuré au sein du langage d'expression de l'annexe comportementale.

Le second patron est basé uniquement sur la spécification des propriétés de l'annexe `Data_Model` et ne permet pas la manipulation de ces éléments.

Les algorithmes 23 et 24 décrivent les règles de transformation de ces patrons. Un type `record` implique la création de trois nœuds au sein de l'arbre abstrait Ada. Le premier, de type `Record_Definition`, contient la définition des éléments du `record` (l.5), le second est la définition du type (l.6) et le dernier nœud représente la déclaration du type (l.7) associant l'identifiant du type à sa définition du type.

Algorithme 23: Transformation d'un composant `data` en type record - patron (1)

```
1  foreach component data implementation D of AADL package AadlPackage | D.isStruct()
   and not D.hasElementNames() and D.hasDataSubcomponents() do
2      eltNames ← D.get_Data_Subcomponent_Names()
3      eltTypes ← D.get_Data_Subcomponent_Types()
4      id ← Map_Ada_Name(D.name)
5      recordDef ← Gen_Record_Definition(eltNames, eltTypes)
6      recordTypeDef ← Gen_Record_Type_Definition(recordDef)
7      adaType ← Gen_Full_Type_Declaration(id, recordTypeDef)
8      AdaPackage.add(adaType)
9      foreach component data dataCpt in eltTypes do
10         AdaPackage.add_Withed_Packages(dataCpt.get_Ada_Package_From_AADL_Package())
11     end
12 end
```

Dans le cas du patron (1), le composant source est un composant `data implementation`. Les interrogateurs `Get_Data_Subcomponents_Names` et `Get_Data_Subcomponents_Types` permettent, à partir de l'analyse des sous-composants du composant, l'extraction des informations relatives aux identifiants et aux typages des éléments du `record`.

© 2012 Gilles LASNIER

Exemple 8.13 – Type structuré AADL

```
-- patron de modélisation (1),
-- avec sous-composants

data A_Struct1
 properties
  Data_Model::Data_Representation => Struct;
end A_Struct1;

data implementation A_Struct1.impl
 subcomponents
  f1 : data Base_Types::Float;
  c2 : data Base_Types::Character;
end A_Struct1.impl;

-- patron de modélisation (2),
-- avec propriétés

data A_Struct2
 properties
  Data_Model::Data_Representation => Struct;
  Data_Model::Base_Type =>
          (classifier (Base_Types::Float),
           classifier (Base_Types::Character));
  Data_Model::Element_Names => ("f1", "c2");
end A_Struct2;
```

Exemple 8.14 – Type record Ada

```
.

type A_Struct1_impl is
  record
    f1 : Base_Types.AADL_Float;
    c2 : Base_Types.AADL_Character;
  end record;

type A_Struct2 is
  record
    f1 : Base_Types.AADL_Float;
    c2 : Base_Types.AADL_Character;
  end record;

.
```

Dans le cas du patron (2), le composant source est un composant `data type` (interface). Les interrogateurs `Get_Element_Names` et `Get_Base_Types` assurent la même fonction que les interrogateurs précédents mais extraient les informations à partir des propriétés de l'annexe de modélisation des données.

L'exemple de code Ada 8.14 illustre le résultat produit par ces deux transformations.

Algorithme 24: Transformation d'un composant `data` en type record - patron (2)

```
1  foreach component data D of AADL package AadlPackage | D.isStruct() and
   D.hasElementNames() and D.hasBaseTypes() do
2      eltNames ← D.get_Element_Names()
3      eltTypes ← D.get_Base_Types()
4      id ← Map_Ada_Name(D.name)
5      recordDef ← Gen_Record_Definition(eltNames, eltTypes)
6      recordTypeDef ← Gen_Record_Type_Definition(recordDef)
7      adaType ← Gen_Full_Type_Declaration(id, recordTypeDef)
8      AdaPackage.add(adaType)
9      foreach component data dataCpt in eltTypes do
10         AdaPackage.add_Withed_Packages(dataCpt.get_Ada_Package_From_AADL_Package())
11     end
12 end
```

Type union

Les algorithmes 25 et 26 décrivent les transformations des patrons de modélisation d'un type union AADL en un type discriminant Ada. La génération d'un type discriminant repose sur le même principe que la génération du type `record` présentée dans la section précédente (voir 9). L'unique différence est la création de deux nœuds supplémentaires relatifs à la génération d'un type discriminant (l.4-8) conformément aux spécifications du langage Ada (voir annexe C). L'exemple de code Ada 8.16 illustre le résultat de ces deux transformations.

© 2012 Gilles LASNIER

Algorithme 25: Transformation d'un composant `data` en type discriminant - patron (1)

```
1  foreach component data implementation D of AADL package AadlPackage | D.isUnion()
   and not D.hasElementNames() and D.hasDataSubcomponents() do
2      eltNames ← D.get_Data_Subcomponents_Names()
3      eltTypes ← D.get_Data_Subcomponents_Types()
4      idDscr ← Map_Ada_Name(D.name, '_Dscr')
5      enumTypeDef ← Gen_Enumeration_Type_Definition(eltNames)
6      adaTypeDscr ← Gen_Full_Type_Declaration(idDscr, enumTypeDef)
7      knownDscr ← Gen_Known_Discriminant_Part('Dsc', adaTypeDiscr)
8      recordDef ← Gen_Record_Definition('Dsc', eltNames, eltTypes)
9      recordTypeDef ← Gen_Record_Type_Definition(recordDef)
10     idAdaType ← Map_Ada_Name(D.name)
11     adaType ← Gen_Full_Type_Declaration(idAdaType, recordTypeDef, knownDscr)
12     AdaPackage.add(adaTypeDscr, adaType)
13     foreach component data dataCpt in eltTypes do
14         AdaPackage.add_Withed_Packages(dataCpt.get_Ada_Package_From_AADL_Package())
15     end
16 end
```

Algorithme 26: Transformation d'un composant `data` en type discriminant - patron (2)

```
1  foreach component data D of AADL package AadlPackage | D.isUnion() and
   D.hasElementNames() and D.hasBaseTypes() do
2      eltNames ← D.get_Element_Names()
3      eltTypes ← D.get_Base_Types()
4      idDscr ← Map_Ada_Name(D.name, '_Dscr')
5      enumTypeDef ← Gen_Enumeration_Type_Definition(eltNames)
6      adaTypeDscr ← Gen_Full_Type_Declaration(idDscr, enumTypeDef)
7      knownDscr ← Gen_Known_Discriminant_Part('Dsc', adaTypeDiscr)
8      recordDef ← Gen_Record_Definition('Dsc', eltNames, eltTypes)
9      recordTypeDef ← Gen_Record_Type_Definition(recordDef)
10     idAdaType ← Map_Ada_Name(D.name)
11     adaType ← Gen_Full_Type_Declaration(idAdaType, recordTypeDef, knownDscr)
12     AdaPackage.add(adaTypeDscr, adaType)
13     foreach component data dataCpt in eltTypes do
14         AdaPackage.add_Withed_Packages(dataCpt.get_Ada_Package_From_AADL_Package())
15     end
16 end
```

© 2012 Gilles LASNIER

Exemple 8.15 – Type union AADL

```
-- patron de modélisation (1),
-- avec sous-composants

data A_Union1
 properties
  Data_Model::Data_Representation => Union;
end A_Union1;

data implementation A_Union1.impl
 subcomponents
  f1 : data Base_Types::Float;
  c2 : data Base_Types::Character;
end A_Union1.impl;

-- patron de modélisation (2),
-- avec propriétés

data A_Union2
 properties
  Data_Model::Data_Representation => Union;
  Data_Model::Base_Type =>
          (classifier (Base_Types::Float),
           classifier (Base_Types::Character));
  Data_Model::Element_Names => ("f1", "f2");
end A_Union2;
```

.

Exemple 8.16 – Type union Ada

```
type A_Union1_impl_Dscr is
  (f1,
   c2);

type A_Union1_impl (Dsc : A_Union2_impl_Dscr) is
  record
    case Dsc is
      when f1 => f1_DATA : Base_Types.AADL_Float;
      when c2 => c2_DATA : Base_Types.AADL_Character;
      pragma Warnings (Off);
      when others =>null;
      pragma Warnings (On);
    end case;
  end record;

type A_Union2_Dscr is
  (f1,
   f2);

type A_Union2 (Dsc : A_Union2_Dscr) is
  record
    case Dsc is
      when f1 => f1_DATA : Base_Types.AADL_Float;
      when f2 => f2_DATA : Base_Types.AADL_Character;
      pragma Warnings (Off);
      when others =>null;
      pragma Warnings (On);
    end case;
  end record;
```

Type tableau

L'algorithme 27 décrit la règle de transformation d'un composant AADL en un type tableau Ada. Dans notre cas, nous ne supportons que le type tableau spécifié à l'aide des propriétés de l'annexe de modélisation des données. Ainsi, nous utilisons les interrogateurs permettant d'extraire les informations relatives aux typages et aux dimensions du tableau.

La création d'un type tableau en Ada s'effectue en trois étapes. Prémierement, nous créons les différentes bornes du tableau (nœud du type `Range_Constraint`, l.4) en fonction des dimensions spécifiées par l'utilisateur. Puis, nous créons la définition du type tableau (l.5) et enfin, sa déclaration finale (l.7). L'exemple AADL 8.17 et l'exemple de code Ada 8.18 illustrent la source et la cible produites par cette transformation.

Algorithme 27: Transformation d'un composant `data` en type tableau

```
1  foreach component data D of AADL package AadlPackage | D.isArray() and
   D.hasBaseTypes() and D.hasDimension() and D.hasRepresentation()and
   D.hasInitialValue() do
2      baseTypes ← D.get_Base_Types()
3      dims ← D.get_Dimensions()
4      rangeConstraints ← Gen_Range_Constraints(baseTypes, dims)
5      arrayTypeDef ← Gen_Array_Type_Definition(rangeConstraints, baseTypes)
6      idAdaType ← Map_Ada_Name(D.name)
7      adaType ← Gen_Full_Type_Declaration(idAdaType, arrayTypeDef)
8      AdaPackage.add(adaType)
9      AdaPackage.add_Withed_Packages(baseTypes.get_Ada_Package_From_AADL_Package())
10 end
```


© 2012 Gilles LASNIER

Exemple 8.17 – Type tableau AADL

```
data One_Dimension_Array
 properties
  Data_Model::Data_Representation => Array;
  Data_Model::Base_Type =>
       (classifier (Base_Types::Integer));
  Data_Model::Dimension => (42);
end One_Dimension_Array;

data Two_Dimensions_Array
 properties
  Data_Model::Data_Representation => Array;
  Data_Model::Base_Type =>
       (classifier (Base_Types::Integer));
  Data_Model::Dimension => (74, 75);
end Two_Dimensions_Array;
```

Exemple 8.18 – Type tableau Ada

```
type One_Dimension_Array is
  array (1 .. 42)
    of Base_Types.AADL_Integer;


type Two_Dimensions_Array is
  array (1 .. 74, 1 .. 75)
    of Base_Types.AADL_Float;

.
```

8.2.3 Type opaque

Les composants AADL de données *opaques* (voir exemple 8.19) référencent un type de donnée spécifié dans un langage de programmation (Ada, C, etc.). L'algorithme 28 détaille la transformation permettant l'intégration de ces types de données au code généré. Celle-ci repose sur la spécification de types dérivés Ada. L'exemple de code Ada 8.20 décrit le code généré par cette transformation.

Les interrogateurs `has_Source_Language` et `has_Type_Source_Name` sont utilisés pour déterminer si le composant analysé est un type opaque. Les interrogateurs `get_Type_Source_Name` et `get_Source_Language` extraient respectivement les informations sur le nom du type et le langage de programmation utilisé pour son implantation.

Algorithme 28: Transformation d'un composant `data` en type opaque

```
1  foreach component data D of AADL package AadlPackage | D.hasDataRepresentation()
   and D.hasSourceLanguage() and D.has_Type_Source_Name() do
2      typeSrc ← D.get_Type_Source_Name()
3      id ← Map_Ada_Name(D.name)
4      typeDef ← Gen_Derived_Type_Definition('typeSrc')
5      adaType ← Gen_Full_Type_Declaration(id, typeDef)
6      if D.hasInitialValue() then
7          idDefaultValue ← Map_Ada_Name(D.name, '_Default_Value')
8          initialValue ← D.get_Initial_Value()
9          constant ← true
10         defaultValue ← Gen_Default_Value(idDefaultValue, initialValue, constant,
           adaType)
11         AdaPackage.add(adaType, defaultValue)
12     else
13         AdaPackage.add(adaType)
14     end
15
16     AdaPackage.add_Withed_Packages(D.Get_Ada_Package_From_AADL_Package())
17 end
```

© 2012 Gilles LASNIER

Exemple 8.19 – Type opaque AADL

```
-- Ada source language opaque type
data Ada_Type
 properties
  Source_Language => Ada95;
  Source_Text => ("User_Types.ads");
  Type_Source_Name =>
      "User_Types.The_Opaque_Type";
end Ada_Type;

-- C source language opaque
data C_Type
 properties
  Source_Language => C;
  Source_Text => ("types.h");
  Type_Source_Name => "the_type";
end C_Type;
```

Exemple 8.20 – Type opaque Ada

```
-- Ada source language opaque type
type Ada_Type is
  new User_Types.The_Opaque_Type;


-- C source language opaque type
type C_Type is
  new the_type;

.
```

8.2.4 Constantes et variables

Les règles de transformation pour la production des constantes et des variables sont élaborées sur le même principe que les règles de génération des types de données présentés précédemment.

Types basiques

Une seule règle de transformation permet la génération des différentes variables spécifiées à partir des types basiques que nous venons de présenter. Le patron source pour l'identification d'une variable requiert uniquement la présence des propriétés `Data_Model::Base_Types` pour déterminer le type de la variable et `Data_Model::Initial_Value` pour sa valeur. Enfin, si la propriété `Access_Right => read_only` est spécifiée sur le composant, il s'agit alors de générer une constante.

Types complexes

Les composants complexes modélisant des variables nécessitent des règles de transformation distinctes.

L'algorithme 29 décrit la règle de transformation d'un composant modélisant un tableau de données en une variable de type tableau dans le langage Ada. Les propriétés `Data_Model::Representation` et `Data_Model::Initial_Value` sont analysées pour la génération des valeurs des éléments du tableau et la propriété `Access_Right => read_only` détermine s'il s'agit d'une constante.

8.2.5 Objet protégé

Les composants de données partagées AADL (exemple 8.21) sont sémantiquement équivalents aux objets protégés Ada (exemple 8.22). L'algorithme 30 décrit la transformation. Les préconditions du patron d'identification d'un composant partagé sont basées sur la présence d'accesseurs de sous-programmes offerts par la donnée et les propriétés du standard AADL `Priority` et `Concurrency_Control_Protocol`.

La transformation produit deux nœuds de l'arbre Ada : la déclaration de la définition de l'objet protégé (l.9) et la déclaration du corps de l'objet (l.11). Les accesseurs déclarés par le composant AADL sont utilisés pour générer les spécifications des routines offertes par l'objet protégé. Les sous-composants sous-programmes (l.7) sont utilisés pour la génération du

© 2012 Gilles LASNIER

Algorithme 29: Transformation d'un composant `data` en une instance d'un type tableau avec valeurs

```
1  foreach component data D of AADL package AadlPackage | D.isArray() and
   D.hasBaseTypes() and D.hasDimension() and D.hasRepresentation() and
   D.hasInitialValue() do
2      baseType ← D.get_Base_Types()
3      dim ← D.get_Dimension()
4      eltValues ← D.get_Initial_Values()
5      eltReprs ← D.get_Representations()
6      constant ← D.get_Access_Right()
7      rangeConstraint ← Gen_Range_Constraint(baseType, '0', dim − 1)
8      arrayTypeDef ← Gen_Array_Type_Definition(rangeConstraint, baseType)
9      arrayExpr ← Gen_Array_Aggregate(eltValues, eltReprs)
10     idAdaObject ← Map_Ada_Name(D.name)
11     adaObjectDecl ← Gen_Object_Declaration(idAdaType, arrayTypeDef, constant,
       arrayExpr)
12     AdaPackage.add(adaObjectDecl)
13     AdaPackage.add_Withed_Packages(baseType.get_Ada_Package_From_AADL_Package())
14 end
```

corps des routines et les sous-composants de données (I.8) traduisent la structure interne de l'objet protégé. Enfin, la priorité spécifiée sur le composant définit la priorité de l'objet. Les exemples 8.21 et 8.22 illustrent la source et la cible produites par la transformation.

Algorithme 30: Transformation d'un composant `data` partagé en objet protégé Ada

```
1  foreach component data D of AADL package AadlPackage | D.hasAccessors() and
   D.hasPriority() and D.hasConcurrencyCtrlProtocol() do
2      priority ← D.get_Priority()
3      accessors ← D.get_Accessors()
4      dataSubCpts ← D.get_Data_Subcomponents()
5      spgSubCpts ← D.get_Spg_Subcomponents()
6      idAdaObject ← Map_Ada_Name(D.name)
7      opDecl ← Gen_Protected_Op_Declaration(accessors)
8      eltDecl ← Gen_Protected_Elt_Declaration(dataSubCpts)
9      proTypDef ← Gen_Protected_Type_Definition(idAdaObject, priority, opDecl,
       eltDecl)
10     proOperationItems ← Gen_Protected_Op_Item(spgSubCpts)
11     proBody ← Gen_Protected_Body(proOperationItems)
12     AdaPackage.add(proTypDef, proBody)
13 end
```

8.3 Transformation des composants sous-programmes

Les sous-programmes AADL sont des composants traduisant le comportement fonctionnel des tâches du système. Ce sont des entités passives appelées par les composants threads.

© 2012 Gilles LASNIER

Exemple 8.21 – Objet partagé AADL

```
data POS
features
  Update : provides subprogram access
                                Update.Impl;
  Read   : provides subprogram access
                                     Read;
properties
  Priority => 240;
  Concurrency_Control_Protocol =>
                        Priority_Ceiling;
end POS;

data implementation POS.Impl
subcomponents
  spgUpdate : subprogram Update.Impl;
  spgRead   : subprogram Read;
  Field     : data POS_Internal_Type;

connections
  Cnx_Pos_1 : subprogram access
                SpgUpdate -> Update;
  Cnx_Pos_2 : subprogram access
                    SpgRead -> Read;
properties
  Data_Model::Data_Representation =>
                                  Struct;
end POS.Impl;
```

Exemple 8.22 – Objet protégé Ada

```
protected type POS_Impl is
 procedure Update (Field : in out POS_Internal_Type);
 procedure Read (Field : in out POS_Internal_Type);
private
 Field : PolyORB_HI_Generated_Types.POS_Internal_Type;
 pragma Priority
  (240);
end POS_Impl;

protected body POS_Impl is
 procedure Update is
  begin
    Field := Field + 1;
  end Update;

 procedure Read is
  begin
     Put_Line (Normal, "Value Read:"
                & POS_Internal_Type'Image (Field));
  end Read;
end POS_Impl;
```

L'équivalent sémantique de ces composants est une procédure ou une fonction (sous-convention de la valeur de retour) dans le langage Ada.

Comme nous l'avons expliqué dans le chapitre 4, le langage AADL distingue deux catégories de sous-programmes :

1. Le sous-programme *opaque* référence un sous-programme implanté dans un langage de programmation identique ou différente du langage cible, présenté dans l'exemple de code 8.23.
2. Le sous-programme spécifié, à l'aide de l'annexe comportementale, définit un automate modélisant la structure interne (paramètres, variables, etc.), l'ensemble des états d'exécution et des opérations réalisées, présenté dans l'exemple de code 8.25.

Dans le premier cas, le sous-programme est intégré au code généré à la compilation. Dans le second cas, la description comportementale permet la génération complète du sous-programme et ne nécessite aucune intégration de code externe au code généré. La construction des nœuds de l'arbre Ada représentant un sous-programme respecte l'AST Ada (voir l'annexe C).

8.3.1 Sous-programmes opaques

Le point d'entrée de la transformation d'un sous-programme opaque est la description architecturale de son interface (`component type` AADL). Les propriétés `Source_Language`, `Source_Name` et `Source_Text` définissent respectivement le langage implantant le sous-programme, le nom du sous-programme et sa localisation dans un fichier de code source qui le contient. Nous utilisons les interrogateurs du modèle AADL pour extraire ces informations requises par les patrons sources de nos transformations. Les `features` AADL (voir exemple 8.23) décrivent les paramètres d'entrées et de sorties du sous-programme, et sont équivalentes aux paramètres des routines et des fonctions du langage Ada.

Les algorithmes 31 et 32 décrivent deux règles de transformation pour la traduction des sous-programmes opaques en sous-programmes Ada. La règle (1) concerne les sous-programmes implantés dans le langage de programmation cible. Dans le cas du langage Ada, nous utilisons la construction `renames` (algorithme 31, l.9) pour intégrer le code externe au code généré.

© 2012 Gilles LASNIER

Le résultat produit par cette transformation est présenté dans l'exemple de code 8.24 (partie haute).

La règle (2) concerne les sous-programmes spécifiés dans un langage cible différent. Dans ce cas, nous utilisons le mécanisme d'interfaçage du langage Ada avec d'autres langages- *i.e.* construction `pragma Import` (algorithme 32, l. 13). Le résultat de cette transformation est présenté dans l'exemple 8.24 (partie basse).

Algorithme 31: Transformation d'un composant `subprogram` - règle (1)

```
1  foreach component subprogram S of AADL package AadlPackage | S.isAda() and
   S.hasSourceName() do
2      adaPackage ← get_Ada_Package_From_AADL_Package(AadlPackage)
3      srcName ← S.get_Source_Name()
4      features ← S.get_Feature_Parameters()
5      parameters ← Gen_Parameter_Specification(features)
6      return ← Gen_Return_Type(features)
7      idSpgSpec ← Map_Ada_Name(S.name)
8      spgSpec ← Gen_Subprogram_Specification(idSpgSpec, parameters, return)
9      spgRenamesDecl ← Gen_Renaming_Declaration(spgSpec, srcName)
10     AdaPackage.specification.add(spgSpec)
11     AdaPackage.body.add(spgRenamesDecl)
12     AdaPackage.add_Withed_Packages(srcName.get_Ada_Package_From_AADL_Component())
13 end
```

Algorithme 32: Transformation d'un composant `subprogram` - règle (2)

```
1  foreach component subprogram S of AADL package AadlPackage | S.hasSourceLangage()
   and S.hasSourceName() do
2      adaPackage ← get_Ada_Package_From_AADL_Package(AadlPackage) ;
3      srcLangage ← S.get_Source_Langage()
4      srcName ← S.get_Source_Name()
5      features ← S.get_Feature_Parameters()
6      parameters ← Gen_Parameter_Specification(features)
7      return ← Gen_Return_Type(features)
8      idSpgSpec ← Map_Ada_Name(S.name)
9      spgSpec ← Gen_Subprogram_Specification(idSpgSpec, parameters, return)
10     convention ← Gen_Element_Association('Convention', srcLangage)
11     entity ← Gen_Element_Association('Entity', spgSpec)
12     extName ← Gen_Element_Association('External_Name', srcName)
13     spgPragmaImport ← Gen_Pragma_Statement('Import', convention, entity, extName)
14     AdaPackage.specification.add(spgSpec)
15     AdaPackage.body.add(spgPragmaImport)
16 end
```

© 2012 Gilles LASNIER

Exemple 8.23 – Sous-programmes AADL

```
-- sous-programme opaque Ada

subprogram Mul
 features
  x : in parameter My_Integer;
  y : in parameter My_Integer;
  z : out parameter My_Integer;
 properties
  source_language => Ada95;
  source_name     => "Calcul.Mul";
end Mul;

-- sous-programme opaque C

subprogram Mul2
 features
  x : in parameter My_Integer;
  y : in parameter My_Integer;
  z : out parameter My_Integer;
 properties
  source_language => C;
  source_name     => "mul_spg";
  source_text     => ("calcul.c");
end Mul2;
```

Exemple 8.24 – Sous-programmes Ada

```
-- sous-programme opaque Ada

procedure Mul
  (x : My_Integer;
   y : My_Integer;
   z : out My_Integer)
renames Calcul.Mul;

-- sous-programme opaque C

procedure Mul2
  (x : My_Integer;
   y : My_Integer;
   z : out My_Integer)
pragma Import (C, Mul2, "mul_spg");

.
```

Exemple 8.25 – Sous-programme AADL-BA

```
subprogram Mul
 features
  x : in parameter My_Integer;
  y : in parameter My_Integer;
  z : out parameter My_Integer;
end Mul;

subprogram implementation Mul.impl
 annex behavior_specification {**
  states
   s : initial final state;
  transitions
   t : s -[]-> s { z := x * y };
 **};
end Mul.impl;
```

Exemple 8.26 – Sous-programme Ada

```
procedure Mul
  (x : My_Integer;
   y : My_Integer;
   z : out My_Integer)
is
begin
   z :=  x * y;
end Mul;

.
```

8.3.2 Sous-programme avec description comportementale

L'algorithme 33 décrit la règle de transformation d'un sous-programme spécifié à l'aide de l'annexe comportementale. Le point d'entrée de cette règle est un composant sous-programme contenant un automate comportemental (une `behavior_specification`). L'interface du composant est utilisée pour la génération des paramètres d'entrée et de sortie. Dans le cas où le sous-programme spécifie des sous-composants de données - *i.e* des variables locales - nous utilisons la règle `Gen_Declarations` (l.9) pour générer les instances de données locales au sous-programme Ada.

L'interrogateur `Get_Behavior_Annex` (l.10) est utilisé pour récupérer la spécification comportementale attachée au composant. La règle de transformation `Gen_Statements` (l.10 et détaillée dans la section 8.5) analyse les constructions des langages d'action et d'expression de l'annexe comportementale et les traduit en opérations ou instructions sémantiquement équivalentes du langage Ada (expression logique, affectation d'une donnée, appel de fonction, boucle, etc.) définissant le comportement du sous-programme. L'exemple 8.26 présente le code Ada produit par la transformation.

8.4 Transformation des composants threads

Dans le chapitre 5, nous avons défini des patrons pour la modélisation explicite du comportement des composants threads périodiques et sporadiques. Les tâches applicatives spéci-

© 2012 Gilles LASNIER

Algorithme 33: Transformation d'un composant `subprogram` - règle (3)

```
1  foreach component subprogram implementation S of AADL package AadlPackage |
   S.hasBehaviorAnnex() do
2      adaPackage ← get_Ada_Package_From_AADL_Package(AadlPackage) ;
3      features ← S.get_Feature_Parameters()
4      parameters ← Gen_Parameter_Specification(features)
5      return ← Gen_Return_Type(features)
6      idSpgSpec ← Map_Ada_Name(S.name)
7      spgSpec ← Gen_Subprogram_Specification(idSpgSpec, parameters, return)
8      dataSubCpts ← S.get_Data_Subcomponents()
9      declarations ← Gen_Declarations(dataSubCpts)
10     statements ← Gen_Statements(S.get_Behavior_Annex())
11     spgBody ← Gen_Subprogram_Body(spgSpec, declarations, statements)
12     AdaPackage.specification.add(spgSpec)
13     AdaPackage.body.add(spgBody)
14 end
```

fiées par l'utilisateur ont ainsi été raffinées (voir chapitre 7, T^{tasks}) à partir de ces patrons. Nous utilisons ces spécifications comportementales pour générer le comportement des tâches dans un langage de programmation cible.

Comme pour les types de données et les sous-programmes, les patrons que nous avons définis peuvent être réutilisés pour des langages cibles différents. Ainsi, les transformées des composants threads peuvent produire des tâches Ada ou encore des fils d'exécution implantés à l'aide de bibliothèques systèmes (par exemple, le standard POSIX [pos, 2009] pour le langage C).

Dans les sous-sections suivantes, nous présentons les règles de transformation pour la génération de tâches Ada à partir des composants threads AADL.

8.4.1 Génération des tâches périodiques

Notre patron de modélisation nous assure de disposer de toutes les informations nécessaires à la génération d'une tâche cyclique Ada : un identificateur, ses différents éléments d'interface, sa période, son échéance, sa priorité, sa taille de pile, ses sous-programmes d'initialisation et de recouvrement (si spécifiés) et le sous-programme relatif à sa fonction - *i.e.* le travail effectué cycliquement par la tâche.

Les algorithmes 34 et 37 (présentés dans l'annexe C) décrivent la règle de transformation d'un composant thread périodique assurant la génération d'une tâche Ada périodique. La première étape de cette transformation (algorithme 37) est la génération de la spécification de la tâche. Lors de cette étape, nous générons les données (période, échéance, priorité et taille de la pile) requises pour la configuration d'une tâche Ada/Ravenscar.

La seconde étape est la génération de l'implantation de la tâche (algorithme 34). Lors de cette étape, les sous-composants du composant `thread` sont transformés en instances de données (l.10-11) et l'automate comportemental attaché au composant est analysé puis transformé en une séquence de déclarations du langage Ada par la règle de transformation `Gen_Task_Periodic_Loop_Statement` (l.13).

Cette règle permet l'extraction du comportement de la tâche à partir de l'automate comportemental spécifié. Elle assure la génération de la boucle infinie traduisant le caractère cyclique

© 2012 Gilles LASNIER

Algorithme 34: Transformation d'un composant `thread` périodique en tâche Ada périodique (body)

```
1  foreach component thread implementation T of AADL package AadlPackage |
   T.isPeriodic() and T.hasBehaviorAnnex() do
       - body (adb)
2      nextDeadlineVal ← Gen_Object_Declaration('Next_Deadline_Val',
       'Ada.Ada_RealTime.Time')
3      nextStart ← Gen_Object_Declaration('Next_Start', 'Ada.Real_Time.Time')
4      nextStartExpr ← Gen_Expression(+, 'System_Startup_Time', 'Task_Period')
5      nextStartAssign ← Gen_Assignment_Statement('Next_Start', nextStartExpr)
6      nextDeadlineExpr ← Gen_Expression(+, 'System_Startup_Time', 'Task_Deadline')
7      nextDeadline ← Gen_Assignment_Statement('Next_Deadline_Val',
       nextDeadlineExpr)
8      delayUntilExpr ← Gen_Expression(+, 'System_Startup_Time', 'Dispatch_Offset')
9      delayUntil ← Gen_Delay_Statement(true, delayUntilExpr)
10     dataSubCpts ← T.get_Data_Subcomponents()
11     declarations ← Gen_Declarations(dataSubCpts)
12     declarations.add(nextStart)
13     statements.add(Gen_Subprogram_Call(activateEntrypoint), delayUntil,
       nextStartAssign, nextDeadlineAssign,
       Gen_Task_Periodic_Loop_Statement(T.get_Behavior_Annex()))
       - Task body object
14     taskBody ← Gen_Task_Body(taskDecl, declarations, statements)
15     AdaPackage.specification.add(dispatchOffset, taskPeriod, taskDeadline,
       taskPriority, taskStackSize, taskDecl)
16     AdaPackage.body.add(nextDeadlineVal, taskBody)
17 end
```


© 2012 Gilles LASNIER

de la tâche à partir de la structure de l'automate. L'algorithme 35 décrit cette transformation qui produit dans l'ordre les constructions spécifiées par l'annexe comportementale (l.7, utilisation de la règle `Gen_Ada_Statements`, détaillée section 8.5), les instructions pour le calcul de la prochaine échéance et la suspension de la tâche (`delay until`), et la boucle encapsulant l'ensemble des éléments produits.

L'exemple de code Ada 5.2 (présenté dans le chapitre 5) est le résultat de la transformation d'un composant thread périodique en tâche Ada périodique.

Algorithme 35: Génération de la boucle infinie du `thread` périodique

Data : thread behavior annex $behaviorAnnex$; task body $taskBody$; Ada package $AdaPackage$

1 $delayUntilExpr \leftarrow$ `Gen_Expression`$('Next\_Start')$
2 $delayUntil \leftarrow$ `Gen_Delay_Statement`$(true,\ delayUntilExpr)$
3 $nextStartExpr \leftarrow$ `Gen_Expression`$(+, 'Next\_Start', 'Task\_Period')$
4 $nextStartAssign \leftarrow$ `Gen_Assignment_Statement`$('Next\_Start',\ nextStartExpr)$
5 $nextDeadlineExpr \leftarrow$ `Gen_Expression`$(+, 'Ada.Real\_Time.Clock', 'Task\_Deadline')$
6 $nextDeadline \leftarrow$ `Gen_Assignment_Statement`$('Next\_Deadline\_Val',\ nextDeadlineExpr)$
7 $loopStatements \leftarrow$ `Gen_Ada_Statements`$(behaviorAnnex,\ taskBody,\ AdaPackage)$
8 $loopStatements$.`add`$(delayUntil,\ nextStartAssign,\ nextDeadline)$
9 $loop \leftarrow$ `Gen_Loop_Statement`$(loopStatements)$
10 return $loop$

8.4.2 Génération des tâches sporadiques

Les algorithmes 38 et 39 (présentés dans l'annexe C) décrivent la règle de transformation d'un thread sporadique sur le même principe de transformation d'un thread périodique (présenté plus haut). La différence avec la précédente transformation concerne la génération de la boucle infinie de la tâche sporadique qui intègre l'appel à la routine `Wait_For_Incoming_Events` de la file d'attente globale de messages. Celle-ci implante le comportement pour le réveil de la tâche lors de la réception d'un message placé dans la file d'attente globale de messages de la tâche (voir algorithme 40, l.1 et l.8, annexe C).

8.5 Transformation des spécifications comportementales

La dernière transformation que nous présentons concerne la traduction des actions spécifiées au sein d'un automate comportemental attachées aux composants `thread` ou `subprogram`. Nous avons expliqué, dans le chapitre 5, que les différentes actions autorisées par notre profil AADL-HI Ravenscar au sein d'une spécification comportementale se réfèrent aux constructions du langage d'action et d'expression de l'annexe ; celles-ci exprimant des opérations classiques de manipulation de données (déclaration, affectation, modification, expressions) et d'appels de sous-programmes.

Ces constructions simplifient le processus de génération à partir de l'automate comportemental. L'algorithme 36 décrit cette règle de transformation (`Gen_Ada_Statements`). Celle-ci appelle, en fonction du type de l'action analysée, les règles de génération des structures conditionnelles, des appels de sous-programmes et du langage d'expression définis par le langage

Algorithme 36: Transformation des actions de l'annexe comportementale vers des constructions du langage Ada

Data : thread behavior annex $behaviorAnnex$

```
1  statements ← ∅
2  actions ← behaviorAnnex.get_Dispatch_Transition_Actions()
3  foreach action A in actions do
4      switch A do
5          case ConditionnalStatements
6              statements.add(Gen_Conditionnal_Ada_Statements(A))
7          case CommunicationAction
8              statements.add(Gen_Spg_Call_Ada_Statements(A))
9          case AssignmentAction
10             statements.add(Gen_Expression_Ada_Statements(A))
11         otherwise Not_Handle
12     endsw
13 end
```

Ada. La traduction de ces éléments d'annexe est triviale du fait des équivalences sémantiques entre ces éléments et les constructions fournies par le langage Ada (présentées dans le chapitre 5, section 5.4). De plus, nous rappelons que le langage d'expression de l'annexe comportementale est directement basé sur celui du langage Ada.

8.6 Intégration et compilation

Le processus de raffinement présenté dans le chapitre précédent a, de par sa nature, effectué les étapes de configuration et de déploiement des composants applicatifs et intergiciels. Ceci nous a permis de réduire considérablement la complexité du processus de production automatique du code source du système critique, comme les règles présentées dans les sections précédentes le montrent.

La dernière phase est donc l'intégration du code généré à une infrastructure de compilation permettant la production de l'exécutable final. Cette phase consiste à sélectionner, mettre en relation ou produire l'ensemble des éléments (fichiers, options, compilateurs) spécifiques requis pour la compilation des sources. Rendre cette phase automatique est un travail non négligeable qui facilitera la tâche du développeur et qui évitera l'introduction d'erreur manuelle.

Ainsi, lors de la phase d'intégration et de compilation, l'analyse des composants matériels et logiciels du modèle AADL nous permet :

- La sélection du bon type de compilateur et de sa version selon le langage de programmation choisi.
- La sélection et l'intégration des composants pré-compilés et/ou produits par des outils tiers, par génération des options et des directives de compilation comprises par le compilateur sélectionné.
- La production des fichiers permettant la compilation des sources tels les fichiers projets pour le compilateur Ada GNAT ou encore les *Makefiles*.

A l'issue de cette phase d'intégration et de compilation nous disposons d'une application TR²E critique prête à être déployée sur la plate-forme embarquée cible et exécutée.

© 2012 Gilles LASNIER

8.7 Synthèse

Nous avons présenté, dans ce chapitre, l'ensemble des transformations pour la génération automatique du code source des composants architecturaux et comportementaux de notre modèle de code pour le langage Ada (respectant les recommandations du profil Ravenscar).

Les composants AADL considérés dans les préconditions de nos règles de transformation font tous partie de notre sous-ensemble du langage AADL défini par notre profil AADL-HI Ravenscar. La phase de raffinement, présentée dans le chapitre précédent, nous a permis de réduire la complexité des règles de transformation et de traduire simplement les composants AADL vers des constructions du langage Ada. Ainsi, le mapping AADL vers Ada que nous venons de présenter peut être réutilisé pour produire des composants AADL à partir des constructions du langage Ada mais aussi dans une approche différente de la nôtre.

Les constructions produites en sortie des transformations sont basées sur un sous-ensemble des éléments de la syntaxe abstraite du langage Ada respectant la sémantique Ravenscar. Ceci facilite la maintenance et l'évolution du générateur de code.

Enfin, l'expression explicite et simple de l'ensemble des règles de transformation valide le niveau sémantique très proche de notre modèle raffiné du système et son implantation physique. Ceci renforce la cohérence entre le modèle et le code généré et facilite la traçabilité des composants. De plus, ces règles de transformation peuvent être réutilisées et adaptées facilement pour la production des composants AADL vers un autre langage impératif comme le langage C.

Le chapitre suivant présente la mise en œuvre expérimentale de notre approche à travers la réalisation d'un prototype implantant notre chaîne de transformation qui assure le raffinement, l'analyse et la production automatique du système TR^{2}E critique.

© 2012 Gilles LASNIER

© 2012 Gilles LASNIER

Quatrième partie

Mise En Oeuvre et Expérimentation

Chapitre 9

Chaîne de Transformation et de Production de Systèmes TR²E

SOMMAIRE

9.1 Introduction

Dans le chapitre 7, nous avons présenté les règles de transformation pour le raffinement des composants d'une description architecturale AADL et l'intégration des composants de l'architecture intergicielle proposée au chapitre 6. Une bonne partie de ces composants est générée automatiquement à partir des informations du modèle AADL.

Dans le chapitre 8, nous avons tiré bénéfice de notre raffinement pour l'élaboration de patrons de génération permettant une traduction simple des composants AADL vers le langage Ada. La combinaison de ces résultats nous amène à l'implantation d'un nouveau processus automatisé pour l'analyse et la production de systèmes TR²E critiques.

Dans ce chapitre, nous présentons la suite d'outils libre [5] AADL-HI qui supporte ce processus. Les constituants de cette suite d'outils sont écrits à partir de différents langages (ATL, Java et EMF/Ecore) et développés sous la forme de plug-ins ECLIPSE intégrés à la plate-forme OSATE2. AADL-HI permet de raffiner, d'analyser, de générer, de déployer et de configurer automatiquement les composants d'une application TR²E à partir de son modèle AADL.

Différents prototypes implantent les fonctionnalités nécessaires pour le support de notre processus :

1. le compilateur de l'annexe comportementale nous permet d'analyser et de disposer des arbres syntaxiques abstraits et sémantiques des descriptions comportementales AADL. Ces arbres constituent des piliers dans notre approche pour le raffinement et la génération de code ;
2. le prototype AADL-HI-*Analysers* implante les différents analyseurs pour la validation des modèles AADL tout au long de notre processus ;
3. AADL-HI-*Transformation* est le prototype implantant le processus de raffinement du modèle AADL initial de l'application vers un modèle de code analysable. Celui-ci contient les composants intergiciels optimisés pour chaque nœud et selon les besoins de l'application ;
4. AADL-HI-*Generator* implante le générateur de code Ada basé sur les règles de tranformation de modèles AADL que nous avons établis.

Ce chapitre est organisé de la manière suivante. La section 9.2 présente l'architecture d'OSATE2 qui nous sert de partie frontale et de référentiel (espace de stockage) pour l'analyse et le support des modèles AADL. Nous y présentons également l'implantation et l'intégration du compilateur de l'annexe comportementale. La section 9.3 présente l'implantation des analyseurs de modèles AADL de notre processus. La section 9.4 décrit l'architecture et l'implantation ATL de la chaîne de transformation pour le raffinement et la section 9.5 le prototype de générateur de code Ada. La section 9.6 conclut ce chapitre.

9.2 Supports et outillages AADL

Dans un premier temps, cette section s'intéresse à l'outil OSATE2 défini et poussé pour être la plate-forme de référence et fédératrice pour l'intégration de nombreux projets autour du langage AADL. Celle-ci nous servira de référentiel pour les modèles AADL.

Puis, nous présentons nos contributions visant la génération de modèles AADLv2 et la réalisation du compilateur pour le support de l'annexe comportementale. Ces contributions ont été réalisées sous la forme de plug-ins ECLIPSE et intégrées comme partie dorsale à la plate-forme OSATE2.

9.2.1 OSATE2 : espace technologique et plate-forme d'édition

Dans le chapitre 4, nous avons présenté la plate-forme OSATE2 dédiée à l'édition textuelle et à l'analyse de modèles AADL. Les fonctionnalités principales de cet outil sont l'implantation d'une partie frontale pour l'analyse syntaxique et sémantique des modèles AADL et des mécanismes facilitant l'interopérabilité des outils d'analyse et leur intégration comme partie dorsale. Pour ce faire, OSATE2 repose sur un ensemble d'outils et de technologies génératives issues

5. AADL-HI est sous les termes de la licence EPLv10 (*Eclipse Public Licence*)

© 2012 Gilles LASNIER

du développement logiciel reposant sur les canevas de développement ECLIPSE/RCP (*Rich Client Platform*) et EMF. Ces technologies font de cet outil un véritable espace technologique (défini chapitre 2, section 2.5.1) pour le support des modèles AADL .

Architecture générale

L'architecture d'OSATE2 est semblable à celle d'un compilateur classique et s'articule autour du développement de plusieurs plug-ins ECLIPSE réalisant une fonctionnalité. La figure 9.1 schématise l'architecture générale de l'outil.

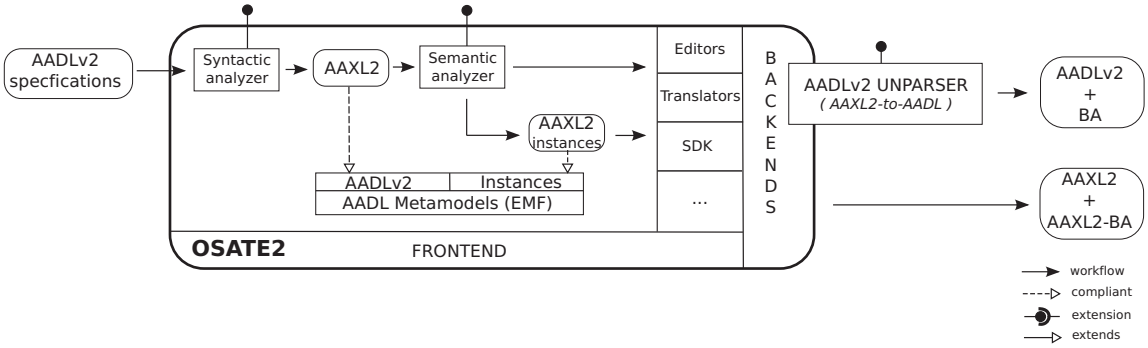


FIGURE 9.1 – Architecture de la plate-forme OSATE2

OSATE2 utilise la technologie UML 2.0 et EMF (présentée chapitre 2, section 2.6.2) pour l'élaboration du méta-modèle du langage cœur (AADLv2). La représentation finale du méta-modèle est un fichier EMF/Ecore à partir duquel sont générés les différentes classes représentant les éléments du méta-modèle. La partie frontale d'OSATE2 utilise ces classes pour construire un arbre syntaxique abstrait du langage cœur - *i.e.* une collection d'objets Java - qui sera analysé syntaxiquement et sémantiquement. L'issue de cette analyse est un arbre, dont la sémantique est conforme au standard du modèle AADL maintenu en mémoire sous la forme d'une collection d'objets Java ou sauvegardé dans un fichier AAXL2 (une représentation XMI issue de la sérialisation des objets EMF et assurant la persistance de données).

A partir de cet arbre sémantique, OSATE2 effectue la phase d'instanciation décrite par le standard AADL permettant de résoudre les prototypes, l'héritage des propriétés, des calculs sur les valeurs de propriétés, etc.

Remarque. Pendant la rédaction de ce manuscrit, une nouvelle partie frontale était en phase d'implantation. L'ancienne basée sur la technologie ANTLR [ANTLR Project, 2012] (détaillée section 9.2.2) a été abandonnée au profit de la technologie XTEXT [Eclipse Project, 2012] apportant de nombreuses améliorations en terme d'édition de modèles (compilation et analyse du modèle AADL à la volée, coloration et complétion syntaxique, etc.). La phase d'instanciation implantée par ces deux parties frontales étant incomplète, nous ne nous baserons que sur les modèles AADL déclaratifs et prendront en charge la résolution des prototypes et l'héritage des propriétés.

© 2012 Gilles LASNIER

Extensibilité et support d'annexes

Les choix technologiques pour le développement d'OSATE2 sont orientés pour faciliter l'intégration et l'interopérabilité des outils d'analyse basés sur les modèles AADL. Notamment OSATE2 permet d'étendre ses capacités et l'intégration d'outils tiers par deux mécanismes :

- des *factories* Java reposant sur le mécanisme des points d'extension ECLIPSE (visible sur la figure 9.1 détaillé section 9.2.2) ont été implantées pour permettre l'intégration et l'exécution automatisée des analyseurs syntaxique et sémantique des annexes ;
- le fichier XMI représentant le modèle AADL peut être réutilisé directement par les outils tiers et offre une plus grande interopérabilité (reposant sur XML). Des interfaces sont fournies pour la spécification d'actions autmatisées ou à réaliser à partir de l'interface graphique d'OSATE2.

Contributions à l'outil OSATE2

Lors de l'intégration des outils réalisés lors de nos travaux de recherche, nous avons contribué à améliorer le support des annexes et les *factories* Java afin de permettre l'intégration du compilateur de l'annexe comportementale. Une autre contribution a porté sur l'implantation d'un générateur de code AADLv2 à partir du XMI du modèle déclaratif permettant de produire des modèles AADL complets. Ce dernier repose sur les *switchs* EMF générés à partir du méta-modèle AADLv2 (fichier Ecore), l'utilisation de cette fontionnalité est détaillée dans la section 9.5. Ces travaux ont été réalisés dans le cadre d'une invitation par le SEI (*Software Engineering Institute*[6]).

9.2.2 Implantation de l'annexe comportementale

Cette section fait l'objet de l'implantation de l'annexe comportementale et de son intégration comme plug-in ECLIPSE au sein de la plate-forme OSATE2. La figure 9.2 décrit l'architecture globale de la partie frontale implantant l'annexe comportementale. Une partie dorsale permettant la production textuelle d'une description comportementale à partir de son arbre sémantique a aussi été implantée.

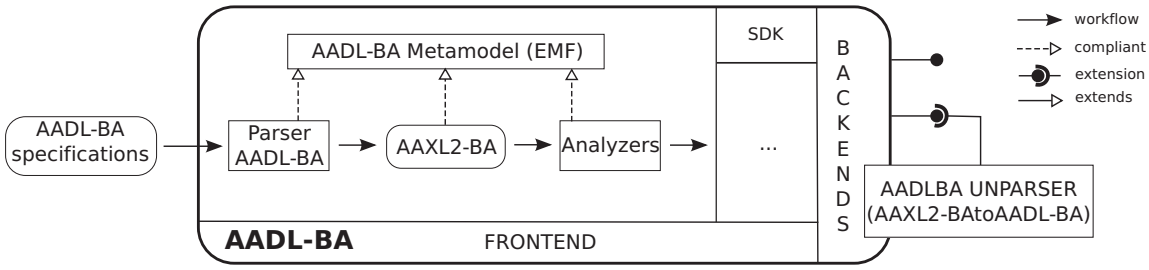


FIGURE 9.2 – Architecture du compilateur de l'annexe comportementale

Choix technologiques

Pour des raisons de simplicité de mise en œuvre et d'interopérabilité, nous avons choisi de nous appuyer sur la représentation textuelle de la description comportementale et de réutitili-

6. Invité par Peter Feiler, responsable technique du standard AADL et du développement de l'outil OSATE2.

© 2012 Gilles LASNIER

ser les technologies identiques au développement de la partie frontale d'OSATE2. Ainsi, nous avons conçu un analyseur syntaxique à l'aide d'ANTLR et un méta-modèle à l'aide d'EMF/Ecore qui permettent d'obtenir, à partir d'une description comportementale, l'arbre syntaxique abstrait (AST) et l'arbre sémantique correspondant. Cet arbre sémantique est ensuite attaché à la description architecturale du composant.

Méta-modèle

EMF (*Eclipse Modeling Framework*) est utilisé pour réaliser le méta-modèle de l'annexe et générer le code correspondant à l'arbre syntaxique abstrait des langages de l'annexe comportementale - *i.e.* les éléments du méta-modèle. Ce-dernier a été réalisé à partir de l'analyse des concepts, de la BNF et des règles sémantiques décrites par l'annexe. Pour ce faire, nous avons défini quatre règles de traduction des éléments de l'annexe vers les éléments du méta-modèle.

- R1. Les concepts spécifiés avec une sémantique forte et une représentation textuelle sont traduits par des objets **EClass** du méta-modèle - *i.e.* une classe Java.
- R2. Les concepts spécifiés avec une sémantique implicite ou introduits pour clarifier ou introduire une hiérarchie pour faciliter la compréhension des éléments de langage de l'annexe sont traduits par des interfaces abstraites **EClass**.
- R3. Les concepts appartenant à une même famille (exemple du langage d'action) sont traduits en respectant le mécanisme d'héritage Java.
- R4. Les relations et les dépendances entre les différents concepts sont traduites à l'aide des **EReferences** spécifiant le lien entre deux classes du méta-modèle.

L'ensemble de ces règles a été défini dans l'objectif de simplifier la réalisation du méta-modèle à partir des éléments décrits par l'annexe. Notamment, ces règles favorisent l'évolution du méta-modèle qui, tout au long de ces travaux de recherche, a connu des modifications du fait des changements ou des corrections apportées sur l'annexe comportementale en vue de sa standardisation finale.

La figure 9.2 (partie droite) décrit un fragment du méta-modèle de l'annexe. L'élément racine du méta-modèle est la classe **BehaviorElement** qui étend la classe **Element** du méta-modèle du langage AADLv2 fourni par OSATE2. Ceci nous permet de rattacher l'arbre sémantique d'une description comportementale à une description architecturale d'un composant AADL.

Analyseur syntaxique

ANTLR (*ANother Tool for Language Recognition*) permet de générer le code dédié à la réalisation d'interpréteurs de langages. A partir d'une description de la grammaire du langage et d'un ensemble d'actions associées, ANTLR génère le code de reconnaissance du langage et exécute les actions associées dès que la règle de grammaire est rencontrée. La figure 9.3 (encadré bas gauche) illustre la règle de grammaire ANTLR permettant l'instanciation d'un objet de type **BehaviorAnnex** généré à partir du méta-modèle de l'annexe comportementale. Notons que la structure de notre modèle simplifie le *mapping* des éléments de la BNF du langage d'annexe comportementale vers des constructions de la grammaire ANTLR.

Analyseur sémantique

Nous avons développé différents analyseurs pour la vérification des règles de nommage, légales, de cohérence et de sémantique définies par l'annexe (voir figure 9.2 précédente). Ces

© 2012 Gilles LASNIER

```
behavior_annex ::=
   [ variables { behavior_variable }+ ]
   [ states { behavior_state }+ ]
   [ transitions { behavior_transition }+ ]

behavior_state ::=
   behavior_state_identifier { , behavior_state_identifier }* :
    behavior_state_kind state ;
```

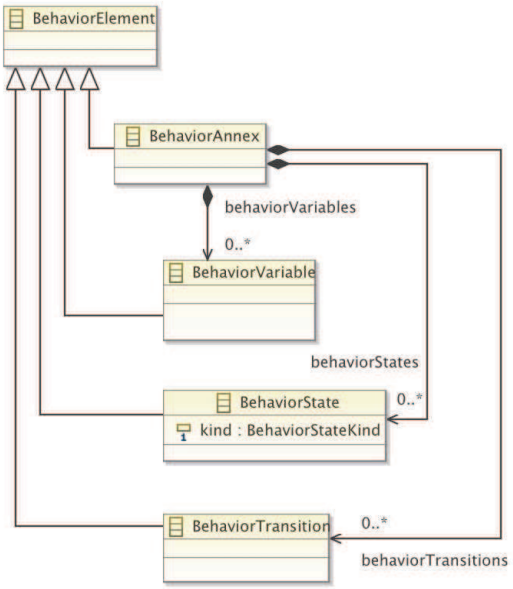
AADL-BA BNF

```
behavior_annex returns [BehaviorAnnex ba]
 @init{ ba = AadlBaF.createBehaviorAnnex();
        ba.setLocationReference
          (new LocationReference(this.getFilename(),
                                 input.get(0).getLine())) ;
} :
  ( VARIABLES ( bVars=behavior_variable
              { ba.getBehaviorVariables().add(bVars); } )+ )?

  ( STATES ( bStates=behavior_state
             { ba.getBehaviorStates().add(bStates); } )+ )?

  ( TRANSITIONS ( bTrans=behavior_transition
            { ba.getBehaviorTransitions().add(bTrans); } )+ )?
;
catch [RecognitionException ex] {
  reportError(ex);
  consumeUntil(input,SEMICOLON);
  input.consume();
}
```

AADL-BA ANTLR GRAMMAR



AADL-BA METAMODEL
(ECORE DIAGRAM)

FIGURE 9.3 – Relations BNF, méta-modèle et grammaire ANTLR

analyseurs effectuent différentes actions telles que la résolution des noms des éléments, la vérification de la cohérence vis-à-vis des composants AADL architecturaux et la vérification de la structure et de la sémantique des automates décrits.

Pour ce faire, nous avons réutilisé des bibliothèques de visiteurs du modèle AADL implantées dans OSATE2 et nous avons développé nos propres visiteurs pour les éléments du modèle comportemental. L'issue de cette analyse est un arbre décoré dont la sémantique est conforme aux spécifications de l'annexe comportementale. Cet arbre sera attaché au composant architectural ou utilisé au sein des parties dorsales du compilateur de l'annexe ou d'OSATE2.

Intégration à la plate-forme OSATE2

La figure 9.4 décrit l'intégration du plug-in de l'annexe comportementale à la plate-forme OSATE2. Cette intégration repose sur les mécanismes des points d'extension ECLIPSE. La plate-forme fournit différents points d'extensions couplés à des *factories* permettant l'enregistrement des différents analyseurs d'une annexe.

Le branchement d'un plug-in tiers à la plate-forme s'effectue en deux étapes. Premièrement, il est nécessaire d'implanter une interface permettant l'enregistrement du plug-in dans une *factory*, puis de compléter le fichier de configuration (`plugin.xml`) du plug-in et d'y référencer le point d'extension défini par OSATE2. Une fois le branchement effectué, les actions d'analyse syntaxique et d'analyse sémantique sont pilotées par les différents analyseurs du modèle AADL architectural d'OSATE2.

© 2012 Gilles LASNIER

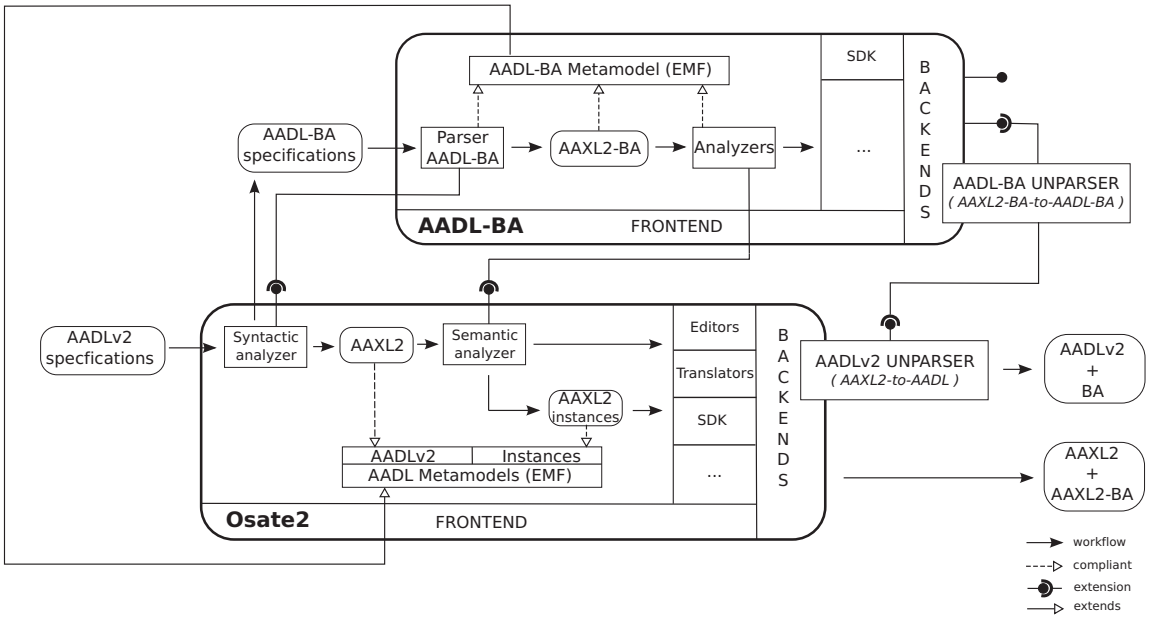


FIGURE 9.4 – Intégration du compilateur AADL-BA à la plate-forme OSATE2

Contributions

Dans cette section, nous avons présenté notre contribution pour le support de l'annexe comportementale à travers l'implantation d'un compilateur intégré à la plate-forme OSATE2. Cet outil est l'implantation référentielle de l'annexe et est maintenu par les ingénieurs de notre équipe. La réalisation de ce compilateur a été présentée à la communauté AADL et à fait l'objet d'une publication, [Lasnier *et al.*, 2011b]).

Les outils que nous venons de présenter servent à valider la syntaxe et la sémantique d'une description architecturale et comportementale d'un système TR^{2}E modélisé dans le langage AADL. Ceci constitue la première phase de l'étape 1 pour la validation du modèle utilisateur. Intéressons-nous maintenant à la vérification des restrictions et des contraintes définies par notre profil AADL-HI Ravenscar.

9.3 L'analyseur de contraintes Ravenscar

Dans le chapitre 5, nous avons présenté le profil AADL-HI Ravenscar définissant un sous-ensemble du langage AADL et un ensemble de restrictions autorisant le raffinement des composants vers des patrons de modélisation préservant l'analyse et la génération de code.

La présente section décrit l'implantation d'un prototype d'analyseur de contraintes sur une description AADL, reposant sur les *ATL queries* et *helpers* de l'outil de transformation de modèles ATL (présenté chapitre 2, section 2.6.2). Dans les sous-sections suivantes, nous rappelons le choix de l'utilisation de la technologie ATL et la réalisation de l'analyseur de contraintes Ravenscar.

© 2012 Gilles LASNIER

9.3.1 Choix technologiques

Dans notre état de l'art, nous avons expliqué que l'outil ATL proposait les *ATL queries* et *ATL helpers* pour l'expression de requêtes (invariants de classes, préconditions, postconditions...) sur les modèles sources. Ces requêtes reposent sur un sous-ensemble du langage formel d'expression de contraintes OCL qui fournit un ensemble d'opérations pour lister, itérer et calculer des propriétés à partir de l'analyse des attributs des classes d'un méta-modèle.

Le langage OCL est un langage formel standardisé par l'OMG [OMG, 2010]. C'est un langage déclaratif permettant l'expression précise de requêtes et de contraintes sur les éléments d'un méta-modèle. L'ensemble de ces caractéristiques convient particulièrement pour l'analyseur de contraintes que nous souhaitons réaliser.

De plus, nous favorisons la réutilisation des compétences d'un outil déjà intégré à notre chaîne de transformation (voir section suivante). Ceci vise à limiter le nombre d'outils hétérogènes utilisés par notre approche et réduit les problématiques d'interopérabilité et de maintenance.

9.3.2 *Mapping* d'une règle de vérification vers le langage ATL

Le *mapping* d'une règle de vérification d'une contrainte sur un élément d'un méta-modèle est simple. L'unique difficulté est de déterminer précisément le chemin d'accès aux attributs de la classe considérée. Ainsi, plus le méta-modèle est complexe - *i.e.* implication de nombreuses classes héritées et de nombreuses références parmi les attributs d'une classe, etc. - plus la requête peut être complexe.

L'exemple de code ATL 9.1 illustre l'implantation de la règle de vérification :

Chaque sous-composant d'un système de type processeur doit spécifier le protocole d'ordonnancement à priorité fixe avec la politique FIFO within priorities.

Dans le langage AADL, cette information est attachée au composant à l'aide de la propriété suivante :

```
Scheduling_Protocol => (Posix_1003_Highest_Priority_First_Protocol);
```

Exemple 9.1 – Exemple d'expression d'une règle de contrainte avec ATL

```
helper context AADL2!SystemImplementation def : checkSchedulingPolicy()
: Boolean =
   self.ownedProcessorSubcomponent->collect(e | e.name + ' : ' + e.isPosix1003HighestPriorityProtocol())
;

helper context AADL2!ProcessorSubcomponent def : isPosix1003HighestPriorityProtocol()
: Boolean =
   self.getAllProperties()
     ->select(e | e.property.name.toLower() = 'scheduling_protocol' and e.ownedValue->first().ownedValue.oclIsKindOf(AADL2!ListValue))

     ->select(e | e.ownedValue->first().ownedValue.ownedListElement->size() = 1
                  and e.ownedValue->first().ownedValue.ownedListElement.first()
                        .literal.name.toLower() = 'posix_1003_highest_priority_first_protocol')

     ->notEmpty()
;
```

Nous utilisons la construction *helper* qui peut être vue comme une méthode Java permettant d'exprimer une fonction exécutable à différentes étapes d'une transformation ou d'une requête. Cette construction est caractérisée par un nom, un contexte, zéro ou plusieurs paramètres, une valeur de retour et une expression ATL. Le contexte détermine le type de l'objet sur lequel est appelé le *helper*. Le nom du *helper* est défini par le mot-clé **def**. La valeur de retour peut être un type primitif (booléen, chaîne de caractères...) ou un élément du

© 2012 Gilles LASNIER

méta-modèle. Enfin, l'expression ATL est spécifiée à l'aide du langage ATL basé sur un sous-ensemble d'OCL.

La vérification de la contrainte présentée plus haut nécessite la définition de deux *helpers*. Le premier **checkSchedulingPolicy** s'applique sur l'implantation d'un composant `system` et exécute l'*helper* **isPosix1003HighestPriorityProtocol** sur l'ensemble de ces sous-composants processeurs. Ce dernier renvoi vrai ou faux si la bonne propriété est spécifiée. Pour ce faire, nous récupérons la liste de l'ensemble de propriétés (y compris les propriétés héritées) à l'aide du *helper* **getAllProperties** puis nous vérifions la présence de la propriété **Scheduling_Protocol** dans cette liste et la valeur de celle-ci.

Remarque. Pour la réalisation de notre analyseur et de notre chaîne de transformation, nous avons spécifié un ensemble d'*helpers* pour l'accès aux informations d'une description architecturale et comportementale AADL. L'ensemble de ces interrogateurs est réparti en trois bibliothèques :

- `AADL_Interrogators` contient les interrogateurs communs des éléments du méta-modèle AADL et du méta-modèle AADL-BA (propriétés de l'annexe de modélisation des données, valeurs des propriétés standard, etc.) ;
- `AADL_Ravenscar_Interrogators` contient des interrogateurs de plus haut niveau utilisant ceux de la première bibliothèque pour la vérification des contraintes Ravenscar (définies dans le chapitre 3) ;
- `AADLtoADA_Interrogators` contient les interrogateurs spécifiques pour la génération de code vers Ada (extraction des identifiants AADL et normalisation pour Ada, etc.).

9.3.3 Prototype d'analyseur Ravenscar

Dans la sous-section précédente, nous avons vu l'utilisation des *helpers* d'ATL pour l'implantation des règles de contrainte. L'exemple de code 9.2 présente l'implantation ATL de notre analyseur de contraintes Ravenscar. Logiquement, le principe de l'analyseur est d'invoqué un ensemble de règles de vérification pré-définies sur un modèle source. Nous utilisons pour ce faire la construction *ATL query*. Celle-ci permet l'exécution de plusieurs règles- *i.e.* des *helpers* - et la production du résultat dans un fichier texte.

La structure d'une *query* (requête) est composée d'un préambule définissant le compilateur ATL et les différents méta-modèles requis, d'un nom et d'une expression ATL. Les dépendances vers les bibliothèques d'*helpers* sont spécifiées après la requête à l'aide du mot-clé `uses`.

Dans notre exemple, la requête s'applique à toute implantation de composant `system`, présent dans le modèle source, et construit une chaîne de caractères exprimant l'ensemble des résultats des *helpers* exécutés. A la fin de l'exécution de la requête, cette chaîne est écrite dans un fichier de rapport présenté à l'utilisateur. Des exemples de rapports (chapitre 10, 10.1 et 10.5) sont présentés dans le chapitre suivant de ce manuscrit.

Remarque. Les mécanismes que nous venons de présenter ont été réutilisés à un niveau meta sur la structure des classes du méta-modèle AADL. Ceci nous a permis d'implanter un prototype de générateur d'interrogateurs pour les propriétés AADL. Ce générateur permet à un utilisateur de produire les interrogateurs d'un ensemble de propriétés spécifiques qu'il a spécifié. Ces interrogateurs peuvent être, par la suite, intégrés aux différents analyseurs pour étendre les capacités d'analyse.

© 2012 Gilles LASNIER

Exemple 9.2 – Implantation de l'analyseur Ravenscar-checker

```
-- @atlcompiler atl2010
-- @nsURI AADL2=http:///AADL2
-- @nsURI AADLBA=http:///AADLBA


-- ATL Query to drive and build the final Ravenscar Checker report

query CheckAadlRavenscar = AADL2!SystemImplementation.allInstancesFrom('IN')->iterate(e; result : String =
    'Check AADL Ravenscar Profil for system implementation << ' | result + e.qualifiedName + ' >> :\n\n'

        + 'RA1. the processor has the good scheduling policy : ' + e.checkSchedulingPolicy() + e.checkPreemptiveScheduler() + '\n\n'
        + if e.verbose_mode then e.detailSchedulingPolicy() + e.detailcheckPreemptiveScheduler() + '\n' else '' endif

        + 'RA2. all tasks are cyclic : ' + e.checkAllTasksCyclic() + '\n\n'
        + if e.verbose_mode then e.detailAllTasksDispatchProtocol() + '\n' else '' endif

        + 'RA3. check pcp : ' + e.checkPcp() + '\n\n'
        + if e.verbose_mode then e.detailCheckPcp() + '\n' else '' endif
        + '     check correct pcp : ' + e.checkCorrectPcp() + '\n\n'
        + if e.verbose_mode then e.detailCorrectPcp() + '\n' else '' endif

        + 'RC1. check ony one dispatch condition : ' + e.checkOneDispatchCondition() + '\n\n'
        + if e.verbose_mode then e.detailOneDispatchCondition() + '\n' else '' endif

        + 'RC2. check no final state : ' + e.checkNoFinalState() + '\n\n'
        + if e.verbose_mode then e.detailNoFinalState() + '\n' else '' endif

        ).writeTo('/ravenscar-query-checker/' + 'ravenscar_report.txt');

-- ATL/OCL library of AADL interrogators

uses AADL_Ravenscar_Interrogators;
```

Nous avons aussi développé un analyseur pour la vérification des contraintes sur les types de données et les informations de déploiement. Ces deux analyseurs et l'analyseur Ravenscar constituent un analyseur globale validant une description AADL d'un système critique vis-à-vis de notre profil AADL-HI Ravenscar.

Enfin, l'intégration de l'outil CHEDDAR au sein de notre suite d'outils est en cours d'implantation. Celle-ci nous permettra d'automatiser les analyses d'ordonnancement qui seront effectuées à partir de notre analyseur global.

Le modèle validé par ces différents outils que nous venons de présenter est utilisé par notre processus de raffinement. Ceci fait l'objet de la section suivante.

9.3.4 Intégration à la plate-forme OSATE2

L'intégration de nos analyseurs est simplifiée par le canevas de développement ADT de la technologie ATL. A partir d'un fichier ATL, celui-ci permet la génération d'un plug-in ECLIPSE et des classes permettant de configurer les dépendances et le moteur d'exécution des règles. Ce plug-in peut ensuite être utilisé de manière autonome ou à partir des mécanismes d'extension offerts par la plate-forme OSATE2.

9.4 Chaîne de transformation de modèles AADL

Dans le chapitre 7, nous avons expliqué l'ensemble des règles de transformation de modèles AADL pour le raffinement des composants architecturaux, l'intégration des aspects comportementaux et l'intégration des composants intergiciels requis par l'application.

Cette section décrit la chaîne de transformation de modèles AADL que nous avons implanté à l'aide du langage ATL et de son canevas de développement ADT. Dans les sous-sections suivantes, nous rappelons les motivations de ce choix, le *mapping* simple de nos

© 2012 Gilles LASNIER

règles de transformation vers les contructions du langage ATL et l'architecture de notre chaîne de transformation.

9.4.1 Choix technologiques

Notre étude préliminaire sur les outils de transformations de modèles (chapitre 2, section 2.6.2) nous a permis de présenter brièvement les caractéristiques du langage ATL et les capacités de son moteur de transformation. Celui-ci est intégré sous la forme d'un plug-in à la plate-forme ECLIPSE (le plug-in ADT). Dans le cadre de notre processus de raffinement, nous devons pouvoir décrire nos transformations selon les critères suivants :

1. Mener des transformations endogènes à partir de méta-modèles produits dans un fichier EMF/Ecore (dépendances du méta-modèle AADLv2 fourni par OSATE2 et du méta-modèle de l'annexe comportementale).
2. Produire un ou plusieurs modèles AADL cibles à partir d'un ou plusieurs modèles AADL sources.
3. Appliquer des modifications incrémentales sur les éléments d'un modèle source - *i.e* transformations *in-place* (raffinement, suppression d'éléments, etc.).
4. Exprimer nos règles de transformation dans un langage simple et facilitant la compréhension (expressivité et maintenance).
5. Exprimer les règles de transformation sous une représentation textuelle pour faciliter l'automatisation des exécutions de transformation.
6. Utiliser une technologie dont le support des outils est assuré dans la durée (évolution et maintenance).

Enfin, dans le meilleur des cas, nous devons aussi tenir compte des critères pour l'implantation d'un prototype pour la production automatisée du code source Ada des composants AADL. Ceci nécessite que le moteur transformation soit capable d'exécuter des transformations exogènes - *i.e.* d'un méta-modèle vers un autre.

ATL et son moteur de tranformation répondent à l'ensemble de ses critères. De plus, son développement et son intégration comme projet de la plate-forme ECLIPSE est un gage sur son support et son évolution dans la durée.

9.4.2 Mapping des règles de transformation avec ATL

L'exemple de code ATL 9.3 décrit la description de transformation $T^{deployment}$ dans le langage ATL. Une description de transformation ATL est caractérisée par :

1. La spécification d'un préambule identifiant et localisant les méta-modèles utilisés (– **@nsURI** ...).
2. Un **module** identifiant la description de transformation, le(s) modèle(s) cible(s) produit(s) (section **create**) et le(s) modèle(s) source(s) requis (section **from**).
3. Zéro ou plusieurs bibliothèques d'*helpers* (mot-clé **uses**), zéro ou plusieurs **helpers** locaux.
4. Une ou plusieurs règles de type *matching rules*.
5. Zéro ou plusieurs règles de type *called rules*.
6. Zéro ou plusieurs règles de type *lazy rules*.

© 2012 Gilles LASNIER

Rappel. Les *matching rules* permettent l'application d'une règle sur un élément correspondant à un patron source déterminé. Les *called rules* sont des règles impératives qui produisent à chaque exécution un nouvel élément du méta-modèle cible. Une *lazy rule* produit un nouvel élément du méta-modèle cible une seule fois. Dans le cas de plusieurs appels à la même *lazy rule*, celle-ci renverra l'élément lors du premier appel.

La règle **GeneratedDeploymentPackage** de l'exemple 9.3 (l.19-91) décrit la règle principale de la transformation $T^{deployment}$ présentée dans le chapitre 7, section 7.3.2. La section **from** décrit le patron source de la règle. Celui-ci s'applique pour tout sous-composant processus d'un composant système.

La section **using** de la règle décrit un ensemble de variables initialisées par des éléments du méta-modèle source ou par des valeurs calculées à partir d'*helpers*. Par exemple, la variable **connectedThreads** (l.32) représente la liste des threads connectés au composant processus analysé, produite par l'interrogateur **getAllThreadEntitiesConnected** (décrit dans l'annexe B).

La section **to** de la règle décrit les objets produits par la règle. Dans notre exemple, nous retrouvons les différents composants produits par notre transformation $T^{deployment}$ (détaillés dans le tableau 7.2 du chapitre 7). La production d'un composant correspond à la définition d'un objet typé par une classe du méta-modèle et à la définition des objets correspondant à ses attributs et tels qu'ils sont spécifiés au sein du méta-modèle.

Ainsi, l'objet **pohiEntityTable** (l.74) correspond au composant de donnée produit par le règle `RG.1` de la transformation $T^{deployment}$. Les propriétés de l'annexe de modélisation des données requises sont produites à l'aide de *called rules* (règle nommée par convention **gen-<nom_propriété>PropAssoc**, l.77-80). Nous utilisons les *called rules* pour factoriser le code pour la production des éléments du méta-modèle.

Enfin, l'ensemble des objets produits par la règle principale est stocké dans l'objet **Deployment** (l.39) qui est une instance de la classe `AadlPackage` représentant le paquetage AADL produit par la transformation.

Nous avons défini pour chacune des transformations de notre processus de raffinement une description de transformation - *i.e.* un module ATL. Dans la section suivante, nous présentons leur intégration dans une chaîne de transformation pour l'automatisation du processus de raffinement.

9.4.3 Architecture de la chaîne de transformation

La figure 9.5 présente notre prototype implantant notre chaîne de transformation pour le raffinement d'un modèle AADL. Le point d'entrée du prototype est la représentation XMI du modèle AADL produit par la plate-forme d'OSATE2. A partir de cette représentation, nous appliquons les transformations de notre processus une à une. Un seul ordre d'exécution des transformations est possible. Celui-ci est déterminé par les dépendances des paquetages et des composants réutilisés par les différentes transformations. Nous utilisons le générateur de code AADLv2 pour générer la représentation textuelle AADL de l'ensemble des paquetages générés et produire le modèle AADL du système complet.

Ce modèle AADL servira de point d'entrée pour les outils d'analyse. Les paquetages AADL produits serviront tous pour la production automatique du code source Ada des composants de l'application.

© 2012 Gilles LASNIER

Exemple 9.3 – Description de tranformation $T^{deployment}$ avec ATL

```
1  -- @nsURI AADL2=http:///AADL2
2
3  module TransformationDeployment;
4  create targetDeployment : AADL2
5  from   cibleUserModel   : AADL2, cibleDataModel : AADL2
6
7  uses AADL_Interrogators;
8
9  --  Attributs for prefix, suffix and separator names
10 helper def : prefixPackage : String = 'PolyORB_HI_Generated_';
11 helper def : suffixNode : String = '_K';
12 helper def : separator : String = '_';
13
14
15 -- RP rule : top-level rule to generate the PolyORB_HI_Generated_<PROCESS>_Deployment AadlPackage
16 --  context: thisModule
17 --  return : AADL2!AadlPackage
18 --
19 rule GenerateDeploymentPackage {
20         from
21                 --  Match each process subcomponent
22                 cmpSub : AADL2!ProcessSubcomponent
23         using {
24                 cmpImpl : AADL2!ProcessImplementation = cmpSub.classifier;
25                 cmpName : String = thisModule.prefixPackage.concat(cmpSub.name);
26                 sysImpl : AADL2!SystemImplementation = cmpSub.namespace;
27
28                 --  List of all node connected to the current process including himself
29                 connectedNodes : Sequence(String) = thisModule.getAllStringNodesConnected(sysImpl, cmpSub).debug(cmpSub.name);
30
31                 --  List of all thread connected to the current process including its owns threads
32                 connectedThreads : Sequence(String) = thisModule.getAllThreadEntitiesConnected(sysImpl, cmpSub);
33
34                 --  List of all port's threads connected to the current process including its owns port's threads
35                 connectedPorts : Sequence(String) = thisModule.getAllPortEntitiesConnected(sysImpl, cmpSub);
36         }
37         to
38                 --  Create 'PolyORB_HI_Generated_<node_name>_Deployment' AADLPackage
39                 Deployment : AADL2!AadlPackage (name <- cmpName.concat('_Deployment'), ownedPublicSection <- pubSection),
40                 pubSection : AADL2!PublicPackageSection (name <- cmpName.concat('_Deployment'),
41
42                 --  Create Data_Model properties import
43                 importedPropertySet <-
44                  Sequence{AADL2!PropertySet.allInstancesFrom('dataModel')->select(e | e.name.toLower() = 'data_model')->first()},
45
46                 ownedDataType <- Sequence{pohiNodeType, pohiEntityType, pohiPortType, pohiEntityTable, pohiPortTable}
47             ),
48
49             --  Rule RT.1 : create Node_Type AADL data type component
50             pohiNodeType : AADL2!DataType
51             (   name <- 'Node_Type',
52                 ownedPropertyAssociation <-
53                         Sequence{thisModule.genDataRepresentationPropAssoc('Enum'),
54                                  thisModule.genEnumeratorsPropAssoc(connectedNodes),
55                                  thisModule.genRepresentationPropAssoc(thisModule.genConsistentRepresentation(connectedNodes))} ),
56
57             --  Rule RT.2 : create Entity_Type AADL data type component
58             pohiEntityType : AADL2!DataType
59             (   name <- 'Entity_Type',
60                 ownedPropertyAssociation <-
61                   Sequence{thisModule.genDataRepresentationPropAssoc('Enum'),
62                            thisModule.genEnumeratorsPropAssoc(connectedThreads),
63                            thisModule.genRepresentationPropAssoc(thisModule.genConsistentRepresentation(connectedThreads))} ),
64
65             --  Rule RT.3 : create Port_Type AADL data type component
66             pohiPortType : AADL2!DataType
67             (    name <- 'Port_Type',
68                  ownedPropertyAssociation <-
69                    Sequence{thisModule.genDataRepresentationPropAssoc('Enum'),
70                             thisModule.genEnumeratorsPropAssoc(connectedPorts),
71                             thisModule.genRepresentationPropAssoc(thisModule.genConsistentRepresentation(connectedPorts))} ),
72
73             --  Rule RG.1 : create Entity_TABLE AADL data type component
74             pohiEntityTable : AADL2!DataType
75             (    name <- 'Entity_Table',
76                  ownedPropertyAssociation <-
77                    Sequence{thisModule.genDataRepresentationPropAssoc('Array'),
78                             thisModule.genDimensionClassifierPropAssoc('', pohiNodeType),
79                             thisModule.genInitialValuePropAssoc(connectedThreads),
80                             thisModule.genRepresentationPropAssoc(connectedNodes)} ),
81
82             --  Rule RG.2 : create Port_Table AADL data type component
83             pohiPortTable : AADL2!DataType
84             (   name <- 'Port_Table',
85                 ownedPropertyAssociation <-
86                   Sequence{thisModule.genDataRepresentationPropAssoc('Array'),
87                            thisModule.genDimensionClassifierPropAssoc('', pohiPortType),
88                            thisModule.genBaseTypePropAssoc(pohiEntityType),
89                            thisModule.genInitialValuePropAssoc(connectedPorts),
90                            thisModule.genRepresentationPropAssoc(connectedThreads)})
91 } --  end of rule RP
```

© 2012 Gilles LASNIER

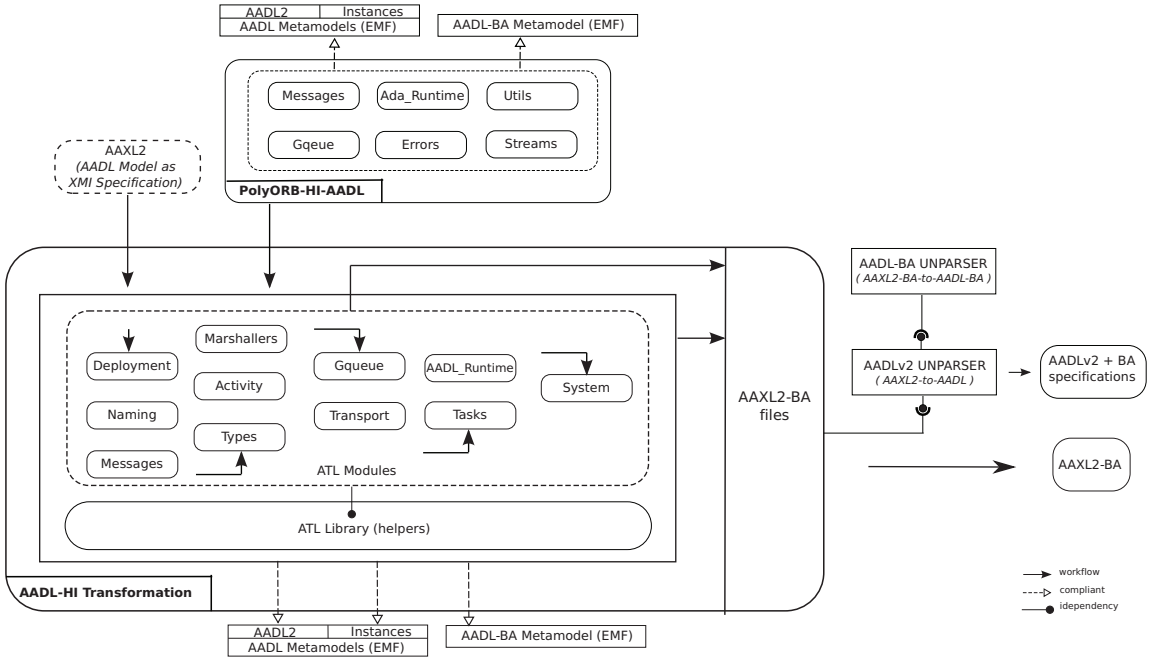


FIGURE 9.5 – Architecture de la chaîne de transformation

9.4.4 Intégration à la plate-forme OSATE2

L'intégration de notre chaîne de transformation de modèles AADL à la plate-forme OSATE2 est simplifiée par le canevas de développement ADT. Cet outil facilite la réalisation de plug-in ECLIPSE basé sur l'utilisation de descriptions de transformation ATL. Il génère à partir de ces dernières : la structure du plug-in ECLIPSE, un fichier pour la configuration des propriétés des différentes transformations et une API Java offrant différentes méthodes pour la gestion des ressources (chargement des méta-modèles, modèles source, modèles cible, bibliothèques d'*helpers*...) et pour l'exécution automatisée des différentes transformations.

Nous utilisons cette dernière pour implanter l'interface offerte par la plate-forme OSATE2 pour la définition d'actions exécutées sur les modèles AADL. Celle-ci nous permet notamment de définir différentes actions en fonction du composant AADL sélectionné.

9.5 Génération de code Ada/Ravenscar

Dans le chapitre 8, nous avons présenté l'ensemble de nos règles de transformation des composants AADL vers le langage Ada. Nous avons montré que, grâce à notre processus de raffinement (décrit section précédente) et à nos patrons de génération de code analysable (profil AADL-HI Ravenscar), ces règles de transformation se résument à des traductions simples de composants, souvent équivalentes à une construction précise du langage cible.

Nous présentons, dans cette section, la réalisation de notre prototype de générateur de code AADL vers Ada.

© 2012 Gilles LASNIER

9.5.1 Choix technologiques

Pour l'implantation de notre générateur de code, nous avons réutilisé les technologies intégrées pour la réalisation de nos précédents prototypes.

Ainsi, nous avons élaboré le méta-modèle Ada selon les mêmes règles définies pour la réalisation du méta-modèle de l'annexe comportementale. Le sous-ensemble de BNF considéré (voir annexe C) est restreint aux seuls éléments autorisés par le profil Ravenscar.

Les transformations des éléments du méta-modèle AADL vers les éléments du méta-modèle Ada ont été réalisées avec les même règles de *mapping* et nous avons défini une bibliothèque d'*helpers* spécifiques pour l'extraction des informations à partir des composants AADL conformément à nos patrons de génération de code.

Enfin, nous avons réutilisé EMF pour générer un *switch* (sorte de patron de visiteurs) à partir des éléments décrits dans le méta-modèle. Celui-ci s'apparente à un simple *pretty printer* parcourant les nœuds (resp. les classes) de l'arbre syntaxique abstrait (resp. du méta-modèle) pour produire la syntaxe textuelle Ada.

9.5.2 Architecture détaillée

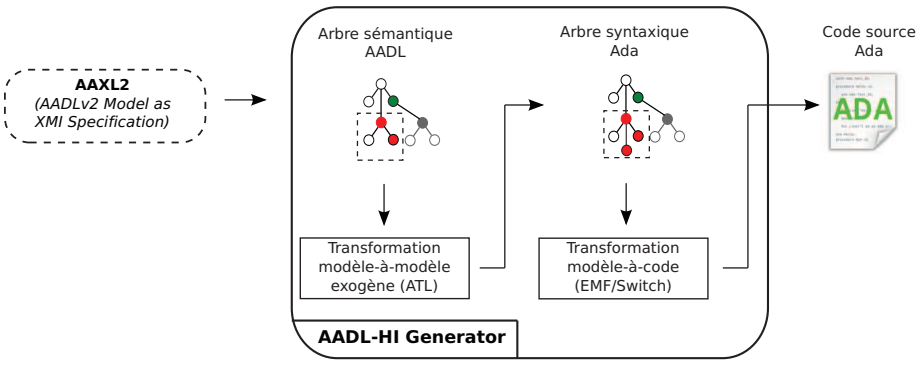


FIGURE 9.6 – Architecture du générateur de code Ada

La figure 9.6 décrit l'architecture de notre prototype de générateur de code. Celui-ci est implanté comme un plug-in ECLIPSE venant s'interfacer aux différents outils présentés dans ce chapitre. Le point d'entrée pour la génération est la représentation XMI d'un paquetage AADL produit par notre prototype de raffinement de modèles. La description de transformation ATL spécifiée permet la traduction des paquetages, des clauses d'importations, des composants AADL de données, sous-programmes et threads. Le résutat de la transformation est la représentation XMI de l'arbre syntaxique abstrait Ada. Celui-ci est pris en entrée par le *pretty-printer* (EMF/*switch*) pour la production du code source Ada correspondant au paquetage AADL pris en entrée du prototype.

Une transformation supplémentaire est effectuée à partir du paquetage AADL généré par la dernière transformation de notre processus de raffinement (transformation T^{system_final}). Celle-ci permet la génération des fichiers de support pour la compilation du code source (*Makefile* et fichier de projet GNAT).

© 2012 Gilles LASNIER

9.5.3 Intégration à la plate-forme OSATE2

Ce dernier prototype complète notre suite d'outils - *i.e.* de plug-ins ECLIPSE - AADL-HI et s'interfaee de la même manière avec la plate-forme OSATE2. Nous avons aussi défini une API Java permettant d'automatiser les étapes de notre processus de génération de code. Cette API permet la configuration et l'exécution de transformation de modèles et l'exécution du *pretty-printer* à partir de l'AST Ada produit.

9.6 Synthèse

Dans ce chapitre, nous avons présenté notre suite d'outils AADL-HI qui permet d'analyser, de générer, de déployer et de configurer les applications TR²E à partir de leurs modèles AADL. Nous avons implanté différents prototypes permettant d'assurer les étapes d'analyses et de validation de modèles AADL à différentes phases de notre processus de raffinement et d'implantation de systèmes critiques. L'ensemble de ces prototypes s'interface avec la plate-forme OSATE2 qui nous sert de référentiel pour le support des modèles AADL produits (partie frontale AADL, gestion des modèles, persistance des représentations, etc.).

Nous avons aussi présenté l'architecture de notre prototype implantant la chaîne de transformation supportant le raffinement du modèle AADL architecturale d'une application vers un modèle de code complet (applicatif + exécutif AADL) analysable. Lors de ce raffinement, les étapes de sélection, de déploiement et de configuration des composants intergiciels sont menées de manière automatisée.

Enfin, nous avons présenté notre prototype de générateur de code basé sur une transformation de modèles exogène. Celui-ci permet de produire, de manière très simple l'ensemble des composants AADL du modèle de code complet vers les constructions du langage Ada respectant le profil Ravenscar. Les fichiers de support pour la configuration et la compilation des exécutables finaux des nœuds de l'application sont aussi produits par ce générateur.

Le prochain chapitre validera notre approche sur un cas d'étude complexe et permettra de vérifier l'adéquation des solutions que nous avons apportées au problème de la cohérence entre les modèles dédiés à l'analyse et le code des systèmes TR²E produit automatiquement.

© 2012 Gilles LASNIER

Chapitre 10

Validation et Analyse des Performances

SOMMAIRE

10.1 Introduction

Dans les chapitres précédents, nous avons défini un processus pour l'analyse et la production d'une application TR^{2}E critique à partir de la description architecturale et comportementale complète de ses composants applicatifs et intergiciels. Ce modèle est obtenu par un processus de transformation raffinant la description architecturale initiale de l'application selon les restrictions et les patrons de notre profil AADL-HI Ravenscar visant à garantir l'analysabilité et la génération des composants modélisés. La bibliothèque de composants intergiciels POLYORB-HI-AADL, basée sur l'implantation POLYORB-HI-Ada, couplée à ce processus permet l'intégration des composants intergiciels (sélectionnés et configurés) requis par l'application.

Ce chapitre présente l'application de notre processus d'analyse et d'implantation de systèmes critiques à travers l'étude de cas sur lequel nous nous sommes appuyés tout au long de

ce manuscrit. Dans la section 10.2, nous rappelons brièvement l'étude de cas et sa description AADL initiale. Dans la section 10.3, nous présentons le modèle raffiné obtenu par notre chaîne de transformation et les résultats des analyses Ravenscar (sur modèle) et d'ordonnancement. La section 10.4 présente nos résultats relatifs au code généré. La section 10.5 présente les comparaisons avec l'approche OCARINA. La section 10.6 conclut ce chapitre.

10.2 L'étude de cas Ravenscar

Le cas d'étude Ravenscar que nous proposons pour valider notre approche est issu du guide pour les recommandations du profil Ravenscar [Burns *et al.*, 2004] et a été repris et revisité pour en faire une application répartie dans le but de valider une partie des travaux de notre équipe sur l'outil OCARINA. Dans cette section, nous présentons les éléments décrits par le cas d'étude et ceux ajoutés pour mettre en œuvre la répartition puis nous rappelons sa modélisation à l'aide du langage AADL.

10.2.1 Description détaillée

Exemple initial

Le cas d'étude Ravenscar (voir figure 10.1, processus `Workload_Manager`) de l'exemple Ravenscar présenté par [Burns *et al.*, 2004] décrit un système de gestion de charges de travail variables, pouvant recevoir des interruptions, assuré par différentes entités actives.

Une entité périodique de forte priorité `Regular_Producer` traite les charges de travail régulières. Sous certaines conditions, celle-ci délégue des charges irrégulières (supplémentaires) à l'entité sporadique `On_Call_Producer` de priorité moins élevée. Le tampon `Request_Buffer` gère la file d'attente des ordres des travaux supplémentaires entre les deux entités.

Les interruptions gérées par le système sont placées dans la file d'attente `Event_Queue` qui est consultée par l'entité sporadique de très forte priorité `External_Event_Server`. Le traitement des interruptions est, par la suite, délégué à l'entité sporadique `Activation_Log_Reader` de très faible priorité réveillé de temps en temps par l'entité périodique principale `Regular_Producer`. Celle-ci consulte le journal `Activation_Log` des interruptions à traiter que remplit l'entité `External_Event_Server`.

Enfin, l'entité passive `Production_Workload` effectue le traitement des charges de travail des différentes tâches.

Exemple distribué

Pour transformer l'exemple Ravenscar initial en application répartie, [Zalila *et al.*, 2008] propose d'encapsuler le système décrit précédemment dans un processus (`Workload_Manager`) et d'ajouter le processus `Interruption_Simulator` (contenant l'entité active `External_Event_Source`) pour la génération aléatoire des interruptions. Pour compléter ce passage au distribué, les deux processus s'exécutent sur deux processeurs distincts et communiquent via un bus.

Caractéristiques temporelles

Le tableau 10.1 présente les caractéristiques des entités actives. Le pire temps d'exécution a été déduit par [Zalila *et al.*, 2008] à partir de l'analyse du code Ada décrivant leur compor-

© 2012 Gilles LASNIER

tement. La colonne `Période` des entités sporadiques indique le temps minimal entre deux déclenchements d'activité.

Entité	Protocole	Priorité	Période	Échéance	WCET
Regular_Producer	Périodique	7	1000 ms	500 ms	498 ms
On_Call_Producer	Sporadique	5	1000 ms	800 ms	250 ms
Activation_Log_Reader	Sporadique	3	1000 ms	1000 ms	125 ms
External_Event_Server	Sporadique	11	5000 ms	100 ms	2 ms
External_Event_Source	Périodique	11	5000 ms	100 ms	10 ms

TABLE 10.1 – Caractéristiques des entités actives

10.2.2 Description architecturale AADL

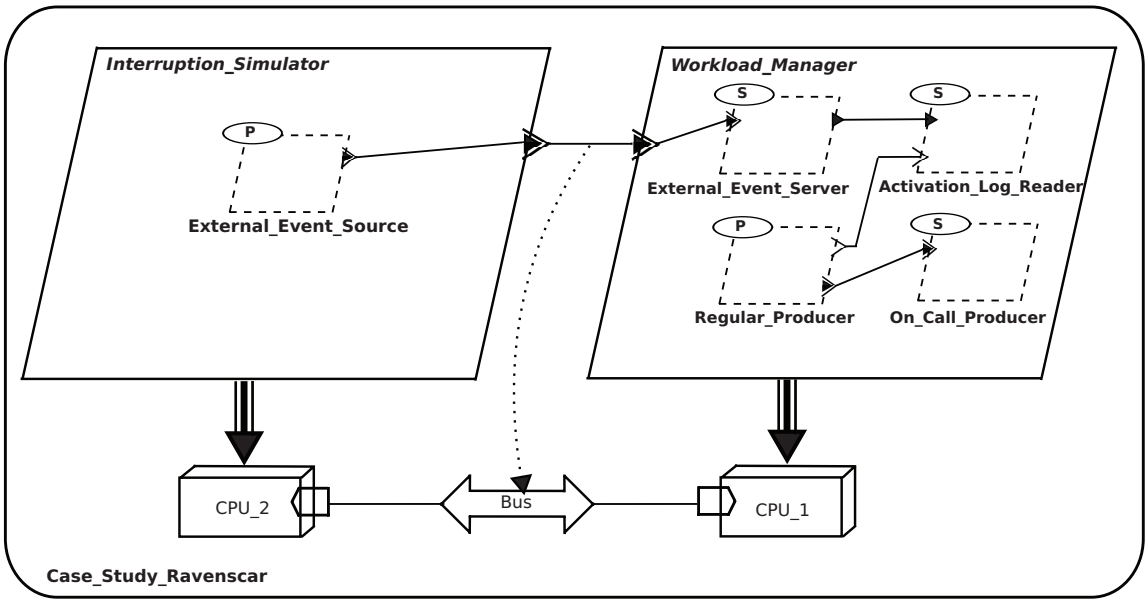


FIGURE 10.1 – Cas d'étude Ravenscar distribué

La figure 10.1 illustre la modélisation architecturale AADL de l'exemple Ravenscar dans sa version distribuée. Nous avons donné, tout au long de ce manuscrit une description des différents constituants du modèle AADL (5.1, 5.2, 7.1, 7.2, 7.3, 7.4, 7.5 et 7.6).

Les entités actives de l'exemple sont naturellement traduites par des composants threads AADL. Les caractéristiques temporelles (exprimées tableau 10.1) et fonctionnelles sont spécifiées à l'aide des propriétés du standard AADL (non représentées sur la figure) prévues à cet effet (`Dispatch_Protocol`, `Period`, `Compute_Entrypoint`, etc.) et permettent de spécifier le même comportement que celui défini dans l'exemple initial. Les différents objets partagés (tampons et files d'attente) sont spécifiés à l'aide des éléments d'interface des composants threads - *i.e.* les ports - et de leurs attributs spécifiques (`Queue_Size`, `Compute_Entrypoint`, etc). Les connexions entre ces éléments d'interface expriment les communications inter-tâches.

Pour compléter la description AADL, ces composants threads sont encapsulés dans les composants processus `Interruption_Simulator` et `Workload_Manager` attachés à des composants processeurs distincts (`CPU1` et `CPU2`) et reliés par un bus.

© 2012 Gilles LASNIER

La description AADL de l'exemple que nous venons de présenter illustre l'expressivité du langage AADL. De plus, la spécification de la communication inter-tâches et inter-processus est grandement simplifiée. L'utilisateur a juste besoin de spécifier les propriétés qui conviennent pour le protocole de transport et les informations de déploiement sur les composants processus et processeurs. Notre processus de raffinement se chargera d'intégrer et de lier les ressources du protocole de transport au sein des différents composants *process* et notre processus d'implantation de produire le code généré correspondant.

10.3 Processus de raffinement et analyses

Cette section présente nos résultats sur l'application de notre processus de raffinement sur la description architecturale AADL du système que nous venons de présenter.

10.3.1 Analyses préliminaires

Comme nous l'avons expliqué dans le chapitre 9, nous utilisons l'outil OSATE2 pour valider la syntaxe et la sémantique du modèle AADL. Une fois cette étape validée, nous procédons à l'analyse de la conformité du modèle avec notre profil AADL-HI Ravenscar et à l'analyse de son ordonnancement avec l'outil CHEDDAR.

Conformité AADL-HI Ravenscar

Nous utilisons notre analyseur (AADL-HI-*Analyzer*) pour vérifier que la description architecturale spécifie l'ensemble des informations requises et que les composants modélisés respectent notre profil AADL-HI Ravenscar. Notamment, l'analyseur Ravenscar-CHECKER vérifie les restrictions architecturales du profil Ravenscar. Le rapport 10.1 présente les résultats de cette analyse.

Nous remarquons que les trois restrictions architecturales RA1, RA2, RA3 sont respectées (définies chapitre 3). Cependant, nous ne pouvons conclure pour les restrictions définies sur les descriptions comportementales (RC1 et RC2) car aucun composant thread ne spécifie d'automate. Nous faisons l'hypothèse qu'elles sont conformes.

Analyse d'ordonnancement avec CHEDDAR

Nous utilisons l'outil CHEDDAR [Singhoff *et al.*, 2005] pour analyser l'ordonnancement du système à partir de sa description architecturale. En plus des propriétés AADL standards pour la spécification des aspects temps-réel, CHEDDAR utilise un ensemble de propriétés spécifiques (`Cheddar_Properties`) sur les composants threads et processeurs (`Fixed_Priority`, `Preemptive_Scheduler`). Nous associons à chaque composant thread les propriétés nécessaires pour l'analyse d'ordonnancement. L'exemple 10.2 montre les propriétés temporelles du composant thread `Regular_Producer`.

Pour une analyse d'ordonnancement RMA, les priorités des tâches doivent être déduites de leurs périodes respectives. Pour notre cas d'étude nous avons donner des périodes cohérentes et nous effectuons un ordonnancement de type *Fifo within priorities*.

L'exemple 10.3 montre le résultat produit par CHEDDAR pour le nœud `Workload_Manager` s'exécutant sur le processeur `CPU_1`. Nous obervons qu'une première analyse sur le taux d'utilisation processeur est réalisée. Le taux d'utilisation dans le pire cas est de `0.87340` et ne per-

© 2012 Gilles LASNIER

Exemple 10.1 – Rapport du plug-in Ravenscar-CHECKER - Modèle initial

```
RA1. the processor has the good scheduling policy and preemptive properties : true

    Ravenscar::CPU_1::Processor_Native
        [Scheduling_Protocol = Posix_1003_Highest_Priority_First_Protocol]
        [Preemptive_Scheduler = true]
    Ravenscar::CPU_2::Processor_Native
        [Scheduling_Protocol = Posix_1003_Highest_Priority_First_Protocol]
        [Preemptive_Scheduler = true]

RA2. all tasks are cyclic : true

    Ravenscar::Workload_Manager.Impl.Regular_Producer [periodic]
    Ravenscar::Workload_Manager.Impl.On_Call_Producer [sporadic]
    Ravenscar::Workload_Manager.Impl.External_Event_Server [sporadic]
    Ravenscar::Workload_Manager.Impl.Activation_Log_Reader [sporadic]
    Ravenscar::Interrupt_Simulator.Impl.InS.External_Event_Source [periodic]

RA3. check pcp : true

    Ravenscar::Workload Priority_Ceiling[false] and accessor threads [0] []
    Ravenscar::Interrupt_Counter Priority_Ceiling[false] and accessor threads [0] []

     check correct pcp : true

    Ravenscar::Workload Priority[** NULL **] MaxThreadPriority[11]
    Ravenscar::Interrupt_Counter Priority[** NULL **] MaxThreadPriority[11]

RC1. check only one dispatch condition : true

    Ravenscar::Workload_Manager.Impl.Regular_Producer [**NO BEHAVIOR_SPECIFICATION**]
    ...
    Ravenscar::Interrupt_Simulator.Impl.External_Event_Source [**NO BEHAVIOR_SPECIFICATION**]

RC2. check no final state : true

    Ravenscar::Workload_Manager.Impl.Regular_Producer [**NO BEHAVIOR_SPECIFICATION**]
    ...
    Ravenscar::Interrupt_Simulator.Impl.External_Event_Source [**NO BEHAVIOR_SPECIFICATION**]
```

Exemple 10.2 – Modèle du composant `Regular_Producer`

```
thread Regular_Producer
  features
    Additional_Workload: out event data port Workload;
    Handle_External_Interrupt: out event port ;
  properties
    Compute_Entrypoint_Source_Text => "Work.Regular_Producer_Operation";
    Dispatch_Protocol => Periodic;
    Period => 1000 ms;
    Deadline => 500 ms;
    Compute_Execution_Time => 0 ms .. 498 ms;
    Cheddar_Properties::Fixed_Priority => 7;
end Regular_Producer;
```

Exemple 10.3 – Résultat du test d'ordonnancement sur le modèle initial

```
[...]

Scheduling feasibility, Processor component case_study.native.cpu_1 :

1) Feasibility test based on the processor utilization factor :

- The base period is 5000 (see [18], page 5).
- 633 units of time are unused in the base period.
- Processor utilization factor with deadline is 1.45350 (see [1], page 6).
- Processor utilization factor with period is 0.87340 (see [1], page 6).
- Invalid scheduler : can not apply the feasibility test on processor utilization factor.

2) Feasibility test based on worst case thread component response time :

- Bound on thread component response time : (see [2], page 3, equation 4).
    case_study.native.wom.activation_log_reader => 875
    case_study.native.wom.on_call_producer => 750
    case_study.native.wom.regular_producer => 500
    case_study.native.wom.external_event_server => 2
- All thread component deadlines will be met : the thread component set is schedulable.

[...]
```

met pas de conclure sur l'ordonnançabilité des tâches par une analyse de type RMA. CHEDDAR effectue alors une analyse basée sur le temps de réponse de chacune des tâches. Celle-ci permet de conclure que les tâches du nœud `Workload_Manager` respectent leur échance, le système est ordonnançable.

Les analyses préliminaires que nous venons de présenter ont validé la description architecturale intiale du système. Nous pouvons, maintenant, passer au processus de raffinement.

10.3.2 Modèle raffiné

Nous utilisons notre outil AADL-HI pour le raffinement du modèle initial. La figure 10.2 présente le composant processus `Workload_Manager`[7] raffiné et les différents composants intergiciels intégrés. Nous remarquons particulièrement la génération des différentes files d'attente globale de messages (`EES_GQ`, `ALR_GQ`, `RP_GQ` et `OCP_GQ`) pour l'ensemble des composants threads. Le composant thread `Receiver_Task` traitant la réception des messages sur le nœud et la délivrance du message dans la file d'attente correspondante a été intégré uniquement au nœud récepteur `Workload_Manager`. L'exemple 10.4 présente la description AADL textuelle de ce composant.

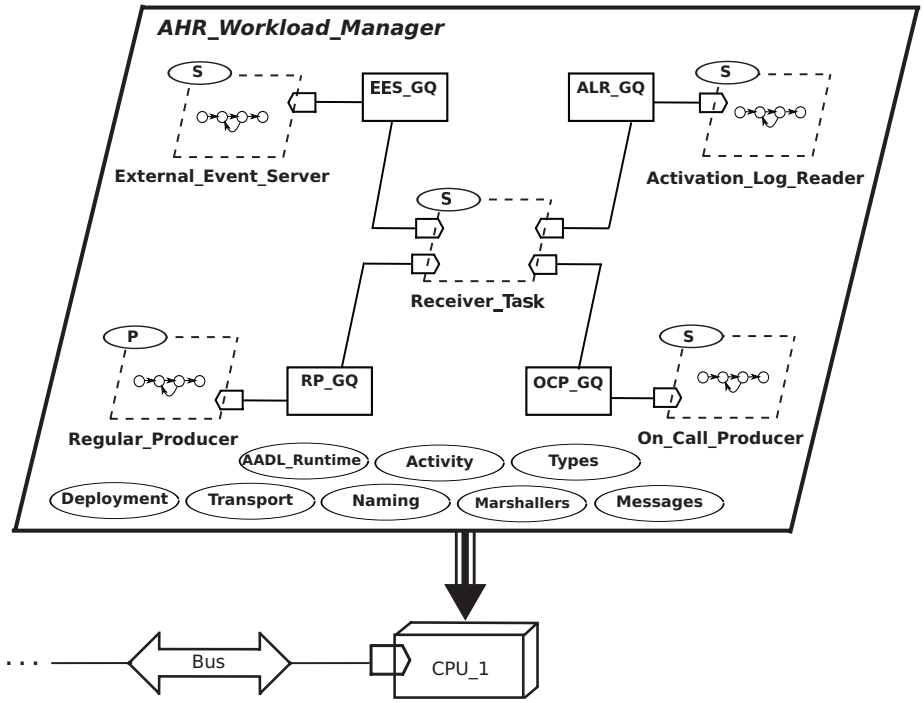


FIGURE 10.2 – Processus `Workload_Manager` du modèle raffiné

10.3.3 Analyse Ravenscar

Nous effectuons maintenant la seconde étape de validation. Nous commencons par vérifier que notre modèle est toujours conforme aux restrictions définies par notre profil AADL-HI

7. Pour des raisons de visibilité, les noms réels des files d'attente globale de messages ont été tronquées.

© 2012 Gilles LASNIER

Exemple 10.4 – Modèle du composant `Workload_Manager`

```
process implementation AHR_Workload_Manager.Impl
subcomponents
  Regular_Producer: thread Ravenscar_Workload_Manager_Regular_Producer_Task;
  On_Call_Producer: thread Ravenscar_Workload_Manager_On_Call_Producer_Task;
  External_Event_Server: thread Ravenscar_Workload_Manager_External_Event_Server_Task;
  Activation_Log_Reader: thread Ravenscar_Workload_Manager_Activation_Log_Reader_Task;
  Receiver_Task : thread Ravenscar_Workload_Manager_TCP_IP_Receiver_Task;
  Regular_Producer_Global_Queue: data WoM_Regular_Producer_Gqueue::Global_Queue.impl;
  On_Call_Producer_Global_Queue: data WoM_On_Call_Producer_Gqueue::Global_Queue.impl;
  External_Event_Server_Global_Queue: data WoM_External_Event_Server_Gqueue::Global_Queue.impl;
  Activation_Log_Reader_Global_Queue: data WoM_Activation_Log_Reader_Gqueue::Global_Queue.impl;
connections
  Regular_Producer_GQ_Access : data access Regular_Producer_Global_Queue -> Regular_Producer.GQ;
  On_Call_Producer_GQ_Access : data access On_Call_Producer_Global_Queue -> On_Call_Producer.GQ;
  External_Event_Server_GQ_Access : data access External_Event_Server_Global_Queue -> External_Event_Server.GQ;
  Activation_Log_Reader_GQ_Access : data access Activation_Log_Reader_Global_Queue -> Regular_Producer.GQ;
  TCP_IP_Receiver_Task_Regular_Producer_GQ_Access : data access
                                        Regular_Producer_Global_Queue -> Receiver_Task.Regular_Producer_GQ;
  TCP_IP_Receiver_Task_On_Call_Producer_GQ_Access : data access
                                        On_Call_Producer_Global_Queue -> Receiver_Task.On_Call_Producer_GQ;
  TCP_IP_Receiver_Task_External_Event_Server_GQ_Access : data access
                                External_Event_Server_Global_Queue -> Receiver_Task.External_Event_Server_GQ;
  TCP_IP_Receiver_Task_Activation_Log_Reader_GQ_Access : data access
                                Activation_Log_Reader_Global_Queue -> Receiver_Task.Activation_Log_Reader_GQ;
properties
 Deployment::Port_Number => 12001;
 Cheddar_Properties::Critical_Section =>
  (     "Regular_Producer_Global_Queue", "Regular_Producer", "1", "498",
        "On_Call_Producer_Global_Queue", "On_Call_Producer", "1", "250",
        "External_Event_Server_Global_Queue", "External_Event_Server", "1", "2",
        "Activation_Log_Reader_Global_Queue", "Activation_Log_Reader", "1", "125",
        "Regular_Producer_Global_Queue", "Receiver_Task", "1", "10",
        "On_Call_Producer_Global_Queue", "Receiver_Task", "1", "10",
        "External_Event_Server_Global_Queue", "Receiver_Task", "1", "10",
        "Activation_Log_Reader_Global_Queue", "Receiver_Task", "1", "10" );
end AHR_Workload_Manager.Impl;
```

Ravenscar. Le rapport 10.5 montre le résultat de cette analyse. Nous remarquons que les restrictions RA1, RA2 et RA3 sont toujours vérifiées et prennent en compte les différentes files d'attente globale de messages générées pour les threads (le nombre et le nom des composants threads accédant à une file d'attente est renseigné).

Contrairement à la première analyse, les restrictions comportementales RC1 (sur l'unique point de suspension d'une tâche) et RC2 (sur le comportement infini d'une tâche) sont maintenant vérifiées à partir des descriptions comportementales des tâches rendues explicites par notre processus de raffinement (sous la forme d'automate).

10.3.4 Analyse d'ordonnancement

CHEDDAR est de nouveau utilisé pour l'analyse d'ordonnancement. Nous gardons les mêmes hypothèses temporelles pour les composants threads qui ont été raffinés.

Les files d'attente globales de messages sont prises en compte par l'analyse (voir le nœud `Workload_Manager` raffiné, exemple 10.4). Celles-ci sont partagées entre leur thread respectif et la tâche de réception de la couche basse de transport. CHEDDAR définit aussi des propriétés spécifiques pour la prise en compte des données partagées lors de l'analyse d'ordonnancement (`Data_Concurrency_State` et `Critical_Section`), celles-ci sont générées à partir des hypothèses temporelles initiales et ajoutées à la description raffinée.

Conformément à notre processus de raffinement, la tâche de réception est intégrée uniquement sur le nœud récepteur `Workload_Manager` et a une priorité supérieure à l'ensemble des priorités des tâches du nœud. De plus, sa période est égale à la plus petite période des tâches émettrices connectées au nœud afin de respecter les contraintes temporelles spécifiées par l'utilisateur. Dans notre cas, la période considérée est celle de la tâche `External_Event_Source` du nœud `Interrupt_Simulator` (`5000 ms`, voir tableau 10.1).

© 2012 Gilles LASNIER

Exemple 10.5 – Rapport du plug-in Ravenscar-CHECKER sur le modèle raffiné

```
RA1. the processor has the good scheduling policy and preemptive properties : true
     ...

RA2. all tasks are cyclic : false

     AHR_Ravenscar::WoM::AHR_Workload_Manager.Impl.Regular_Producer [periodic]
     AHR_Ravenscar::WoM::AHR_Workload_Manager.Impl.TCP_IP_Receiver_Task [sporadic]
     ...

RA3. check pcp : true

     ...WoM_Types::Workload Priority_Ceiling[false] and accessor threads [0] []
     ...WoM_Types::Interrupt_Counter Priority_Ceiling[false] and accessor threads [0] []
     ...InS_Types::Interrupt_Counter Priority_Ceiling[false] and accessor threads [0] []
     AHR_...WoM...Regular_Producer_Global_Queue and accessor thread [2] [Regular_Producer, Receiver_Task];
     ...
     AHR_...WoM...InS_External_Event_Source_Global_Queue and accessor thread [1] [External_Event_Source];

      check correct pcp : true

     ...WoM_Types::Workload Priority[** NULL **] MaxThreadPriority[11]
     ...WoM_Types::Interrupt_Counter Priority[** NULL **] MaxThreadPriority[11]
     ...InS_Types::Interrupt_Counter Priority[** NULL **] MaxThreadPriority[11]
     AHR_...WoM...Regular_Producer_Global_Queue Priority[240] MaxThreadPriority[11]
     ...
     AHR_...InS...External_Event_Source_Global_Queue Priority[240] MaxThreadPriority[11]

RC1. check only one dispatch condition : true

     AHR_Ravenscar::WoM::AHR_Workload_Manager.Impl.Regular_Producer [1]
     ...
     AHR_Ravenscar::InS::AHR_Interrupt_Simulator.Impl.External_Event_Source [1]

RC2. check no final state : true

     AHR_Ravenscar::WoM::AHR_Workload_Manager.Impl.Regular_Producer [0]
     ...
     AHR_Ravenscar::InS::AHR_Interrupt_Simulator.Impl.External_Event_Source [0]
```

Exemple 10.6 – Résultat du test d'ordonnancement sur le modèle raffiné

```
[...]

Scheduling feasibility, Processor component case_study.native.cpu_1 :

1) Feasibility test based on the processor utilization factor :

- The base period is 5000 (see [18], page 5).
- 583 units of time are unused in the base period.
- Processor utilization factor with deadline is 1.45550 (see [1], page 6).
- Processor utilization factor with period is 0.87540 (see [1], page 6).
- Invalid scheduler : can not apply the feasibility test on processor utilization factor.

2) Feasibility test based on worst case thread component response time :

- Bound on thread component response time : (see [2], page 3, equation 4).
     case_study.native.wom.activation_log_reader => 875
     case_study.native.wom.on_call_producer => 760
     case_study.native.wom.regular_producer => 510, missed its deadline (deadline = 500)
     case_study.native.wom.external_event_server => 12
     case_study.native.wom.receiver_task => 10
- All thread component deadlines will be met : the thread component set is schedulable.

[...]
```


© 2012 Gilles LASNIER

Le résultat obtenu par l'outil CHEDDAR est présenté dans l'exemple 10.6. Nous observons que le taux d'occupation du processeur a augmenté (`0.87540`) en raison de l'ajout des composants intergiciels. L'ensemble des tâches du nœud `Workload_Manager` n'est plus ordonnançable. Les hypothèses temporelles initiales de l'utilisateur sont trop strictes et ne permettent pas l'intégration des files d'attente et de la tâche de réception sans compromettre la faisabilité du système.

Nous constatons que le thread `Regular_Producer` rate son échéance. CHEDDAR a calculé de nouvelles hypothèses temporelles pour cette tâche afin d'obtenir un système ordonnançable. En modifiant l'échéance de cette tâche (`510 ms` au lieu de `500 ms`), l'ordonnancement du système est validé.

10.3.5 Synthèse

Nous venons de présenter le modèle AADL complet du système obtenu à partir de notre processus de raffinement. Notre processus automatique d'analyse (incluant Ravenscar-CHECKER pour la conformité Ravenscar et CHEDDAR pour l'ordonnancement) montre l'intégration et l'impact des ressources intergicielles sur le système.

Ainsi, en explicitant la tâche de réception du nœud `Workload_Manager`, nous avons détecté une anomalie dans la conception du système. Celle-ci a pu être corrigée en modifiant les spécifications initiales de l'utilisateur et nous avons pu obtenir un système valide.

Cette seconde étape de validation étant achevée, nous pouvons maintenant procéder à la génération de code du système à partir de ce modèle complet.

10.4 Génération de code

Nous utilisons notre prototype AADL-HI pour générer le code Ada du système décrit par notre modèle raffiné. Nous présentons dans cette section les traces d'exécution, les résultats en terme de génération de code et des statistiques sur les nœuds de l'application.

10.4.1 Traces d'exécution

L'exemple 10.7 montre la trace d'exécution du nœud `Interruption_Simulator`. Le comportement attendu est respecté. Les interruptions sont envoyées de manière pseudo aléatoire (critère arbitraire) par la tâche `External_Event_Source` dont le temps minimal entre deux déclenchements a été fixé à 5 secondes (voir tableau 10.1) pour respecter les contraintes temporelles du thread `External_Event_Server` qui reçoit les interruptions sur le nœud `Workload_Manager`.

Exemple 10.7 – Trace d'exécution du nœud `Interruption_Simulator`

```
[ 0.000000000] External Events: starting
[ 0.000103000] External Events: send an new event: 1. Next divisor = 3
[ 0.000130000] Using user-provided TCP/IP stack to send
[ 15.000076000] External Events: send an new event: 2. Next divisor = 5
[ 15.000099000] Using user-provided TCP/IP stack to send
[ 25.000079000] External Events: send an new event: 3. Next divisor = 7
[ 25.000098000] Using user-provided TCP/IP stack to send
```

La trace d'exécution du nœud `Workload_Manager` est présentée dans l'exemple 10.8. Nous remarquons l'activité des quatre tâches du système. La tâche `Regular_Producer` effectue sa charge périodiquement et délègue une partie à la tâche `On_Call_Producer`. Toutes les interruptions externes sont reçues et traitées par la tâche `External_Event_Server`.

© 2012 Gilles LASNIER

Exemple 10.8 – Trace d'exécution du nœud `Workload_Manager`

```
[ 0.000227000] Using user-provided TCP/IP stack to receive
[ 0.000268000] External Event Server: received an external interrupt
[ 0.000274000] External Event Server: end of sporadic activation.
[ 0.000321000] Regular Producer: doing some work.
[ 1.016714000] Regular Producer: end of cyclic activation
[ 1.016894000] Regular Producer: doing some work.
[ 2.015650000] Sending extra work to 'On_Call_Producer': 250
[ 2.020319000] Regular Producer: end of cyclic activation
[ 2.020431000] On Call Producer: doing some work.
[ 2.036213000] Regular Producer: doing some work.
[ 3.015800000] On Call Producer: end of sporadic activation.
[ 3.529128000] Signaling 'Activation Log Reader'
[ 3.532286000] Regular Producer: end of cyclic activation
[ 3.532388000] Activation Log Reader: do some work.
[ 3.548246000] Regular Producer: doing some work.
[ 4.036544000] Read external new interruption: 1. Arrived at [ 0.100266000]
[ 4.040284000] Activation Log Reader: end of parameterless sporadic activation.
[ 4.795274000] Regular Producer: end of cyclic activation
...
[ 14.316238000] Regular Producer: doing some work.
[ 15.004260000] Using user-provided TCP/IP stack to receive
[ 15.004364000] External Event Server: received an external interrupt
[ 15.004440000] External Event Server: end of sporadic activation.
[ 15.077115000] Read external new interruption: 2. Arrived at [ 15.104292000]
[ 15.087733000] Activation Log Reader: end of parameterless sporadic activation.
[ 15.608263000] On Call Producer: end of sporadic activation.
[ 16.108015000] Regular Producer: end of cyclic activation
[ 16.112293000] Regular Producer: doing some work.
[ 17.112569000] Regular Producer: end of cyclic activation
[ 17.116232000] Regular Producer: doing some work.
[ 18.111582000] Signaling 'Activation Log Reader'
...
[ 25.184411000] Activation Log Reader: do some work.
[ 25.200249000] Regular Producer: doing some work.
[ 25.677438000] Read external new interruption: 3. Arrived at [ 25.100176000]
[ 25.687750000] Activation Log Reader: end of parameterless sporadic activation.
```

10.4.2 Empreinte mémoire

Cette sous-section présente des statistiques calculées à partir des mesures sur les tailles de fichiers sources et les empreintes mémoire des binaires [8] produit pour notre étude de cas.

Dans le tableau 10.2, nous donnons la taille des fichiers sources pour chaque nœud de l'application. La partie minimale de notre intergiciel représente les paquetages "utiles" écrits à la main, pour la gestion des erreurs, des entrées/sorties, etc.

Nous observons que la taille du code écrit par l'utilisateur et la taille du code écrit de l'intergiciel (appelé minimal) est très petite par rapport à la taille globale du code de l'application (respectivement 2,5% et 15% pour `Interruption_Simulator` et, 4% et 7% pour `Workload_Manager`). Notamment, nous générons environ 90% du code du nœud `Workload_Manager`. Les mesures sur ce nœud sont plus expressives car il contient plus de composants, cependant elles restent tributaires de la taille des composants utilisateurs qui est ici minimale.

	Interruption_Simulator	**Workload_Manager**
Code de l'utilisateur	51	200
Code de l'intergiciel minimal	304	304
Code de l'intergiciel généré	1647	4343
Total	2002	4847

TABLE 10.2 – Taille des fichiers sources (en ligne de code) des nœuds raffinés

Le tableau 10.3 décrit quant à lui l'empreinte mémoire des exécutables des nœuds raffinés. L'empreinte totale du nœud `Workload_Manager` est de `560 Ko` dont `448 Ko` correspondent à la runtime Ada. Nous observons que la taille des composants intergiciels est différente en

8. Nous utilisons `size` des `binutils` pour effectuer ces mesures, www.gnu.org/software/binutils/.

© 2012 Gilles LASNIER

fonction du nœud (`103 Ko` pour `Workload_Manager` et `35 Ko` pour `Interruption_Simulator`. Ceci est du à la personnalisation des composants en fonction des besoins de chaque nœud.

	Interruption_Simulator	**Workload_Manager**
Code de l'utilisateur	4	4
Code de l'intergiciel minimal	12	12
Code de l'intergiciel généré	23	91
Support d'exécution du compilateur	421	448
Empreinte mémoire de l'exécutable	457	560

TABLE 10.3 – Empreinte mémoire des binaires (en kilo-octet) des nœuds raffinés

10.5 Comparaisons avec l'approche OCARINA

Dans cette section, nous présentons une comparaison avec l'approche OCARINA afin d'illustrer la non régression et les améliorations de notre étude. Pour cette comparaison, nous nous concentrons sur le nœud `Workload_Manager`.

10.5.1 Analyses du modèle AADL

OCARINA supporte le langage REAL pour la vérification de contraintes architecturales sur les modèles AADL. Les analyses architecturales Ravenscar peuvent ainsi être réalisées à l'aide de théorème REAL. En revanche, aucune analyse sur les spécifications comportementales ne peuvent être effectuées car ni OCARINA ni REAL ne supporte l'annexe comportementale. Notre approche intégrant les automates de cette annexe facilite, quant à elle, la traduction du comportement de l'application vers un formalisme dédié à ce type d'analyse.

De plus, par rapport à l'approche OCARINA, nos analyses (Ravenscar, ordonnancement...) portent sur l'ensemble des composants applicatifs et intergiciels du système et sont réalisées sur le modèle AADL complet. OCARINA n'intègre les composants intergiciels que lors de la compilation de l'exécutable final. Par conséquent, l'analyse d'ordonnancement du système sur le modèle AADL n'est pas pertinente et n'est pas réalisable de manière automatique sur le code Ada. Il faudrait la faire manuellement.

10.5.2 Code généré

Pour les comparaisons sur le code généré par les deux approches, nous utilisons la partie dorsale d'OCARINA dédiée à la génération et à la configuration des composants de l'intergiciel POLYORB-HI-Ada. Nous avons choisi de compiler et de déployer le nœud `Workload_Manager` sur une architecture LEON/ERC32 plus pertinente dans le cas des applications TR^{2}E critiques (l'outil TSIM-ERC32 est utilisé pour simuler son exécution).

Le tableau 10.4 donne la taille des différents fichiers sources qui forment l'application. Nous remarquons que le code source utilisateur reste minime vis-à-vis du code de l'intergiciel. La différence du nombre de ligne entre les deux versions du code de l'utilisateur, vient de l'utilisation de la structure de donnée fournie par le service `AADL_Runtime` de notre approche.

Enfin, nous constatons que notre approche génère deux fois plus de lignes de code soit 90% de l'application. Ceci est du à notre processus de raffinement intégrant la modélisation d'un plus grand nombre de composants (incluant la totalité des composants intergiciels). Excepté les paquetages *utiles* de notre intergiciel et le code de l'utilisateur (implantant la partie

fonctionnelle des tâches), nous générons la totalité des composants applicatifs et intergiciels de l'application.

	Exemple OCARINA	**Exemple AADL-HI**
Code de l'utilisateur	188	200
Code de l'intergiciel minimal	2285	304
Code de l'intergiciel généré	2067	4289
Total	4540	4793

TABLE 10.4 – Taille des fichiers sources (en ligne de code) du nœud `Workload_Manager`

Le tableau 10.5 donne la taille des binaires mesurés à l'aide de l'outil `erc32-elf-size` fourni par le compilateur. Nous notons que nous avons réduit par deux (environ) l'empreinte mémoire de l'intergiciel minimal et que nous avons réduit de **10%** l'empreinte mémoire de l'exécutable final. La différence se situe au niveau de la génération des files d'attente globale de messages où nous tirons partie des bénéfices de la génération complète de la structure au lieu d'avoir recours à l'instanciation d'un générique Ada (paquetage `PolyORB_HI.Thread_Interrogators` de POLYORB-HI-Ada) implantant le comportement de la file.

	Exemple OCARINA	**Exemple AADL-HI**
Code de l'utilisateur	4	4
Code de l'intergiciel minimal	11	6
Code de l'intergiciel généré	577	509
Empreinte mémoire de l'exécutable	732	660

TABLE 10.5 – Empreinte mémoire des binaires (en kilo-octet) du nœud `Workload_Manager`

Enfin, par rapport à l'approche OCARINA implantée comme un compilateur, notre approche basée sur les techniques de transformation de modèles nous a permis d'expliciter l'ensemble des règles de transformation pour l'analyse et la génération. Ceci facilite la maintenance et l'évolution de notre approche et de nos outils.

10.6 Synthèse

Ce chapitre a présenté la mise en application de notre méthode pour l'analyse et l'implantation automatisées des systèmes critiques. Nous avons illustré celle-ci à l'aide d'un cas d'étude exprimant les caractéristiques des systèmes TR^{2}E critiques et déjà usité dans le contexte d'une modélisation architecturale AADL.

Ainsi, après une première analyse préliminaire du cas d'étude originel, nous avons pu appliquer notre processus de raffinement et mener des analyses statiques sur le modèle raffiné complet de l'application (applicatif et exécutif). Notre processus automatique d'analyse nous a permis de détecter une anomalie de conception du système (anomalie potentielle du fait des analyses pessimistes effectuées par les outils). La correction de cette anomalie à partir des spécifications initiales de l'utilisateur et la réutilisation de notre processus nous a permis d'obtenir un modèle complet et de valider le système. A partir de ce dernier, nous avons généré le code Ada de l'application respectant le profil Ravenscar. Puis, nous avons compilé, exécuté et analysé le code généré avec succés.

Ces différents résultats valident notre approche, le comportement et la sémantique de notre modèle raffiné analysable très proches du code généré.

© 2012 Gilles LASNIER

Les analyses et les mesures que nous avons relevées (ordonnancement, tailles du code source, empreinte mémoire, etc) nous ont permis d'effectuer une comparaison avec l'approche OCARINA. Par rapport à celle-ci, les analyses réalisées portent sur un modèle AADL complet du système et s'avèrent plus pertinentes vis-à-vis de l'implantation finale du système (exemple de l'analyse d'ordonnancement).

Nous générons la quasi totalité des composants applicatifs et exécutifs de l'application à partir de ce même modèle. Ceci renforce la cohérence entre le modèle d'analyse et le code généré et les possibilités de traçabilité. De plus, nous avons réduit de moitié la taille de l'intergiciel développé dans le langage de programmation cible et d'environ 10% l'empreinte mémoire de l'exécutable final par rapport à l'approche OCARINA.

Aussi, l'intégration des spécifications comportementales au sein de notre approche facilite la traduction des composants du système vers un formalisme dédié à l'analyse de son comportement. Enfin, l'implantantion de notre prototype à l'aide des techniques de transformation de modèles nous a permis d'expliciter les règles de raffinement et de génération de notre processus. Ce dernier point rend notre approche maintenable et facilite son évolution.

L'ensemble de ces résultats nous amènent à conclure que nous avons réalisé l'ensemble des objectifs que nous nous étions fixés, à savoir fournir un processus automatisé d'analyse et d'implantation renforçant la cohérence entre le modèle analysé et le codé généré du système critique.

© 2012 Gilles LASNIER

© 2012 Gilles LASNIER

Cinquième partie

Conclusion et Perspectives

Chapitre 11

Conclusions et Perspectives

SOMMAIRE

Dans ce mémoire, nous avons présenté la problématique de notre travail de recherche : renforcer la cohérence entre les modèles d'analyse et le code généré pour améliorer la fiabilité du processus d'analyse et de production des systèmes TR^{2}E critiques. Puis, nous avons proposé et implanté des solutions pour répondre à notre problématique. Ce chapitre rappelle l'ensemble des contributions réalisées et présente nos conclusions. Ensuite, nous discutons des perspectives pour étendre notre travail de recherche.

11.1 Rappel des contributions et résultats

Pour répondre à notre problématique principale, nous avons proposé un processus automatisé, flexible et évolutif, pour l'analyse et la génération de code des systèmes TR^{2}E critiques. Celui-ci est basé sur le raffinement d'une description architecturale AADL d'un système critique menant deux objectifs majeurs.

Premièrement, il réduit le niveau d'abstraction des composants afin de limiter les différences sémantique entre les composants modélisés et les constructions du système généré. Ceci est possible en supprimant les éléments de modélisation non concrets pour l'analyse et la génération, en réécrivant les composants avec une forte abstraction en un assemblage de

composants exprimant une sémantique proche d'un langage de programmation et en explicitant les aspects comportementaux des éléments architecturaux.

Deuxièmement, le niveau d'abstraction que nous proposons permet l'intégration des ressources intergicielles requises par l'application lors de la phase de modélisation. Ainsi, les analyses sont effectuées sur un modèle architectural et comportemental définissant une représentation pertinente de l'application intégrant sa partie applicative mais aussi sa partie exécutive.

Pour permettre une telle démarche, nous avons présenté une contribution scientifique : une méthode de raffinement reposant sur trois contributions techniques. Nous résumons ces éléments dans les sections suivantes.

11.1.1 Méthode pour le raffinement, l'analyse et la production

La méthode de raffinement que nous proposons produit à partir d'une description architecturale AADL initiale, un modèle de code complet et analysable de l'application. Notre méthode automatise les étapes de raffinement, d'analyse et de production du code source en proposant de :

1. Raffiner le niveau sémantique d'une description architecturale AADL ;
2. Expliciter le comportement et autoriser l'intégration des ressources intergicielles par un processus de sélection, de génération et de personnalisation des composants ;
3. Analyser le modèle complet obtenu ;
4. Produire l'implantation finale du système.

La mise en œuvre de cette méthode s'appuie sur des techniques qui ont nécessité la définition d'un langage intermédiaire (un profil de modélisation pour la génération de code préservant l'analyse) et d'une bibliothèque de composants modélisant une architecture d'intergicielle dédiée aux systèmes critiques. Nous rappelons ces éléments dans les sous-sections suivantes ainsi que l'architecture de notre prototype implantant notre méthode.

11.1.2 Le langage intermédiaire AADL-HI Ravenscar

Nous avons choisi le langage AADLv2 et son annexe comportementale pour décrire une application TR^{2}E. Notre méthode s'appuie sur un sous-ensemble des éléments de ces langages pour définir le langage intermédiaire AADL-HI Ravenscar : un profil de modélisation intriquant analyse et génération de code. Ce langage intermédiaire définit un niveau d'abstraction pour la modélisation spécifiant à la fois des détails pertinents pour différentes analyses de l'application et l'ensemble des informations nécessaires à l'implantation du système. Celui-ci n'a pas vocation de remplacer les formalismes tiers utilisés par les outils d'analyse mais de limiter les hypothèses et les transformations nécessaires pour la traduction des composants du système vers ces formalismes.

AADL-HI Ravenscar permet ainsi de modéliser des composants architecturaux et d'expliciter leur comportement. Ces composants et leur assemblage sont analysables statiquement (présenté chapitre 5) et autorise une démarche de production automatique de leur code source dans un langage de programmation impératif comme le langage Ada ou le langage C.

Pour renforcer l'analyse statique, nous avons choisi de respecter le profil Ravenscar et de respecter des restrictions spécifiques aux systèmes critiques (définies chapitre 4). De plus, le niveau sémantique proposé par notre langage est volontairement proche d'un langage de

© 2012 Gilles LASNIER

programmation impératif. Ceci nous permet de simplifier la complexité du processus de génération (élimination des hypothèses sur la runtime, traduction explicite et bi-directionnelle entre composants AADL et constructions du langage) et, par conséquent, de renforcer la cohérence entre le modèle analysé et le code généré.

Nous avons réutilisé ce langage intermédiaire pour modéliser la bibliothèque de composants intergiciels PolyORB-HI-AADL à partir d'une implantation concrète d'un intergiciel dédié aux systèmes critiques. Nous rappelons l'architecture de cette bibliothèque dans la sous-section suivante.

11.1.3 Modélisation d'une architecture intergicielle

L'une des solutions que nous avons proposée consiste à définir une architecture d'intergiciel sous la forme de composants architecturaux et comportementaux, intégrés par raffinement incrémental au sein de la description architecturale AADL du système.

Le pouvoir d'expression de notre profil AADL-HI Ravenscar nous a permis de raffiner et de modéliser les ressources intergicielles à partir de l'implantation d'un intergiciel dédié aux systèmes critiques, PolyORB-HI. La majeure partie de ces composants intergiciels est fortement personnalisable et est produite automatiquement à partir de l'analyse de la description architecturale AADL initiale du système. L'autre partie constitue une bibliothèque de composants pré-existants qui sont sélectionnés puis configurés en fonction des besoins de l'application.

Le chapitre 6 décrit PolyORB-HI-AADL, la bibliothèque de composants modélisant l'architecture intergicielle que nous avons réalisée durant ce travail de thèse. Cette contribution nous permet ainsi de compléter la description architecturale et d'effectuer les analyses sur une modélisation complète et pertinente du système critique.

11.1.4 Implantation du processus de raffinement, d'analyse et de production

Pour mettre en œuvre notre méthode, nous avons implanté un processus automatisé pour le raffinement, l'analyse et la production de systèmes TR^{2}E critiques, basé sur les techniques de transformation de modèles.

Processus de raffinement incrémental

Nous avons proposé un premier processus automatique pour le raffinement incrémental de la spécification initiale vers un modèle de code complet et analysable du système. Ce processus, décrit dans le chapitre 7, raffine un ensemble de composants applicatifs et intégre les composants intergiciels par génération ou sélection à partir de notre bibliothèque PolyORB-HI-AADL. Les composants applicatifs et intergiciels sont déployés et configurés en fonction des besoins de l'application.

Notre processus de raffinement est implanté sous la forme d'une chaîne de transformation. Celle-ci est décomposée en un ensemble de règles de transformation endogènes (voir chapitre 7) assurant la préservation des propriétés du système. L'expressivité de nos règles de transformation et l'architecture de l'outil de transformation que nous avons développée nous amènent une plus grande flexbilité et facilitent la maintenance et l'évolution de cet outil.

© 2012 Gilles Lasnier

Analyses automatisées

Nous avons implanté un analyseur pour automatiser et vérifier, à partir de notre modèle complet raffiné, le respect des différentes restrictions que nous avons définies et la présence des informations requises pour la génération de code.

Ces restrictions sont implantées dans notre analyseur sous la forme de règles de vérification de contraintes interrogeant le modèle raffiné. Cette architecture amène une plus grande flexibilité. Il est ainsi facile de maintenir et de faire évoluer notre analyseur par l'ajout de règles de vérification supplémentaires. Les résultats des règles de vérification sont remontés automatiquement à l'utilisateur, à chaque étape d'analyse, sous la forme de rapport.

L'architecture de notre analyseur permet aussi l'intégration d'outils d'analyse tiers. Il est ainsi possible d'intégrer une transformation de modèle pour traduire les composants de notre modèle raffiné vers un formalisme spécifique. Nous avons ainsi pu automatiser l'analyse d'ordonnancement du système réalisée par l'outil CHEDDAR.

Processus de production automatique simplifié

Enfin, nous avons défini un processus de production automatique. Celui-ci, décrit chapitre 8, repose sur des transformations simples des composants du profil AADL-HI Ravenscar vers des constructions équivalentes du langage de programmation Ada contraint par le profil Ravenscar. Le processus de raffinement que nous avons présenté précédemment simplifie ces règles de transformation. De plus, le mapping généraliste que nous avons défini permet aussi de traduire les composants AADL vers les constructions d'un autre langage de programmation impératif comme le langage C.

Ce processus est implanté sous la forme d'un générateur de code basé sur les techniques de transformation de modèles (modèle-à-modèle et modèle-à-code) et peut être facilement réutilisé dans une autre approche.

La suite d'outils AADL-HI

L'ensemble des outils que nous avons implanté pour démontrer la faisabilité de notre approche constitue la suite libre d'outils AADL-HI (présentée dans le chapitre 9). Celle-ci s'exécute au-dessus de la plate-forme OSATE2, référentiel AADL qui fournit l'environnement pour la modélisation et la manipulation des modèles AADLv2. Nous avons développé un compilateur supportant l'annexe comportementale AADL (pour permettre la prise en compte des descriptions comportementales au sein d'OSATE2) et un générateur de code AADLv2. Ces outils complètent notre suite d'outils AADL-HI.

La figure 11.1 rappelle notre processus automatisé pour l'analyse et la production des systèmes TR^{2}E critiques, le rôle et l'intégration des différents constituants de notre suite d'outils AADL-HI.

11.2 Conclusions

La validation de notre approche, des solutions réalisées et de leur adéquation aux problématiques identifées dans ce travail de thèse, a été effectuée à travers une étude de cas concrète. Les résultats de certaines de nos solutions nous ont aussi permis de contribuer au développement du standard de l'annexe comportementale AADL.

© 2012 Gilles LASNIER

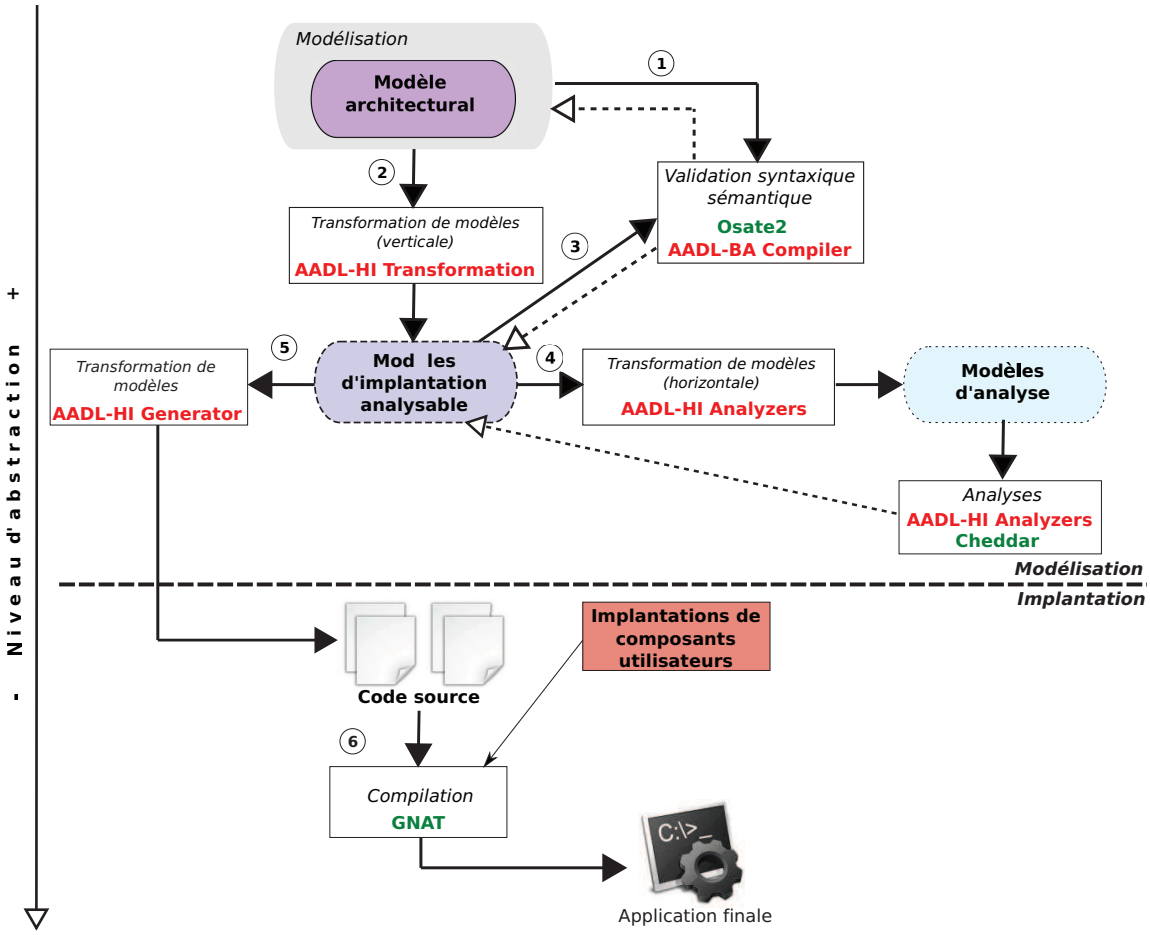


FIGURE 11.1 – Processus et outils pour la réalisation des systèmes TR^{2}E critiques

11.2.1 Expérimentation

Nous avons repris une étude de cas d'une application Ravenscar critique adaptée pour la répartition par [Zalila *et al.*, 2008]. Le chapitre 10 détaille cette étude de cas et donne les résultats des analyses, du raffinement, de la génération et de l'exécution de l'application générée. Ces résultats expérimentaux illustrent le bien-fondé de notre approche. Les analyses sont ainsi effectuées sur une représentation du système fidèle à son implantation et s'avèrent pertinentes (détection des anomalies de conception...). La cohérence entre les modèles et le code généré est par conséquent renforcée.

L'exemple que nous avons choisi nous a aussi permis de comparer nos résultats avec des travaux existants réalisés par notre équipe de recherche dans le même domaine. Par rapport à l'ancienne approche, nous avons affiné les analyses sur les modèles AADL (par la prise en compte des ressources intergicielles), simplifié la complexité du générateur de code, maximisé le nombre de composants générés vis-à-vis du code écrit à la main et réduit l'empreinte mémoire des exécutables finaux (d'environ **10%**).

11.2.2 Contributions autour d'AADL

A partir de nos contributions sur l'annexe comportementale AADL (présentées chapitre 4), l'équipe S3 de TELECOM ParisTech s'est vue confier la reponsabilité de rédiger les améliorations sur la future version de l'annexe comportementale. Le compilateur qui a été développé lors de ce travail de thèse, est devenue l'implantation officielle du support de l'annexe comportementale intégrée à la plate-forme OSATE2. Celui-ci est maintenu par notre équipe.

En conclusion, notre contribution adresse les questions sur la cohérence entre les modèles d'analyse et le code généré d'un système critique. Elle automatise le raffinement, l'analyse et la production de l'architecture et du comportement de l'application à partir de la description architecturale fournie par l'utilisateur. Les analyses s'effectuent sur une représentation pertinente et complète de l'application. Cette même représentation est utilisée pour produire l'ensemble des composants de l'application. Nous avons donc atteint notre objectif initial d'améliorer la cohérence entre modèles d'analyse et code généré pour renforcer le processus de réalisation des systèmes critiques.

11.3 Perspectives

Nous discutons maintenant des perspectives de notre contribution. La solution que nous proposons est ouverte à tous. Les outils que nous avons développés sont sous licence libre et peuvent être utilisés aussi bien par des ingénieurs que des chercheurs dans le contexte de la réalisation des systèmes critiques.

11.3.1 Modélisation et analyses

L'architecture intergicielle que nous proposons a été modélisée à partir de l'implantation Ada de PolyORB-HI, ce dernier influe sur certains choix de modélisation des composants intergiciels. Les travaux de notre équipe sur l'implantation de PolyORB-HI en C peuvent être réutilisés pour affiner et rendre plus générique la modélisation de certains services interigiciels dépendants de la plate-forme d'exécution (par exemple, le service de protocole et le service

© 2012 Gilles Lasnier

d'interrogation). Ceci permettrait d'implanter les éléments manquants pour générer l'application dans le langage C.

De plus, notre travail de recherche est focalisé sur la modélisation des systèmes TR^{2}E critiques. Nous pouvons envisager l'application de notre approche pour d'autres familles de systèmes comme les systèmes partitionnés. Des travaux de recherche au sein de notre équipe sont en cours à ce sujet.

Dans le cadre de ces travaux de thèse, nous n'avons pas eu le temps d'intégrer un autre outil d'analyse que CHEDDAR. Notamment, le pouvoir d'expression de l'annexe comportementale permettrait d'affiner les analyses comportementales. Ainsi, les travaux de [Renault, 2009] sur les patrons de modélisation comportementale pourraient servir de base et être intégrés à notre suite d'outils. Ceci autoriserait l'analyse du comportement de l'application à l'aide des réseaux de Petri.

11.3.2 Génération de code

Le prototype de générateur de code implanté propose une architecture flexible et modulaire. Nous avons fait ce choix afin de permettre l'intégration futur d'une autre partie dorsale que le langage Ada, par exemple le C.

11.3.3 Traçabilité des composants

L'approche que nous avons définie et les choix technologiques que nous avons faits favorisent et simplifient la génération d'information de traçabilité à tous les niveaux de notre processus (raffinement, analyse et production). Il serait intéressant d'étudier les différentes applications possibles de la traçabilité avant une mise en œuvre concrète. Ainsi, nous pouvons envisager : la détection de composants fautifs lors d'une analyse, la traçabilité des exigences ou encore la mise en relation des résultats sur la couverture du code généré vis-à-vis du modèle de code pour l'amélioration de la description du système (applicatif et exécutif). Ceci contribuerait à améliorer la cohérence entre modèle et code généré.

Cette section sur les perpectives s'articulant autour de notre processus de validation et de génération automatisé de systèmes TR^{2}E critiques, basé sur une démarche IDM, conclut notre manuscrit de thèse. Les travaux de recherche que nous avons présenté contribue à la thématique *"Ingénierie dirigée par les modèles pour les systèmes critiques"*.

© 2012 Gilles LASNIER

Sixième partie

Annexes

Annexe A

Paquetages de composants paramétrables de la bibliothèque PolyORB-HI-AADL

Sommaire

Cette annexe présente les paquetages de composants AADL paramétrables modélisant le service de protocole et le service d'interrogation (modélisant la file d'attente globale de messages).

A.1 Paquetage PolyORB_HI_Messages

A.2 Paquetage PolyORB_HI_Global_Queue

Exemple A.1 – Paquetage paramètrable `PolyORB_HI_Messages`

```
package PolyORB_HI_Generated_InS_Messages
public

[...]

data Message_Type
  properties
    Data_Model::Data_Representation => Struct;
end Message_Type;

data implementation Message_Type.Impl
  subcomponents
    Content : data PolyORB_HI_Streams::Stream_Element_Array[PolyORB_HI_Generated_InS_Messages_Properties::PDU_Size];
    First : data PolyORB_HI_Streams::Stream_Element_Count
                { Data_Model::Initial_Value => ("1"); };
    Last : data PolyORB_HI_Streams::Stream_Element_Count
                { Data_Model::Initial_Value => ("0"); };
end Message_Type.Impl;

[...]

subprogram Internal_To_Length extends Ada_Runtime::Unchecked_Conversion
  features
    InP : in parameter;
    OutP : out parameter;
end Internal_To_Length;

subprogram Encapsulate
  features
    Message : in parameter PolyORB_HI_Generated_InS_Messages::Message_Type.Impl;
    From : in parameter PolyORB_HI_Generated_InS_Deployment::Entity_Type;
    Entity : in parameter PolyORB_HI_Generated_InS_Deployment::Entity_Type;
    PO_HI_Return : out parameter PolyORB_HI_Streams::Stream_Element_Array;
end Encapsulate;

subprogram implementation Encapsulate.Impl
  subcomponents
    L : data PolyORB_HI_Streams::Stream_Element_Count
            { Data_Model::Initial_Value => ("Length (Message) + Header_Size");
              Access_Right => read_only; };
    R : data PolyORB_HI_Streams::Stream_Element_Count[]
            { Data_Model::Dimension_Value => ("L"); };
    Tmp_R : data PolyORB_HI_Generated_InS_Messages::Message_Size_Buffer;
    Tmp_Entity : data Base_Types::Integer_8;
    Tmp_From : data Base_Types::Integer_8;
    Tmp_Length : data PolyORB_HI_Streams::Stream_Element_Count;

  annex behavior_specification {**
  states
    s0 : initial final state;

  transitions
    t0_0 : s0 -[ ]-> s0 {
           To_Buffer!(L - 1, Tmp_R);

            R[1 .. Message_Length_Size] := PolyORB_HI_Streams::Stream_Element_Array[Tmp_R];

            PolyORB_HI_Utils::Internal_Code!(Entity, Tmp_Entity);
            PolyORB_HI_Utils::Internal_Code!(From, Tmp_From);

            R[Receiver_Offset] := PolyORB_HI_Streams::Stream_Element[Tmp_Entity];
                R[Sender_Offset] := PolyORB_HI_Streams::Stream_Element[Tmp_From];

                Length!(Message, Tmp_Length);

            for (J : PolyORB_HI_Streams::Stream_Element_Count in 1 .. Tmp_Length)
            {
                R[PolyORB_HI_Generated_InS_Messages_Properties::Header_Size + J] := Message.Content[J];
            };

            PO_HI_Return := R
    };
  **};
end Encapsulate.Impl;

subprogram Length
  features
    M : in parameter Message_Type.Impl;
    PO_HI_Return : out parameter PolyORB_HI_Streams::Stream_Element_Count;
end Length;

subprogram implementation Length.Impl
  annex behavior_specification {**
  states
    s0 : initial final state;

  transitions
    t0_0 : s0 -[ ]-> s0 { PO_HI_Return := M.Last + 1 - M.First }
  **};
end Length.Impl;
```


© 2012 Gilles LASNIER

Exemple A.2 – Paquetage paramètrable `PolyORB_HI_Messages (suite)`

```
subprogram Payload
  features
    M : in parameter Message_Type.Impl;
    PO_HI_Return : out parameter PolyORB_HI_Streams::Stream_Element_Array;
end Payload;

subprogram implementation Payload.Impl
  subcomponents
    Payload : data PolyORB_HI_Streams::Stream_Element_Array[];
                      { Data_Model::Dimension_Value => ("M.First .. M.Last"); };

  annex behavior_specification {**
  states
    s0 : initial final state;

  transitions
    t0_0 : s0 -[ ]-> s0 {
             for ( J : PolyORB_HI_Streams::Stream_Element_Count in M.First .. M.Last )
               {
                Payload[J] := M.Content[J]
           };

           PO_HI_Return := Payload
    }
  **};
end Payload.Impl;

subprogram Sender_Of_Msg
  features
    M : in parameter Message_Type.Impl;
    PO_HI_Return : out parameter PolyORB_HI_Generated_InS_Deployment::Entity_Type;
end Sender_Of_Msg;

subprogram implementation Sender_Of_Msg.Impl
  annex behavior_specification {**
  states
    s0 : initial final state;

  transitions
    t0_0 : s0 -[ ]-> s0 {
      Sender!(M.Content[M.First .. M.Last], PO_HI_Return)
    }
  **};
end Sender_Of_Msg.Impl;

subprogram Sender
  features
    M : in parameter PolyORB_HI_Streams::Stream_Element_Array;
    PO_HI_Return : out parameter PolyORB_HI_Generated_InS_Deployment::Entity_Type;
end Sender;

subprogram implementation Sender.Impl
  subcomponents
    Tmp_Integer_8 : data Base_Types::Integer_8;
    R : data PolyORB_HI_Generated_InS_Deployment::Entity_Type;
  annex behavior_specification {**
  states
    s0 : initial final state;

  transitions
    t0_0 : s0 -[ ]-> s0 {
      Interfaces::Integer_8!(M[Sender_Offset], Tmp_Integer_8);

      Corresponding_Entity!(Tmp_Integer_8, R);

      PO_HI_Return := R
    }
  **};
end Sender.Impl;

subprogram Read
  features
    Stream : in out parameter Message_Type.Impl;
    Item : out parameter PolyORB_HI_Streams::Stream_Element_Array;
    Last : out parameter PolYORB_HI_Streams::Stream_Element_Offset;
end Read;

subprogram implementation Read.Impl
  subcomponents
   L1 : data PolyORB_HI_Streams::Stream_Element_Count
              { Data_Model::Initial_Value => ("Item'Length");
                Access_Right => read_only; };
   L2 : data PolyORB_HI_Streams::Stream_Element_Count
              { Data_Model::Initial_Value => ("Length (Stream)"); };
   Tmp_L : data PolyORB_HI_Streams::Stream_Element_Count;
   Item_First : data PolyORB_HI_Streams::Stream_Element_Offset
              { Data_Model::Initial_Value => ("Item'First"); };
  annex behavior_specification {**
  states
    s0 : initial final state;

  transitions
    t0_0 : s0 -[ ]-> s0 {
      if (L1 < L2)
        L2 := L1
      end if;
      Tmp_L := L2 - 1;

      for ( J : PolyORB_HI_Streams::Stream_Element in 0 .. Tmp_L ) {
        Item[Item_First + J] := Stream.Content[Stream.First + J] };

      Last := Item_First + L2 - 1;
      Stream.First := Stream.First + L2
```

© 2012 Gilles LASNIER

Exemple A.3 – Paquetage paramètrable `PolyORB_HI_Messages (suite)`

```
subprogram Reallocate
  features
    M : in out parameter Message_Type.Impl;
end Reallocate;

subprogram implementation Reallocate.Impl
  annex behavior_specification {**
  states
    s0 : initial state;
    s1 : final state;

  transitions
    t0_0 : s0 -[ ]-> s1 {
      M.First := 1;
      M.Last  := 0
    }
  **};
end Reallocate.Impl;

subprogram Write
  features
    Stream : in out parameter Message_Type.Impl;
      Item : in parameter PolyORB_HI_Streams::Stream_Element_Array;
end Write;

subprogram implementation Write.Impl
 subcomponents
  Item_First : data PolyORB_HI_Streams::Stream_Element_Offset
               { Data_Model::Initial_Value => ("Item'First"); };
  Item_Last : data PolyORB_HI_Streams::Stream_Element_Offset
               { Data_Model::Initial_Value => ("Item'Last"); };
  Item_Length : data PolyORB_HI_Streams::Stream_Element_Offset
               { Data_Model::Initial_Value => ("Item'Length"); };
  annex behavior_specification {**
  states
    s0 : initial final state;

  transitions
    t0_0 : s0 -[ ]-> s0 {
      if (Stream.First > Stream.Last)
         Stream.First := 1;
         Stream.Last := 0
      end if;

      for (J : Base_Types::Integer in Item_First .. Item_Last)
        { Stream.Content[Stream.Last + Stream_Element_Offset[J - Item_First + 1] := Item[J]; };

      Stream.Last := Stream.Last + Item_Length
    }
  **};
end Write.Impl;

[...]

subprogram To_Buffer
  features
    L : in parameter PolyORB_HI_Streams::Stream_Element_Count;
    PO_HI_Return : out parameter Message_Size_Buffer;
end To_Buffer;

subprogram implementation To_Buffer.Impl
  subcomponents
      Tmp_L : data Base_Types::Unsigned_16;
      Tmp_Swap_Bytes : data Base_Types::Unsigned_16;
      Tmp_Wrapper : data Wrapper;
      R : data Message_Size_Buffer;

  annex behavior_specification {**
  states
    s0 : initial final state;

  transitions
    t0_0 : s0 -[ ]-> s0 {
      Ada_Runtime::Interfaces.Unsigned_16!(L, Tmp_L);
      PolyORB_HI_Utils::Swap_Bytes!(Tmp_L, Tmp_Swap_Bytes);
      Wrapper!(Tmp_Swap_Bytes, Tmp_Wrapper);
      Internal_To_Buffer!(Tmp_Wrapper, R);
      R := PO_HI_Return
      }
  **};
end To_Buffer.Impl;

end PolyORB_HI_Generated_InS_Messages;
```


© 2012 Gilles LASNIER

Exemple A.4 – Paquetage paramètrable `PolyORB_HI_Global_Queue`

```
package PolyORB_HI_Generated_InS_External_Event_Source_Gqueue
public

[...]

data Ravenscar_External_Event_Source_Port_Kinds
properties
  Data_Model::Base_Type => (classifier (Ravenscar_External_Event_Source_Port_Kind_Array));
  Data_Model::Initial_Value => ("External_Interrupt");
  Data_Model::Representation => ("PolyORB_HI_Port_Kinds.Out_Event_Data_Port");
  Access_Right => read_only;
end Ravenscar_External_Event_Source_Port_Kinds;

data Ravenscar_External_Event_Source_FIFO_Sizes
properties
  Data_Model::Base_Type => (classifier (Ravenscar_External_Event_Source_Integer_Array));
  Data_Model::Initial_Value => ("External_Interrupt");
  Data_Model::Representation => ("-1");
  Access_Right => read_only;
end Ravenscar_External_Event_Source_FIFO_Sizes;

data Ravenscar_External_Event_Source_Offsets
properties
  Data_Model::Base_Type => (classifier (Ravenscar_External_Event_Source_Integer_Array));
  Data_Model::Initial_Value => ("External_Interrupt");
  Data_Model::Representation => ("0");
  Access_Right => read_only;
end Ravenscar_External_Event_Source_Offsets;

data Ravenscar_External_Event_Source_Destinations
properties
  Data_Model::Base_Type => (classifier (Ravenscar_External_Event_Source_Integer_Array));
  Data_Model::Initial_Value => ("External_Interrupt");
  Data_Model::Representation => ("External_Event_Source_External_Interrupt_Destinations'Address");
  Access_Right => read_only;
end Ravenscar_External_Event_Source_Destinations;

------------------
-- GLOBAL_QUEUE --
------------------

data Global_Queue
  features
    -- read write
    Wait_Event             : provides subprogram access Wait_Event;
    Read_Event             : provides subprogram access Read_Event.impl;
    Dequeue                : provides subprogram access Dequeue;
    Set_Invalid            : provides subprogram access Set_Invalid;
    Store_In               : provides subprogram access Store_In.impl;
    Store_Out              : provides subprogram access Store_Out.impl;
    Set_Most_Recent_Value  : provides subprogram access Set_Most_Recent_Value.impl;

    -- read only
    Read_In                : provides subprogram access Read_In.impl;
    Read_Out               : provides subprogram access Read_Out.impl;
    Is_Invalid             : provides subprogram access Is_Invalid.impl;
    Count                  : provides subprogram access Count.impl;
    Get_Time_Stamp         : provides subprogram access Get_Time_Stamp.impl;
    Get_Most_Recent_Value  : provides subprogram access Get_Most_Recent_Value.impl;
end Global_Queue;

data implementation Global_Queue.Impl
  subcomponents
    Firsts   : data Ravenscar_External_Event_Source_Integer_Array
                 { Data_Model::Initial_Value => ("1"); } ;
                 -- means all cases with value 1
    Lasts    : data Ravenscar_External_Event_Source_Integer_Array
                 { Data_Model::Initial_Value => ("0"); } ;
    Empties  : data Boolean_Array
                 { Data_Model::Initial_Value => ("True"); } ;

    Global_Data_Queue    : data Big_Port_Stream_Array;
    Global_Data_History  : data Big_Port_Type_Array;

    GH_First : data Base_Types::Integer { Data_Model::Initial_Value => ("1"); } ;
    GH_Last  : data Base_Types::Integer { Data_Model::Initial_Value => ("0"); } ;

    Most_Recent_Values : data Port_Stream_Array;
    Time_Stamps : data Time_Array;
    Initialized : data Boolean_Array
                    { Data_Model::Initial_Value => ("False"); } ;
    Value_Put   : data Boolean_Array
                    { Data_Model::Initial_Value => ("False"); } ;
    Not_Empty   : data Base_Types::Boolean
                    { Data_Model::Initial_Value => ("False"); } ;
    N_Empties   : data Base_Types::Integer
                    { Data_Model::Initial_Value => ("N_Ports"); } ;
    properties
         Priority => 240;
         Concurrency_Control_Protocol => Priority_Ceiling;
end Global_Queue.Impl;
```

© 2012 Gilles LASNIER

Exemple A.5 – Paquetage paramètrable `PolyORB_HI_Global_Queue (suite)`

```
-----------------
-- SUBPROGRAMS --
-----------------

subprogram Wait_For_Incoming_Events
  features
   A_Port : out parameter PolyORB_HI_Generated_InS_External_Event_Source_Activity::Ravenscar_External_Event_Source_Port_Type;
   GQ     : requires data access Global_Queue.Impl;
end Wait_For_Incoming_Events;

subprogram implementation Wait_For_Incoming_Events.Impl
  annex behavior_specification {**
    states
      s0 : initial final state;
    transitions
      t0_0 : s0 -[]-> s0 { GQ.Wait_Event!(A_Port) };
  **};
end Wait_For_Incoming_Events.Impl;

subprogram Read_Event
  features
   P      : out parameter PolyORB_HI_Generated_InS_External_Event_Source_Activity::Ravenscar_External_Event_Source_Port_Type;
   Valid  : out parameter Base_Types::Boolean;
   GQ     : requires data access Global_Queue.Impl;
end Read_Event;

subprogram implementation Read_Event.Impl
  annex behavior_specification {**
    states
      s0 : initial final state;
    transitions
      t0_0 : s0 -[]-> s0 { Valid := GQ.Not_Empty;
                           if ( Valid = true ) P := GQ.Global_Data_History[GQ.GH_First] end if
      }
  **};
end Read_Event.Impl;

subprogram Dequeue
  features
   T  : in  parameter PolyORB_HI_Generated_InS_External_Event_Source_Activity::Ravenscar_External_Event_Source_Port_Type;
   P  : out parameter Port_Stream_Entry.Impl;
   GQ : requires data access Global_Queue.Impl;
end Dequeue;

subprogram implementation Dequeue.Impl
  subcomponents
   Is_Event_B : data Base_Types::Boolean
     { Data_Model::Initial_Value => ("Is_Event_B"); };
  annex behavior_specification {**
    states
      s0 : initial final state;
    transitions
      t0_0 : s0 -[]-> s0 {
        PolyORB_HI_Port_Kinds.Is_Event!(Ravenscar_External_Event_Source_Port_Kinds[T], Is_Event_B);
        if ( GQ.Empties[T] )
           GQ.Get_Most_Revent_Value (T, P);
        elsif ( Ravenscar_External_Event_Source_Fifo_Sizes[T] )
           GQ.Get_Most_Recent_Value!(T, P)
        else
           if ( GQ.Firsts[T] = GQ.Lasts[T] )
              if ( not QG.Empties[T] and Is_Event_B )
                 GQ.N_Empties := GQ.N_Empties + 1
              end if;

              GQ.Empties[T] := True  --  Is_Empty := true
           end if;

           P := GQ.Global_Data_Queue[Firsts[T] + Ravenscar_External_Event_Source_Offsets[T] - 1];

           if ( GQ.Firsts[T] = Ravenscar_External_Event_Source_Fifo_Sizes[T] )
              GQ.Firsts[T] := 1
           elsif ( Fifo_Size > 1 )
              GQ.Firsts[T] := Firsts[T] + 1
           end if;

           GQ.H_Increment_First!(GQ.GH_First)
        end if;

        if ( GQ.N_Empties < GQ.N_Ports )
           GQ.Not_Empty := True
        else
           GQ.Not_Empty := False
        end if
      };
  **};
end Dequeue.Impl;

[...]

end PolyORB_HI_Generated_InS_External_Event_Source_Gqueue;
```


© 2012 Gilles LASNIER

Annexe B

Interrogateurs du modèle AADL

SOMMAIRE

Nous donnons dans cette annexe un récapitulatif des interrogateurs du modèle AADL que nous avons développé dans le cadre de ces travaux de thèse. Nous donnons aussi les algorithmes d'une partie de ces interrogateurs implantés sous la forme d'*helpers* ATL dont la syntaxe et la sémantique sont basées sur un sous-ensemble du langage de spécification de contraintes OCL.

B.1 Tableau récapitulatif des principaux interrogateurs AADL

B.2 Exemples d'implantation d'interrogateur en ATL

Exemple B.6 – ATL helpers Get_Connected_Entities

```
helper context AADL2!ProcessSubcomponent def : getConnectedEntities() : Sequence(String) =
    -- Get all threads connected as source to destination node connection
    let srcThreads : Sequence(String) =
        self.namespace.ownedPortConnection->select(i | i.sourceContext.name = self.name)
                                          ->collect(p | p.getSrcThreadConnected())
                                          ->flatten()
    in -- Get all threads connected as destination to destination node connection
    let dstThreads : Sequence(String) =
        srcThreads.union(self.namespace.ownedPortConnection
                                          ->select(i | i.sourceContext.name = self.name)
                                          ->collect(p | p.getDstThreadConnected())
                                          ->flatten())
    in -- Get all threads connected as destination to source node connection
    let srcNodes : Sequence(String) =
        dstThreads.union(self.namespace.ownedPortConnection
                                          ->select(i | i.destinationContext.name = self.name)
                                          ->collect(p | p.getSrcThreadConnected())
                                          ->flatten())
    in -- Get all threads connected as source to source node connection
    srcNodes.union(self.namespace.ownedPortConnection
                                          ->select(i | i.destinationContext.name = self.name)
                                          ->collect(p | p.getDstThreadConnected())
                                          ->flatten())
                                          ->asOrderedSet()->sortedBy(p | p.toString())
;
```

Get_Connected_Nodes(<system>, <process>) contexte : ∅ entrée : *système* S, *sous-composant process* P | sortie : *liste_noeuds*<>>br>Fonction : Renvoi la liste des sous-composants processus (nœuds) d'un système S connectés au nœud P
Get_Clause_Representation(list_elements<>) contexte : ∅ entrée : *liste_elements*<> L | sortie : *liste_entiers*<> Fonction : Renvoi une liste de clause de représentation (des entiers distincts) de la taille la liste L
Get_Connected_Entities(<system>, <process>) contexte : ∅ entrée : *système* S, *sous-composant process* P | sortie : *liste_threads*<> Fonction : Renvoi la liste des sous-composants threads (entités) connectés au nœud P (y compris les siens) appartenant au système S
Get_Element(<package>, <component_name>) contexte : ∅ entrée : *package* P, *string* S | sortie : *composant* C Fonction : Renvoi le composant de nom S localisé dans le paquetage P
Get_Port_Information(<thread>) contexte : ∅ entrée : *sous-composant thread*<> T | sortie : *port_information*<> Fonction : Renvoi la structure de donnée *port_information*<> contenant l'ensemble des informations (direction, type, propriétés...) des ports du thread T
Get_Port_Name(<port_information<>>) contexte : ∅ entrée : *port_information*<> PI | sortie : *liste_string*<> Fonction : Renvoi la liste des noms des ports contenus dans la strucure PI
Get_Port_Queue_Size(<port_Information<>>) contexte : ∅ entrée : *port_information*<> PI | sortie : *liste_string*<> Fonction : Renvoi la liste des valeurs de la propriété AADL `Queue_Size` des ports contenus dans la strucure PI
Get_Deployment_Port_Number_Property(<process>) contexte : ∅ entrée : *sous-composant process* P | sortie : *string* Fonction : Renvoi la valeur de la propriété AADL `Deployment::Port_Number` attachée au nœud P
Get_Deployment_Location_Property(<process>) contexte : ∅ entrée : *sous-composant process* P | sortie : *string* Fonction : Renvoi la valeur de la propriété AADL `Deployment::Location` attachée au nœud P
<thread>.Get_Initialize_EntryPoint() contexte : *sous-composant thread* T entrée : ∅ | sortie : *string* Fonction : Renvoi la valeur de la propriété AADL `Initialize_Entrypoint` attachée au thread T
<data>.get_Data_SubComponents() contexte : *composant data* D entrée : ∅ | sortie : *liste_sous-composants*<> Fonction : Renvoi la liste des sous-composants de donnée du composant D
<data>.get_Spg_Subcomponents() contexte : *composant data* D entrée : ∅ | sortie : *liste_sous-composants*<> Fonction : Renvoi la liste des sous-composants sous-programme du composant D

TABLE B.1 – Principaux interrogateurs du modèle AADL

© 2012 Gilles LASNIER

<data>.get_Accessors() contexte : *composant data* D entrée : ∅ | sortie : *liste_accesseurs<>* Fonction : Renvoi la liste des accesseurs spécifiés par le composant de donnée D
<data>.get_Initial_Value() contexte : *composant data* D entrée : ∅ | sortie : *string* Fonction : Renvoi la valeur de la propriété AADL `Data_Model::Initial_Value` attachée au composant D
<data>.get_Data_Size() contexte : *composant data* D entrée : ∅ | sortie : *string* Fonction : Renvoi la valeur de la propriété AADL `Source_Data_Size` attachée au composant D
<data>.get_Number_Representation() contexte : *composant data* D entrée : ∅ | sortie : *string* Fonction : Renvoi la valeur de la propriété AADL `Data_Model::Number_Representation` attachée à D
<data>.get_Type_Of_Ada_Integer(<string>dataSize, <string>dataSign) contexte : *composant data* D entrée : *string* dataSize, *string* dataSign | sortie : *string* Fonction : Renvoi le type entier correspondant à la taille et au signe spécifiés pour composant D
<data>.get_Dimension() contexte : *composant data* D entrée : ∅ | sortie : *string* Fonction : Renvoi la valeur de la propriété AADL `Data_Model::Dimension` attachée au composant D
<data>.get_Enumeration() contexte : *composant data* D entrée : ∅ | sortie : *string* Fonction : Renvoi la valeur de la propriété AADL `Data_Model::Enumeration` attachée au composant D
<data>.get_Representation() contexte : *composant data* D entrée : ∅ | sortie : *string* Fonction : Renvoi la valeur de la propriété AADL `Data_Model::Representation` attachée au composant D
<data>.get_Data_Subcomponent_Names() contexte : *composant data* D entrée : ∅ | sortie : *liste_string<>* Fonction : Renvoi la liste des noms des sous-composants du composant D
<data>.get_Data_Subcomponent_Types() contexte : *composant data* D entrée : ∅ | sortie : *liste_composants<>* Fonction : Renvoi la liste des types (composants) des sous-composants de donnée du composant D
<data>.get_Element_Names() contexte : *composant data* D entrée : ∅ | sortie : *string<>* Fonction : Renvoi la liste des noms d'énumération spécifiée par la propriété AADL `Data_Model::Element_Names` attachée au composant de donnée D
<data>.get_Element_Base_Types() contexte : *composant data* D entrée : ∅ | sortie : *string<>* Fonction : Renvoi la liste des composants spécifiée par la propriété AADL `Data_Model::Base_Type` attachée au composant de donnée D
<data>.get_Access_Right() contexte : *composant data* D entrée : ∅ | sortie : *string* Fonction : Renvoi la valeur de la propriété AADL `Access_Right` attachée au composant de donnée D

TABLE B.2 – Principaux interrogateurs du modèle AADL (suite)

© 2012 Gilles LASNIER

<data_or_thread>.get_Priority() contexte : *composant data* ou *composant thread* C entrée : ∅ | sortie : *string*<> Fonction : Renvoi la valeur de la propriété AADL `Priority` attachée au composant de donnée D
<subprogram>.get_Source_Name() contexte : *composant subprogram* S entrée : ∅ | sortie : *string*<> Fonction : Renvoi la valeur de la propriété AADL `Source_Name` attachée au composant S
<subprogram>.get_Feature_Parameters() contexte : *composant subprogram* S entrée : ∅ | sortie : *parameters*<> Fonction : Renvoi la liste des éléments d'interface `parameter` spécifié dans le composant S
<subprogram>.get_Source_Langage() contexte : *composant subprogram* S entrée : ∅ | sortie : *string*<> Fonction : Renvoi la valeur de la propriété AADL `Source_Langage` attachée au composant S
<subprogram>.get_Behavior_Annex() contexte : *composant subprogram* S entrée : ∅ | sortie : *behavior_specification* Fonction : Renvoi la description comportementale spécifiée pour le composant S
<thread>.get_Activate_Entrypoint() contexte : *composant thread* T entrée : ∅ | sortie : *string* Fonction : Renvoi la valeur de la propriété AADL `Activate_Entrypoint` attachée au composant T
<thread>.get_Period_In_Milliseconds() contexte : *composant thread* T entrée : ∅ | sortie : *string* Fonction : Renvoi la valeur de la propriété AADL `Period` attachée au composant T
<thread>.get_Deadline_In_Milliseconds() contexte : *composant thread* T entrée : ∅ | sortie : *string* Fonction : Renvoi la valeur de la propriété AADL `Deadline` attachée au composant T
<thread>.get_Stack_Size() contexte : *composant thread* T entrée : ∅ | sortie : *string* Fonction : Renvoi la valeur de la propriété AADL `Source_Stack_Size` attachée au composant T
<behavior_specification>.Get_Dispatch_Transition_Actions() contexte : *behavior_specification* behaviorAnnex entrée : ∅ | sortie : *behavior_specification* Fonction : Renvoi la liste des actions spécifiées dans l'automate comportemental behaviorAnnex

TABLE B.3 – Récapitulatif des interrogateurs du modèle AADL (suite)

© 2012 Gilles LASNIER

Exemple B.7 – ATL helpers Get_Port_Information

```
helper context AADL2!ThreadSubcomponent def : getPortInformation()
: Sequence(TupleType(portCpt : AAD2!Port,
                     thParent : AADL2!ComponentClassifier,
                     portName : String,
                     queueSize : String,
                     overflowProcotol : String,
                     urgency : String,
                     computeEntrypoint : String,
                     destinations : Sequence(TupleType(portDst : AADL2!Port,
                                                       thSubOwner : AADL2!ComponentClassifier))))
=
  if self.classifier.oclIsKindOf(AADL2!ThreadImplementation) then
      self.classifier.type.ownedFeature
       ->select(e | e.oclIsKindOf(AADL2!Port))
       ->collect(p | Tuple{portCpt = p, thParent = self, portName = p.name,
                           queueSize = p.getQueueSize(),
                           overflowProtocol = p.getOverflowHandlingProtocol(),
                           urgency = p.getUrgency(),
                           destinations = p.getPortDestinations(self)->flatten(),
                           computeEntrypoint = p.getComputeEntrypointFromPort()})
  else
      self.classifier.ownedFeature
      ->select(e | e.oclIsKindOf(AADL2!Port))
      ->collect(p | Tuple{portCpt = p, thParent = self, portName = p.name,
                          queueSize = p.getQueueSize(),
                          overflowProtocol = p.getOverflowHandlingProtocol(),
                          urgency = p.getUrgency(),
                          destinations = p.getPortDestinations(self)->flatten(),
                          computeEntrypoint = p.getComputeEntrypointFromPort()})
  endif
;
```

Exemple B.8 – ATL helpers Get_All_Data_Properites

```
helper context AADLBA!ComponentType def : getAllDataProperties()
: Sequence(AADLBA!PropertyAssociation) =
    let propAssocs : Sequence(AADLBA!PropertyAssociation) = self.ownedPropertyAssociation
    in
        if not self.ownedExtension.oclIsUndefined() then
              propAssocs.append(self.ownedExtension.extended.getAllDataProperties()
               ->select(e | let propAssocNames : Sequence(String) = propAssocs
               ->collect(p | p.property.qualifiedName)
                  in not propAssocNames.includes(e.property.qualifiedName)))->flatten()
        else
            propAssocs
        endif
;

helper context AADLBA!ComponentImplementation def : getAllDataProperties()
: Sequence(AADLBA!PropertyAssociation) =
    let propAssocs : Sequence(AADLBA!PropertyAssociation) = self.ownedPropertyAssociation
    in
        let propAssocs2 : Sequence(AADLBA!PropertyAssociation) =
            if not self.type.oclIsUndefined() then
                  propAssocs.append(self.type.getAllDataProperties()
                   ->select(e | let propAssocNames : Sequence(String) = propAssocs
                   ->collect(p | p.property.qualifiedName)
                      in not propAssocNames.includes(e.property.qualifiedName)))->flatten()
            else
                propAssocs
            endif
        in
            if not self.ownedExtension.oclIsUndefined() then
                  propAssocs2.append(self.ownedExtension.extended.getAllDataProperties()
                  ->select(e | let propAssocNames : Sequence(String) = propAssocs2
                  ->collect(p | p.property.qualifiedName)
                     in not propAssocNames.includes(e.property.qualifiedName)))->flatten()
            else
                propAssocs2
            endif
;
```

© 2012 Gilles LASNIER

© 2012 Gilles LASNIER

Annexe C

Éléments pour la génération vers les constructions du langage Ada

SOMMAIRE

Cette annexe présente la syntaxe abstraite des constructions du langage Ada que nous avons retenues pour notre approche de génération de code. Le sous-ensemble des constructions Ada que nous supportons respecte les recommandations du prodil architectural Ravenscar pour l'implantation de systèmes critiques.

C.1 BNF des constructions principales

C.2 BNF des constructions liées au sous-programme

C.3 BNF des constructions liées au support des tâches

C.4 BNF des constructions liées aux objets protégés

C.5 Règles de transformations des tâches périodiques

C.6 Règles de transformation des tâches sporadiques

Exemple C.9 – BNF des constructions principales du langage Ada

```
full_type_declaration ::=
      ( "type" defining_identifier [ known_discriminant_part ]
      "is" type_definition ";" )
      | task_type_declaration | protected_type_declaration

type_definition ::=
      enumeration_type_definition | integer_type_definition
      | real_type_definition | array_type_definition
      | record_type_definition | access_type_definition
      | derived_type_definition

enumeration_type_definition ::=
      "(" enumeration_literal_specification
      { "," enumeration_literal_specification } ")"

integer_type_definition ::=
      ( "range" simple_expression ".." simple_expression )
      | ( "mod" expression )

array_type_definition ::=
      unconstrained_array_definition | constrained_array_definition

constrained_array_definition ::=
      "array" "(" discrete_subtype_definition
      { "," discrete_subtype_definition } ")" "of" component_definition
```

Exemple C.10 – BNF des constructions principales du langage Ada (suite)

```
record_type_definition ::= [ [ "abstract" ] "tagged" ]
      [ "limited" ] record_definition

record_definition ::=
      ( "record" (
      ( component_item { component_item } )
      | ( { component_item } variant_part )
      | ( "null" ";" ) )
      "end" "record" )
      | ( "null" "record" )

derived_type_definition ::= [ "abstract" ]
      "new" subtype_indication [ record_extension_part ]

representation_clause ::=
      attribute_definition_clause
      | enumeration_representation_clause
      | record_representation_clause
      | at_clause

enumeration_representation_clause ::=
      "for" first_subtype_local_name "use" enumeration_aggregate ";"

record_representation_clause ::=
      "for" first_subtype_local_name "use"
      "record" [ mod_clause ]
      { component_clause }
      "end" "record" ";"

component_declaration ::=
      defining_identifier_list ":" component_definition
      [ ":=" default_expression ] ";"

object_declaration ::=
      ( defining_identifier_list ":" [ "aliased" ] [ "constant" ]
      subtype_indication [ ":=" expression ] ";" )
      | ( defining_identifier_list ":" [ "aliased" ] [ "constant" ]
      array_type_definition [ ":=" expression ] ";" )
      | single_task_declaration
      | single_protected_declaration

if_statement ::=
      "if" condition "then"
      sequence_of_statements
      { "elsif" condition "then"
      sequence_of_statements }
      [ "else"
      sequence_of_statements ]
      "end" "if" ";"

loop_statement ::= [ statement_identifier ":" ]
      [ ( "while" condition ) | ( "for"
      defining_identifier "in" [ "reverse" ] discrete_subtype_definition ) ]
      "loop"
      sequence_of_statements
      "end" "loop" [ statement_identifier ] ";

delay_until_statement ::= "delay" "until" expression ";"
```


© 2012 Gilles LASNIER

Exemple C.11 – BNF des sous-programmes Ada

```
subprogram_declaration ::= subprogram_specification ";"

subprogram_specification ::=
      ( "procedure" defining_program_unit_name [ formal_part ] )
      | ( "function" defining_designator [ formal_part ] "return" subtype_mark )

formal_part ::=
      "(" parameter_specification { ";" parameter_specification } ")"

subprogram_body ::=
      subprogram_specification "is"
      declarative_part
      "begin"
      handled_sequence_of_statements
      "end" [ designator ] ";"
```

Exemple C.12 – BNF des tâches Ada

```
task_type_declaration ::=
      "task" "type" defining_identifier [ known_discriminant_part ]
      [ "is" task_definition ] ";"

task_definition ::=
      { task_item }
      [ "private"
      { task_item } ]
      "end" [ task_identifier ]

task_item ::= entry_declaration | representation_clause

task_body ::=
      "task" "body" defining_identifier "is"
      declarative_part
      "begin"
      handled_sequence_of_statements
      "end" [ task_identifier ] ";"
```

Exemple C.13 – BNF des objets protégés Ada

```
protected_type_declaration ::=
      "protected" "type" defining_identifier [ known_discriminant_part ]
      "is" protected_definition ";"

protected_definition ::=
      { protected_operation_declaration }
      [ "private"
      { protected_element_declaration } ]
      "end" [ protected_identifier ]

protected_body ::=
      "protected" "body" defining_identifier "is"
      { protected_operation_item }
      "end" [ protected_identifier ] ";"

protected_element_declaration ::=
      protected_operation_declaration
      | component_declaration
```

© 2012 Gilles LASNIER

Algorithme 37: Transformation d'un composant `thread` AADL périodique en tâche périodique Ada (specification)

```
1  foreach component thread implementation T of AADL package AadlPackage |
   T.isPeriodic() and T.hasBehaviorAnnex() do
2      adaPackage ← get_Ada_Package_From_AADL_Package(AadlPackage) ;
3      if T.hasActivateEntrypoint() then
4          activateEntrypoint ← T.get_Activate_Entrypoint()
5      end
6
       - specification (ads)
7      dispatchOffsetExpr ← Gen_Expression(Gen_Subprogram_Call('To_Time_Span',
       'Ada_Real_Time'), 0.0)
8      dispatchOffset ← Gen_Object_Declaration('Dispatch_Offset',
       'Ada.Real_Time.Time_Span', dispatchOffsetExpr)
9      period ← T.get_Period_In_Milliseconds()
10     taskPeriodExpr ← Gen_Expression(Gen_Subprogram_Call('Milliseconds',
       'Ada.Real_Time'), period)
11     taskPeriod ← Gen_Object_Declaration('Task_Period',
       'Ada.Real_Time.Time_Span', taskPeriodExpr)
12     deadline ← T.get_Deadline_In_Milliseconds()
13     taskDeadlineExpr ← Gen_Expression(Gen_Subprogram_Call('Milliseconds',
       'Ada.Real_Time'), deadline)
14     taskDeadline ← Gen_Object_Declaration('Task_Deadline',
       'Ada.Real_Time.Time_Span', taskDeadlineExpr)
15     priority ← T.get_Priority()
16     taskPriority ← Gen_Object_Declaration('Task_Priority', 'System.Any_Priority',
       priority)
17     stackSize ← T.get_Stack_Size()
18     taskStackSize ← Gen_Object_Declaration('Task_Stack_Size', 'Natural', stackSize)
19     priorityPragma ← Gen_Pragma_Statement('Priority', Task_Priority)
20     storageSizePragma ← Gen_Pragma_Statement('Storage_Size', taskStackSize)
21     taskTypeDef ← Gen_Task_Type_Definition(priorityPragma, storageSizePragma)
       - Task type declaration object
22     idTaskDecl ← Map_Ada_Name(T.name)
23     taskDecl ← Gen_Task_Type_Declaration(idTaskDecl, taskTypeDef)
24 end
```


© 2012 Gilles LASNIER

Algorithme 38: Transformation d'un composant `thread` sporadique en tâche sporadique (specification)

```
1  foreach component thread implementation T of AADL package AadlPackage |
   T.isSporadic() and T.hasBehaviorAnnex() do
2      adaPackage ← get_Ada_Package_From_AADL_Package(AadlPackage)
3      if T.hasActivateEntrypoint() then
4          activateEntrypoint ← T.get_Activate_Entrypoint()
5      end
6
       - specification (ads)
7      period ← T.get_Period_In_Milliseconds()
8      taskPeriodExpr ← Gen_Expression(Gen_Subprogram_Call('Milliseconds',
       'Ada.Real_Time'), period)
9      taskPeriod ← Gen_Object_Declaration('Task_Period',
       'Ada.Real_Time.Time_Span', taskPeriodExpr)
10     deadline ← T.get_Deadline_In_Milliseconds()
11     taskDeadlineExpr ← Gen_Expression(Gen_Subprogram_Call('Milliseconds',
       'Ada.Real_Time'), deadline)
12     taskDeadline ← Gen_Object_Declaration('Task_Deadline',
       'Ada.Real_Time.Time_Span', taskDeadlineExpr)
13     priority ← T.get_Priority()
14     taskPriority ← Gen_Object_Declaration('Task_Priority', 'System.Any_Priority',
       priority)
15     stackSize ← T.get_Stack_Size()
16     taskStackSize ← Gen_Object_Declaration('Task_Stack_Size', 'Natural', stackSize)
17     priorityPragma ← Gen_Pragma_Statement('Priority', Task_Priority)
18     storageSizePragma ← Gen_Pragma_Statement('Storage_Size', taskStackSize)
19     taskTypeDef ← Gen_Task_Type_Definition(priorityPragma, storageSizePragma)
       - Task type declaration object
20     idTaskDecl ← Map_Ada_Name(T.name)
21     taskDecl ← Gen_Task_Type_Declaration(idTaskDecl, taskTypeDef)
22 end
```

© 2012 Gilles LASNIER

Algorithme 39: Transformation d'un composant `thread` sporadique en tâche sporadique (body)

foreach *component thread implementation* T *of AADL package* $AadlPackage$ | T.isSporadic() *and* T.hasBehaviorAnnex() **do**

```
    - body (adb)
2   nextDeadlineVal ← Gen_Object_Declaration('Next_Deadline_Val',
    'Ada.Ada_RealTime.Time')
3   nextStart ← Gen_Object_Declaration('Next_Start', 'Ada.Real_Time.Time')
4   portType ← Gen_Object_Declaration('Port_Type', 'XXXXX − TOFIX')
5   delayUntilExpr ← Gen_Expression('System_Startup_Time')
6   delayUntil ← Gen_Delay_Statement(true, delayUntilExpr)
7   declarations.add(nextStart)
8   declarations.add(portType)
9   dataSubCpts ← T.get_Data_Subcomponents()
10  declarations ← Gen_Declarations(dataSubCpts)
11  statements.add(Gen_Subprogram_Call(activateEntrypoint), delayUntil,
    Gen_Task_Sporadic_Loop_Statement(T.get_Behavior_Annex()))
    - Task body object
12  taskBody ← Gen_Task_Body(taskDecl, declarations, statements)
13  AdaPackage.specification.add(, taskPeriod, taskDeadline, taskPriority,
    taskStackSize, taskDecl)
14  AdaPackage.body.add(nextDeadlineVal, taskBody)
```

end

Algorithme 40: Boucle infinie du `thread` sporadique

Data : thread behavior annex $behaviorAnnex$; task body $taskBody$; Ada package $AdaPackage$

```
1  waitForIncEvents ← Gen_Subprogram_Call('Wait_For_Incoming_Events', 'Entity',
   'Port')
2  nextStartExpr ← Gen_Expression(+, 'Ada.Real_Time.Clock', 'Task_Period')
3  nextStartAssign ← Gen_Assignment_Statement('Next_Start', nextStartExpr)
4  nextDeadlineValExpr ← Gen_Expression(+, 'Ada.Real_Time.Clock', 'Task_Deadline')
5  nextDeadlineVal ← Gen_Assignment_Statement('Next_Deadline_Val',
   nextDeadlineExpr)
6  delayUntilExpr ← Gen_Expression('Next_Start')
7  delayUntil ← Gen_Delay_Statement(true, delayUntilExpr)
8  loopStatements.add(waitForIncEvents, nextStart, nextDeadlineVal)
9  loopStatements ← Gen_Ada_Statements(behaviorAnnex, taskBody, AdaPackage)
10 loopStatements.add(delayUntil)
11 loop ← Gen_Loop_Statement(loopStatements)
12 return loop
```


© 2012 Gilles LASNIER

Bibliographie

[Alizon *et al.*, 2007] F. Alizon, G. Belaunde, M. DuPre, B. Nicolas, S. Poivre, and Simonin J. Les modèles dans l'action à france télécom avec smartqvt. In *Génie Logiciel : Congrès Journées Neptune*, volume No5, 2007.

[Allen, 1997] Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, 1997.

[Alur and Dill, 1994] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2) :183–235, April 1994.

[AndroMDA, 2012] AndroMDA. Andromda. http://www.andromda.org/docs/index.html, 2012.

[ANTLR Project, 2012] ANTLR Project. Antlr parser generator, 2012.

[Armstrong, 1996] Joe Armstrong. Erlang - A survey of the language and its industrial applications. In *INAP'96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, volume pages 16–18, 1996.

[ATL, 2012] ATL. ATL - A model transformation technology. http://www.eclipse.org/atl, 2012.

[Barnes, 2008] John Barnes. *Safe and secure software : an invitation to Ada 200*. Adacore, April 2008.

[Berthomieu *et al.*, 2010] B. Berthomieu, Jean-Paul Bodeveix, Silvano Dal Zilio, Pierre Dissaux, Mamoun Filali, Pierre Gaufillet, Sebastien Heim, and F. Vernadat. Formal Verification of AADL models with Fiacre and Tina (regular paper). In *European Congress on Embedded Real-Time Software (ERTS), Toulouse*. SIA/3AF/SEE, 2010.

[Bézivin and Jouault, 2006] Jean Bézivin and Frédéric Jouault. Using atl for checking models. *Electron. Notes Theor. Comput. Sci.*, 152 :69–81, March 2006.

[Biehl, 2010] Matthias Biehl. Literature study on model transformations. Technical report, Royal Institute of Technology, jul 2010.

[Blanc, 2005] Xavier Blanc. *MDA en action*. EYROLLES, mar 2005.

[Borde *et al.*, 2009] E. Borde, G. Haik, and L. Pautet. Mode-based reconfiguration of critical software component architectures. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 1160 –1165, april 2009.

[Borde, 2009] Etienne Borde. *Configuration et Reconfiguration des Systèmes Temps-Reél Répartis Embarqués Critiques et Adaptatifs*. PhD thesis, Télécom ParisTech, dec 2009.

[Bordin and Vardanega, 2005] Matteo Bordin and Tullio Vardanega. Automated model-based generation of ravenscar-compliant source code. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, ECRTS '05, pages 59–67, Washington, DC, USA, 2005. IEEE Computer Society.

[Bordin *et al.*, 2008] M. Bordin, M. Panunzio, and T. Vardanega. Fitting schedulability analysis theory into model-driven engineering. In *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*, pages 135 –144, july 2008.

[Bray *et al.*, 2008] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml) 1.0 (fifth edition). Technical report, W3C, nov 2008.

[Brun *et al.*, 2008] M. Brun, J. Delatour, and Y. Trinquet. Code generation from aadl to a real-time operating system : An experimentation feedback on the use of model transformation. In *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on*, pages 257 –262, 31 2008-april 3 2008.

[Burns *et al.*, 2004] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems. *Ada Lett.*, XXIV(2) :1–74, June 2004.

[Cancila *et al.*, 2010] D. Cancila, R. Passerone, T. Vardanega, and M. Panunzio. Toward correctness in the specification and handling of non-functional attributes of high-integrity real-time embedded systems. *Industrial Informatics, IEEE Transactions on*, 6(2) :181 –194, may 2010.

[Combemale, 2008] Benoît Combemale. Ingénierie dirigée par les modèles (idm) état de l'art. état de l'art, Institut de Recherche en Informatique de Toulouse, aug 2008.

[Consortium, 2012] OW2 Consortium. Fractal adl. http://fractal.ow2.org/fractaladl, 2012.

[Czarnecki and Helsen, 2003] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation ap- proaches. *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[Czarnecki and Helsen, 2006] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3) :621–645, July 2006.

[Dirckze, 2002] R. Dirckze. Java metadata interface (jmi) specification, v1.0. http://java.sun.com/products/jmi, jun 2002.

[Eclipse,] Eclipse. Umlx. http://www.eclipse.org/gmt/umlx, 2012.

[Eclipse Community,] Eclipse Community. Viatra : Visual Automated model TRAnsformations. http://www.eclipse.org/gmt/VIATRA2, 2012.

[Eclipse Project, 2012] Eclipse Project. Xtext. http://www.eclipse.org/Xtext/, 2012.

[Ellidiss Technologies, 2012] Ellidiss Technologies. Aadl tools. AADL committee (presentation), apr 2012.

[Esterel Technologies, 2012] Esterel Technologies. Scade-suite. http://www.esterel-technologies.com/products/scade-suite, 2012.

[Faugere *et al.*, 2007] Madeleine Faugere, Thimothee Bourbeau, Robert de Simone, and Sebastien Gerard. Marte : Also an uml profile for modeling aadl applications. In *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, ICECCS '07, pages 359–364, Washington, DC, USA, 2007. IEEE Computer Society.

[Fleurey, 2006] Franck Fleurey. *Langage et méthode pour une ingénierie des modèles fiable*. PhD thesis, Université de Rennes 1 - Triskell - IRISA, oct 2006.

[Flex-eWare Project, 2012] Flex-eWare Project. Flex-eware project. http://www.flex-eware.org, 2012.

© 2012 Gilles LASNIER

[Fujaba Core Development Group, 2012] Fujaba Core Development Group. The fujaba tool suite. http://www.fujaba.de, 2012.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*, volume 0-201-63361-2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Garlan *et al.*, 1997] David Garlan, Robert Monroe, and David Wile. Acme : an architecture description interchange language. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '97, pages 7–. IBM Press, 1997.

[Gilles, 2008] Olivier Gilles. Real : Requirement enforcement analysis language. Technical report, TELECOM ParisTech, 2008.

[Gokhale *et al.*, 2003] Aniruddha Gokhale, Douglas Schmidt, Nanbor Wang, Tao Lu, Balachandran Natarajan, and Ran Natarajan. Cosmic : An mda generative tool for distributed real-time and embedded applications, 2003.

[Gorrieri and Siliprandi, 1994] Roberto Gorrieri and Glauco Siliprandi. Real-time system verification using p/t nets. In *Proceedings of the 6th International Conference on Computer Aided Verification*, CAV '94, pages 14–26, London, UK, UK, 1994. Springer-Verlag.

[Hugues, 2005] Jérôme Hugues. *Architectures et Services des Intergiciels Temps Réel*. PhD thesis, École Nationale Supérieure des Télécommunications, sep 2005.

[Hugues, 2011] Jérôme Hugues. Analytic virtual integration of cyber-physical systems and aadl : challenges, threats and opportunities. *The Second Analytic Virtual Integration of Cyber-Physical Systems Workshop*, Nov 2011.

[IEEE, 2000] IEEE. Ieee recommended practice for architectural description of software-intensive systems. IEEE Std 1471-2000 ISO/IEC 42010 :2007, IEEE, 2000.

[IEEE, 2009] IEEE. Ieee standard vhdl language reference manual. Ieee std 1076-2008, IEEE, jan 2009.

[Jézéquel *et al.*, 2011] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model driven language engineering with kermeta. In *Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*, GTTSE'09, pages 201–221, Berlin, Heidelberg, 2011. Springer-Verlag.

[Jézéquel, 2008] Jean-Marc Jézéquel. Model transformation techniques. http://model-ware.inria.fr/staticpages/slides/ModelTransfo.pdf, 2008.

[Jouault and Tisi, 2010] Frédéric Jouault and Massimo Tisi. Towards incremental execution of atl transformations. In *Proceedings of the Third international conference on Theory and practice of model transformations*, ICMT'10, pages 123–137, Berlin, Heidelberg, 2010. Springer-Verlag.

[Jouault *et al.*, 2006] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl : a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 719–720, New York, NY, USA, 2006. ACM.

[Jouault *et al.*, 2008] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl : A model transformation tool. *Sci. Comput. Program.*, 72(1-2) :31–39, June 2008.

[Kiczales and Hilsdale, 2001] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. *SIGSOFT Softw. Eng. Notes*, 26(5) :313–, September 2001.

© 2012 Gilles LASNIER

[Kleppe *et al.*, 2003] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained. The Model Driven Architec- ture : Practice and Promise*. Addison-Wesley, 2003.

[Kordon and Pautet, 2005] Fabrice Kordon and Laurent Pautet. Toward next-generation toward next-generation middleware ? *IEEE DIstributed Systems Online*, 5 1 :ONLINE, mar 2005.

[Kwon *et al.*, 2002] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-java : a high integrity profile for real-time java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, JGI '02, pages 131–140, New York, NY, USA, 2002. ACM.

[LAAS, 2012] LAAS. Time petri net analyzer (tina) toolbox. http://projects.laas.fr/tina, 2012.

[Lara and Vangheluwe, 2002] Juan Lara and Hans Vangheluwe. Atom3 : A tool for multi-formalism and meta-modelling. In Kutsche, Ralf-Detlef and Weber, Herbert, editor, *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin / Heidelberg, 2002.

[Lasnier *et al.*, 2009] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérome Hugues. Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications. In *Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe '09, pages 237–250, Berlin, Heidelberg, 2009. Springer-Verlag.

[Lasnier *et al.*, 2010] G. Lasnier, T. Robert, L. Pautet, and F. Kordon. Architectural and behavioral modeling with aadl for fault tolerant embedded systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*, pages 87 –91, may 2010.

[Lasnier *et al.*, 2011a] G. Lasnier, L. Pautet, and J. Hugues. A model-based transformation process to validate and implement high-integrity systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011 14th IEEE International Symposium on*, pages 67 –74, march 2011.

[Lasnier *et al.*, 2011b] G. Lasnier, L. Pautet, J. Hugues, and L. Wrage. An implementation of the behavior annex in the aadl-toolset osate2. In *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, pages 332 –337, april 2011.

[LIP6 - MoVe, 2012] LIP6 - MoVe. Cpn-ami. http://move.lip6.fr/software/CPNAMI, 2012.

[Liu and Layland, 1973] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1) :46–61, January 1973.

[Luckham *et al.*, 1995] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Trans. Softw. Eng.*, 21(4) :336–355, April 1995.

[Mazzini *et al.*, 2009] Silvia Mazzini, Stefano Puri, and Tullio Vardanega. An mde methodology for the development of high-integrity real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1154–1159, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[Medvidovic and Taylor, 2000] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1) :70–93, January 2000.

© 2012 Gilles LASNIER

[Mens and Van Gorp, 2006] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152 :125–142, March 2006.

[MetaCase, 2012] MetaCase. Metaedit+ modeler. http://www.metacase.com/mep, 2012.

[Miranda and Schonberg, 2004] Javier Miranda and Edmond Schonberg. Gnat : The gnu ada compiler. http://www.adacore.com/gap-static/GNAT_Book/gnat-book.pdf, 2004.

[Muller, 2006] Pierre-Alain Muller. *De la modélisation objet des logiciels à la metamodélisation des langages informatiques*. Habilitation à diriger des recherches, Université de Rennes 1, nov 2006.

[OMG, 2003a] OMG. Light weight corba component model. Revised submission, OMG, may 2003.

[OMG, 2003b] OMG. Mda guide version 1.0.1. Omg/2003-06-01, OMG, jun 2003.

[OMG, 2004] OMG. Common object request broker architecture : Core specification, version 3.0.3. Omg technical document formal/04-03-12, OMG, mar 2004.

[OMG, 2005] OMG. Real-time corba specification version 1.2. Omg technical document formal/05-01-04, OMG, jan 2005.

[OMG, 2006a] OMG. Corba component model specification version 4.0. Omg technical document formal/06-04-01, OMG, apr 2006.

[OMG, 2006b] OMG. Deployment and configuration of component-based distributed applications specification, version 4.0. Omg technical document formal/06-04-02, OMG, apr 2006.

[OMG, 2006c] OMG. Meta object facility (mof) 2.0 core specification. Final adopted specification, OMG, jan 2006.

[OMG, 2007a] OMG. Data distribution service for real-time systems version 1.2. Omg technical document formal/07-01-01, OMG, jan 2007.

[OMG, 2007b] OMG. Mof 2.0 / xmi mapping specification, v2.1.1. Technical report, OMG, dec 2007.

[OMG, 2007c] OMG. Omg unified modeling language (omg uml), infrastructure, version 2.1.2. Final adopted specification, OMG, nov, 2007.

[OMG, 2007d] OMG. Omg unified modeling language (omg uml), superstructure, version 2.1.2. Final adopted specification, OMG, nov 2007.

[OMG, 2007e] OMG. Uml profile for marte, beta 1, ptc/07-08-04. http ://www.omg.org/cgi-bin/doc ?ptc/2007-08-04, OMG, 2007.

[OMG, 2009] OMG. Mof 2.0 query / view / transformation. Technical report, OMG, dec 2009.

[OMG, 2010] OMG. Object constraint language (ocl). Technical report, OMG, 2010.

[Panunzio and Vardanega, 2011] Marco Panunzio and Tullio Vardanega. Schedulability analysis of ravenscar systems with mast+. *ArtistDesign Workshop on Real-Time System Models for Schedulability Analysis*, feb 2011.

[Pautet, 2001] Laurent Pautet. *Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition*. Habilitation à diriger des recherches, Université Pierre et Marie Curie – Paris VI, dec 2001.

[Perrotin *et al.*, 2011] Maxime Perrotin, Thanassis Tsiodras, Julien Delange, and Jérôme Hugues. taste : the assert set of tools for engineering. Technical report, European Space Agency - the Assert Project, sep 2011.

[pos, 2009] Information technology - portable operating system interface (posix) operating system interface (posix). *ISO/IEC/IEEE 9945 (First edition 2009-09-15)*, pages c1 –3830, 15 2009.

[Quinot, 2003] Thomas Quinot. *Conception et réalisation d'un intergiciel schizophrène pour la mise en oeuvre de systèmes répartis interopérables*. PhD thesis, École Nationale Supérieure des Télécommunications, 2003.

[Renault, 2009] Xavier Renault. *Mise en oeuvre de notations standardisées, formelles et semi-formelles dans un processus de développement de systèmes embarqués temps-réel répartis*. PhD thesis, Université Pierre et Marie Curie, nov 2009.

[Rushby, 1994] John Rushby. Critical system properties : Survey and taxonomy. Technical Report SRI-CSL-93-1, Computer Science Laboratory, SRI International, 1994.

[SAE Aerospace, 2004] SAE Aerospace. Architecture analysis and design language (aadl). Aerospace Standard SAE AS55066 - AADL1.0 - final draft, SAE, 2004.

[SAE Aerospace, 2009a] SAE Aerospace. Annex x behavior annex. Aerospace Standard SAE AS5506-X - Behavior Annex - draft v2.12, SAE, 2009.

[SAE Aerospace, 2009b] SAE Aerospace. Architecture analysis and design language (aadl). Aerospace Standard SAE AS5506A - AADL v2 - final draft, SAE, jan 2009.

[SAE Aerospace, 2009c] SAE Aerospace. Architecture analysis and design language (aadl) v2 code generation annex document. Aerospace Standard SAE AS5506 -Code Generation Annex - draft, SAE, jun 2009.

[SAE Aerospace, 2009d] SAE Aerospace. Architecture analysis and design language (aadl) v2 data modeling annex document. Aerospace Standard SAE AS5506-X - Data Modeling Annex - draft v0.17, SAE, nov 2009.

[SAE Aerospace, 2011] SAE Aerospace. Architecture analysis and design language (aadl) annex volume 3 : Annex e : Error model annex. Aerospace Standard SAE AS5506/3-E - Error Model Annex - draft, SAE, sep 2011.

[Salazar, 2010] Emilio Salazar. Automatic generation of schedulability analysis - consistent code. Presentation, 2010.

[Schmidt and Cleeland, 2001] Douglas C. Schmidt and Chris Cleeland. Applying a pattern language to develop extensible orb middleware. In Linda Rising, editor, *Design patterns in communications software*, chapter Applying a pattern language to develop extensible ORB middleware, pages 393–438. Cambridge University Press, New York, NY, USA, 2001.

[Schmidt *et al.*, 1998] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the tao real-time object request broker. *Elsevier Science UK*, 21(4) :294–324, apr 1998.

[SEI/CMU, 2011] SEI/CMU. OSATE2 : Open-Source AADL Tool Environment. http://www.aadl.info, 2011.

[Services, 2012] Tata Consultancy Services. Modelmorf. http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm, 2012.

[Sha *et al.*, 1990] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9) :1175–1185, September 1990.

[Singhoff *et al.*, 2004] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar : a flexible real time scheduling framework. In *SIGAda '04 : Proceedings of the 2004 annual ACM SIGAda international conference on Ada*, pages 1–8. ACM, 2004.

© 2012 Gilles LASNIER

[Singhoff *et al.*, 2005] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and memory requirements analysis with aadl. *Ada Lett.*, XXV(4) :1–10, November 2005.

[Software, 2012] Objecteering Software. Objecteering : the model-driven development tool. http://www.objecteering.com, 2012.

[Soley, 2000] Richard Soley. Model driven architecture (mda), draft 3.2. Technical report, OMG, nov 2000.

[Srivastava *et al.*, 2006] Sangeeta Srivastava, Naveen Prakash, and Sangeeta Sabharwal. The classification framework for model transformation. *Computer Science*, 2((2)) :166 – 170, 2006.

[Steinberg *et al.*, 2009] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF : Eclipse Modeling Framework 2.0*, volume isbn/0321331885. Addison-Wesley Professional, 2nd edition, 2009.

[SUN, 1988] SUN. Rpc : remote procedure call protocol specification version 2 ; rfc1058. Internet request for comments, (1057), SUN, 1988.

[TELECOM ParisTech, 2011] TELECOM ParisTech. Behavior annex errata, dec 2011.

[TELECOM ParisTech, 2012] TELECOM ParisTech. Ocarina : An AADL model processing suite. http://aadl.enst.fr, 2012.

[The Assert Project, 2012] The Assert Project. The assert project. http://www.assert-project.net, 2012.

[Triskell, 2012] Triskell. Kermeta. IRISA, Rennes, 2012.

[Universidad de Cantabria, 2010] Universidad de Cantabria. MAST : Modeling and Analysis Suit for Real Time Applications. http://mast.unican.es, 2010.

[Uppsala University - Aalborg University, 2012] Uppsala University - Aalborg University. Uppaal. http://www.uppaal.org, 2012.

[Vanderbilt University, 2012] Vanderbilt University. Great : Graph rewriting and transformation. http://www.isis.vanderbilt.edu/tools/GReAT, 2012.

[Vergnaud, 2006] Thomas Vergnaud. *Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées*. PhD thesis, École Nationale Supérieure des Télécommunications, 2006.

[Vestal, 2000] Steve Vestal. Metah. *SIGSOFT Softw. Eng. Notes*, 25(1) :105–, January 2000.

[Vidal *et al.*, 2009] Jorgiano Vidal, Florent de Lamotte, Guy Gogniat, Philippe Soulard, and Jean-Philippe Diguet. A co-design approach for embedded system modeling and code generation with uml and marte. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 226–231, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[W3C, 1999] W3C. Xsl transformations (xslt) version 1.0. http://www.w3.org/TR/1999/REC-xslt-19991116., nov 1999.

[W3C, 2010] W3C. Xquery 1.0 : An xml query language (second edition). http://www.w3.org/TR/2010/REC-xquery-20101214, dec 2010.

[Working Group, 2005] Working Group. Ada 2005 language reference manual, 2005.

[Zalila *et al.*, 2008] B. Zalila, L. Pautet, and J. Hugues. Towards automatic middleware generation. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 221 –228, may 2008.

© 2012 Gilles LASNIER

[Zalila, 2008] Bechir Zalila. *Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture*. PhD thesis, École Nationale Supérieure des Télécommunications, nov 2008.

© 2012 Gilles LASNIER

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

THÈSE