Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture

Bechir Zalila

TELECOM ParisTech

École Doctorale d'Informatique, Télécommunications et Électronique de Paris

Thèse

Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture

présentée et soutenue publiquement le 07 Novembre 2008

pour l'obtention du

Doctorat de l'École Nationale Supérieure des Télécommunications

spécialité : Informatique et Réseaux

par

Bechir ZALILA

Composition du jury

Président :	Jacques MALENFANT	Professeur, LiP6 - Université Pierre & Marie Curie
Rapporteurs :	Yvon KERMARREC	Professeur, TELECOM Bretagne
	Lionel SEINTURIER	Professeur, Université de Lille 1
Examinateurs :	Peter FEILER	Directeur de recherche, SEI - Carnegie Mellon University
	Franco GASPERONI	Directeur général, Europe - AdaCore
Directeurs de thèse :	Laurent PAUTET	Professeur, TELECOM ParisTech
	Jérôme HUGUES	Maître de conférences, TELECOM ParisTech

Á mes parents,
Á mon frère.

Remerciements

Ce mémoire, fruit de trois années de travail, n'a pu voir le jour sans l'aide et le support de personnes exemplaires. Cette partie n'est qu'une modeste tentative de reconnaissance à ces personnes.

Je tiens tout d'abord à remercier mes deux directeurs de thèse Laurent PAUTET, professeur à TELECOM ParisTech et Jérôme HUGUES, maître de conférences à TELECOM ParisTech. L'aide et les conseils qu'il n'ont jamais cessé de me donner m'ont été d'une valeur inestimable. Ils m'ont fait sentir que je suis plus un ami qu'un disciple. Par ailleurs, ayant été parmi mes enseignants en école d'ingénieurs et en Master, Laurent et Jérôme m'ont orienté vers le domaine fascinant de l'informatique temps-réel répartie embarquée et m'ont fait découvrir le meilleur langage de programmation de tous les temps, Ada. Ma gratitude envers eux est sans limite.

Je remercie Yvon KERMARREC, professeur à TELECOM Bretagne et Lionel SEINTURIER, professeur à l'Université de Lille 1 pour avoir accepté d'être les rapporteurs de ce mémoire. Les remarques pertinentes qu'ils ont émis ont permis de le consolider.

Je remercie également Jacques MALENFANT, professeur au LiP6 - Université Pierre & Marie Curie, Peter FEILER, directeur de recherche au Software Engineering Institute - Carnegie Mellon University et Franco GASPERONI, directeur général d'AdaCore - Europe pour l'intérêt qu'ils ont porté à mes travaux et pour avoir accepté de faire partie du jury de ma soutenance de thèse.

Effectuer cette thèse au sein du département INFRES de TELECOM ParisTech m'a donné l'occasion de faire la connaissance de plusieurs amis. Le temps passé avec ces amis ainsi que les nombreuses discussions, souvent vives et intéressantes, ont constitué la cerise sur le gâteau de ces trois années de thèse. Merci donc à Irfan HAMID, Hoa HA DUONG, Raja CHIKY, Nesrine GABSI, Billel GUENI, Julien DELANGE, Olivier GILLES, Etienne BORDE, Xavier RENAULT, Gilles LASNIER, Khaled JOUINI, Khalil MAHRSI, Nora DEROUICHE, Khaled BARBARIA, Thomas VERGNAUD et Isabelle PERSEIL. Je ne vous oublierai jamais.

Je remercie vivement Fabrice KORDON, Isabelle DEMEURE, Elie NAJM, Bertrand DUPOUY et Sylvie VIGNES 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.

Ma gratitude va également à Philippe DAX, Serge GADRET et Pierre BEYSSAC pour avoir toujours été disponibles et pour m'avoir fournis tous les supports matériels et logiciels nécessaires pour effectuer mes expérimentations et mener à bon port mes travaux de recherche.

Je n'oublie pas non plus mes amis ingénieurs Tunisiens, grâce à qui le mal du pays a été considérablement amoindri durant six années passées en France. Je remercie particulièrement Youssef GHORBAL, Ali KEFIA, Amine ELLEUCH, Mohamed TRIGUI, Walid BOUJELBEN, Taoufik HNIA et Khalil GHORBAL.

Enfin, la totalité de ma reconnaissance et de mes pensées vont à mes parents Mohamed et Radhia et à mon petit frère Baligh. Ils n'ont jamais cessé de m'encourager et de m'apporter du support surtout durant les dernières phases, pas toujours faciles, de la rédaction de ce manuscrit. C'est donc tout naturellement que ce document leur soit dédié.

Résumé

La production de systèmes temps-réel répartis embarqués (TR^{2}E) est une opération lourde en temps et en coût de développement. De plus, les applications temps-réel doivent satisfaire des contraintes dures pour assurer leur bon fonctionnement (respect des échéances temporelles...). L'utilisation des langages de description d'architecture vise à réduire le coût de développement de ces applications. AADL (*Architecture Analysis & Design Language*) fait partie de cette famille de langages. Il propose la notion de "composant" (matériel ou logiciel) dont la sémantique bien définie permet de décrire plusieurs aspects d'un système TR^{2}E. Les différentes contraintes qui doivent être satisfaites sont intégrées dans le modèle AADL sous forme de propriétés. Ce travail de thèse exploite les fonctionnalités offertes par AADL pour spécifier les besoins exacts d'une application TR^{2}E afin de la produire automatiquement. En effet, le processus de production que nous proposons (1) génère automatiquement le code pour intégrer les composants applicatifs à la plate-forme d'exécution, (2) génère automatiquement une importante partie des composants intergiciels taillés sur mesure pour l'application et (3) déploie automatiquement les composants applicatifs et intergiciels afin d'obtenir un système fortement dédié à l'application. Notamment, la plate-forme d'exécution supportant les composants AADL est elle même configurée statiquement en fonction des propriétés spécifiées. L'approche adoptée réduit le temps de développement et permet d'obtenir un code personnalisé et analysable. La configuration et le déploiement (souvent séparés du processus de développement) sont désormais automatiques et intégrés à la chaîne de production.

Mots-clés: Systèmes répartis, systèmes embarqués, temps-réel, configuration, déploiement, génération de code, OCARINA, POLYORB-HI, AADL

Abstract

Building distributed real-time embedded systems (DRE) is a tedious task. In addition, real-time applications must satisfy hard constraints to ensure they work correctly (meeting deadlines...). The use of architecture description languages aims at reducing the development cost of these applications. AADL (*Architecture Analysis & Design Language*) belongs to this family of languages. It uses the concept of "component" (hardware or software) whose well defined semantics makes possible the description of many aspects of DRE systems. The various constraints that must be met are integrated into the AADL model as properties. This thesis work exploits the features offered by AADL to specify the exact requirements of a DRE application and automatically generate its code. The production process we propose (1) automatically produces the code to integrate the applicative components to the runtime platform, (2) automatically produces a significant part of the middleware components customised for the application and (3) automatically deploys the applicative and middleware components to get a system which is strongly dedicated to the application. In particular, the AADL executive is itself statically configured. The adopted approach reduces the development time and allows for an customised and analyzable code. The configuration and deployment (often separated from the development process) are now automated and integrated into the production chain.

Keywords: Distributed systems, embedded systems, real-time, configuration, deployment, code generation, OCARINA, POLYORB-HI, AADL

Table des matières

Liste des illustrations

Liste des exemples de code

Liste des algorithmes

Liste des tableaux

Chapitre 1

Introduction Générale

SOMMAIRE

1.1 Contexte général

Concevoir des applications temps-réel réparties embarquées constitue de nos jours une tâche très difficile. Ceci est dû à la complexité de ces applications, au nombre de contraintes qu'elles doivent respecter et aux problèmes à prendre en compte et à résoudre lors de leur construction [Vinoski, 2002]. Un système de gestion de missions d'un ensemble de satellites (autonomes ou bien pilotés par une station terrestre) est un exemple réaliste d'une application temps-réel répartie embarquée [ESA, 2007]. Plusieurs problèmes doivent être pris en compte dans une telle situation : respect des échéances des activités de chacun des satellites, gestion des canaux de connexion et des protocoles de communication robustes, tolérance aux pannes...

L'introduction des standards de répartition (CORBA, RMI, RPC...) a permis de simplifier la conception des systèmes répartis : l'utilisateur n'a plus à interagir avec les couches basses de

transport pour gérer la communication entre les différents nœuds de son application. Il définit, dans un langage spécifique (IDL, Java...), l'interface de communication entre ces nœuds. L'intergiciel gère d'une manière transparente la gestion de ces communications.

Cependant, la plupart de ces standards sont mal-adaptés au domaines temps-réel et embarqués. La grande taille des bibliothèques d'intergiciels qui offrent une large panoplie de fonctionnalités contribue à élargir la taille des applications et rend difficile leur utilisation dans des systèmes embarqués. Pour rendre les intergiciels modulaires et configurables, les développeurs ont recours à des patrons de conception spécifiques [Schmidt *et al.*, 2000]. Cependant ces patrons utilisent des mécanismes tels que l'aiguillage dynamique (orienté-objet) et l'allocation dynamique de mémoire. Ceci compromet fortement le déterminisme de l'application, requis dans les systèmes temps-réel. Enfin, la majorité des composants de l'intergiciel (que ce soient ceux faisant partie de la bibliothèque ou ceux générés automatiquement) ne subissent aucune sorte d'analyse ; ils sont souvent complexes et donc difficilement analysables. Ceci rend leur utilisation risquée dans le contexte des systèmes critiques.

Plusieurs travaux ont tenté de réconcilier les standards avec le domaine du temps-réel embarqué (RT-CORBA, CORBA/e). Mais les gains en performances et en analysabilité restent limités car les aspects temps-réel et embarqués ne sont pas au cœur de ces standards pour la répartition. Dans le cas de RT-CORBA, on note même des divergences entre la spécification et les règles d'ingénierie pour les systèmes temps réel répartis embarquées (concurrence et comportement dynamique...).

Ce travail de thèse s'articule autour de la conception, la configuration et le déploiement d'applications réparties. Nous donnons une nouvelle approche pour mener à bien cette conception dans les domaines temps-réel embarqués critiques.

Dans la suite de ce chapitre, nous définissons les problématiques de cette thèse, les contraintes qui s'opposent à leur résolution, les objectifs à atteindre après leur étude et enfin l'approche adoptée pour réaliser ces objectifs.

1.2 Problématiques

Il s'agit de construire un processus de conception d'applications temps-réel réparties embarquées (que nous noterons TR^{2}E dans tout le reste de ce mémoire). Ce processus inclut l'analyse (sémantique, ordonnancement...), le déploiement (sélection des composants requis), la configuration automatique (paramétrisation des composants sélectionnés) de l'intergiciel et enfin l'intégration des composants applicatifs, intergiciels et des composants fournis par l'utilisateur pour aboutir à une application prête à être exécutée.

1.2.1 Analyse de l'application répartie

Un système critique est un système dans lequel, une panne peut engendrer des pertes humaines ou matérielles dramatiques. Dans le contexte de ces systèmes, plusieurs analyses doivent être effectuées sur les applications pour garantir leur bon fonctionnement. Les analyses auxquelles nous nous intéressons dans ce mémoire sont :

- l'analyse sémantique (cohérence de la description donnée par l'utilisateur),
- l'analyse d'ordonnancement pour vérifier le respect des échéances de l'application comme l'analyse de temps de réponse des tâches (RTA), l'analyse monotone par taux (RMA).

Il faut noter que ces analyses ne sont pas corrélées et qu'elles sont souvent effectuées par des outils différents qui prennent en entrée des descriptions dans des formalismes différents

(graphes de dépendances de tâches, arbres syntaxiques, réseaux de Petri...). Le passage d'un formalisme à un autre s'effectue parfois manuellement et rien ne garantit que les propriétés soient préservées lors de cette transformation.

La première problématique est donc la suivante : *Comment effectuer de façon automatisée ou semi-guidée les différentes analyses sur une application répartie à partir d'une description initiale de cette application?*

1.2.2 Déploiement de l'application répartie

Déployer une application répartie revient à placer les différents constituants de cette application sur leurs emplacements physiques respectifs et préparer leur exécution. De plus, la tâche de déploiement consiste à déduire le placement implicite de certains composants. En particulier, le placement des entités responsables de l'envoi (resp. la réception) des messages sur le réseau du côté des nœuds clients (resp. serveurs) est une action de déploiement. Le déploiement assure aussi que les différents nœuds d'une application répartie puissent atteindre les nœuds auxquels ils sont connectés. Ceci nécessite la présence de tables d'adressage/nommage sur chaque nœud pour qu'il puisse communiquer avec les autres.

Cette tâche devient rapidement complexe si le nombre d'interactions entre les nœuds de l'application est grand. De plus, elle est source d'erreur pour le développeur lorsqu'elle est effectuée manuellement. La deuxième problématique de ce travail est par conséquent la suivante : *Comment assurer que ces actions de placement d'entités implicites soient effectuées automatiquement et correctement?*

1.2.3 Configuration de l'application répartie

Après le déploiement vient la phase de configuration qui consiste à personnaliser les composants sélectionnés en fonction des besoins et des propriétés de l'application. Par exemple, la spécification de la taille des tampons d'envoi et de réception des messages est un paramètre de configuration.

Le moindre changement dans les propriétés du système induit un changement dans sa configuration. La troisième problématique est donc : *Comment assurer une configuration automatique et correcte de l'application répartie?*

1.2.4 Intégration des composants de l'application

La dernière phase de la réalisation d'une application est l'intégration des différents composants qui la forment. Ces composants peuvent être produits par des acteurs différents (code fourni par l'utilisateur, code généré dans le contexte du déploiement, code de l'intergiciel, code généré par des outils tiers...). L'intégration consiste à rassembler les composants de l'application pour les compiler et produire les exécutables représentants les nœuds. Il est important que tous ces composants puissent être intégrés aisément par le développeur. Idéalement, ils doivent être intégrés automatiquement.

L'intégration des composants, en particulier ceux fournis par l'utilisateur nécessite trois conditions :

1. Les noms de ces composants (sous-programmes, types de données...) doivent être déduits à partir de la spécification pour que les autres composants puissent les *référencer*,
2. Les caractéristiques de ces composants (signatures de sous-programmes...) doivent être déduites à partir de la spécification pour que les autres composants puissent les *utiliser*,

3. Les binaires implantant ces composants (fichiers objets, bibliothèques) doivent pouvoir être retrouvés (ou construits) à partir de la spécification de l'application. Ceci permet une *édition de liens* correcte de ces composants.

La dernière problématique de ce travail est donc : *Comment intégrer automatiquement les différents composants d'une application répartie?*

1.3 Contraintes

Les contraintes auxquelles sont soumis les systèmes TR^{2}E rendent plus difficile la résolution des problématiques de la section 1.2. Dans cette section nous exposons les contraintes principales de ces systèmes.

1.3.1 Modèle de concurrence analysable

Les systèmes TR^{2}E sont souvent utilisés dans des domaines critiques où l'enjeu peut parfois être des vies humaines (ils sont d'ailleurs souvent qualifiés de *critiques*). Ils doivent donc être vérifiés et leur fonctionnement correct doit être démontré à des autorités de certification avant même leur exécution. Ceci explique les règles strictes de développement qui régissent ce genre de systèmes.

La manière dont les entités actives de l'application TR^{2}E interagissent (appelée aussi le *modèle de concurrence*) est un des aspects les plus importants à devoir être analysé. En effet, les applications dont les nœuds sont multi-tâches deviennent de plus en plus courantes et l'analyse d'ordonnançabilité et d'absence d'interblocage est la moindre des vérifications à effectuer dans ce cas.

1.3.2 Restrictions pour les systèmes critiques

Développer un système critique requiert beaucoup plus d'attention (cohérence, sûreté de fonctionnement, sécurité...) que développer un système classique. Le code doit respecter certaines restrictions pour éviter que des constructions jugées dangereuses apparaissent. Typiquement, pour les systèmes critiques embarqués, une des restrictions les plus connues est l'interdiction de l'allocation dynamique de mémoire car elle peut être source d'indéterminisme [Puaut, 2002] mais aussi car elle peut être une source de *fuites* de mémoire dues à des erreurs du programmeur ou parfois même du compilateur. Ce genre de fuites ne peut être toléré dans les systèmes embarqués dont la durée d'exécution est très longue et dont les ressources en mémoire sont très limitées (e.g. les satellites).

Pour les systèmes répartis, l'ouverture et la fermeture dynamiques des canaux de communication entre les différents nœuds est une source potentielle de problèmes. En effet, elle induit un risque de mauvaise gestion de la bande passante disponible. Il serait judicieux de pouvoir ouvrir tous ces canaux lors du démarrage de l'application.

Enfin, la plupart des standards de répartition reposent sur le paradigme orienté-objet (CORBA, DDS...). Certains patrons de conception (usines à objets, aiguillage dynamique) introduisent de l'indéterminisme dans le comportement de l'application ce qui rend l'analyse de l'ordonnancement difficile.

1.3.3 Augmentation de la complexité

Avec la croissance continue des performances des plates-formes embarquées, la taille et la complexité des applications qu'elles exécutent sont devenues de plus en plus grandes. Les techniques de conception classiques ont très vite atteint leur limites [The ASSERT Consotium, 2005]. Ces techniques consistent à développer manuellement l'application même si, parfois, un outil de modélisation permet d'assister le développeur au cours de son travail. En effet, construire manuellement une application embarquée qui contient des dizaines de processus légers et gérer la communication et la synchronisation entre ces processus légers est une tâche fastidieuse. La difficulté de cette tâche et la complexité des outils assistant pour sa réalisation peuvent induire le développeur en erreur.

1.3.4 Synthèse

Les contraintes listées dans les sections 1.3.1, 1.3.2 et 1.3.3 rendent la résolution des problématiques énoncées dans les sections 1.2.2, 1.2.3 et 1.2.4 plus difficile. De plus, l'analysabilité de l'application répartie (section 1.2.1) nécessite de définir une nouvelle approche de développement. Dans la suite, nous exposerons les objectifs de ce travail de thèse (section 1.4) et l'approche adoptée pour atteindre ces objectifs (section 1.5).

1.4 Objectifs

Nous voulons propager les propriétés exprimées dans le modèle jusqu'à la production des composants de l'application TR^{2}E ainsi que ceux de l'intergiciel. Par conséquent nous voulons utiliser un formalisme de description capable d'exprimer non seulement les caractéristiques de l'application, mais aussi celles de l'intergiciel. L'analyse des modèle écrit dans ce formalisme permet de produire automatiquement du code dédié pour les besoins de l'application.

1.4.1 Modélisation et analyse des applications TR^{2}E

Il s'agit de trouver un processus de modélisation d'applications réparties permettant de décrire les particularités des systèmes TR^{2}E (contraintes temporelles, ressources...). Il faut noter que différentes analyses peuvent être effectuées en utilisant des outils différents prenant souvent en entrée des descriptions dans des formalismes différents.

Pour automatiser ces analyses, le modèle initial doit contenir toutes les propriétés requises pour les mener à bien. Il faut aussi que ce modèle soit hiérarchique pour masquer les différents niveaux de complexité et qu'il puisse être transformé automatiquement en un formalisme compréhensible par les outils d'analyse.

Dans ce contexte, les langages de description d'architecture (ADL) visent à réduire le coût de développement d'une application répartie. Certains de ces langages offrent une syntaxe précise et assez détaillée qui permet d'effectuer des analyses sur le modèle et aussi de générer du code automatiquement. Nous avons choisi le langage AADL (*Architecture Analysis & Design Language*) car, pour des raisons que nous exposerons (chapitre 4), il nous semble le meilleur candidat pour atteindre nos objectifs. Ceci permet de résoudre la problématique énoncée dans la section 1.2.1 mais aussi d'adresser les contraintes énoncées dans 1.3.1 et 1.3.3.

1.4.2 Intergiciel adapté à l'application

L'introduction des intergiciels et des standards de répartition a simplifié la conception des applications réparties. En effet, l'utilisation des intergiciels permet d'élever le niveau d'abstraction particulièrement en ce qui concerne les communications. Le développeur n'a plus à manipuler les routines d'envoi et de réception de tampons de communication ni à gérer la manière dont les donnée sont insérées et extraites à partir de ces tampons.

Cependant, un intergiciel offre véritablement un nombre de fonctionnalités beaucoup plus grand que ceux dont l'application a besoin. Par exemple, les intergiciels CORBA doivent contenir des routines gérant la compatibilité entre les nœuds s'exécutant sur des architectures différentes (alignement des données dans les tampons de communication, boutisme...). Ceci rend la taille des applications réparties très grande et par la suite, très mal adaptée pour être utilisée dans des systèmes embarqués. Ce type d'intergiciel influe sur le cycle de vie d'une application répartie classique. Ce cycle comporte généralement une première phase de configuration automatique et de sélection des paramètres et qualités de services requis depuis de larges palettes fournies par l'intergiciel.

Notre second objectif est de pouvoir configurer et déployer un intergiciel *ad hoc* pour une application répartie. Cet intergiciel a une taille plus petite que celle des intergiciels classiques. Il peut être configuré classiquement en fonction des propriétés de l'application. Pour ce faire, nous aurons recours à certains éléments de l'architecture intergicielle schizophrènes [Pautet, 2002]. L'intergiciel produit joue le rôle d'un support d'exécution pour les modèles AADL des applications TR^{2}E. Ceci permet de répondre aux problématiques présentées dans les sections 1.2.2 et 1.2.3 tout en adressant la contrainte 1.3.2.

1.4.3 Génération automatique de code

Le meilleur moyen d'avoir un intergiciel dédié à une application donnée sans pour autant le réécrire à la main pour chaque application est d'écrire manuellement les composants qui sont invariants d'une application. Le reste des composants doit être généré automatiquement à partir d'une une description précise de l'architecture et de ses propriétés (exigences). Ainsi ces derniers composants seront *taillés sur mesure* pour l'application en cours.

Dans le contexte des systèmes critiques, on commencera par la génération vers le langage Ada dont le compilateur peut renforcer plusieurs restrictions que doivent subir ces systèmes (comme le profil Ravenscar [Burns *et al.*, 2004]). Ensuite nous étendrons ces constructions pour le langage C.

Le générateur de code aura donc un rôle double : (1) accueillir le code fourni par l'utilisateur (ou généré par des outils tiers) et (2) produire les composants intergiciels dédiés à l'application en question. La génération automatique et massive de code permet de réduire le cycle de développement de l'application et réduit le risque d'erreur par le développeur. Ceci complète la résolution des problématiques 1.2.2, 1.2.3 et 1.2.4 tout en adressant la contrainte 1.3.2.

1.5 Approche

Dans l'approche que nous avons adoptée, le déploiement et la configuration de l'application sont fortement pris en compte dès les phases de modélisation. De plus, la conception de l'intergiciel est fondée sur la génération massive de code.

1.5.1 Intergiciel

Pour chaque application TR^{2}E, nous visons à construire un intergiciel qui lui soit dédié. Ceci nécessite la définition d'une nouvelle architecture intergicielle. L'intergiciel est formé par des composants suffisamment indépendants pour être personnalisés séparément.

Nous identifions les composants de l'intergiciel qui sont fortement personnalisables en fonction des propriétés de l'application répartie. Ces composants sont optimisés en fonction des propriétés de l'application et sont générés automatiquement à partir de son modèle. les composants qui sont faiblement personnalisables sont rassemblés pour former un intergiciel minimal. Il peuvent par la suite être configurés en fonction des propriétés de l'application.

1.5.2 Modélisation

Du coté de la modélisation, le langage AADL offre des abstractions de bas niveaux permettant à la fois de modéliser les systèmes TR^{2}E avec un niveau de détail élevé mais aussi d'avoir des modèles qui sont précis et analysables. Les informations architecturales incorporées dans le modèle permettent de produire automatiquement les composants intergiciels qui sont dédiées à l'application.

AADL étant un langage riche et généraliste, nous spécifions un sous-ensemble de ce langage qui sera utilisé par les développeurs pour donner lieu à la génération d'un code "valide". Ce coté modélisation de l'approche implique la définition des règles pour permettre au code de l'utilisateur d'être encapsulé par le code généré (types de données, sous-programmes...). D'autres règles de transformation doivent permettre à ce code d'utiliser les composants intergiciels (envoi et réception de messages, partage de données...).

Aussi, pour renforcer la modélisation afin de simplifier l'intégration finale des composants de l'application, nous utilisons le mécanisme de propriétés offert par AADL pour étendre ses capacités de description.

1.5.3 Génération de code

Il s'agit de générer les composants applicatifs et intergiciels dédiés à l'application. Un générateur de code par cible (langage de programmation, plate-forme) doit être construit.

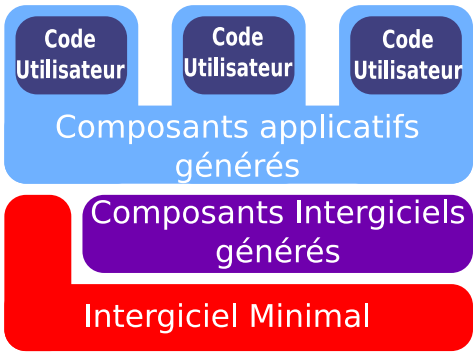

FIGURE 1.1 – Couches de code constituant un nœud de l'application

La figure 1.1 montre la structure globale d'un nœud : (1) l'intergiciel minimal se place au plus bas niveau (le plus près du matériel). (2) Au dessus de cet intergiciel, viennent se placer les composants intergiciels qui sont générés automatiquement en fonction des propriétés de

l'application. (3) Ensuite, viennent se placer les composants applicatifs qui encapsulent (4) le code utilisateur.

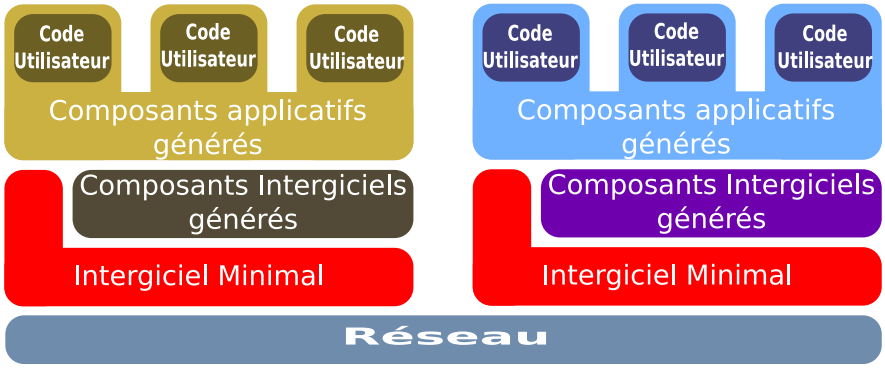

FIGURE 1.2 – Interaction entre les nœuds d'une application répartie

Les nœuds d'une même application répartie n'ont en commun que la partie minimale de l'intergiciel. La figure 1.2 montre une vision globale d'une application répartie formée par deux nœuds.

La suite de ce mémoire de thèse sera consacrée au développement détaillé de cette approche.

1.6 Plan du Mémoire

Ce document présente les trois étapes décrites précédemment : l'intergiciel dédié aux applications TR^{2}E, la manière de spécifier les caractéristiques de ces applications et la génération de code correspondant à ces caractéristiques. Mais en premier lieu (chapitre 2), nous donnons un état de l'art des technologies traitant les thématiques de notre travail de thèse. Nous présentons d'abord des architectures intergicielles pouvant être optimisées en fonction des besoins des applications. Ensuite nous présentons quelques formalismes utilisés pour décrire les applications réparties. Puis, nous nous intéressons aux outils qui permettent de configurer et déployer automatiquement des applications réparties embarquées. Enfin, nous présentons les contraintes des systèmes TR^{2}E et quelques moyens utilisés pour garantir le respect de ces contraintes. Nous concluons le chapitre 2 en précisant les limites des technologies présentées vis-à-vis du respect de ces contraintes et restrictions.

Le chapitre 3 décrit les grandes lignes de la solution que nous proposons dans ce travail de thèse. Nous définissons l'architecture d'un intergiciel qui est dédié à une application TR^{2}E. Nous expliquons l'impact du respect des contraintes des systèmes critiques sur une telle architecture. Nous concluons que le meilleur moyen d'avoir un intergiciel dédié est de produire automatiquement la plus grande partie de ses composants en fonction des propriétés de l'application. Ceci nous mènera logiquement vers le besoin d'un formalisme précis et rigoureux pour décrire les systèmes TR^{2}E.

Le chapitre 4 est une présentation du langage de description d'architecture AADL. Nous avons choisi ce formalisme pour décrire les applications réparties. Nous présentons les différents aspects de ce langage ainsi que les avantages qu'il apporte pour la description de ces systèmes. Ensuite, nous spécifions les restrictions supplémentaires et le sous-ensemble du langage AADL que nous utilisons pour décrire les applications TR^{2}E.

Le chapitre 5 complète la partie théorique de la solution proposée dans notre travail de thèse. Il décrit le processus de production automatique qui, à partir d'un modèle d'une application répartie :

- effectue les analyses nécessaires pour garantir son bon fonctionnement,
- produit automatiquement une grande partie des composants intergiciels et les optimise aux besoins de l'application,
- rassemble les différents composants pour déployer et configurer l'application.

Le chapitre 6 présente la réalisation pratique d'un intergiciel minimal pour les systèmes TR^{2}E. Nous décrivons les versions de cet intergiciel écrites dans les langages de programmation Ada et C. Nous nous attardons particulièrement sur les performances de cet intergiciel et nous montrerons le déterminisme des services qu'il offre.

Le chapitre 7 décrit la suite d'outils qui réalise le processus de production décrit plus haut. Nous décrivons l'architecture de cette suite et de ses générateurs de code. Nous expliquons aussi la manière dont les applications réparties sont configurées automatiquement à l'aide de cet suite d'outils.

Un cas d'étude et des mesures de performances sont donnés dans le chapitre 8 pour vérifier l'adéquation de notre solution avec les problématiques posées.

Enfin, nous concluons ce mémoire de thèse dans le chapitre 9 et nous présentons les perspectives possibles d'extension pour notre travail.

Première partie

Étude Théorique

Chapitre 2

État de l'Art et Problématique

SOMMAIRE

2.1 Introduction

La conception d'applications TR^{2}E retient de plus en plus d'attention aussi bien de la part des chercheurs que de la part des industriels. En effet, le nombre de micro-contrôleurs dans des appareils d'usage quotidien (voitures...) ne cesse d'augmenter et le besoin de concevoir des applications qui utilisent ces nouvelles architectures ne cesse de se manifester. Ce type d'applications devient complexe (nombre de lignes de codes, complexité des algorithmes et des graphes de connexions entre les nœuds). Par conséquent, leurs déploiement et configuration deviennent des tâches ardues car plusieurs problèmes doivent être résolus en même temps.

L'automatisation de la conception d'applications TR^{2}E et en particulier du déploiement et de la configuration simplifie considérablement le processus de production de ce genre d'applications.

La figure 2.1 illustre un processus classique de conception d'applications TR^{2}E. Le modèle de l'application sert à produire automatiquement les composants applicatifs. Il pilote aussi les deux phases de déploiement et de configuration des composants intergiciels. Il permet enfin d'intégrer les composants applicatifs et intergiciels pour produire l'application finale.

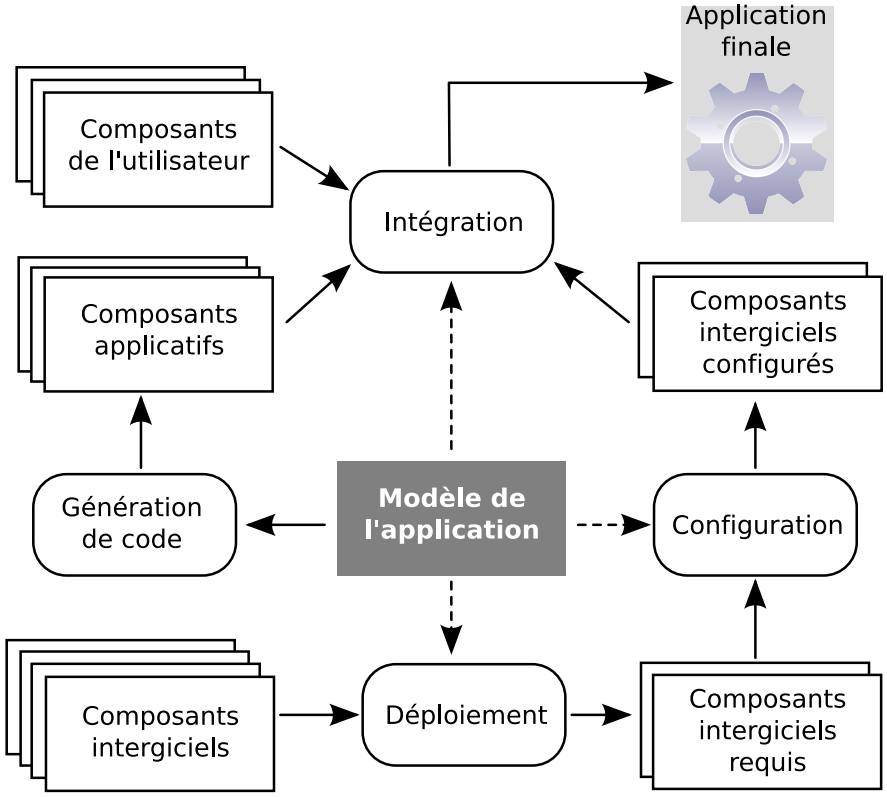

FIGURE 2.1 – Conception d'une application TR^{2}E

Dans ce chapitre, nous présentons l'état de l'art dans le domaine de la conception d'applications réparties et plus particulièrement dans l'automatisation de cette conception. Tout naturellement, nous commençons par l'état de l'art sur les intergiciels fortement configurables, ceux qui constituent l'épine dorsale de toute application. Ensuite, nous donnons différents langages de modélisation utilisés pour spécifier précisément certains aspects des applications réparties (premier pas vers l'automatisme). Puis nous présentons quelques outils et projets qui s'articulent autour du déploiement et de la configuration automatiques des applications réparties dans le contexte des systèmes embarqués. Nous donnons après une vison des différentes spécifications et standards régissant les systèmes critiques (contraintes à respecter...). Enfin, nous donnons une synthèse globale du chapitre en critiquant les solutions existantes et en présentant les grandes lignes de notre travail.

2.2 Intergiciels

Les intergiciels sont des couches logicielles intermédiaires permettant de faire communiquer plusieurs nœuds distants d'une application. Ils ont été introduits pour minimiser le coût (en temps

et en argent) de conception des applications réparties. Comme l'indique leur nom, les intergiciels se placent entre le matériel (y compris le système d'exploitation) et le logiciel (restreint au code de l'utilisateur). Dans certains cas comme pour ERLANG [Armstrong, 1996], l'intergiciel implante toutes les fonctionnalités requises d'un système d'exploitation (concurrence...). Ainsi, au lieu d'avoir le code d'une application répartie qui s'interface directement avec les primitives du système d'exploitation ou le matériel (pour gérer la communication entre les nœuds par exemple), l'application invoque des primitives de l'intergiciel qui sont de plus *haut niveau* et qui offrent une portabilité à son code. La figure 2.2 montre l'emplacement de la couche intergicielle dans un nœud d'une application répartie.

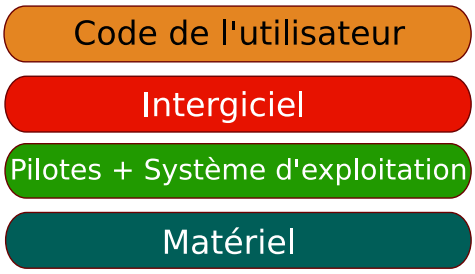

FIGURE 2.2 – Emplacement de la couche intergicielle

Souvent, la gestion directe de la communication entre les nœuds de l'application répartie (construction des tampons de communication avant l'envoi d'un message, vidage de ces tampons après la réception...) est générée automatiquement grâce à des outils fournis par l'intergiciel. Il en est de même pour d'autres tâches fastidieuses comme l'exécution par le serveur de l'opération demandée par le client. Ces composants générés utilisent une ou plusieurs bibliothèques de l'intergiciel qui implantent toutes les fonctionnalités requises pour le dialogue entre les nœuds d'une application faisant abstraction ainsi des routines et primitives du système d'exploitation. La génération se produit à partir d'une description du service entre les nœuds, fournie par l'utilisateur dans un langage de haut niveau (généralement simple et facilement maîtrisable).

Plusieurs mécanismes ont été élaborés pour normaliser la conception d'applications réparties. Des mécanismes allant de la simple invocation distante de procédures dans RPC [SUN, 1988] jusqu'à des niveaux plus complexes comme les objets répartis CORBA [OMG, 2004] et la distribution de données DDS [OMG, 2007a]. Il existe aussi des versions spécialisées de ces mécanismes comme RT-CORBA [OMG, 2005] pour les systèmes temps-réel et CORBA/e [OMG, 2008] pour les systèmes embarqués.

Étant donné que les besoins des applications réparties en termes de ressources intergicielles sont souvent différents, les concepteurs d'intergiciels ont été très rapidement face au problème suivant : *doivent-ils concevoir un intergiciel dédié à un domaine ou une catégorie donnés d'applications réparties ou bien doivent-ils faire en sorte que leur intergiciel couvre plusieurs domaines à la fois?* Dans la suite, nous présentons deux stratégies de conception d'intergiciels adoptées pour accomplir les deux choix en faisant des compromis. Ces deux stratégies seront présentées à travers des exemples de leurs implantations.

2.2.1 Intergiciels Configurables

Définition 2.1 ***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.

En particulier, dans le contexte des systèmes TR^{2}E, les intergiciels configurables ont une utilité double : (1) réduire la taille mémoire de l'application en ne sélectionnant que les composants dont elle a besoin et (2) respecter les contraintes de l'application en personnalisant les composants sélectionnés.

TAO

TAO (The ACE ORB) a été construit en partant du concept que la plupart des implantations de CORBA ne sont pas appropriées pour les systèmes temps-réel parce qu'elles manquent de fonctionnalités pour assurer la qualité de service de la transmission de requêtes. Dans [Schmidt *et al.*, 1998], les auteurs donnent la description de TAO qui est un intergiciel de très haute performance. TAO fonctionne sur plusieurs plates-formes natives et embarquées. Ils donnent aussi les fonctionnalités d'ordonnancement temps-réel de TAO permettant de fournir des garanties de qualité de service (QdS) pour les applications réparties temps-réel.

Description TAO permet de fournir des assurances de qualité de service (QdS) de bout en bout dans une application répartie. En effet, il permet de gérer les priorités dans les requêtes (pour limiter l'inversion de priorité). Il supporte les priorités pour les tâches exécutant les opérations. Il permet de contrôler les ressources requises pour exécuter une opération sur le serveur. En particulier, TAO permet d'éviter le monopole des services du serveur par un client.

La forte portabilité de TAO est acquise grâce à l'utilisation de ACE, un canevas de communication orienté-objet développé en C++ dont les composants sont développés en utilisant des patrons de conception bien définis [Schmidt and Cleeland, 2000].

Le développement de TAO a été motivé par les sujets suivants :

- contribuer à l'amélioration des standards (notamment CORBA) pour y intégrer la spécification de paramètres de QdS de l'application,
- évaluer de façon empirique les fonctionnalités requises pour construire un ORB (*Object Request Broker*) déterministe,
- caractériser de façon précise les propriétés de la liaison de bout en bout (bande passante, latence) pour pouvoir les garantir,
- avoir un intergiciel basé sur des patrons de conception ce qui permet un développement, une maintenance, une configuration et une extension plus faciles.

Grâce à son architecture extensible, TAO permet aux développeurs de choisir le contexte dans lequel ils se situent (performance ou faisabilité). Il permet aussi d'effectuer le "contrôle d'admission" lors de l'ordonnancement des tâches : refuser d'ajouter une tâche à l'ensemble courant si ceci va compromettre l'ordonnançabilité du système.

Les patrons de conception utilisés dans TAO sont des patrons classiques des systèmes répartis concurrents [Schmidt *et al.*, 2000] développés en C++. Pour configurer les politiques d'ordonnancement dans une application répartie, TAO utilise les patrons *Strategy* et *Abstract Factory*. Si plusieurs politiques d'ordonnancement sont supportées dans un intergiciel, le patron *Bridge* permet de sélectionner la politique désirée selon la demande de l'utilisateur. Pour configurer les algorithmes d'ordonnancement dynamiquement à l'exécution, TAO utilise le patron *Service Configurator*. Le patron *Acceptor-Connector* découple l'établissement d'une connexion des différentes actions effectuées par les extrémités de cette connexions (envoi, réception). Le patron *Reactor* désigne une entité qui réagit à la réception d'un événement extérieur.

Dans [Schmidt *et al.*, 1998; Schmidt and Cleeland, 2000], les auteurs montrent comment l'utilisation de ces patrons dans TAO permet d'abandonner progressivement la programmation des applications à partir de zéro pour leur intégration à partir de composants déjà existants. Ils proposent et illustrent l'utilisation d'un langage de "patrons" pour avoir un intergiciel (plus spécifiquement un ORB) dynamiquement configurable et personnalisable chose qui réduit la complexité et augmente la maintenabilité de plusieurs fonctionnalités de l'ORB. L'utilisation d'une technologie de répartition reposant sur les patrons de conception simplifie considérablement le développement des applications réparties (de préférence se basant sur des standards qui gèrent l'hétérogénéité).

TAO a fortement influencé la spécification RT-CORBA 2.0 supportée par les dernières versions de cet intergiciel [Krishnamurthy *et al.*, 2004]. RT-CORBA 2.0 introduit les fonctionnalités d'ordonnancement dynamique qui étaient absents dans la première édition du standard. Il permet de brancher dynamiquement des ordonnanceurs, supprimer dynamiquement des tâches, interagir d'une manière plus flexible avec l'ordonnanceur, disposer d'un moyen portable pour faire la différence entre les tâches de l'utilisateur et celles du système.

Discussion Le fait que TAO repose sur RT-CORBA rend son utilisation délicate dans le contexte de systèmes critiques et embarqués. En effet, dans [Hugues, 2005], plusieurs incompatibilités entre RT-CORBA et les règles de codage utilisées dans de tels systèmes sont notées. À titre d'exemple, nous pouvons citer la possibilité de modification dynamique de priorité des tâches par l'utilisateur ce qui compromet l'analysabilité statique d'ordonnancement.

De plus, l'utilisation d'un standard de répartition quel qu'il soit impose un respect minimum des interfaces de programmation de ce standard et limite le champ d'action des optimisations. Par exemple, dans le cas d'applications réparties non hétérogènes, il n'y a pas besoin de faire appel à tous les mécanismes qui assurent la compatibilité entre les nœuds. Ce type d'optimisations n'est pas supporté dans TAO. L'utilisateur qui souhaite avoir des routines des communication "dédiées" à son application doit les implanter lui-même.

L'utilisation des routines de communication du canevas ACE est présenté comme un avantage puisque le développement de TAO n'a plus à considérer des paramètres d'hétérogénéité entre les plates-formes. Mais ceci n'est qu'un décalage du problème puisque tous la portabilité de code doit être considérée lors du développement de ACE qui n'est pas un canevas standard de développement comme POSIX.

RTZEN/ZEN-KIT

RTZEN est une implantation en Java de RT-CORBA. La machine virtuelle Java [SUN, 2006] souffre de plusieurs problèmes d'indéterminisme dûs essentiellement à son ramasse-miettes. Ce dernier s'exécute d'une manière imprévisible et compromet le déterminisme des actions. Elle ne supporte pas des mécanismes comme PCP [Sha *et al.*, 1990] utiles pour les systèmes temps-réel. Pour cela, RTZEN a été basé sur la technologie RTSJ [Gosling and Bollella, 2000], une spécification temps-réel pour Java qui limite le champ d'action du ramasse-miettes rendant son comportement prévisible sous certaines conditions et qui offre des fonctionnalités utiles pour les systèmes temps-réel.

Description Dans [Raman *et al.*, 2005], les auteurs exposent les contraintes qui ont motivé le développement de RTZEN : assurer le déterminisme des applications réparties, être conforme

au standard RT-CORBA, garantir la performance des applications construites, minimiser la complexité du code de l'utilisateur et utiliser efficacement la mémoire et enfin pouvoir personnaliser l'intergiciel aux besoins de l'application. Ces contraintes sont satisfaites en utilisant des patrons de conception offerts par la spécification RTSJ comme les *threads* temps-réel qui sont plus prioritaires que le ramasse-miettes et qui rendent ainsi les actions déterministes.

RTZEN est implanté de manière à rendre ses fonctionnalités hautement personnalisables. Par exemple, le mode de concurrence est sélectionné parmi un ensemble de modes disponibles et la mémoire est gérée de différentes façons selon les besoins et la nature de l'application. Cette personnalisation peut conduire à des applications réparties dont la performance est très proche de celles écrites dans des langages compilés. En effet, des mesures de performances donnés dans [Raman *et al.*, 2005] montrent que RTZEN est proche de TAO en terme de performance.

RTZEN est aussi accompagné d'un outil graphique pour rendre plus facile sa configuration pour les utilisateurs qui connaissent peu l'implantation interne de l'intergiciel. Il s'agit de ZEN-KIT [Gorappa *et al.*, 2005] qui utilise la plate-forme ECLIPSE et qui permet de configurer RTZEN en insérant ou retirant les fonctionnalités selon les spécifications de l'application répartie. Il permet d'optimiser l'intergiciel (réduire l'empreinte mémoire par exemple) en contrôlant les composants intergiciels inclus dans l'application. Cet outil tend à faciliter cette personnalisation et la rendre assez simple pour être réalisée par l'utilisateur.

ZEN-KIT présente à l'utilisateur une vue hiérarchique de l'intergiciel RTZEN pour faciliter sa personnalisation. Trois niveaux de représentation sont fournis :

1. Une vue conceptuelle de l'ORB qui donnent la liste des modules qui le forment,
2. Les fonctionnalités de chacun des modules de l'ORB,
3. Une vision détaillée de chacune des options de chaque fonctionnalité (IOR, protocole...)

Le passage d'une vision à une autre se fait par l'intermédiaire d'une fonctionnalité intéressante de mise au point (*zoom*) qui permet à l'utilisateur de contrôler la granularité de sa personnalisation.

Discussion Tout comme TAO, RTZEN est basé sur RT-CORBA, ce qui ne permet pas de pousser la personnalisation jusqu'à modifier l'architecture de l'intergiciel en ajoutant des composants qui divergent du standard (utilisation d'un protocole "léger" pour invoquer les requêtes par exemple).

De plus, le fait qu'il soit programmé en Java et qu'il utilise les mécanismes issus du monde orienté-objet (aiguillage dynamique) compromet l'analysabilité statique des application puisque le flux de contrôle n'est pas connu statiquement.

2.2.2 Intergiciels Schizophrènes

Définition 2.2 ***intergiciel schizophrène*** *[Quinot, 2003]*
Un intergiciel est dit schizophrène lorsqu'il permet la cohabitation simultanée de plusieurs paradigmes de répartition au sein d'une même instance et met en place un mécanisme efficace d'interaction entre ces paradigmes (personnalités).

Cette catégorie d'intergiciels à été introduite pour répondre aux besoins d'interopérabilité entre des différents paradigmes de répartition de façon transparente à l'application. Une nouvelle architecture d'intergiciels a été définie [Pautet, 2002] et ensuite concrétisée [Quinot, 2003] sous la forme de l'intergiciel POLYORB. Elle identifie les fonctionnalités de base effectuées par un

intergiciel pour mener à bien l'acheminement d'une requête de l'expéditeur vers le destinataire. Ces fonctionnalités sont appelées les "services canoniques" de l'intergiciel.

Ces services (l'adressage, la liaison, l'interaction, le typage, la représentation, le protocole, le transport, l'activation et l'exécution) sont implantés dans un cœur indépendant de tout standard de répartition appelé *la couche neutre*. L'implantation des aspects applicatifs d'un standard de répartition donné (représentation des données, gestion des objets répartis...) est effectuée dans des *personnalités applicatives*. Les aspects liés à la communication (protocoles, transport...) sont quant à eux implantés dans des *personnalités protocolaires*. Les *personnalités applicatives* et les *personnalités protocolaires* utilisent et spécialisent les services offerts par la *la couche neutre*.

Des éléments de cette architecture qui simplifient la configuration et le déploiement des applications réparties (organisation du système en couches pour mieux séparer les aspects et personnaliser chacune des couches indépendamment des autres) seront utilisés par nos travaux.

POLYORB

Description POLYORB est la première implantation de l'architecture schizophrène. Il propose cinq personnalités applicatives (CORBA, DSA, AWS, MOMA et DDS) et trois personnalités protocolaires (GIOP, SOAP et SRP). Grâce a son architecture, il permet de personnaliser plusieurs aspects de l'application répartie. Il permet par exemple de choisir un des profils de concurrence disponibles : mono-tâche, multi-tâches, groupe de tâches....

Les travaux décrits dans [Kordon and Pautet, 2005] montrent que la séparation des interactions entre les nœuds d'une application des implantations de ces interactions permet de renforcer la généricité de l'intergiciel pour supporter plusieurs modèles de distribution, patrons de concurrences ou protocoles de communication. Cette approche étend les travaux sur les intergiciels configurables [Schmidt *et al.*, 1998] et les intergiciels génériques [Dumant *et al.*, 1998] et permet de faciliter l'adaptabilité de l'intergiciel aux besoins de l'application. Mais elle reste toujours du domaine des constructeurs de l'intergiciel et ne peut pas être effectuée par le développeur de l'application [Hugues *et al.*, 2005].

Discussion Bien que son architecture en couches lui permette d'être statiquement configurable, la configuration de POLYORB est réalisée à l'exécution de l'application en utilisant des mécanismes comme l'aiguillage dynamique pour pouvoir sélectionner l'instance appropriée de chaque service. Tous les services spécialisés dans les personnalités applicatives et protocolaires sont implantés sous la forme d'objet héritant de parents existant dans la couche neutre de POLYORB.

Ceci est dû essentiellement à l'absence, dans POLYORB, d'un langage de modélisation suffisamment détaillé pour décrire plus en détail les propriétés d'une application permettant ainsi sa configuration statique. Des travaux antérieurs ont permis d'utiliser l'intergiciel POLYORB pour des applications réparties modélisées à l'aide d'un langage de description d'architecture [Vergnaud, 2006]. Cependant, la configuration de l'intergiciel est dynamique dans ce cas aussi et non adaptée aux systèmes TR^{2}E critiques. De plus, elle ne tire pas parti à 100% du gain possible permis par le langage de description d'architecture.

2.3 Description des systèmes répartis

Pour pouvoir automatiser le plus possible la conception d'applications réparties, il faut disposer de moyens pour décrire facilement les éléments dont nous voulons automatiser la production.

Dans cette section, nous présentons divers langages de description et de modélisation utilisés pour définir différents aspects des applications réparties.

2.3.1 CCM

"CORBA Component Model" [OMG, 2006a] est une spécification publiée par l'OMG (*Object Management Group*) pour permettre le développement d'applications réparties à l'aide de composants. Les composants sont définis à l'aide du langage CIDL (*Component Implementation Definition Language*) qui est une extension du langage IDL (*Interface Description Language*) de CORBA.

Description

Un composant est une collection de fonctionnalités (*features*) destinée à accomplir une sémantique donnée. Il est inclus dans un objet CORBA. Les composants réalisant des fonctionnalités proches sont regroupés sous forme de bibliothèques.

Les composants CCM peuvent interagir en connectant leur ports respectifs. Cinq types de ports existent :

Les facettes sont des interfaces (au sens CORBA du terme) offertes par le composant pour être utilisées par des clients,

Les réceptacles sont des points d'entrée indiquant que le composant peut accéder à des références CORBA fournies par des entités extérieures,

Les sources d'événements indiquent que le composant est une source potentielle d'un événement d'un type donné qui peut être destiné à un ou plusieurs consommateurs,

Les destinations d'événements indiquent que le composant est un consommateur d'un événement d'un type donné,

Les attributs sont des valeurs exposées par le composant par l'intermédiaire d'accesseurs (`Get_` et `Set_`). Ils sont le plus souvent utilisés pour configurer le composant.

CCM propose un processus de déploiement bien précis qui décrit les différentes étapes de la sélection, le placement, la configuration et l'exécution des composants d'une application répartie. Ce processus est réalisé à l'aide d'une "application de déploiement" grâce aux informations extraites du modèle.

Discussion

Les composants CCM sont implantés en utilisant des patrons de conception issus du monde de l'orienté-objet (usine à objets, héritage et aiguillage multiples...). Ils sont instanciés et configurés en allouant dynamiquement de la mémoire.

Dans le contexte des systèmes temps-réel critiques, cette manière de configuration et de déploiement est potentiellement nuisible au déterminisme et à la sûreté de fonctionnement de l'application. En effet, la plupart des compilateurs des langages orientés-objet produisent implicitement des structures de données lors de la compilation du code [Comar and Porter, 1994]. Pour supporter l'aiguillage dynamique par exemple, les compilateurs construisent des "tables d'aiguillage". Ces structures intermédiaires échappent à toute analyse. Nous reviendrons sur ce problème plus en détails lorsque nous parlerons de l'inadéquation avec les systèmes critiques dans la section 2.6.1.

De plus, le formalisme CIDL ne permet pas de spécifier des détails non-fonctionnels importants tels que la nature d'un *thread* (périodique, sporadique...) et sa période. Ces caractéristiques sont d'une importance cruciale si nous souhaitons effectuer une analyse statique d'ordonnancement. Pour pouvoir utiliser CCM pour les systèmes TR^{2}E, le seul moyen de contourner cette limitation est de définir des sémantiques non standards pour certains composants afin de pouvoir représenter de telles caractéristiques. Ceci compromet l'interopérabilité entre les différents outils supportant le standard.

2.3.2 UML/MARTE

Modéliser les applications, analyser leurs modèles et ensuite générer une quantité de code pour ces applications est une approche qui a été adoptée par l'OMG sous le nom de MDA (*Model Driven Architecture* [OMG, 2003]). Cette approche est réalisée en utilisant le langage de modélisation UML [OMG, 2007b].

Description

À ses débuts, UML avait pour but de faire la synthèse des méthodes orientées-objet. Il a ensuite été étendu pour permettre la modélisation, l'analyse, le développement, le déploiement et la configuration d'applications réparties en appliquant l'approche MDA :

1. Un modèle indépendant de la plate-forme matérielle doit être construit (PIM, *platform independant model*). Il s'agit d'une description de haut niveau représentant une vision abstraite de l'application,
2. Le modèle PIM, analysé et validé, est transformé en un modèle qui est dépendant de la plate-forme d'exécution de l'application (PSM, *platform specific model*),
3. Le modèle PSM, analysé et validé, est utilisé pour générer le code l'application.

Les différentes actions qui doivent être effectuées pour aboutir à une application prête à être exécutée sont effectuées à l'aide de syntaxes UML appropriées, appelée diagrammes du fait de leur nature graphique. Il existe des diagrammes de classes, de paquetages, de déploiement, de cas d'utilisation... Toutes ces syntaxes sont généralistes et ne permettent pas de décrire des aspects très spécialisés des applications. Pour ceci, UML introduit la notion de profil qui est un enrichissement de la sémantique des entités des différentes syntaxes du langages afin de couvrir des domaines donnés.

Pour permettre le développement et l'analyse d'applications TR^{2}E à l'aide de UML, un nouveau profil, MARTE [OMG, 2007c], a été proposé et est actuellement dans les dernières phases d'adoption par l'OMG.

MARTE

MARTE(*Modeling and Analysis of Real-Time and Embedded systems* [OMG, 2007c]) spécialise UML en introduisant les fondements de la modélisation des systèmes temps-réel et embarqués. Ces concepts de base sont ensuite raffinés à la fois pour des raisons de modélisation et d'analyse. La partie modélisation fournit le support requis pour une spécification détaillée des systèmes temps-réel embarqués. MARTE permet aussi une analyse des modèles construits.

L'objectif de MARTE n'est pas de définir de nouvelles techniques d'analyse des systèmes temps-réel embarqués, mais plutôt de soutenir les techniques existantes. Ainsi, il prévoit des facilités pour annoter les modèles avec des informations nécessaires pour effectuer une analyse

avec des outils spécifiques. MARTE est axé particulièrement sur la performance et l'analyse d'ordonnancement des systèmes temps-réel embarqués. Mais il définit aussi un canevas général d'analyse qui tend à raffiner et spécialiser les autres types d'analyses.

MARTE permet d'avoir un moyen commun pour modéliser à la fois la partie matérielle et la partie logicielles des systèmes temps-réel embarqués. Il permet l'interopérabilité entre les différents outils utilisés lors de la spécification, conception, vérification et génération de code. Il permet de modéliser les propriétés non fonctionnelles des systèmes temps-réel embarques comme les caractéristiques temporelles des threads ou les fréquences de processeurs.

MARTE introduit la notion de "composant" en raffinant les classes structurées de UML. Un composant MARTE est une entité autonome du système qui peut avoir à la fois des données et un comportement. Un composant peut disposer de propriétés et de ports pour spécifier explicitement son interaction avec son environnement extérieur. La conception des ports ont été fortement influencée par la notion de même nom existant dans divers langages de description d'architecture comme AADL.

Enfin, MARTE introduit un modèle d'allocation des éléments logiciels du système sur les éléments matériels de l'architecture sous-jacente tout en respectant les contraintes temporelles et topologiques de l'application.

Discussion

Dans [Demathieu *et al.*, 2008], les auteurs et les initiateurs du projet de standardisation de MARTE, présentent une étude de cas pour tester les capacités de modélisation du profil. Il s'agit d'un exemple issu du monde de la robotique. L'utilisation de MARTE s'avère utile et l'implantation de l'étude de cas est menée à bien.

Toutefois, le processus de production décrit est compliqué. En effet, l'approche de développement adaptée dans MARTE et plus généralement dans UML utilise des syntaxes différentes pour décrire chacun des aspects d'une application. Ceci requiert la consolidation de toutes les représentations si le modèle de l'application est modifié ce qui pénalise le temps de développement.

Aussi, peu d'outils implantent MARTE et l'intégration avec les outils d'analyse n'est pas encore effectuée ce qui rend très limitée l'exploitation de MARTE pour des fins d'analyse.

2.3.3 Ada/DSA

Le langage Ada a été utilisé, depuis la version Ada 95, pour supporter les applications réparties [Working Group, 2005, Annexe E]). Il permet à certaines entités du langage d'être distribuées sans pour autant modifier la syntaxe du langage. Des **pragmas** sont insérés à différents endroits du code de l'application pour indiquer les caractéristiques des entités permettant ainsi à l'application d'être développée et testée comme une application monolithique avant de procéder à sa distribution. Ainsi, les unités de compilation peuvent être qualifiées de **Remote_Call_Interface** pour indiquer que les sous-programmes qu'elles exposent peuvent être invoqués à distance. Les données partagées entre plusieurs nœuds sont qualifiées de **Shared_Passive**. Enfin les types de qui sont utilisés pour échanger les données entre les nœuds sont qualifiés de **Remote_Types** pour indiquer que leur déclaration doit figurer sur tous les nœuds de l'application.

Le standard spécifie l'interface du sous-système de communication entre les partitions (PCS, *Partition Communication Subsystem* [Working Group, 2005, Section E.5]). Cette interface gère les communications entre les partitions. Cependant, une grande liberté est laissée aux implantations du standard pour ajouter à ce sous-système des fonctionnalités supplémentaires (répli-

cation, sécurité...). Aussi, la manière dont les applications réparties sont configurées n'a pas été spécifiée par le standard et a été laissée à la charge de l'implantation. Ceci est dû essentiellement à la diversité des architectures visées.

Ce mécanisme a été implanté dans GLADE [Pautet and Tardieu, 2000]. GLADE est un ensemble d'outils qui complètent le compilateur Ada GNAT en offrant les capacités de développer des applications réparties en Ada. Il est formé de **GARLIC** [Kermarrec *et al.*, 1995], une bibliothèque qui implante toutes les fonctionnalités requises pour un PCS et de **GNATDIST**, un outil qui configure les partitions d'une application répartie donnée.

GNATDIST

GNATDIST [Kermarrec *et al.*, 1996] est un outil qui prend en entrée une description de la configuration de l'application dans un langage dédié (proche de Ada). Cette description indique les partitions de l'application répartie, leurs emplacements physiques respectifs, la nature des communications, le profil de concurrence... La partie liée au déploiement et la configuration se trouve ainsi indépendante de la partie implantation et le même code source Ada utilisé conjointement avec des fichiers de configurations différents peut donner lieu à des applications réparties complètement différentes.

Discussion

La nature fortement liée au langage Ada de **GNATDIST** permet de configurer et déployer uniquement des applications réparties écrite dans ce langage. Construire des nœuds qui contiennent du code dans un autre langage de programmation n'est certes pas impossible mais demande beaucoup plus de travail, en particulier si ce code concerne la partie communication (sous-programmes distants, données partagées).

Par contre, les concepts utilisés dans GLADE et dans **GNATDIST** peuvent être adaptés pour pouvoir configurer des applications réparties en utilisant d'autres mécanismes. Des travaux ont été effectués pour pouvoir sélectionner un PCS autre que **GARLIC** à l'aide de **GNATDIST**. Ils ont été intégrés dans l'intergiciel schizophrène POLYORB [Quinot, 2003]. D'autres travaux ont essayé de replacer le langage de configuration de **GNATDIST** par un autre langage plus riche (AADL) pour offrir plus de possibilités d'analyse de l'application [Vergnaud *et al.*, 2005].

Ces dernière années, l'intergiciel schizophrène POLYORB, décrit dans la section 2.2.2, est venu prendre la place de GLADE comme le PCS par défaut pour le compilateur GNAT.

2.4 Déploiement et Configuration Automatiques

Pour perfectionner le processus d'automatisation de la conception des applications réparties, divers outils ont été développés pour intégrer les éléments décrits précédemment (les intergiciels et les langages de description/modélisation) et fournir à l'utilisateur un canevas relativement confortable pour concevoir ses applications. Ce processus a été normalisé par l'OMG qui a publié un ensemble de recommandations dans sa spécification de déploiement et de configuration (D&C [OMG, 2006b]). Cette spécification propose six phases pour lancer une application répartie.

1. L'empaquetage consiste à rassembler une suite de modules binaires et d'éventuelles métadonnées,

2. L'installation consiste à peupler des endroits bien déterminés par les paquetages de la phase 1 (généralement les nœuds de l'application),
3. La configuration est la paramétrisation des paquetages pour satisfaire la sémantique et l'exigence de l'application,
4. Le plan est un ensemble de décisions concernant le déploiement proprement dit des paquetages (exécution sur des processeurs...),
5. La préparation est le déplacement physique des binaires vers les entités décidées lors de la phase 4,
6. Le lancement est le déclenchement des binaires installés.

Le processus entier est désigné par "déploiement" tandis que la "configuration" se résume à la 3^{e} étape de ce processus. Pour accentuer l'aspect statique qui caractérise les systèmes TR^{2}E que nous désirons concevoir, nous avons adapté ces définitions comme suit :

Définition 2.3 ***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 destinations (*squelettes*).*

Cette tâche peut très facilement induire en erreur et ne devrait pas être effectuée manuellement par le développeur. Dans la plupart des cas, un déploiement correct requiert que les nœuds de l'application sachent "comment" atteindre les autres nœuds avec lesquels ils communiquent. Dans les systèmes critiques, cette information doit être connue à l'initialisation pour que tous les canaux de communications puissent être ouverts au tout début de la vie de l'application répartie. Pour ce faire, une "table de nommage/routage" doit être définie pour chaque nœud pour qu'il puisse récupérer les informations de la couche transport et communiquer avec les autres nœuds.

Définition 2.4 ***Configuration***
La configuration d'une application répartie est l'opportunité de paramétrer individuellement chacun des composants intergiciels sélectionnés lors de la phase de déploiement (définition 2.3) et de les assembler par la suite avec les composants générés automatiquement ainsi que les composants fournis par l'utilisateur.

Par exemple, déterminer et fixer la taille des tampons de communication pour les requêtes entrantes d'un nœud donné est un paramètre de configuration.

Dans les systèmes embarqués, les ressources matérielles sont très limitées. Par conséquent, la phase de configuration doit optimiser l'utilisation de ces ressources en allouant uniquement les quantités requises.

Dans la suite, nous présenterons certains des outils de déploiement et de configuration qui sont les plus proches du domaine des systèmes TR^{2}E.

2.4.1 FRACTAL

FRACTAL est un modèle de composants permettant de concevoir et de déployer des systèmes logiciels complexes comme les intergiciels ou les systèmes d'exploitation. Il permet aussi d'administrer les systèmes construits en effectuant par exemple de la configuration dynamique et de l'observation ou du contrôle.

Description

FRACTAL [Coupaye *et al.*, 2007] se situe dans la lignée de l'intergiciel générique JONATHAN [Dumant *et al.*, 1998]. Il vise à produire simplement des systèmes répartis complexe à partir de leur description architecturale. FRACTAL est fondé sur les quatre principes suivants autour de la notion de composant :

1. Composants composites : les composants peuvent contenir d'autres composants. Ceci permet de définir des applications d'une manière hiérarchique et uniforme,
2. Composants partagés : un composant peut appartenir à plusieurs composites. Ceci permet de modéliser le partage des ressource tout en préservant l'encapsulation des composants,
3. Capacités d'introspections : pour permettre l'observation de l'exécution du système. Chaque échange d'information à l'interface d'un composant peut être intercepté,
4. Capacités de (re)configuration : pour permettre de déployer et configurer le système dynamiquement.

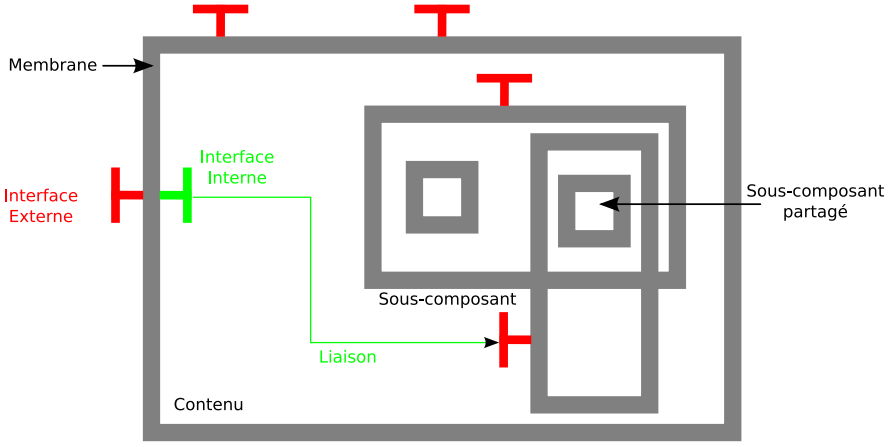

FIGURE 2.3 – Exemple de composants FRACTAL

Un composant FRACTAL [Bruneton *et al.*, 2002] est une unité d'exécution possédant une membrane, des interfaces et un contenu. La figure 2.3 décrit un décrit un ensemble de composants FRACTAL. Un composant peut être primitif (ne contenant aucun autre composant) ou composite (contenant d'autres composants). La membrane d'un composant est formée de contrôleurs. Les contrôleurs pilotent plusieurs aspects du composant (visibilité des interfaces, cycle de vie du composant...). Les interfaces d'un composant peuvent être fournies (serveur) ou bien requise (client). Les interfaces externes (en rouge sur la figure) permettent de lier les sous-composants d'un même composite. Les interface internes (en vert) permettent de lier un composite à ses propres sous-composants.

Les composants FRACTAL peuvent être imbriqués dans d'autres composants. Il peuvent être liés en spécifiant des chemins entre leurs interfaces. Enfin, les composants FRACTAL peuvent être partagés entre plusieurs composites. Cette notion de partage permet de modéliser les ressources. Les composants peuvent avoir un type optionnel.

Trois acteurs principaux interviennent dans le processus de développement de FRACTAL : (1) les *concepteurs d'applications* construisent des composants et des assemblages de composants en utilisant les politiques de contrôle et des supports d'exécution existants. (2) les *concepteurs de contrôleurs* spécifient des nouvelles les politiques de contrôle offertes aux composants.

Ils permettent d'adapter les applications construites aux différents contextes (embarqué, temps réel...). (3) les *concepteurs de plates-formes* fournissent des implantations du support d'exécution de FRACTAL dans un langage de programmation donné.

La construction des applications en FRACTAL se fait en décrivant l'ensemble des composants et leurs interactions dans un langage de description d'architecture. Un générateur de code permet de produire les implantations des composants et de leur interfaces en utilisant les entités fournies par la plate-forme et par l'utilisateur.

FRACTAL ADL

FRACTAL ADL est le langage de description d'architecture utilisé pour spécifier les composants en FRACTAL. Il est fondé sur le langage XML. FRACTAL ADL est extensible car il permet la définition de nouveaux types de contrôleurs d'interface.

Une description en FRACTAL ADL permet de préciser pour chaque composant son contenu, ses contrôleurs et ses interfaces. Pour les composants primitifs, le contenu est un objet Java fourni par l'utilisateur. Pour les composites, le contenu est spécifié en terme d'autres composants FRACTAL.

Plusieurs plates-formes ont été développées pour FRACTAL. JULIA [Bruneton *et al.*, 2006] est l'implantation de référence de FRACTAL. Il s'agit d'un canevas écrit en Java pour développer des applications réparties. JULIA offre les outils nécessaires pour programmer les membranes des composants (contrôleurs). JULIA permet de produire automatiquement le code des composants à partir de leurs descriptions données en FRACTAL AADL. Ce code utilise les éléments de la plate-forme ainsi que les objets fournis par l'utilisateur.

Discussion

FRACTAL étant un processus extensible, aucune vérification statique de cohérence des composants n'est effectuée. Chacun des objets implantant les composants de l'application est chargé à l'exécution. La vérification de la cohérence de l'assemblage est effectuée à l'exécution. Ceci rend l'utilisation de FRACTAL délicate dans le cadre des systèmes TR^{2}E critiques.

Par ailleurs, le langage FRACTAL ADL est un langage *ad hoc* et non pas un standard. De plus sa syntaxe est extensible. Ceci limite la possibilité d'interfacer des outils tiers d'analyse pour vérifier le bon fonctionnement des applications.

Enfin, l'utilisation de XML pour exprimer le langage FRACTAL ADL compromet le passage à l'échelle lors de la conception d'applications répartie très grandes. En effet, la taille des modèles XML pour de telles applications atteint rapidement des tailles très grandes rendant leur manipulation et analyse très coûteuse en temps et en performance.

2.4.2 COSMIC

COSMIC : *Component Syntheses using Model Integrated Computing* [Lu *et al.*, 2003] est une suite d'outils pour concevoir des applications réparties temps-réel embarquées en se basant sur la spécification CCM de l'OMG. Il repose sur le paradigme MIC (*Model Integrated Computing*) [Sztipanovits and Karsai, 1997] et suit les recommandations données par la spécification D&C de l'OMG.

Description

L'implantation de COSMIC est fondée sur l'intergiciel CIAO, une implantation temps-réel de CORBA/CCM qui utilise l'intergiciel TAO (section 2.2.1). Pour décrire le processus de déploiement et d'assemblage de composants, COSMIC utilise le langage CADML de CCM (voir la section 2.3.1) sous une forme graphique. COSMIC permet de spécifier les contraintes sur les composants de l'application et peut éventuellement supporter un module d'analyse pour les modèles. Par défaut, il supporte l'intégration d'applications CORBA [OMG, 2004] développées selon le modèle CCM.

Les descriptions de l'assemblage des composants sont générées à partir de modèle CADML sous la forme d'un fichier XML représentant un scénario possible d'exécution de l'application.

Pour configurer les applications réparties, COSMIC utilise un langage spécifique : OCML (*Options Configuration Modeling Language*) qui est aussi un langage graphique (représenté en interne par XML). Il permet de définir les dépendances entre différentes options qui sont à sélectionner parmi une multitude d'options de l'intergiciel.

Dans [Gokhale *et al.*, 2002], les auteurs montrent comment l'utilisation de COSMIC permet d'assembler des composants logiciels et matériels préfabriqués tout en assurant une compatibilité entre leurs différents paramètres QdS et un déploiement correct de l'application. Si cette tâche est effectuée manuellement, les risques de commettre des erreurs deviennent élevés. Le passage à l'échelle devient très faible surtout avec des applications TR^{2}E de plus en plus grandes et complexes. De plus, cela compromet la possibilité de validation et vérification de l'application.

Dans [Shankaran *et al.*, 2007] sont présentés les bénéfices de l'utilisation d'un intergiciel à composants comme CIAO à la place d'un intergiciel à objets répartis classique. Les intergiciels à composants offrent plusieurs capacités qui permette de séparer le développement de l'application de son déploiement et configuration comme par exemple la standardisation des interfaces entre les composants de l'application et la possibilité d'utiliser des outils de modélisation pour déployer et connecter les composants.

Pour modéliser les composants proprement dits, un langage spécifique est utilisé : PICML (*Platform Independant Component Modeling Language*). Il permet d'effectuer les actions suivantes pour construire un système TR^{2}E :

1. Modéliser les interfaces des composants et composer les applications en connectant ces interfaces,
2. Spécifier les paramètres QdS requis ainsi que les ressources matérielles que l'application consomme,
3. Spécifier le système d'exploitation, l'intergiciel et les paramètres de configuration du réseau,
4. Estimer la disponibilité des ressources du système TR^{2}E,
5. Allouer les ressources,
6. Produire les informations relatives au déploiement de l'application.

L'inconvénient de l'approche "objet réparti" par rapport à celle décrite dans [Shankaran *et al.*, 2007] ("intergiciels orientés composants") est que la première "estime" les ressources de l'application qui sont à allouer et ne vérifie pas la véracité des estimations. COSMIC propose une solution pour vérifier et raffiner le cas échéant les allocations de ressources : RACE (*Resource Allocation and Control Engine*). Il s'agit d'un canevas placé au dessus de l'intergiciel orienté composants CIAO. Il permet de contrôler et de raffiner l'utilisation des ressources. Pour chacune

des 6 étapes décrites plus haut, RACE fournit un composant qui s'occupe de la gestion de cette étape.

Un ensemble de moniteurs de ressources et de moniteurs de QdS surveillent l'exécution de l'application. Un contrôleur récupère les résultats du monitorat et modifie le comportement de l'application pour que la QdS reste acceptable et les ressources ne soient pas sur-allouées.

Pour pousser encore plus loin l'approche décrite ci-dessus, une chaîne d'outils a été crée : QUICKER (*QUality of service pICKER* [Kavimandan *et al.*, 2007]). Elle comble le fossé qui existait entre les différentes familles d'outils :

1. les outils de spécification fonctionnelle et d'analyse qui permettent de modéliser la structure de l'application et de l'analyser,
2. les outils d'analyse d'ordonnancement qui effectuent différentes analyses temporelles pour vérifier le respect des contraintes temporelles (échéances...) de l'application,
3. les canevas de configuration et de déploiement (typiquement le canevas RACE vu plus haut).

L'article note trois approches pour configurer et déployer une application répartie :

1. L'approche statique : en codant "en dur" tous les paramètres de configuration et de déploiement dans le code de l'application,
2. L'approche semi-statique : en utilisant des méta-données qui sont lues au tout début de l'application pour effectuer la configuration et le déploiement,
3. L'approche dynamique : en modifiant dynamiquement les paramètres de configuration et de déploiement de l'application pendant toute la durée de vie de l'application.

Dans les trois cas, il faut un moyen pour (1) traduire les paramètres de QdS qui sont indépendants de la plate-forme en paramètres de configuration fortement dépendants de la plate-forme, (2) choisir des valeurs valides pour ces paramètres et (3) gérer les dépendances et résoudre les conflits entre les différents paramètres de configuration.

QUICKER améliore le langage PICML avec de nouvelles constructions pour permettre la construction et l'analyse des paramètres QdS systèmes TR^{2}E.

Discussion

Le fait que COSMIC repose sur plusieurs langages différents pour spécifier une application (IDL, CADML, OCML...) oblige à consolider chacun des modèles lors des modifications de l'application et à assurer leur cohérence. Ceci augmente le temps de prototypage des systèmes.

La sur-couche utilisée pour raffiner dynamiquement les propriétés des composants pose un problème pour les systèmes critiques où toutes les ressources doivent être allouées statiquement. Avoir une certaine qualité de service fréquemment peut s'avérer insuffisant dans le cas où un système correct par construction est nécessaire.

De plus l'utilisation de UML et de XML pose des problèmes de passage à l'échelle quand il s'agit de modéliser des applications avec un très grand nombre de composants [Sriplakich *et al.*, 2008].

Tous ces inconvénients limitent fortement l'usage de COSMIC pour les systèmes TR^{2}E où un comportement correct par construction est requis. Par ailleurs, l'utilisation de CCM et de l'intergiciel TAO provoque pour COSMIC les mêmes problèmes que nous avons notés pour ces deux derniers (non respect des règles de codage pour les systèmes critiques, utilisation de la programmation orientée-objet, complexité du processus et du support d'exécution...).

2.4.3 AUTOSAR

AUTOSAR [1] (AUTomotive Open System ARchitecture) est le résultat d'un partenariat entre constructeurs automobiles afin de concevoir un standard ouvert pour les architectures électriques et électroniques des systèmes automobiles. Il tend à augmenter la qualité, la maintenabilité et le passage à l'échelle des logiciels dans le domaine automobile, à favoriser l'usage des outils commerciaux (COTS) et à optimiser le coût et le temps de développement des logiciels [Schreiner and Goschka, 2007].

Motivations

Plusieurs défis ont motivé l'élaboration de cette spécification [AUTOSAR Gbr, 2006] :
- La complexité du processus de développement *ad hoc* : ce processus consiste à construire à partir de zéro les systèmes automobiles. Ceci engendre une perte en temps importante,
- La très forte dépendance vis-à-vis du matériel : celle-ci pousse à redévelopper le même système chaque fois que l'architecture change,
- Le coté compétitif du développement des systèmes automobiles : seules les parties du processus de développement qui sont coûteuses mais non compétitives doivent être adressées.

Description

AUTOSAR s'articule autour de 3 axes :
- La standardisation du format d'échange des spécifications,
- L'abstraction du micro-contrôleur pour éviter de redévelopper tout un système lorsqu'une seule la plate-forme change,
- La standardisation des interfaces des différents composants, la partie compétitive étant réduite à l'implantation des composants.

Un composant AUTOSAR est une unité atomique qui ne peut être répartie. Il contient (1) les opérations et les données qui sont requises et fournies par ce composant, (2) les besoins du composant vis-à-vis de l'infrastructure (accès à des capteurs, bus...) et (3) les ressources du composant (mémoire, CPU...). Deux catégories principales de composants existent : (a) les composants logiciels et (b) les capteurs. Plusieurs instances d'un même composant peuvent exister sur le système.

Comme AADL (voir le chapitre 4), un paquetage AUTOSAR est formé d'une description des composants et leurs implantations en code source ou en objets binaires. L'implantation source doit être indépendante du matériel.

La communication entre composants se fait en connectant leurs ports respectifs. Un port appartient à exactement un composant. Les ports sont fournis (PPorts) ou requis (RPorts).

L'interface définit le mode de communication entre ces composants. Par exemple, l'interface client/serveur définit un ensemble d'opérations qui peuvent être invoquées depuis le client sur le serveur. L'interface envoyeur/receveur définit un ensemble de données qui se font à travers un bus virtuel.

Les composants sont connectés à travers bus virtuel (VFB, *Virtual Functional Bus*). Il s'agit d'un composant permettant de séparer l'implantation de l'application de l'architecture matérielle. L'utilisation de ce bus permet aussi l'intégration des composants dans les toutes premières phases du développement.

1. http://www.autosar.org

Les descriptions des contraintes du système et des unités de contrôle électroniques servent à intégrer automatiquement les composants logiciels dans un réseau d'unités de contrôle électroniques à l'aide d'outils dédiés.

À la phase d'intégration, les fonctionnalités offertes par le VFB aux composants sont remplacée par celles des unités de contrôle (voir plus loin dans cette section) pour dialoguer avec les composants. Ces fonctionnalités sont le plus souvent des outils de délégation qui communiquent avec les programmes de base de l'unité de contrôle.

Le support d'exécution pour AUTOSAR est appelé ECU (*Electronic Control Unit*). Il s'agit d'une interface de programmation définie par le standard pour abstraire toutes les hétérogénéités matérielles. Elle définit une couche logicielle *de base* qui contient les éléments suivants :

- Les services comme la gestion de la mémoire, les protocoles de diagnostic...,
- La communication avec les bus, entrée/sortie...,
- Le système d'exploitation qui est :
  - Configuré et dimensionné statiquement,
  - Conçu pour le temps-réel,
  - Doté d'un ordonnanceur basé sur les priorités,
- L'abstraction du micro-contrôleur pour effectuer des opérations de bas niveau (chiens de garde, conversion analogique/numérique...),
- L'abstraction de l'ECU qui présente une interface logicielle pour toutes les valeurs électriques,
- Les pilotes de périphériques complexes (CDD) pour accéder aux ressources matérielles en cas d'applications critiques.

Une méthodologie est introduite dans [AUTOSAR Gbr, 2006] pour concevoir une application TR^{2}E avec AUTOSAR. (1) Il faut tout d'abord décrire les composants, les ressources et les contraintes en utilisant un méta-modèle basé sur XML. (2) La configuration du système est ensuite déduite de ce modèle. (3) Les ECU sont extraites de cette configuration. (4) Le système est paramétré en utilisant cette configuration et (5) l'exécutable final est produit (en utilisant aussi les implantations des composants fournies par l'utilisateur).

Le *logiciel de base* est vu comme un intergiciel configuré automatiquement selon les propriétés et les exigences de l'application. Le projet COMPASS [Schreiner and Goschka, 2007] étend la sémantique des composants AUTOSAR en introduisant de nouvelles catégories (données...) et propose un modèle orienté composant pour le *logiciel de base*. L'intergiciel final est construit à partir de composants personnalisés pour l'application ainsi que de composants préfabriqués. Les composants de COMPASS sont spécifiés à l'aide du langage UML.

L'utilisation du VFB permet à AUTOSAR de supporter plusieurs types de bus. Pour implanter des systèmes TR^{2}E critiques, un bus temps-réel est nécessaire. Dans [Pimentel, 2007], les auteurs montrent comment ils peuvent améliorer les performances de l'ordonnancement des tâches et des messages dans les systèmes TR^{2}E en utilisant une architecture basée sur le bus FLEXCAN [Pimentel and University, 2006] qui est une adaptation du bus CAN [Fredriksson, 2002] pour les systèmes critiques.

Discussion

Dans [Sangiovanni-Vincentelli and Natale, 2007], les auteurs donnent un ensemble de limitations constatées lors de leur utilisation d'AUTOSAR. Il s'agit plus généralement de problèmes présents dans de nombreux langages de modélisation :

1. Manque de séparation entre la partie fonctionnelle et l'architecture,

2. Manque de support pour la définition des tâches et des ressources qu'elles utilisent,
3. Manque de support pour l'analysabilité et pour des retro-annotations dans le cadre d'un processus itératif de conception. Par ailleurs, pour intégrer le code de l'utilisateur, AUTOSAR génère des squelettes à remplir. Or, la possibilité d'édition automatique de squelettes modifiés par l'utilisateur est laissée à la charge des implantations [AUTOSAR Gbr, 2006, Section 3.2]). À chaque modification du modèle, les nouveaux squelettes générés doivent être remplis ou au moins édités,
4. Manque de préservation de la sémantique après la génération de code. Il faut noter ici que le code est généré dans le langage C sur lequel la plupart des compilateurs n'effectuent pas des analyses poussées de sémantique et de cohérence.

Selon la même étude, deux propriétés fondamentales doivent être garanties par les outils de modélisation :

– La composition externe [Lam and Shankar, 1994] qui garantit, pour un système, la "satisfaction" de ses interfaces si chacun de ses composants "satisfait" sa propre interface. Ceci veut dire que chaque composant doit fournir ce qu'il doit fournir aux autres composants et doit consommer ce qu'il reçoit de leur part,
– La composition interne [Medvidovic and Taylor, 2000] qui permet de modéliser les systèmes avec des niveaux différents de détail en utilisant le même formalisme. Un ensemble de composants peut ainsi être vu comme un seul composant dont les propriétés sont déduites de celles de ces constituants.

AUTOSAR ne satisfait pas ces deux propriétés en raison de son méta-modèle inspiré de UML qui ne fournit pas une sémantique claire pour la communication et la synchronisation. Par ailleurs, AUTOSAR offre une description comportementale incomplète. Il ne fournit pas les moyens pour gérer les aspects temporels ou liés à la performance sous-estimant ainsi la complexité des futurs systèmes.

Tout ceci met en question l'utilisation de AUTOSAR dans le contexte de systèmes de plus en plus complexes. En effet, l'analyse statique de ces systèmes est cruciale et l'absence, dans AUTOSAR, de capacités comme la définition des propriétés des tâches (période...) rend ce genre d'analyses très difficiles à effectuer.

2.4.4 SYNDEX

AAA/SYNDEX [Sorel, 2004] est un canevas formel basé sur la théorie des graphes. Il sert à spécifier les fonctionnalités et l'architecture d'une application répartie, à assister le développeur à implanter ces fonctionnalités et à vérifier que les contraintes temporelles de l'application sont vérifiées tout en essayant de minimiser l'utilisation des ressources.

Description

AAA/SYNDEX offre un environnement graphique qui permet à l'utilisateur d'explorer et d'optimiser aussi bien manuellement qu'automatiquement le modèle de son application. Les premières motivations de AAA/SYNDEX proviennent du domaine du calcul distribué.

La génération de code à partir des spécifications de l'application conduit à la production d'un noyau temps-réel dédié à l'application ou bien à la production d'une configuration dédiée à l'application pour un système d'exploitation temps-réel préexistant. La communication entre les nœuds de l'application est assurée par l'intermédiaire d'un bus CAN.

AAA/SYNDEX peut être facilement intégré à un ensemble de langages spécifiques à différents domaines (AIL [Panday *et al.*, 2001], SCILAB [Gomez, 1998]...) qui permettent de garantir les caractéristiques temporelles de la partie comportementale l'application. Il repose sur la méthodologie AAA [Sorel, 2005] (Algorithmes, Architecture, Adéquation). Cette méthodologie permet de spécifier à l'aide de graphes les fonctionnalités logicielles ainsi que l'architecture matérielle. Elle assure la traçabilité et la cohérence entre les différentes étapes du cycle de développement. Elle permet d'effectuer l'analyse et la vérification formelles ainsi que certaines optimisations. Enfin, elle permet de générer du code correspondant à l'application répartie. Ceci consiste à :

- Spécifier les fonctionnalités de l'application ("Algorithmes"),
- Spécifier l'architecture de l'application, les ressources distribuées et les interfaces avec le matériel ("Architecture"),
- Explorer manuellement ou automatiquement l'espace des solutions d'implantations des algorithmes dans l'architecture en utilisant des heuristiques et choisir une solution qui minimise les ressources utilisées ("Adéquation").

Algorithmes Les algorithme sont modélisés sous la forme de graphes orientés dont les sommets représentent les opérations. Les arêtes représentent les flux de données et les dépendances entre ces opérations. Cette description de l'algorithme est hiérarchique et chaque opération peut à son tour être décrite de la même façon jusqu'à obtenir des opérations élémentaires appelées aussi "opérations atomiques" car elles ne peuvent être scindées. La structure des graphes a été étendue pour supporter la diffusion (les graphes classiques ne permettent que des connexions point-à-point) et les constructions de base pour les algorithmes comme les boucles, les structures conditionnelles et les délais (en temps logique).

Architecture L'architecture est aussi modélisée à l'aide de graphes orientés dont les sommets représentent quatre types de composants : les opérations, les communicateurs (DMA...), les mémoires et les bus/multiplexeurs/démultiplexeurs. Les arêtes du graphe représentent les connexions entre les composants. Cette modélisation est aussi hiérarchique (un microprocesseur peut être vu comme un composant qui est lui même formé d'autres composants). Cette modélisation se situe entre les approches haut niveau contenant très peu de détails (PRAM/-DRAM [Zomaya, 1996]) et les approches très bas niveau qui offrent beaucoup plus de détails qu'il n'est nécessaire dans le contexte du travail [Mead and Conway, 1979].

Adéquation L'adéquation se divise en deux sous-phases :

1. L'implantation : elle consiste à allouer des opérations sur les opérateurs physiques. Ceci conduit à un ensemble de "partitions" et induit aussi l'allocation de l'espace mémoire nécessaire pour exécuter les programmes. L'implantation consiste aussi à l'introduction de nouvelles opérations (implicites) qui s'occupent de la gestion de l'envoi et de la réceptions des données (souches et squelettes). Ceci permet de déterminer la quantité de mémoire nécessaire pour chaque processeur. Ensuite ces opérations sont ordonnancées sur les opérateurs physiques. L'implantation consiste aussi à (2) la distribution et l'ordonnancement des transferts des données entre les opérations.
2. L'optimisation : plusieurs solutions peuvent être trouvées lors de la phase d'implantation. Il est important d'en choisir une minimisant l'utilisation des ressources tout en respectant les contraintes de l'application. Ce problème est un problème NP-complet. La méthodologie AAA utilise des heuristiques pour le résoudre [Grandpierre *et al.*, 1999].

Dans [Grandpierre and Sorel, 2003], les auteurs montrent comment la méthodologie AAA permet de combiner le prototypage rapide avec l'ordonnancement hors-ligne des applications TR^{2}E en utilisant la transformation de graphes sur des algorithmes spécifiques. Une suite de transformations et d'analyses conduit à partir des modèles applicatifs et architecturaux à la génération d'un exécutif pour l'application TR^{2}E. Les transformations s'opèrent sur des graphes profitant ainsi des outils mathématiques de la théorie des graphes pour montrer la conservation des différentes propriétés de l'application. L'exécutif généré est correct par construction.

Discussion

SYNDEX-AAA offre un canevas complet pour construire les applications TR^{2}E et produire automatiquement des intergiciels dédiés à leurs besoin. SYNDEX utilise un mode synchronisé pour la communication entre les nœuds ce qui rend l'analyse d'ordonnançabilité très pessimiste. En effet, l'approche synchrone rend l'analyse statique de ces systèmes très compliquée car le pire temps d'exécution sur chacun des nœuds d'une application répartie va dépendre non seulement des propriétés du nœud en question, mais aussi de celles des nœuds auxquels il est connecté.

Ceci peut être toléré pour le domaine du calcul parallèle, domaine principal des applications construites à l'aide de SYNDEX. Cependant, si l'on veut construire des systèmes TR^{2}E, une analyse pessimiste est souvent inacceptable car elle implique un sur-dimensionnement des ressources du système.

2.5 Systèmes Critiques

Pour qu'un système (réparti ou non) puisse être utilisé dans des domaines qualifiés de critiques (espace, aéronautique, médecine...), il doit respecter certaines exigences garantissant sa sécurité et sa sûreté de fonctionnement [Barnes, 2008]. En particulier, il faut que le déploiement et la configuration du système (section 2.4) soient effectués d'une manière statique qui ne compromet pas le respect des exigences. Dans la suite, nous présenterons deux approches utilisées pour développer des systèmes critiques en Ada.

2.5.1 Le profil Ravenscar

Le profil Ravenscar [Working Group, 2005, Annexe D.13.1] est un ensemble de restrictions pour limiter l'usage du langage Ada ainsi que d'autres langages comme Java [Kwon *et al.*, 2002] aux seules constructions qui garantissent une analyse statique d'ordonnancement, l'absence d'interblocage et une borne temporelle d'inversion de priorités entre les tâches.

Description

Les constructions telles que les *rendez-vous*, les entrées dans les tâches et les entrées multiples d'objets protégés rendent complexe l'analyse statique d'un système puisqu'elles augmentent le nombre de chemins d'exécution et les mécanismes d'interaction entre les tâches. Elles sont interdites dans le profil Ravenscar. Pour garantir que l'ensemble de tâches à analyser soit invariant, celles-ci doivent ne jamais finir et aucune création de tâche ne doit survenir à un moment autre que lors de l'initialisation de l'application (désigné par l'*élaboration* en Ada).

Pour garantir l'absence d'interblocage et borner l'inversion de priorité, l'accès aux objets protégés (seul moyen d'interaction entre tâches en Ravenscar) doit être effectué selon le protocole de plafonnement de priorité (PCP, *Priority Ceiling Protocol*).

Afin de garantir une analysabilité selon RMA ou RTA, le guide Ravenscar [Burns *et al.*, 2004] recommande l'utilisation exclusive de tâches périodiques ou sporadiques. Les tâches périodiques sont les tâches qui sont déclenchées à des intervalles réguliers par un événement temporel. Les tâches sporadiques sont déclenchées à la réception d'un événement extérieur avec une durée minimale connue entre la réception de deux événements successifs.

Discussion

Le profil Ravenscar ne s'applique que sur les applications monolithiques [Urueña and Zamorano, 2007]. Par contre si les communications entre les nœuds utilisent un protocole temps-réel et si toutes ces communications sont asynchrones, ce profil peut être étendu pour les systèmes TR^{2}E.

De plus, le profil Ravenscar ne couvre que les aspects liés à la concurrence. D'autres aspects importants doivent être pris en compte lors de la conception de systèmes critiques (gestion de la mémoire...). Un ensemble de restrictions additionnelles pour les systèmes critiques a été ajouté au standard Ada 2005 [Working Group, 2005, Annexe H]. Il s'agit de restrictions indépendantes interdisant l'utilisation des constructions du langage Ada jugées compromettantes pour les systèmes critiques (allocation dynamique de mémoire, aiguillage dynamique, nombre à virgules flottantes...). Le développeur sélectionne un sous-ensemble de ces restrictions à être supportées par son application. Le compilateur Ada permet de garantir le respect des restrictions à la compilation. Par conséquent, pour qu'une application TR^{2}E soit statiquement analysable, elle doit respecter non seulement le profil Ravenscar, mais aussi les restrictions pour les systèmes critiques qui ont été sélectionnées.

2.5.2 SPARK

Le respect du profil Ravenscar n'est pas suffisant pour garantir la sécurité et la sûreté de fonctionnement d'un système. Le flux de données doit être analysé et certaines propriétés fonctionnelles doivent être prouvées. Plus généralement, certaines situations imposent que le système soit correct "par construction".

Description

L'approche SPARK [Barnes, 2003] vise à développer des programmes corrects Ada grâce aux techniques utilisées pour leur construction. Elle est utilisée dans les domaines de l'avionique, des banques et des chemins de fer. Le langage de programmation utilisé pour développer des applications SPARK est un sous ensemble du langage Ada dont les constructions sont "annotées". Les annotations expriment des contraintes de précédence, de droit d'accès et divers autres aspects pour décrire les flux d'exécution et de données.

Des outils spécifiques analysent ensuite le code source Ada et vérifient la conformité du code aux annotations. Ceci permet d'avoir un programme correct "par construction" grâce à des théorèmes établis et prouvés.

Discussion

De manière similaire au profil Ravenscar, SPARK ne gère pas les applications réparties. Étendre ce langage à cette catégorie de systèmes devrait être possible avec les mêmes réserves que celles pour Ravenscar.

SPARK garantit certes d'avoir une application correcte par construction. Mais cette application correcte le sera uniquement vis-à-vis des annotations ajoutées au code. Pour pouvoir garantir un système correct par construction, il faut que les annotations soient complètes, correctes et cohérentes. Ceci pourrait être réalisé en générant automatiquement du code Ada garni par des annotations SPARK à partir d'une description de l'application et de ses contraintes.

Par ailleurs, SPARK ne supporte pas les constructions génériques en Ada. De telles constructions permettent de factoriser des grandes quantités de code en définissant des archétypes pour les instancier. Les constructions génériques n'affectent en rien l'analysabilité statique des applications et l'absence de leur support limite considérablement l'utilisation de SPARK.

2.6 Limites des Solutions Existantes

Dans cette section, nous donnons les limites et inconvénients qui empêchent l'utilisation des solutions présentées plus haut dans le cadre de notre travail. Deux problèmes principaux s'imposent : (1) l'inadéquation de la plupart de ces solutions dans le contexte des systèmes TR^{2}E critiques et (2) l'automatisation uniquement partielle du processus de production. Ce que nous voulons réaliser est un processus complètement automatique pour produire des systèmes TR^{2}E critiques.

2.6.1 Inadéquation aux systèmes TR^{2}E critiques

Une grande partie des standards de répartition et des intergiciels qui les supportent, y compris ceux qui sont présentés comme dédiés aux systèmes répartis temps-réel (RT-CORBA), sont fondés sur des patrons de conception empruntés au paradigme orienté-objet [Schmidt *et al.*, 2000]. Les composants sélectionnés lors du déploiement sont instanciés dynamiquement en utilisant des "usines à objets" et l'interaction entre ces composants est généralement assurée en utilisant de l'aiguillage dynamique. Par ailleurs, la paramétrisation de ces composants lors de la configuration est effectuée le plus souvent au moment de l'exécution. Elle utilise l'allocation dynamique de mémoire. Le comportement de l'intergiciel s'approche de celui d'un interpréteur de code dans le sens où une grande quantité d'actions possibles existe et que l'action convenable est choisie dynamiquement en fonction des propriétés de l'application. Par exemple, chaque fois qu'un nœud de l'application répartie envoie ou reçoit des messages, la taille des tampons de communication est calculée "à la volée" et le tampon est alloué dynamiquement ou, pire encore, le tampon de communication est construit progressivement par une série d'allocations de mémoire. À la fin de la communication, le tampon est désalloué.

Cette approche, que l'on peut qualifier de "dynamique" possède l'avantage de rendre le code de l'application (celui fourni par l'utilisateur mais aussi celui généré automatiquement) plus simple d'un point de vue d'un concepteur d'intergiciel et permet de construire rapidement des générateurs de code. Cependant cette approche présente deux inconvénients majeurs qui empêchent son utilisation dans le contexte des systèmes critiques :

1. La programmation orientée-objet et en particulier l'aiguillage dynamique contient de nombreuses failles de sûreté de fonctionnement. Par exemple, il est difficile de garantir une

initialisation correcte des tables d'aiguillages, moyen le plus répandu pour implanter l'aiguillage dynamique dans les langages de programmation compilés [Gasperoni, 2006]. Dans [FAA, 2004], les auteurs posent un ensemble de questions quant à la compatibilité de la technologie orientée-objet avec les systèmes. Ils présentent un ensemble de problèmes pour montrer la difficulté de développer des systèmes TR^{2}E compatibles avec des spécifications comme DO-178B [RTCA and EUROCAE, 1992]. En particulier, l'utilisation de la technologie orientée-objet complique l'analyse du flux de données dans un programme. L'introduction de l'héritage et du polymorphisme rend les interactions entre les données plus complexes et demande par conséquent un nombre supplémentaires de tests pour vérifier le bon fonctionnement du système. L'utilisation de l'aiguillage dynamique engendre des situations où des variables encore non initialisées sont lues tout en échappant aux différentes analyses destinées à prévenir cela. Des analyses comme la recherche de code mort et des tests comme les tests de couverture deviennent extrêmement difficiles à réaliser du fait du volume et de la complexité du code produit par le compilateur, à l'insu de l'utilisateur et de l'outil d'analyse, lors la phase d'expansion. Un tel code échappe à tout contrôle de sémantique et à toute sorte d'analyse.

2. L'allocation dynamique de mémoire rend non déterministe l'application à partir du moment où la désallocation et la réallocation sont autorisées. En effet ces deux dernières actions causent la fragmentation de l'espace mémoire et le temps de recherche d'espace libre dans un bloc de mémoire fragmenté est difficilement prévisible. Ceci peut causer un problème lors du calcul du pire temps d'exécution (*worst case exécution time*, WCET) dans un système temps-réel.

Des recommandations sont apparues pour rendre les systèmes avioniques développés en orienté-objet compatibles avec DO-178B. [FAA, 2004, Volume 3] liste ces recommandations. L'aiguillage dynamique et le polymorphisme sont interdits. Si le développeur veut utiliser de l'orienté-objet pour bénéficier de ses avantages (encapsulation, héritage...), il doit implanter "à la main" le comportement équivalent à l'aiguillage dynamique et au polymorphisme.

Toutes ces recommandations montrent que l'utilisation de la programmation orientée-objet en tant que telle n'affecte pas la sûreté de fonctionnement des systèmes TR^{2}E. Par contre les fonctionnalités qui accompagnent généralement ce mode de programmation induisent une approche dynamique qui compromet la sûreté de fonctionnement de l'application. Idéalement, nous voulons un moyen de modélisation qui offre les avantages de la programmation orientée-objet (composants, encapsulation, héritage...) et qui préserverait l'aspect statique de l'application.

2.6.2 Non passage à l'échelle

Certains des outils que nous avons décrits dans la section 2.4, à savoir COSMIC et AUTOSAR utilisent des formalismes issus du langage UML, et stockent les modèles en utilisant le langage XML. Ceci compromet le passage à l'échelle de tels outils quand il s'agit de modéliser des systèmes volumineux (quelques milliers de composants). En effet, la taille de la représentation interne en XML peut atteindre 100 fois la taille du modèle dans son formalisme initial. Pour pouvoir traiter des modèles volumineux, un moyen plus efficace que XML est nécessaire pour représenter les arbres et stocker les données. Nous voulons que ce moyen soit compact aussi bien du point de vue formalisme que celui de la représentation interne de ce formalisme.

2.6.3 Automatisation partielle

Ces solutions ne rendent pas automatique tout le processus de production. Notre but consiste à fournir un processus de production automatique ayant pour résultat un ensemble de binaires prêts à être exécutés. FRACTAL, COSMIC et AUTOSAR génèrent automatiquement du code et configurent automatiquement les composants intergiciels en fonction des propriétés du modèle, mais ils ne permettent pas d'intégrer des composants utilisateurs générés automatiquement à partir d'outils tiers. La compilation et le déploiement du code généré sont à la charge de l'utilisateur.

Quant à SYNDEX, bien qu'il offre un processus automatique de production, il s'oriente plutôt vers le domaine du calcul distribué et la répartition d'algorithmes. Le langage qu'il utilise pour modéliser les applications est, d'une part un langage personnalisé et non pas un standard et d'autre part, il n'offre pas toutes les capacités et l'expressivité des langages de description d'architecture.

Enfin, la plupart de ces solutions ont en commun d'utiliser des formalismes différents pour chacune des étapes du processus de production. Avoir un seul formalisme utilisé lors de toutes les étapes de la production est un avantage considérable car il simplifie les outils de développement. Toutefois il faut que ce formalisme possède la caractéristique de "composition interne" définie lors de la discussion de la section 2.4.3. Ainsi les modèles seront-ils plus clairs et plus compréhensibles.

2.6.4 Synthèse

Aucun des outils présentés dans ce chapitre ne décrit un processus de production complet et automatique intégrant la modélisation, l'analyse et la génération à la fois du code et de l'intergiciel dédiés pour les systèmes TR^{2}E. Il existe certes des éléments dans certains des travaux décrits qui vont nous être d'une grande utilité pour mener à bien ce travail de thèse. L'architecture schizophrène, par exemple, semble un très bon candidat pour créer l'intergiciel minimal et générer le reste des composants intergiciels dédiés à une application TR^{2}E. En effet, la définition des services canonique de répartition et l'organisation par couche de l'intergiciel offrent des moyens efficaces pour séparer les différents composants intergiciels afin de les optimiser séparément.

Les moyens de description d'application réparties présentés dans les sections 2.3 et 2.4 permettent certes de modéliser et spécifier les propriétés des applications réparties. Mais certains (**GNATDIST** et CCM) ne permettent pas de définir les caractéristiques du matériel sous-jacent. D'autres (FRACTAL, COSMIC, AUTOSAR) utilisent un langage de description ayant une sémantique pauvre et mal adaptée pour ce genre de modélisation. SYNDEX, quant à lui, utilise un processus de production permettant très difficilement d'intégrer des phases (d'analyse, d'intégration de code utilisateur) produites par les outils autres que ceux du processus lui même ou par des outils alliés.

Nous avons opté pour l'utilisation du langage AADL qui permet à la fois de modéliser précisément la partie logicielle et matérielle des systèmes TR^{2}E. Par ailleurs, grâce à son mécanisme de propriétés, il propose un processus de production assez ouvert et pouvant être facilement enrichi. Le chapitre 4 introduit ce langage et justifiera notre choix.

Enfin, les éléments présentés dans la section 2.5, notamment le profil Ravenscar, vont être utilisés pour renforcer la sûreté de fonctionnement des applications produites.

Nous avons adapté le processus classique décrit au début de ce chapitre (figure 2.1) pour optimiser les composants intergiciels en fonctions des propriétés de l'application TR^{2}E. La figure 2.4 montre le nouveau processus. La différence par rapport au processus classique est

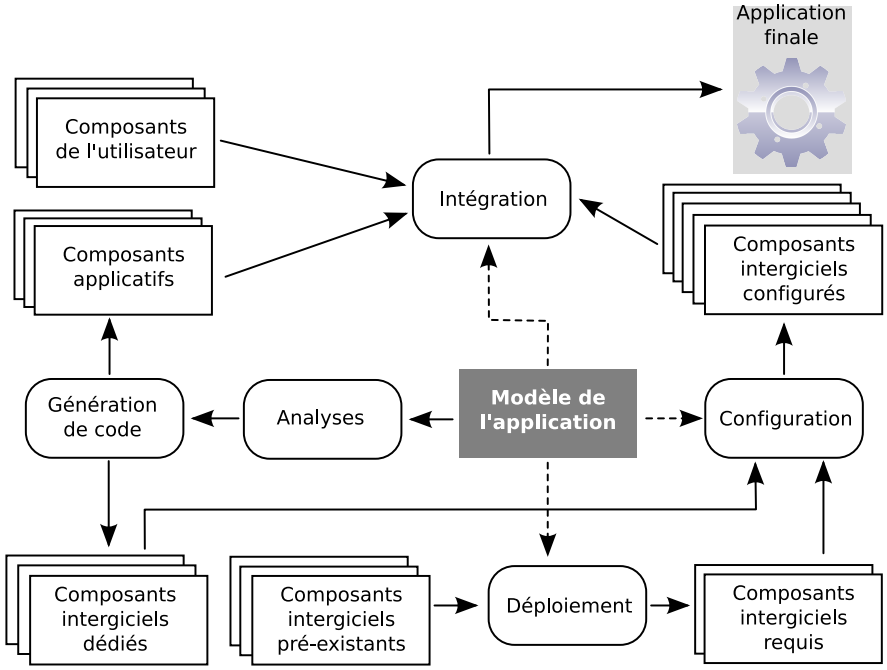

FIGURE 2.4 – Nouveau processus de conception d'une application TR^{2}E

qu'un grand nombre de composants intergiciels sont produits automatiquement à partir du modèle de l'application et sont ainsi dédiés à ses besoins. De plus, une étape d'analyse du modèle de l'application a été intégrée au processus. La suite de ce document décrit en détails chacune des étapes de ce nouveau processus.

Chapitre 3

Proposition d'une Architecture d'un Intergiciel Dédié aux Systèmes TR^{2}E

SOMMAIRE

3.1 Introduction

Dans l'état de l'art, nous avons présenté certaines implantations et outils qui proposent de produire applications réparties. Nous avons aussi noté que ces implantations et architectures ne permettaient pas d'avoir un intergiciel dédié à une application répartie donnée. Ceci est dû

à pour plusieurs raisons (architecture de l'intergiciel inadéquate pour la personnalisation, absence d'un formalisme de modélisation permettant de personnaliser fortement et automatiquement l'intergiciel...). Enfin, nous avons noté le manque de compatibilité entre les spécifications des applications et la plupart des outils d'analyse.

Dans ce chapitre, ainsi que dans les deux chapitres suivants, nous présentons la première partie de la solution aux problématiques citées dans l'introduction générale (Chapitre 1) et dont nous avons montré les limites des solutions existantes (chapitre 2).

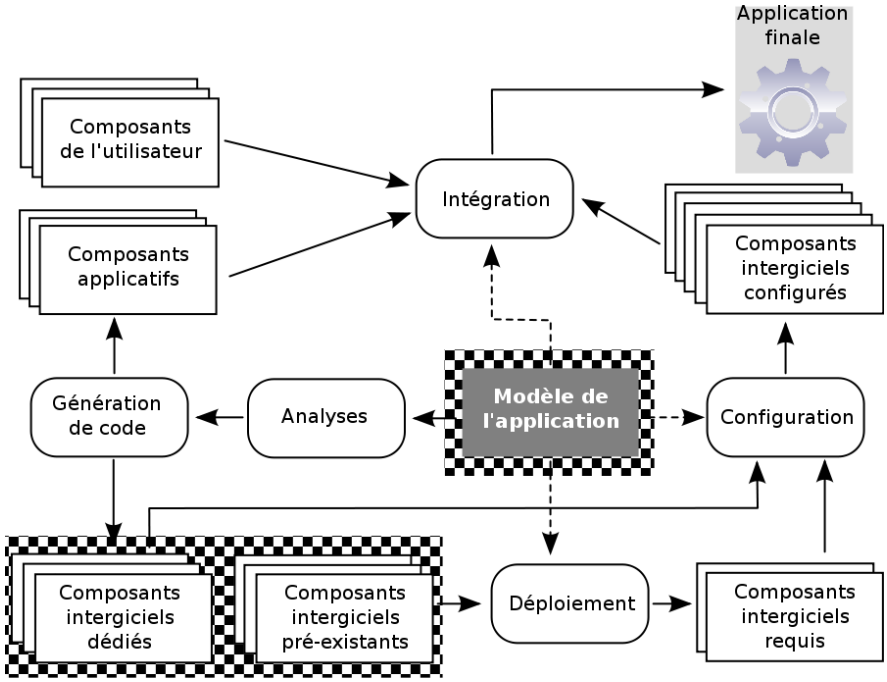

FIGURE 3.1 – Nouveau processus de conception d'une application TR²E

La figure 3.1 met en évidence les étapes du processus de production auxquelles nous nous intéressons dans ce chapitre : la définition d'une architecture d'un intergiciel dédié et le choix d'un formalisme de modélisation pour les applications TR²E. Dans la section 3.2, nous rappelons les services intergiciels canoniques définis par l'architecture schizophrène. Cette architecture est ensuite revue pour l'adapter aux systèmes TR²E et pour avoir un intergiciel spécifique aux besoins de l'application critiques. Certains services sont parfois découpés en deux pour mettre en évidence deux familles séparées de ces services : les services faiblement personnalisables (mais toutefois paramétrables) et les services fortement personnalisables.

La première famille de composants formant cet intergiciel, celle des composants dont l'implantation est indépendante des propriétés de l'application répartie, est présentée dans la section 3.3. Ces composants constituent le noyau minimal de l'intergiciel. La seconde famille, celle des composants qui devront être personnalisés pour chaque application répartie, est décrite dans la section 3.4.

Nous déduisons que la dernière famille de composants doit être produite automatiquement à partir d'une description de l'application. Nous constatons, que pour produire automatiquement ces composants et configurer ceux de l'intergiciel minimal, notre système doit être décrit à l'aide d'un langage de modélisation. La section 3.5 donne les exigences que nous fixons pour un tel langage et justifiera notre choix. La section 3.6 conclut ce chapitre.

3.2 Architecture d'un intergiciel dédié

Une instance d'un intergiciel dédiée à une application répartie, contient uniquement les entités dont cette application a besoin. L'architecture d'intergiciel doit être flexible et modulaire pour que les différents constituants soient configurés et personnalisés finement. L'architecture schizophrène [Pautet, 2002; Quinot, 2003], grâce à la notion de "services intergiciels canoniques" qu'elle introduit, est un excellent candidat pour construire une instance d'intergiciel dédiée. En effet, cette architecture sépare clairement les différents composants de l'intergiciel et ainsi les configure indépendamment les uns des autres. Ces composants sont choisis en fonction des besoins de l'application et des exigences de son environnement. Cependant, deux applications utilisant le même modèle de répartition obtiennent globalement les mêmes intergiciels. Une variante pour un modèle de répartition donné requiert de réécrire une partie de l'intergiciel. Cette architecture doit être revue pour l'adapter aux systèmes TR^{2}E et permettre à chaque application d'obtenir des composants spécifiques à ses besoins.

Dans ce qui suit, nous rappelons les "services intergiciels canoniques" introduits par l'architecture schizophrène. Ensuite nous analysons l'aptitude de ces services à être personnalisés aux besoins d'une application TR^{2}E. Ensuite nous proposons une révision de cette architecture adaptée pour les systèmes TR^{2}E. Cette révision consiste à distinguer et, pour ce faire, éventuellement découper les services fortement personnalisables des services faiblement personnalisables.

3.2.1 Rappel des services canoniques d'un intergiciel

Bien qu'il existe de nombreux standards de répartition qui utilisent —en apparence— des mécanismes très différents pour gérer la transmission de l'information entre les nœuds d'une application répartie, ces standards partagent la mise en œuvre d'un ensemble de services communs : *les services intergiciels canoniques* [Pautet and Kordon, 2004]. L'architecture schizophrène introduit des services "neutres" qui correspondent aux étapes clefs du traitement d'une requête entre deux nœuds de l'application répartie. La figure 3.2 illustre les différents services intergiciels canoniques que nous rappelons ci-dessous.

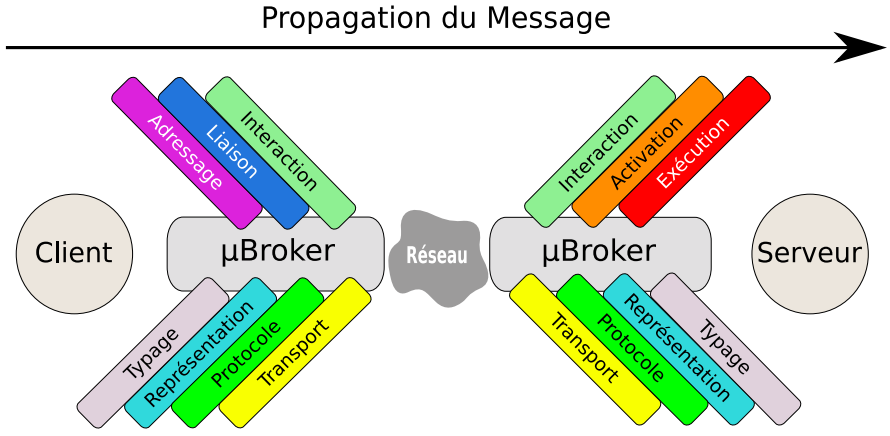

FIGURE 3.2 – Les services canoniques dans un intergiciel

Le *µBroker* [Hugues, 2005] est le composant central de l'intergiciel. Il fournit les supports nécessaires pour l'interaction et la coordination entre les différents services canoniques. Tout le comportement de l'intergiciel est piloté par ce composant. Dans la suite, nous définissons chacun de ces services et nous donnons leurs principales fonctionnalités.

Adressage

Le service d'adressage permet la gestion des références des entités extérieures à l'intergiciel. Une référence est une adresse créée par le serveur qui reçoit les requêtes venant des clients pour permettre de retrouver sans ambiguïté l'entité traitant ces requêtes. Cette adresse permet aux entités de se connecter et d'envoyer des messages à l'entité adressée. La partie de ce service qui crée les références à partir des entités agit exclusivement sur les nœuds logiques recevant les messages tandis que la partie qui retrouve les entités extérieures à partir de leurs références agit exclusivement sur les nœuds logiques envoyant des messages.

Par exemple, un serveur CORBA crée, pour chaque objet qu'il abrite, une référence de type IOR (*Interoperable Object Reference*). Cette référence est utilisée par les clients CORBA pour retrouver l'adresse du serveur (adresse IP dans le cas où la communication se fait en utilisant le protocole IIOP) et l'adresse de l'objet distant dans le serveur.

Liaison

Le service de liaison se situe au dessus du service d'adressage. Il permet de construire, à partir d'une adresse associée, les ressources de communication nécessaires pour dialoguer avec une entité distante ou locale. Dans le cas d'une communication distante, ce service sélectionne les instances requises des services d'interaction, de représentation, de protocole et de transport.

Le mode de communication entre deux entités est aussi géré par ce service. Plusieurs modes de communication existent. Parmi les plus connus, nous trouvons :

- Le passage de message où chaque nœud possède une boîte aux lettres recevant les messages envoyés par les autres nœuds. Le destinataire consulte sa boîte aux lettres pour lire les messages reçus,
- L'appel de procédures distantes où un nœud logique invoque d'une manière distante une procédure localisée sur un autre nœud,
- Les objets distants où un nœud logique invoque des méthodes sur des objets situés sur un autre nœud.

Le choix du mode de communication entre les différentes entités d'une application TR²E influe fortement sur son architecture. En effet, l'échange de message, par exemple, impose la présence de "boites aux lettres" pour recevoir les messages en attente de consommation.

Représentation

Le service de représentation assure la transmission correcte des données à travers le réseau entre l'émetteur et le récepteur. Il contrôle la manière dont ces données sont emballées dans un tampon de communication par l'entité émettrice et déballées de ce tampon par une entité réceptrice. Ce service se justifie du fait que les mécanismes de stockage de données en mémoire et de leur envoi à travers le réseau sont généralement différentes d'un système d'exploitation à un autre ou d'une architecture physique à une autre (systèmes petits boutistes *vs* systèmes gros boutistes, présence *vs* absence d'un alignement automatique lors du stockage des données en mémoire...). Les deux entités échangeant les messages sont parfois exécutées sur des plates-formes hétérogènes stockant et transmettant les données de manières différentes. Dans ce cas, des traitements supplémentaires sont effectués par ce service sur ces données lors de leur échange (alignement, codage...). Ce service garantit l'interopérabilité entre les différents nœuds logiques (CDR [OMG, 2004, 15.2.1], SOAP [Nielsen *et al.*, 2007], XDR [Eisler, 2006]...).

Dans le cas CORBA, les messages indiquent en premier lieu le boutisme de la plate-forme qui les envoie. Ainsi, l'entité réceptrice adapte sa procédure de déballage des données. De plus, CORBA impose l'alignement des données dans tous les tampons de communication.

Typage

Le service de typage s'occupe de la gestion des types de données transmises entre les nœuds de l'application et fournies à l'application. La simplicité de ce service varie d'un intergiciel à l'autre. Il est extrêmement sophistiqué pour des intergiciels utilisés dans des systèmes complexes et versatiles (applications scientifiques, systèmes bancaires...) où des structures de données complexes doivent être manipulées. Ce service opère conjointement avec le service de représentation pour emballer les types complexes de données dans les tampons de communication et les déballer depuis ces tampons.

Dans une description IDL de CORBA, l'utilisateur définit tous les types de données qui sont échangés (types simples, structures, union, séquences...). Ensuite, un générateur de code garantit la production automatique des types de données correspondants dans le langage de programmation choisi.

Interaction

Le service d'interaction permet de gérer le mode d'interaction entre deux nœuds logiques. Plusieurs modes d'interaction existent, parmi lesquels nous citons :

- le mode synchrone qui consiste à bloquer l'entité émettrice jusqu'à l'obtention d'une réponse de la part du récepteur,
- Le mode asynchrone qui consiste à libérer immédiatement l'entité émettrice après l'envoi d'une requête.

Toujours dans le cas de CORBA, lors d'un échange synchrone, ce service gère l'invocation des routines correspondantes au mode de communication choisi. Dans le cas d'un échange synchrone, ce service bloque le client jusqu'à l'arrivée d'une réponse de la part du serveur.

Protocole

Le service de protocole effectue les étapes nécessaires pour transmettre un message entre deux entités de l'application. Il prend en entrée un tampon de communication construit par le service de représentation et lui ajoute les informations nécessaires sur la destination (généralement sous forme d'en-tête). Il existe des situations où le message est fragmenté en plusieurs parties avant d'être passé à la couche transport (taille maximale d'un message limitée...). Le service de protocole s'occupe aussi de la gestion du mode de communication (piloté par le service de liaison). Ainsi, dans le cas d'une communication synchrone, il bloque l'entité expéditrice jusqu'à l'arrivée d'une réponse ou d'un accusé de réception de la part du destinataire.

Dans CORBA, IIOP est une des implantations du service de protocole. Il permet de gérer toutes les fonctionnalités que nous avons citées ci-dessus.

Transport

Le service de transport assure l'ouverture, la fermeture et l'utilisation des canaux de communication pour permettre la transmission de messages entre les nœuds. L'implantation de ce service est fortement liée au système d'exploitation et aux mécanismes matériels de transport

qui sont utilisés (bus de communication, couche physique...). Les routines de ce services sont généralement invoquées par ceux du service protocole.

Dans le cas où les nœuds d'une application répartie sont connectés par un bus Ethernet, une implantation possible du service de transport pourra utiliser les sockets *BSD*.

Activation

Le service d'activation permet, sur une entité réceptrice, à l'issue de la livraison d'un message, d'activer l'entité "concrète" nécessaire au traitement de la requête. Il engendre, lorsque ceci est nécessaire, la création de nouveaux objets et leur destruction à la fin du traitement voulu (variables intermédiaires pour stocker les données en entrée provenant de l'expéditeur ou la réponse à renvoyer...). Ce service existe exclusivement sur les entités qui reçoivent les messages.

Par exemple, à la réception d'une requête, un serveur CORBA détermine l'objet à qui elle est destinée et la passe au squelette correspondant. Le squelette extrait le nom de l'opération demandée par le client et invoque la méthode correspondante sur l'objet.

Exécution

Le service d'exécution se situe au dessus du service d'activation. Il utilise l'entité concrète sélectionnée lors de l'activation et effectue l'exécution proprement dite du traitement requis par la réception d'un message. Il engendre, lorsque ceci est nécessaire, l'allocation de nouvelles tâches pour l'exécution de la requête et leur destruction à la fin de l'exécution. Ce service existe exclusivement sur les entités qui reçoivent les messages.

Par exemple, il existe des politiques différentes de qualité de service (QdS) qui influent sur la création de nouvelle tâches pour traiter les requêtes, parmi lesquelles nous citons :

- La politique *mono-tâche* utilise le fil d'exécution principal pour traiter les requêtes. Ceci empêche la réception de nouvelles requêtes pendant le traitement de la requête en cours,
- La politique *tâche par session* consiste à créer une tâche pour un client donné pendant toute la durée de sa connexion au serveur,
- La politique *tâche par requête* consiste à créer une tâche pour exécuter chaque nouvelle requête.

Le service d'exécution permet de gérer la création et la destruction des tâches selon la politique choisie.

3.2.2 Personnalisation des services

Construire un intergiciel dédié à une application TR²E est équivalent à personnaliser chacun des services décrits dans la section 3.2.1 pour qu'il soit optimisé en fonction des besoins de l'application. Notons ici que, dans le contexte des systèmes TR²E, deux caractéristiques importantes doivent être vérifiées par l'intergiciel. Tout d'abord la sûreté de fonctionnement et le déterminisme car, avant tout, il s'agit de systèmes temps-réel critiques. Ensuite, la haute performance et la faible empreinte mémoire de l'application car il s'agit de systèmes embarqués. Ces deux caractéristiques ne sont pas toujours orthogonales.

Par exemple, le service d'exécution crée des tâches nécessaires au traitement des requêtes et le service de transport ouvre les canaux de communication nécessaires pour établir une connexion donnée. Deux techniques possibles existent pour personnaliser ces deux services aux besoins d'une application :

1. Une technique qui utilise le moins de ressources matérielles possibles sans donner une importance aux recommandations des systèmes critiques et particulièrement au profil Ravenscar (section 2.5.1). Dans ce cas, le service d'exécution crée dynamiquement les tâches effectuant le traitement des requêtes au moment précis où ce traitement est demandé. Il les détruit à la fin du traitement de la requête. D'une manière similaire, le service de transport ouvre les canaux de communication au moment où un envoi est requis et les ferme une fois l'envoi accompli. Cette technique, minimise, très efficacement les ressources utilisées, mais elle empêche l'analysabilité statique et le déterminisme de l'application. En effet, nous avons discuté les mécanismes utilisés pour mettre en œuvre ces optimisations (allocation dynamique de mémoire, création dynamique de tâches...) à la fin du chapitre 2 et nous avons conclu qu'ils étaient inadéquats pour les systèmes critiques.

2. Une technique qui privilégie le caractère statique de l'application même si cela va parfois à l'encontre d'une optimisation avancée des performances. Dans ce cas, le service d'exécution crée toutes les tâches dont l'application a besoin à son initialisation. Ces tâches ne finissent jamais, ainsi il est possible de déterminer de manière statique l'ordonnançabilité de ces tâches puisque leur nombre est connu à l'avance. De manière similaire, le service de transport ouvre tous les canaux de communication nécessaires selon la topologie de l'application et ne les ferme jamais. Cette technique n'optimise pas nécessairement les ressources utilisées par l'application TR^{2}E, mais elle possède l'avantage de fournir une architecture statiquement analysable et déterministe.

Il existe des optimisations qui augmentent les performances de l'application et renforcent son caractère statique et déterministe. Par exemple, l'implantation du service d'adressage sous la forme d'une table statique d'association entre les entités de l'application et leurs adresses respectives améliore à la fois les performances et l'analysabilité statique de l'application. En effet, elle rend constant le temps pour trouver un objet distant à partir de sa référence (déterminisme). Par ailleurs, ce temps est égal au temps d'accès à un élément d'un tableau (performance). Dans tout ce qui suit, nos choix d'architecture et d'implantation privilégieront les optimisations renforçant le caractère statique, dans le cas d'un conflit entre ces deux caractéristiques.

Il est important ici de noter la différence entre un *service personnalisable* et un *service configurable* dans le contexte de notre travail. Un service intergiciel personnalisable est un service dont des aspects fondamentaux de l'implantation changent complètement selon les propriétés d'une application. Par exemple le service de typage est personnalisable. Toute son implantation change d'une application à une autre. Par contre, la configurabilité d'un service est la capacité de modifier des paramètres de ce service sans toucher à ses fondements. Par exemple, le service de protocole est configuré en spécifiant l'ensemble des entités émettrices et réceptrices ou encore la taille maximale d'un message mais les implantations des routines d'envoi et de réception ne sont pas affectées par le changement de ces entités.

Tous les services canoniques d'un intergiciel sont configurables. Ils possèdent tous des paramètres dont les valeurs sont spécifiées selon les propriétés de l'application. En revanche, ces services sont divisés en trois familles selon leur aptitude à être personnalisés pour une application TR^{2}E particulière. Il existe une famille de services hautement personnalisables selon les propriétés de l'application. D'autres services restent pratiquement invariants en fonction des applications. Entre ces deux familles, un troisième ensemble de services est constitués de composants fortement personnalisables et d'autres très faiblement personnalisables. Dans la suite de cette section, nous présenterons ces trois familles de services.

Services fortement personnalisables

Il existe des services canoniques fortement dépendants des caractéristiques d'une application donnée et sont entièrement reproduits pour chaque application. Il s'agit des services suivants :

- L'adressage : la production des adresses est fortement liée au nombre de nœuds dans l'application répartie et à la nature des connexions entre ces nœuds. Deux applications réparties différentes ont généralement des implantations complètement différentes de ce service (utilisation d'une couche de transport Ethernet, SPACEWIRE...),
- La liaison : la sélection des modèles de répartition utilisés dépend fortement des caractéristiques de l'application. En particulier la décision d'envoyer un message à une entité distante ou une entité locale dépend de la topologie de l'application. Toute l'implantation de ce service change d'une application à une autre,
- L'activation : les entités concrètes à activer sont généralement différentes pour chaque application répartie car faisant partie du comportement (spécifié par l'utilisateur). Elle dépendent aussi de la topologie de l'application (connexions entre les nœuds),
- Le typage : les types de données utilisées dans une application particulière sont généralement définis par l'utilisateur. Ainsi, ce service est entièrement personnalisable en fonction des caractéristiques de l'application.

Services partiellement personnalisables

Il existe des services intergiciels canoniques qui sont partiellement personnalisables en fonction des caractéristiques de l'application répartie. Ce genre de service possède généralement deux aspects :

- Un aspect de *bas niveau* comprend les fonctionnalités basiques du service et son implantation. Cet aspect ne dépend pas des propriétés de l'application. Il s'agit, la plupart du temps d'un archétype décrivant une fonctionnalité donnée de manière générique ou bien de primitives très bas biveau pour dialoguer avec le pilote d'un périphérique donné,
- Un aspect de *haut niveau* comprend l'utilisation des fonctionnalités du service d'une manière complètement dépendante des propriétés de l'application. Il s'agit, la plupart du temps d'une instanciation des archétypes de bas niveau ou bien d'un ensemble de primitives qui encapsulent les appels aux routines de bas niveau.

Les services partiellement personnalisable sont les suivants :

- L'exécution : ce service est fortement lié à la présence de tâches dans l'application répartie. Une partie de ce service est invariante. Les catégories des tâches que l'on crée dans un système TR²E sont d'un nombre fini (périodique, sporadique...). Donc, si nous disposons d'un moyen pour définir les patrons de conception correspondant à ces catégories, nous avons notre service invariant. La partie complètement personnalisable de ce service est le nombre d'instances de tâches de chaque catégorie. Ce service est donc divisé en deux services dans la nouvelle architecture que nous proposons : un premier service "générique" et faiblement personnalisable que nous appelons **parallélisme** et un service fortement personnalisable qui héritera du nom originel du service, **exécution**,
- La représentation : les manières dont les données sont emballées et déballées ne sont pas nombreuses surtout pour les systèmes TR²E. Par conséquent, une partie de ce service consiste à implanter toutes les routines de représentation que nous désirons supporter dans notre intergiciel et en particulier, les routines pour emballer et déballer chacun des types basiques (entiers, booléens...). Nous notons ce service **représentation élémen-**

taire. La partie qui est personnalisable est le choix et l'utilisation des méthodes d'emballages existantes et leur combinaison pour former une méthode plus complexe correspondant à un type complexe décrit par l'utilisateur. Nous la notons **représentation avancée**,
- L'interaction : de même que pour les services d'exécution et de représentation, ce service est formé d'une partie qui contient les archétypes des différents types d'interactions qui existent entre les tâches (routines d'envoi et de lecture de message, routine d'attente d'un événement...) et d'une autre partie qui consiste à instancier ces archétypes en fonction des interfaces des composants présents dans l'application. Nous les appelons respectivement **interrogation** et **interaction**,
- Le transport : ce service est formé naturellement de deux couches. La couche basse de transport qui rassemble des routines de bas niveau et dialogue généralement avec le pilote de l'interface réseau. L'implantation de cette couche est indépendante de l'application répartie en question. Nous la notons **couche basse de transport**. Par contre, la couche haute de transport est complètement personnalisable selon les caractéristiques de l'application car son travail (sélection de la couche basse de transport selon une information extraite du service d'adressage) est différente pour chaque application. Nous la notons **couche haute de transport**.

Chacun de ces services est donc divisé en deux services : un service faiblement personnalisable contenant l'aspect de bas niveau (*parallélisme*, *représentation élémentaire*, *interrogation* et *couche basse de transport*) et un autre fortement personnalisable qui contient l'aspect de haut niveau (*exécution*, *représentation avancée*, *interaction* et *couche haute de transport*).

Services faiblement personnalisables

Le seul service dont l'implantation est indépendante de l'application répartie est le service protocole. En effet le protocole de communication entre les nœuds d'une application répartie est complètement défini lors des phases de conception de l'intergiciel et il n'est affecté par aucun des aspects particuliers de l'application. Bien entendu, ce service est, lui aussi, configuré en fonction des caractéristiques (matérielles et logicielles de l'application). Par exemple, la taille maximale d'un message est déduite à la fois de la taille des données dans l'application et aussi des caractéristique de l'infrastructure matérielle de communication).

Synthèse

Après avoir divisé les "services intergiciels canoniques" en trois familles, nous les rassemblons pour avoir un premier groupe de services faiblement personnalisables en fonction de l'application TR^{2}E. Ce groupe de faible taille constitue le noyau d'un intergiciel minimal. Le deuxième groupe de services qui sont fortement personnalisables en fonction des caractéristiques de l'application répartie est produit entièrement en fonction de ces caractéristiques. Ce nouveau découpage redéfinit l'ensemble des services canoniques qui sont présentés dans [Hugues, 2005] pour adapter l'architecture de l'intergiciel aux système TR^{2}E critiques. Le tableau 3.1 montre la correspondance entre l'ancien ensemble et le nouvel ensemble de services et précise la nature de chacun des nouveaux services (personnalisable ou non). Les deux prochaines sections présentent le nouveau découpage en services pour un intergiciel dédié à une application TR^{2}E.

Anciens services	Nouveaux services	Degré de personnalisation
Adressage	Adressage	Très fort
Liaison	Liaison	Très fort
Activation	Activation	Très fort
Typage	Typage	Très fort
Exécution	Parallélisme	Très faible
	Exécution	Très fort
Représentation	Représentation élémentaire	Très faible
	Représentation avancée	Très fort
Interaction	Interrogation	Très faible
	Interaction	Très fort
Protocole	Protocole	Très faible
Transport	Couche basse de transport	Très faible
	Couche haute de transport	Très fort

TABLE 3.1 – Nouveau découpage des services intergiciels canoniques

3.3 Intergiciel minimal

Dans cette section, nous donnons une vue globale sur l'architecture de l'intergiciel minimal. D'abord, nous détaillons les motivations qui nous ont poussé à créer un intergiciel minimal. Ensuite, nous décrivons la manière dont les services de cet intergiciel seront implantés.

Motivations

Les services dont l'implantation est indépendante de l'application répartie sont rassemblés pour former l'intergiciel minimal. Cet intergiciel constituera un noyau léger sur lequel viennent se greffer les services fortement personnalisables en fonction de l'application répartie.

Pour chaque langage de programmation que nous désirons supporter, une version de l'intergiciel minimal sera créée. Dans la prochaine section, nous donnons une vision globale de l'architecture de l'intergiciel minimal. Ensuite, nous détaillons la définition des nouveaux services intergiciels faiblement personnalisables identifiés dans la section 3.2.2. Les architectures détaillées pour chaque langage de programmation seront présentées dans le chapitre 6.

Services de l'intergiciel minimal

Les entités qui constituent l'intergiciel minimal sont les nouveaux services de parallélisme, de représentation élémentaire, d'interrogation, de protocole ainsi que la couche basse de transport. Seuls quelques paramètres de configuration de ces services sont affectés par les propriétés de l'application (taille maximale d'un message pour le service de protocole...). Dans la suite, nous décrivons chacun de ces nouveaux services et la manière la plus adaptée pour l'implanter.

3.3.1 Parallélisme

Pour chaque type de tâche supporté dans l'intergiciel, nous créons un archétype qui décrit le comportement de la tâche. Pour chaque tâche présente effectivement dans un des nœuds de l'application, nous instancions l'archétype correspondant à sa catégorie. Vu que nous nous

situons dans le contexte de systèmes TR^{2}E et que nous voulons que nos applications respectent les recommandations du profil Ravenscar (détaillées dans la section 2.5.1). L'ensemble des catégories de tâches supportées se trouve très restreint. En effet, pour que l'application soit analysable statiquement, il faut que l'ensemble des tâches à ordonnancer soit le même pendant toute la durée de l'exécution. Ceci implique que seules les tâches cycliques ne finissant pas sont supportées. Il faut aussi que chacune des tâches ait un temps minimal entre deux déclenchements successifs. Ceci restreint les tâches cycliques supportées aux seules tâches périodiques, sporadiques, ou toute combinaison statique de ces deux comportements.

Les tâches que nous supportons dans notre intergiciel sont donc : (1) les tâches périodiques dont le déclenchement est effectué sur l'attente d'un événement temporel, (2) les tâches sporadiques dont le déclenchement est effectué sur l'attente d'un événement extérieur avec une garantie statique de temps minimal entre deux déclenchements successifs et (3) les tâches hybrides qui sont une combinaison entre les deux précédentes catégories citées (attente effectuée sur un événement temporel superposée à une attente effectuée sur un événement extérieur). Pour chacune de ces trois catégories, nous créons un archétype qui décrit son comportement.

La manière dont ces archétypes sont implantés est fortement dépendante du langage de programmation. Certains langages de programmation supportent intrinsèquement la généricité (comme les *templates* du langage C++ ou encore les entités génériques du langage Ada). Ceci nous permet d'implanter les archétypes de manière naturelle. D'autres langages de programmation (comme le langage C) ne supportent ni la notion de généricité ni celle de tâche. Nous sommes obligés dans ce cas d'utiliser des bibliothèques du système (POSIX par exemple) pour créer nos archétypes. Par ailleurs, le support de la concurrence existe d'une manière intrinsèque dans des langages de programmation comme Ada. Il rend plus simple la création des tâches. Dans d'autres langages ne supportant pas intrinsèquement la concurrence (C++, C), la création des tâches est faite par l'intermédiaire d'interfaces de programmation.

3.3.2 Représentation élémentaire

L'emballage et le déballage des données dans les tampons de communication sont décrits de manière précise indépendamment de l'application répartie. L'architecture et les propriétés de l'application (logicielles et matérielles) poussent parfois le choix vers une technique d'emballage et de déballage particulière. Par exemple, pour une application répartie dont les nœuds s'exécutent sur des plates-formes homogènes, nous sélectionnons une méthode d'emballage et de déballage très simple : la copie mémoire des données dans le tampon. Dans le cas de plates-formes hétérogènes, il existe des méthodes d'emballage sophistiquées pour garantir l'interopérabilité. Elles font intervenir des mécanismes comme l'alignement en mémoire ou encore la précision du boutisme de la plate-forme expéditrice au tout début du tampon.

Pour chacune des méthodes de représentation supportées (CDR, XDR...), nous implantons l'ensemble des mécanismes d'emballage et de déballages pour les types de données élémentaires supportées dans l'intergiciel (entiers, booléen...). Chacun de ces mécanismes est donc implanté une fois pour toute dans l'intergiciel minimal. Ensuite, le service de représentation avancée gérera la sélection des mécanismes élémentaires de la méthode choisie pour construire les routines d'emballage et de déballage pour les types de données complexes.

3.3.3 Interrogation

Ce service implante la manière dont les informations sont échangées entre les interfaces des différentes tâches de l'application. Plusieurs manières d'échange d'information existent :

– L'envoi de donnée d'une tâche vers une autre,
– Le déclenchement d'une tâche sporadique,
– Le déclenchement d'une tâche sporadique tout en lui envoyant une donnée.

Le service d'interrogation offre les implantations basiques pour réaliser les actions citées ci-dessus du coté des tâches émettrices. Il permet aussi, du côté des tâches réceptrices de gérer le mécanisme de files d'attente. Enfin, il permet à la tâche réceptrice de lire les données reçues et de connaître l'identité de l'expéditeur.

L'implantation de ce service décrit le comportement des routines d'envoi et de réception ainsi que les structures de données qui sont utilisées pour permettre des actions conformes aux recommandations du profil Ravenscar. Le nouveau service d'interaction instancie les mécanismes offerts par le service d'interrogation. Ensuite, il implante le comportement désiré pour chaque tâche en fonction des propriétés et la topologie de l'application.

3.3.4 Protocole

Le service de protocole garde la même définition donnée dans la section 3.2.1. L'implantation d'un protocole particulier ne dépend pas de propriétés de l'application TR²E. En effet, ces protocoles sont généralement spécifiés d'une manière claire qui décrit comment les messages sont construits (en-tête, corps du message...) et ceci quelque soit l'application. Par contre, l'architecture (matérielle et logicielle) d'une application peut influencer le choix d'une implantation particulière de protocole donné.

Pour chaque protocole que nous désirons supporter, nous fournissons une implantation qui expose les routines suivantes :

– Une primitive (INIT) initialise les structures internes du service de protocole et qui invoque la routine d'initialisation de la couche haute transport.
– Une primitive envoie un message à une destination donnée (SEND). Cette routine construit un message conformément au protocole à l'aide de routines d'emballage fournis par le service de représentation. Elle le passe à la routine SEND de la couche haute de transport (qui est produite automatiquement),
– Une primitive (DELIVER délivre un message reçu à sa destination correspondante. Cette routine est invoquée par la couche haute de transport. Elle passe ensuite le message reconstitué au service de représentation avancée qui restitue les données depuis ce message.

3.3.5 Couche basse de transport

La couche basse de transport rassemble les routines d'accès direct au médium de communication entre deux nœuds de l'application. Elle constitue le moyen de dialogue entre l'application répartie et le pilote du périphérique de communication. Par conséquent, une application monolithique n'a pas besoin de cette couche. Selon les caractéristiques de l'application et particulièrement celles du médium de communication, nous sélectionnons la couche basse de transport qui convient. Chacune des implantations des couches de transport supportées par l'intergiciel offre les fonctionnalités suivantes :

– Une primitive (INIT) initialise les tâches et les structures de données nécessaires à la couche basse de transport. Une deuxième partie du travail de cette tâche consiste à participer à un algorithme distribué pour initialiser tous les canaux de communication liant les nœuds de l'application répartie. À l'issue de l'exécution de cet algorithme, tous les canaux

de communication sont ouverts et chacun des nœuds est autorisé à commencer son travail effectif. L'implantation de cet algorithme dépend de la couche basse de transport choisie.
– Une primitive envoie un tampon de communication à travers le réseau en utilisant les routines offerts par le pilote du médium de communication (SEND). Cette primitive n'effectue aucun traitement sur le message à envoyer.
– Un mécanisme d'attente reçoit les messages venant du réseau. Il s'agit, le plus souvent d'une tâche sporadique qui se bloque en attendant l'arrivée de nouveaux messages. À l'arrivée d'un nouveau message, la routine DELIVER de la couche haute de transport est utilisée pour le faire parvenir à sa destination.

La couche haute de transport gère la sélection de la couche basse correspondant aux propriétés de l'application.

3.4 Intergiciel produit automatiquement

Les nouveaux composants, identifiés comme fortement personnalisables à partir des caractéristiques de l'application répartie, sont produits automatiquement. Pour ce faire, nous effectuons une analyse poussée de l'application et nous produisons ces composants à l'issue de cette analyse. Ainsi, au moment de la compilation, les composants exacts de l'intergiciel dont l'application a besoin sont sélectionnés (déploiement) et leurs propriétés exactes sont calculées afin de les paramétrer statiquement (configuration).

Dans la suite, nous expliquons les motivations de cette production automatique de composants personnalisables. Ensuite, nous donnons pour chaque composant la stratégie pour sa production automatique ainsi que caractéristiques de l'application TR^{2}E influençant son implantation.

Motivations

Pour produire les nouveaux services fortement personnalisables (deuxième colonne du tableau 3.1), une analyse des caractéristiques de l'application est nécessaire. Les composants sont personnalisés en utilisant les résultats de cette analyse. Effectuer cette analyse manuellement est très fastidieux et inducteur en erreur car tous les composants doivent être revus au moindre changement effectué sur l'architecture de l'application.

Pour accomplir nos objectifs énoncés dans le chapitre 1 (processus automatique de production de systèmes TR^{2}E statiquement analysables), les composants fortement personnalisables doivent être produits automatiquement à partir d'un formalisme précis et analysable. Ainsi, les risques d'erreurs sont moindres et nous augmentons considérablement la productivité. Le choix de ce formalisme fera l'objet de la section 3.5.

Services personnalisés selon l'architecture

Les composants hautement personnalisables sont produits en extrayant les informations nécessaires de la description de l'application TR^{2}E. Dans la suite nous donnons, pour chaque service, l'ensemble des composants automatiquement produits et personnalisés selon les propriétés du nœud qui les abrite.

3.4.1 Adressage

Du fait du caractère statique de l'application TR²E, l'implantation de ce service se trouve très simplifiée. En effet, l'ensemble de tâches qui interagissent dans l'application est constant et les canaux de communication sont ouverts au tout début de l'application. L'association entre les entités destinataires et les ressources utilisées pour les atteindre est implanté en utilisant des tables statiques de nommage/adressage. Pour chaque couche basse de transport T supportée, nous créons une table qui associe les entités connectés l'intermédiaire T. Cette table contient, pour chaque entité N, l'information de la ressource de transport (Socket...) qui a été créée pour atteindre une entité E connectée à N.

3.4.2 Liaison

De manière similaire au service d'adressage, l'implantation du service de liaison se trouve simplifiée du fait du caractère statique de l'application. En effet, l'analyse de la topologie de l'application permet de connaître à coup sûr si une communication entre deux entités est locale ou distante.

L'implantation de ce service fournit des mécanismes qui induisent l'inclusion des services de protocole, d'interaction, de représentation élémentaire et de couche basse de transport requis par le nœud considéré. Dans notre contexte, il s'agit de sélectionner les instances pertinentes pour les services de l'intergiciel minimal.

3.4.3 Activation

L'analyse de l'application répartie détermine les tâches à activer à la réception d'un message. L'implantation du service d'activation inclut aussi l'acheminement correct des messages entre la couche protocolaire et la couche d'interaction. Ainsi, un message est délivré correctement à la tâche de réception. Si le message est aussi un signal et que la tâche de réception est sporadique, alors cette tâche est activée.

3.4.4 Typage

L'analyse des données dans l'application nous permet de générer automatiquement tous les types qui sont utilisés. Elle permet par la même occasion de vérifier si ces types sont conformes aux critères des systèmes critiques (taille bornée, nombres réel à virgule fixe, interdiction des types récursifs...). Elles devront toutes être respectées (voir section 4.9.7).

3.4.5 Exécution

À partir du nombre de tâches dans l'application et de leur caractéristiques (périodicité, priorité, taille de pile...), nous créons statiquement l'intégralité de l'ensemble de tâches dont l'application a besoin. Il s'agit ici de l'ensemble des tâches de l'utilisateur (celles qu'il a modélisées explicitement dans son application). L'ensemble des tâches dans le système réparti pourrait aussi être influencé par la nature des différentes couches du service de transport. Celles-ci peuvent requérir l'ajout d'une ou plusieurs tâches supplémentaires comme nous l'avons vu dans la description de la couche basse de transport dans la section 3.3.5. Cette dernière information est aussi déduite automatiquement à partir de la description architecturale de l'application.

3.4.6 Représentation avancée

Ce service fournit l'implantation des méthodes d'emballage et de déballage pour les types de données définis par l'utilisateur. Il sélectionne les méthodes d'emballage des types de base à partir du service de représentation élémentaire choisi pour l'application. En cas de support de plusieurs mécanismes élémentaires d'emballages de données (CDR, XDR...), la méthode correspondante est soit déduite automatiquement des caractéristiques de l'application soit spécifiée explicitement par l'utilisateur.

L'analyse des types définis par l'utilisateur permet de combiner automatiquement un ensemble de méthodes élémentaires d'emballage et de déballage pour construire des routines d'emballage et de déballage spécifiques au type analysé.

3.4.7 Interaction

Pour chaque entité interagissante, nous déduisons les paramètres nécessaires pour déterminer le mode de communication de cette entité ainsi que les informations qu'elle est susceptible d'envoyer ou de recevoir. Ensuite, nous instancions l'ensemble des mécanismes d'envoi et de réception de données ou de signaux offerts par le service d'interrogation.

Du point de vue de l'utilisateur, ceci rend simple la communication entre les nœuds puisque chaque composant n'a que sa propre interface à gérer tandis que les connexions entre les interfaces (multiples destinations...) sont traitées automatiquement par l'instance de ce service. Ceci nous pousse à opter pour une modélisation de notre application sous la forme de composants interagissant à travers leurs interfaces respectives.

3.4.8 Couche haute de transport

L'implantation de la couche haute de transport expose trois routines qui se situent entre le service de protocole et les différentes couches basses de transport supportées par le nœud.

- Une routine INIT initialise toutes les couches basses utilisées par le nœud courant en invoquant leurs routines INIT respectives.
- Une routine SEND est invoquée par la routine de même nom du service de protocole. Selon l'entité destinataire du message, cette routine invoque soit la routine SEND de la couche basse de transport qui lie l'entité expéditrice à l'entité destinataire soit la routine DELIVER de la même couche haute de transport s'il s'agit d'une communication locale,
- Une routine DELIVER délivre un message reçu à la couche de protocole. Cette routine est appelée soit par la couche basse de transport (dans le cas d'une communication répartie) soit par la routine SEND décrite ci-dessus dans la cas d'une communication locale,

3.5 Choix du formalisme de description

Cette approche de construction de l'intergiciel est qualifiée de "statique" (par comparaison à l'approche dynamique décrite dans la section 2.6.1). Elle est plus appropriée pour les systèmes critiques distribués. Elle exige toutefois une phase préliminaire d'analyse. Cette phase permet de déterminer statiquement les ressources dont l'application a besoin (mémoire, bus de communication...) ainsi que les paramètres de ces ressources (taille des tampons de communication, bande passante...). Par exemple, pendant cette phase d'analyse, la taille maximale d'un message envoyé ou reçu par un nœud de l'application est déterminée, la taille de piles de chaque

tâche est déduite. A la compilation, les types définissant les tampons de communication ont une taille fixe et les piles des tâches sont allouées statiquement. La performance de l'application répartie et sa sûreté de fonctionnement sont considérablement améliorées de cette manière. En effet, configurer tous les composants à l'exécution est semblable au comportement d'un interpréteur de code source ; faire ceci au moment de la compilation permet de gagner en performance. D'autre part, allouer toute la mémoire nécessaire à l'application de façon statique diminue considérablement les risques d'erreur dus à la mauvaise gestion de la mémoire qui surviennent lors de l'exécution.

Dans cette section nous expliquons les motivations qui nous poussent à choisir un formalisme précis pour décrire nos applications TR²E et nous justifions notre choix pour l'un des ces formalismes.

3.5.1 Motivations

L'analyse décrite plus haut est évidemment très laborieuse si elle est effectuée manuellement par le développeur. De plus, elle doit être refaite chaque fois que le moindre paramètre de l'application se trouve modifié. Cependant, si l'application est modélisée avec un langage approprié, le modèle est analysé automatiquement pour évaluer toutes les ressources. Une large partie du code de l'application peut être générée automatiquement. Le code généré contient, de manière statique, la configuration de toutes les ressources dont l'application a besoin. Pour ce faire, nous avons besoin d'un formalisme pour décrire l'architecture du système réparti et d'exprimer les exigences qui influent sur sa configuration et son déploiement. Ce formalisme doit être assez détaillé pour représenter les différents composants de l'application TR²E ainsi que les connexions entre eux.

Par ailleurs, nous n'avons pas besoins uniquement des propriétés logicielles de l'application TR²E pour produire les composants fortement personnalisables. Les propriétés matérielles de l'application (type de bus, type du processeur...) jouent un rôle important sur le choix des composants intergiciels. À cet effet, le formalisme qui décrit l'application doit être capable de spécifier ces caractéristiques matérielles. Les langages de description d'architecture sont d'excellents candidats pour réaliser cette tâche.

3.5.2 Choix d'un langage de description d'architecture

Dans [Medvidovic and Taylor, 2000], les auteurs présentent les caractéristiques qui permettent de qualifier de "langage de description d'architecture" un formalisme donné. En particulier, il faut que ce formalisme permette de modéliser explicitement des "composants".

Définition 3.1 ***Composant***
Un composant, d'après [Medvidovic and Taylor, 2000] est une unité de calcul ou un dépôt de données. Il possède un état interne et peut être qualifié soit de simple soit de composé (dans ce dernier cas, il contient d'autres composants).

Nous ajoutons à cette définition qu'un composant est soit logiciel (donnée, processus légers, sous-programme...) soit matériel (mémoire, processeur...) et que les composants logiciels sont actifs (les processus légers) ou bien passifs (les données...). Les composants d'un système TR²E communiquent ensemble à travers des "interfaces" fournies ou requises. Les interfaces des composants sont connectées par l'intermédiaire de "connecteurs".

Définition 3.2 ***Connecteur***
Un connecteur, toujours d'après [Medvidovic and Taylor, 2000], est un module de l'architecture qui est utilisé pour décrire les interactions entre les composants ainsi que les règles qui régissent ces interactions. Il ne correspond pas nécessairement à une entité sémantique à part entière.

FIGURE 3.3 – Composants et connecteurs

La figure 3.3 décrit l'interaction entre deux composants par l'intermédiaire d'un connecteur. Les interfaces et les connecteurs constituent l'unique moyen d'interaction du composant avec son monde extérieur. L'interface modélise les besoins, les liens ainsi que les contraintes d'un composant donné vis-à-vis de son environnement.

Le formalisme qui décrit l'application TR^{2}E doit être capable de spécifier des propriétés non fonctionnelles concernant le déploiement, la configuration et de l'application (c.f. les définitions 2.3 et 2.4 page 24). En particulier, il doit avoir une syntaxe précise et rigoureuse pour autoriser des analyses comme l'analyse d'ordonnancement. Il doit aussi permettre les spécifications, d'une manière simple et intuitive, des éléments suivants :

- Les caractéristiques des tâches (périodicité, période, priorité...),
- Les types de données,
- La plate-forme d'exécution (processeur, mémoire, bus...),
- Les interfaces des tâches,
- Les comportements effectués par les tâche. À cet effet, la plupart des langages de description d'architecture existant ne permettent pas de spécifier le comportement de manière intrinsèque, nous voulons que le formalisme fournisse au moins un moyen d'interfaçage simple avec le code de l'utilisateur.

Il existe plusieurs familles de langages de description d'architecture [Déplanche and Faucou, 2006]. Les deux familles les plus proches de nos besoins sont :

Les ADLs formels : ces langages permettent de décrire formellement un système TR^{2}E afin de les analyser par la suite. Ils supportent les notions de composants et de connecteurs de manière abstraite sans indiquer à quoi il correspondent dans le système réel. Parmi les ADLs formels, nous trouvons, Wright [Allen, 1997], Rapide [Luckham *et al.*, 1995] et ACME [Garlan *et al.*, 1997]

Les ADLs concrets : ces langages, comme leur nom l'indique, reflètent la réalité mieux que les langages formels. Ils spécialisent la notion de composant en introduisant plusieurs catégories correspondant chacune à une entité du monde réel (logiciel ou matériel). Cette famille de langage est donc mieux adaptée pour la génération automatique de code à partir des modèles. Parmi les langages concrets, nous trouvons UNICON [Zelesnik, 1996] et AADL [SAE, 2004].

Nous avons choisi le langage AADL parce qu'il possède toutes les caractéristiques dont nous avons besoin pour spécifier et générer automatiquement les systèmes TR^{2}E. En effet, il s'agit d'un langage concret. Il permet de spécifier l'architecture logicielle et matérielle d'une application répartie de manière assez détaillée. Ceci nous permet de décrire des aspects précis de la configuration (plate-forme, nature d'un bus de communication...). Aussi, la génération de code à partir des modèles AADL sera-t-elle facilitée du fait de la correspondance quasi immédiate que propose ce langage entre les composants et les entités concrètes.

AADL permet aussi par le biais de la notion de "propriété" d'enrichir les descriptions par des aspects fonctionnels (pointer vers du code sources pour spécifier une implantation donnée) et non fonctionnels (la période ou la priorité d'un processus léger). Ceci nous permet d'exprimer à la fois les exigences de l'utilisateur et les paramètres de déploiement et de configuration du système.

De plus, nous voulons effectuer des analyses sur nos modèles, le langage UNICON n'est pas adapté à cette tâche du fait de sa sémantique peu détaillée.

3.6 Synthèse

Dans ce chapitre, nous avons présenté une architecture pour un intergiciel fondée sur l'architecture schizophrène pour construire des systèmes répartis critiques. Notre architecture reprend la notion de "service" et la redéfinit en séparant les services intergiciels en deux familles : (1) les services faiblement personnalisables en fonction de l'application font partie de ce que nous appelons l'intergiciel minimal et (2) les services fortement personnalisables en fonction de l'application sont produit automatiquement à partir de son modèle.

Nous avons donné l'architecture de l'intergiciel minimal. Ensuite nous avons décrit de manière générale comment des services intergiciels doivent être produits et personnalisés en fonction des propriétés de l'application. Ceci nous a permis de constater qu'il est nécessaire d'avoir un formalisme détaillé et précis pour décrire les applications TR²E et pour rendre automatiques la production et la personnalisation de ces services. Nous avons choisi le langage de description d'architecture AADL pour jouer ce rôle.

Le prochain chapitre présente plus en détail les aspects du langage AADL dont nous avons besoin pour spécifier nos applications. Ensuite, dans le chapitre 5, nous décrivons le processus global de production qui, à partir d'un modèle AADL, donne une application répartie prête à être exécutée.

Chapitre 4

Introduction au Langage AADL 1.0

SOMMAIRE

4.1 Introduction

Dans le précédent chapitre, nous avons défini une nouvelle architecture pour un intergiciel dédié à une application TR²E. Pour réaliser cet intergiciel, une partie importante de ses composants doit être personnalisée en fonction des caractéristiques de l'application. Nous avons déduit que ces composants doivent être produits automatiquement à partir des modèles de l'application. Ce processus diminue les risques d'erreurs et masquent la complexité de la production. Nous avons conclu que l'application doit être modélisée avec un formalisme précis et rigoureux

spécifiant à la fois son architecture et ses propriétés de déploiement et de configuration. Nous avons choisi le langage AADL et justifié ce choix dans la section 3.5.

AADL (*Architecture Analysis and Design Language*) est un langage de description d'architecture qui découle de MetaH [Vestal, 1998]. AADL est un standard international publié par la SAE (*Society of Automotive Engineers*). Comme son prédécesseur, AADL était destiné à ses débuts aux systèmes avioniques et spatiaux, ce qui explique son ancien nom (*Avionics Architecture Description Language*). Il s'est avéré par la suite que le langage était très riche et qu'il offre des capacités d'expression qui dépassent largement le domaine de l'avionique et sont extensibles à d'autres systèmes logiciels et matériels. La première version de ce standard, AADL 1.0 [SAE, 2004] a été publiée en octobre 2004.

Ce langage permet de décrire les systèmes TR^{2}E en assemblant des blocs développés séparément. Il repose principalement sur la notion de "composant" (voir la définition 3.1 page 54). Tout système décrit est un assemblage de composants connectés.

AADL permet de décrire à la fois la partie matérielle et la partie logicielle d'un système. Il est axé autour de la définition d'interfaces précises pour les blocs construits. Il sépare l'implantation interne de la description des interfaces. AADL peut être exprimé en utilisant une syntaxe textuelle aussi bien qu'une représentation graphique.

Un modèle AADL contient (en plus de la description de l'architecture du système) des éléments non architecturaux : caractéristiques temps-réel ou embarquées des composants (temps d'exécution, empreinte mémoire...), descriptions comportementales.... Ainsi, il est possible d'utiliser AADL comme l'épine dorsale pour décrire la majorité des aspects d'un système.

Ce chapitre propose une introduction au langage AADL et particulièrement aux constructions utilisées à la spécification d'applications TR^{2}E critiques. Ceci est nécessaire pour la suite de ce mémoire. En effet, toutes les étapes du processus de production que nous nous proposons d'implanter utilisent le langage AADL comme support de modélisation, de déploiement et configuration.

Dans la suite de ce chapitre, nous présentons les composants de la première version du standard, AADL 1.0. En effet la version définitive du nouveau standard AADLv2 n'est pas suffisamment finalisée pour être utilisée dans notre travail de thèse. Nous listons les différentes catégories de composants ainsi que leurs structures internes (section 4.2). La section 4.3 présente les moyens utilisés pour enrichir les modèles AADL et y incorporer des informations non architecturales. Ensuite, nous indiquons comment les composants et leurs propriétés sont organisés à l'aide de paquetages et d'ensembles de propriétés dans une description AADL(section 4.4). La section 4.5 présente les modes opérationnels des composants AADL. La section 4.6 présente les flots de données et de contrôle dans un modèle AADL. Nous présentons dans la section 4.7 le concept d'instanciation d'un modèle AADL. Nous expliquons dans la section 4.8 comment le langage AADL constitue un choix pertinent pour décrire les systèmes TR^{2}E. Nous constatons que le langage AADL est généraliste et qu'il permet de définir des modèles non conformes aux recommandations des systèmes critiques. Par conséquent, dans la section 4.9, nous définissons un profil (ensemble de restrictions) pour ce langage pour permettre le respect de toutes ces recommandations. La section 4.10 conclut ce chapitre.

4.2 Composants

Une description AADL est faite de "composants". Un composant AADL, selon le standard [SAE, 2004], représente une entité matérielle ou logicielle qui appartient au système en cours de modélisation. Un composant possède un type, qui décrit son *interface*. Ce type joue le rôle d'une

spécification pour le composant. Cette spécification est utilisée par les autres composants du système pour interagir avec le composant spécifié. Le type d'un composant AADL est constitué de trois parties : les éléments d'interface, les flots et les propriétés.

Un composant possède plusieurs implantations ou aucune. Une implantation, contrairement au type du composant, décrit la structure interne. Généralement, un composant est décrit sous la forme d'un assemblage de sous-composants. Ces sous-composants sont des instances de types ou d'implantation d'autres composants. Une implantation contient aussi les connexions qui lient les sous-composants.

Il est possible de décrire les composants du langage AADL en fonction d'autres composants déjà existants. Le langage offre deux mécanismes pour réaliser cette tâche :

1. L'*extension* : un type de composant étend un autre type de composant et acquiert tous ses éléments d'interfaces, ses flots et ses propriétés (ainsi que celles de ses éventuels parents si celui-ci étend lui-même un autre composant). L'extension ressemble à l'héritage de classes dans le monde orienté-objet. Une implantation d'un composant étend soit une autre implantation du même composant, soit une implantation d'un composant parent. Dans les deux cas, elle acquiert toute la structure interne de l'implantation étendue (sous-composants, appels à des sous-programmes, connexions, propriétés...),
2. Le raffinement : un composant qui étend un autre composant a la possibilité de raffiner, selon ses besoins, les entités (éléments d'interfaces, sous-composants...) dont il vient d'hériter.

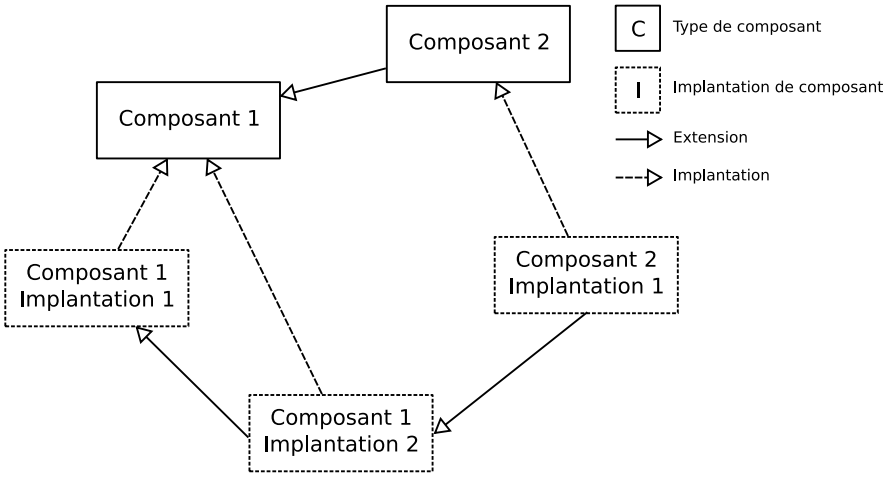

FIGURE 4.1 – Extension des composants en AADL

Ces deux mécanismes permettent de réutiliser des déclarations déjà existantes mais aussi de modéliser les systèmes de manière incrémentale en augmentant à chaque étape le niveau de détail du modèle. La figure 4.1 illustre les mécanismes d'extension entre les composants AADL. Le type de composant **Composant 2** étend le type **Composant 1**. L'implantation **Composant 1 - Implantation 2** étend l'implantation **Composant 1 - Implantation 1** du même type de composant. L'implantation **Composant 2 - Implantation 1**, quant à elle, implante le type de composant **Composant 2** et étendant une implantation de son parent (**Composant 1 - Implantation 2**).

Les composants constituent la notion principale sur laquelle repose tous les modèles d'application TR^{2}E. Une application répartie est une hiérarchie de composants matériels et logiciels interconnectés. Cette hiérarchie est réalisée par deux constructions : les sous-composants

et les appels aux sous-programmes. Les éléments d'interfaces servent à connecter les sous-composants ou les appels aux sous-programmes entre eux ou au composant parent qui les contient.

Dans la suite de cette section, nous donnons les différentes catégories de composants AADL. Ensuite, nous présentons plus en détail les notions de sous-composant et d'appel à un sous-programme. Enfin nous décrivons les éléments d'interface et la manière dont les constituants internes d'un composant sont connectés entre eux pour décrire une sémantique particulière.

4.2.1 Catégories des composants

Il existe trois familles de composants dans le langage AADL. (1) Les composants matériels décrivent des éléments de la plate-forme d'exécution (processeurs, mémoires, bus...). (2) Les composants logiciels décrivent les entités logicielles qui forment une application (processus lourds, processus légers, sous-programmes, données...). Enfin, (3) les composants hybrides servent à hiérarchiser une description AADL. Ils permettent aussi de ne pas préciser la catégorie logicielle ou matérielle des composants dès les premières étapes de la description.

Composants matériels

Il existe quatre catégories de composants matériels : les processeurs, les mémoires, les bus et les périphériques.

Les processeurs sont décrits en utilisant le composant **processor**. Celui-ci modélise un micro-contrôleur et un noyau (système d'exploitation réduit au strict minimum : ordonnanceur, pilotes...). Un processeur exécute les processus légers du système, il peut contenir des mémoires comme sous-composants. Un processeur peut "accéder" à un bus.

Les mémoires sont modélisées avec le composant **memory**. Il représente une mémoire physique quelconque (disque dur, mémoire vive...). Les mémoires servent à stocker les données et le code et sont accessibles par les processus légers en cours d'exécution.

Les bus sont modélisés par l'intermédiaire du composant **bus**. Ils fournissent l'accès entre les processeurs, les périphériques et les mémoires. Les connexions entre les entités logicielles d'un système peuvent être associées à un bus donné auquel cas le flot de données (ou de contrôle) passe par le bus en question.

Le périphériques sont décrits à l'aide des composants **device**. Ils modélisent une large variété de matériel allant des simples capteurs jusqu'aux appareils les plus complexes. Dans tous les cas, un périphérique est vu comme une *boite noire* et sa structure interne ne peut être décrite en AADL. Seule l'interface d'un périphérique est visible par les autres composants du système.

L'ensemble de ces composants matériels permettent de définir l'architecture matérielle d'une application. Le standard précise qu'il est possible d'utiliser les modèles de matériel pour produire automatiquement des descriptions matérielles dans des langages de description matérielles (VHDL [IEEE, 2000]). Dans le contexte de notre travail de thèse, nous utilisons la partie matérielle du modèle de l'application pour spécifier des informations liées au déploiement et la configuration de l'application. Par exemple, le projet ASSERT[2], dans le cadre duquel s'est déroulée la majeure partie de cette thèse, tend à spécifier des applications réparties pour le domaine de l'espace. L'architecture matérielle qui exécute les applications utilise le processeur LEON2 [Research, 2004] et le bus de communication SPACEWIRE [ESTEC, 2003]. Pour modéliser une application répartie s'exécutant sur cette architecture, nous créons des composants

2. ASSERT est un projet financé par la Commission Européenne et animé par l'Agence Spatiale Européenne. http://www.assert-project.net

matériels correspondant au processeur LEON2 et au bus SPACEWIRE et nous lions les composants logiciels à ces composants matériels. Ils nous serviront pour connaître le compilateur à utiliser pour compiler le code source généré à partir du modèle. Ils permettent aussi de vérifier le respect des empreintes mémoire occupées par les composants logiciels pour les tailles réelles des ressources de l'application. Enfin, ils permettent de savoir si une couche de transport est supportée par un bus.

Composants logiciels

Il existe cinq catégories de composants logiciels : les processus lourds, les processus légers (tâches), les groupes de processus légers, les sous-programmes et les données.

Les processus lourds sont modélisés en utilisant le composant **process**. Un processus AADL est un espace mémoire qui sert à contenir les processus légers ainsi que les données partagées entre eux. Par conséquent, pour effectuer un comportement quelconque, un processus lourd doit contenir au moins un processus léger.

Les processus légers sont modélisés grâce aux composants **thread**. Ils modélisent les fils d'exécution qui constituent la partie active de l'application (comme les *threads* POSIX par exemple). Le comportement effectué par un processus léger est spécifié par l'une des manières suivantes :

- en utilisant les propriétés et en pointant vers du code fourni par l'utilisateur,
- en utilisant les annexes pour décrire le comportement au sein même du modèle,
- en appelant des sous-programmes AADL.

Dans le cas où de nombreux processus légers d'un système possèdent des caractéristiques proches et pour éviter la duplication de code, AADL introduit les groupes de processus légers. Ils décrivent des tâches partageant un nombre de propriétés. Ces groupes servent aussi à introduire une hiérarchie entre les processus légers. Ils sont décrits en utilisant le composant **thread group**.

Les sous-programmes sont modélisés avec le composant **subprogram**. Ils décrivent des procédures comme en C ou en Ada. Comme pour les processus légers, le comportement des sous-programmes est spécifié de plusieurs manière (propriétés, annexes, appel à d'autres sous-programmes AADL).

Enfin, les données sont modélisées en utilisant les composants **data**. Les données représentent des types de données, lorsqu'elles sont déclarées sous la forme de composants ou bien quand elles sont utilisées dans les éléments d'interfaces. Elles représentent des variables partagées lorsqu'elles sont instanciées sous la forme de sous-composants.

Composants hybrides

Contrairement aux autres composants, les systèmes, décrits par l'intermédiaire du composant **system**, ne représentent pas une entité concrète. Ils sont utilisés pour créer des blocs qui aident à structurer l'application. Les systèmes servent également dans les premières phases de la modélisation pour remplacer les composants dont la nature n'a pas encore été décidée par le concepteur.

4.2.2 Sous-composants et appels

Comme nous l'avons précisé plus haut, une application répartie est un assemblage hiérarchique de composants. Des composants contiennent d'autres sous-composants pour former l'ar-

chitecture de l'application. Au niveau le plus haut, un composant *system* contient toutes les instances constituant ainsi la racine d'un modèle appelé le modèle d'instance.

La plupart des composants peuvent contenir des sous-composants. Cependant, il existe des règles sémantiques précisées par le standard interdisant à des catégories particulières de composants de contenir certaines catégories de sous-composants. Ces règles découlent de la logique de composition de l'application : ainsi, il est interdit pour un composant de donnée de contenir un processus ou encore un processeur car ceci ne correspond pas à un assemblage réel. Il existe aussi des règles obligeant des catégories de composants à contenir des sous-composants d'autre catégories : ainsi, un processus lourd doit contenir au moins un processus léger pour avoir une activité. Ces règles sont résumées dans le tableau 4.1.

Composant	**Peut contenir**
system	system
	processor
	memory
	bus
	device
	process
	data
processor	memory
memory	memory
bus	*ne peut rien contenir*
device	*ne peut rien contenir*
process	thread (*au moins 1*)
	thread group
	data
thread	data
thread group	thread
	thread group
	data
subprogram	*ne peut rien contenir*
data	data

TABLE 4.1 – Règles de contenance des sous-composants dans AADL 1.0

L'implantation d'un processus léger ou d'un sous-programme peut contenir des séquences d'appels à d'autres sous-programmes décrivant ainsi un flot d'exécution. Seules ces catégories de composants peuvent contenir de tels appels.

4.2.3 Interfaces et connexions

L'interface d'un composant fournit des "éléments d'interface" (les ports de communication, les paramètres...). Les composants communiquent les uns avec les autres en connectant leurs éléments d'interface respectifs.

Un élément d'interface modélise une caractéristique qui est visible par les autres composants. En AADL, ces éléments sont des entités nommées permettant à un composant d'échanger les données et les signaux avec l'extérieur. L'échange des données est effectué grâce aux ports de type donnée et aux paramètres de sous-programmes. L'échange des signaux se fait, quant

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
  {Do_Transmit : subprogram Transmit;};
connections
  parameter Input               -> Do_Transmit.Input;
  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
  event data port Input -> Transmitter.Input;
  event data port Transmitter.Output -> Output;
end Transmitter_Process.Impl;
```

à lui, grâce aux ports de type événement, et aux sous-programmes serveurs (qui fournissent le support d'appels distants de type RPC). Les éléments d'interface sont aussi utilisés pour spécifier un accès à un sous-composant spécifique. Deux caractéristiques orthogonales existent pour chaque accès à un sous-composant : (1) le sens de l'accès (*requires*, *provides*) et (2) la nature de l'accès (*read_only*, *write_only*...).

L'exemple de code 4.1 illustre un exemple utilisant les interfaces et les connexions. Un processus lourd **Transmitter_Process.Impl** contient un processus léger **Transmitter_Thread.Impl** qui appelle un sous-programme **Transmit**. Le port en entrée du processus est connecté à celui du processus léger lui même connecté au paramètre en entrée du sous-programme. Le paramètre en sortie du sous-programme est connecté au port en sortie du processus léger, lui-même connecté au port en sortie du processus.

4.3 Annexes et Propriétés

Comme nous pouvons le voir dans l'exemple donné dans le modèle AADL 4.1, il manque des caractéristiques non architecturales pour compléter la spécification. Par exemple, il manque la nature du processus léger (périodicité, priorité...) ou encore la partie comportementale du sous-programme.

Pour enrichir les descriptions AADL avec les caractéristiques non architecturales, le langage offre deux mécanismes : les annexes et les propriétés.

4.3.1 Annexes

Les annexes dans les modèles AADL permettent d'incorporer dans le modèle d'une application, des éléments écrits dans un langage différent de AADL.

Exemple 4.2 – Utilisation d'annexes OCL dans AADL

```
data Sample end Sample;

thread Collect_Samples
features
  Input_Sample : in data port Sample;
  Output_Average : out data port Sample;
annex OCL {**
  pre: 0 < Input_Sample < maxValue;
  post: 0 < Output_Sample < maxValue;
**};
end Collect_Samples;
```

L'exemple 4.2, issu du standard AADL montre un processus léger qui contient une annexe pour le langage de contraintes OCL. L'annexe spécifie une pré-condition sur le port d'entrée du composant **Collect_Samples** et une post-condition sur la valeur de son port en sortie.

La future version du standard, AADLv2 définit une annexe appelée "l'annexe comportementale". Cette annexe comble l'absence de moyens intrinsèques à AADL pour décrire des comportements (boucles, structures conditionnelles...). Elle permet d'associer un comportement aux sous-programmes et aux processus légers notamment. Avec l'annexe comportementale, l'utilisateur n'est plus obligé de fournir de code source correspondant au comportement des sous-programmes. Il a aussi la possibilité d'implanter le comportement directement dans le modèle AADL.

Il existe un moyen supplémentaire aux annexes, permettant de spécifier les caractéristiques de composants. Il s'agit des propriétés.

4.3.2 Propriétés

AADL introduit la notion de propriété. Les propriétés sont des caractéristiques associées à différentes entités (composants, connections, éléments d'interface...). Il s'agit d'attributs qui permettent de spécifier des caractéristiques ou des contraintes s'appliquant aux éléments de l'architecture : fréquence d'un processeur, pire temps d'exécution d'un processus léger, bande passante d'un bus... Un ensemble de propriétés standards est défini pour le langage ; mais il est possible de définir des propriétés spécifiques à une application donnée.

Exemple 4.3 – Un modèle AADL enrichi par les propriétés

```
subprogram Transmit
features
  Input  : in parameter Message;
  Output : out parameter Message;
properties
  Source_Languege => Ada95;
  Source_Name     => "Transmitter.Do_Transmit";
end Transmit;

thread implementation Transmitter_Thread.Impl
calls
  {Do_Transmit : subprogram Transmit;};
connections
  parameter Input               -> Do_Transmit.Input;
  parameter Do_Transmit.Output -> Output;
properties
  Dispatch_Protocol => Sporadic;
  Period            => 20 ms;
end Transmitter_Thread.Impl;
```

L'exemple de code 4.3 montre quelques composants de l'exemple 4.1 enrichis par les propriétés. Nous remarquons que le processus léger **Transmitter_Thread.Impl** est sporadique et que le temps minimal entre deux déclenchements successifs de ce composant est de $20\ ms$. Notons aussi que le sous-programme **Transmit** est implanté en Ada et que son implantation s'appelle **Transmitter.Do_Transmit**.

4.4 Paquetage et ensembles de propriétés

Pour mieux organiser les déclarations dans un modèle AADL, les notions de paquetage et d'ensemble de propriétés on été introduites. Il s'agit d'espaces de noms qui contiennent les déclarations d'entités faisant partie d'un domaine particulier.

Les paquetages (**package**) servent à organiser les déclarations des composants. Par exemple, il est possible de déclarer les composants matériels et logiciels dans des paquetages différents. Ainsi nous avons des bibliothèques de composants réutilisables sans risquer des conflits de nommage.

Les ensembles de propriétés (**property set**) sont les équivalents des paquetages pour les déclarations de propriétés en AADL. Ils permettent de rassembler des définitions de types, de constantes et de propriétés utilisées dans les modèles. Le standard définit deux ensembles standards de propriétés qui permettent de spécifier des caractéristiques fonctionnelles (implantation

des sous-programmes...) ou non fonctionnelles (période des tâches...) pour les composants. Toutefois, il est permis aux outils manipulant les modèles AADL de spécifier leurs propres ensembles de propriétés si certaines caractéristiques ne sont pas données dans les ensembles standards.

4.5 Modes Opérationnels

Les modes opérationnels représentent les états des composants AADL (toutes familles confondues). Ils permettent de contrôler les valeurs des propriétés, l'activation des connexions et même des sous-composants. Ainsi, un processus léger peut avoir un comportement particulier dans un mode donné et un autre comportement dans un autre mode. Un processus peut voir le nombre et la nature de ses sous composants actifs de type **thread** changer selon le mode.

La transition d'un mode à un autre se fait dynamiquement au cours de l'exécution de l'application répartie. Elle est pilotée par la réception d'événements sur les ports de type événement en entrée. Une machine à états décrit les transitions entre les modes selon les événement reçus.

Exemple 4.4 – Utilisation des modes opérationnels dans AADL

```
thread Worker
features
  Work_Normally       : in event port;
  Emergency_Occurred  : in event port;
  Everything_Is_Cool  : in event port;
  Message             : in event data port Simple_Type;
properties
  Dispatch_Protocol     => Sporadic;
  Period                => 20 Ms;
end Worker;

thread implementation Worker.Impl
modes
  Normal_Mode     : initial mode;
  Emergency_Mode  : mode;
  Lazy_Mode       : mode;

  Normal_Mode,     Lazy_Mode       -[Emergency_Occurred]-> Emergency_Mode;
  Normal_Mode,     Emergency_Mode  -[Everything_Is_Cool]-> Lazy_Mode;
  Emergency_Mode,  Lazy_Mode       -[Work_Normally     ]-> Normal_Mode;
properties
  Compute_Entrypoint => "Repository.CE_Normal_Handler"
    in modes (Normal_Mode);
  Compute_Entrypoint => "Repository.CE_Emergency_Handler"
    in modes (Emergency_Mode);
  Compute_Entrypoint => "Repository.CE_Lazy_Handler"
    in modes (Lazy_Mode);
end Worker.Impl;
```

L'exemple 4.4 montre un processus léger fonctionnant selon trois modes opérationnels. Un mode "nominal", un mode "paresseux" et un mode "urgence". La clause MODES de l'implantation du composant liste les modes, le mode initial (nominal) et la machine à états décrivant les transitions entre les modes. Dans ce cas, le comportement (spécifié par le biais de la propriété standard **Compute_Entrypoint**) dépend de la valeur courante du mode.

4.6 Flots

Il est possible de préciser explicitement les flots de données et de contrôle dans un modèle AADL. Cette fonctionnalité n'introduit aucune nouveauté fonctionnelle ou architecturale : tous les flots sont implicitement déterminés par les connexions entre les différentes entités et l'association des connexions à des bus. Toutefois, la spécification explicite des flots est une fonctionnalité intéressante pour l'analyse. Elle permet de vérifier la cohérence de la topologie de l'application. Il est aussi plus simple pour des outils d'analyse de flots de données ou de contrôle d'avoir à traiter des flots explicites que d'essayer de les évaluer à partir de la topologie de l'application.

Les flots sont décrits dans la section FLOWS d'un type ou d'une implantation de composant AADL. Les flots décrits dans un type de composants sont appelés les "spécifications de flots" : ils précisent uniquement l'existence d'une source de flot ou d'un puits de flot sans les détailler. Les flots décrits dans l'implantation d'un composant sont appelés les "implantations de flots". Ils détaillent les spécifications de flots en précisant le chemin complet que doit suivre la donnée ou le contrôle en fonction des connections et des spécifications de flots des sous-composants.

4.7 Instanciation d'un modèle AADL

Pour modéliser une architecture, les composants AADL doivent être déclarés comme sous-composants (*instances*) d'autres composants AADL. Au niveau le plus haut, un composant *system* contient toutes les instances des autres composants. La plupart des composants AADL peuvent contenir des sous-composants permettant ainsi une description hiérarchique du système. Le processus d'instanciation a été clairement défini par le standard AADL. L'instance d'un modèle AADL est la donnée de la vue hiérarchique de ce modèle en partant du système racine et en évaluant toutes les valeurs de propriétés de tous les composants.

Lors de l'analyse d'un modèle AADL l'instanciation constitue une étape très importante : elle permet d'évaluer les valeurs des propriétés des différents sous-composants. Elle permet aussi, grâce à l'analyse des connexions AADL de construire des chemins absolus (de bout en bout) entre les éléments d'interfaces. L'instanciation peut être rapprochée à la résolution de la surcharge des méthodes faite sur un langage orienté-objet. Toutefois, dans le cas de AADL, la résolution peut se faire statiquement car toutes les instances de composants sont définies dans le modèle (il n'y a pas de polymorphisme en AADL).

Le modèle de l'exemple de code 4.5 décrit une application répartie formée de deux processus. Chacun de ces deux processus contient deux processus légers. Les processus légers **Slow_Fast.Impl.S** et **Slower_Faster.Impl.S** sont de même composant type. Il en est de même pour **Slow_Fast.Impl.F** et **Slower_Faster.Impl.F**. Le composant **Fast** étend le composant **Slow** et redéfinit la valeur de sa période. Certains des sous-composants redéfinissent eux-mêmes la valeur de cette période (les sous composants **S** et **F** de **Slower_Faster.Impl**). Si nous désirons, par exemple, effectuer une analyse de type RMA [Sha *et al.*, 1993] sur cette application en nous basant directement sur le modèle, nous devons évaluer les valeurs correctes des périodes de chacune des instances de composants **thread**.

Effectuer cette tâche d'évaluation au sein même de la partie analyse (ou celle de la génération de code) est une tâche très complexe. Ceci est dû essentiellement à l'absence d'une contrainte d'ordre dans les déclarations AADL (les entités peuvent être utilisées avant d'être déclarées). Par ailleurs, les outils qui combinent l'analyse à l'instanciation deviennent difficilement maintenables. Par contre, transformer le modèle AADL en modèle d'instance sépare les différentes étapes et rend les phases d'analyse et de génération de code plus simples. La figure 4.2

Exemple 4.5 – Exemple de modèle AADL contenant des redéfinitions de propriétés

```
thread Slow
properties
  Dispatch_Protocol => Periodic;
  Period            => 1 sec;
end Slow;

thread Fast extends Slow
properties
  -- La valeur de Dispatch_Protocol est héritée du
  -- thread Slow.
  Period            => 100 ms;
end Fast;

process Slow_Fast end Slow_Fast;
process Slower_Faster end Slower_Faster;

process implementation Slow_Fast.Impl
subcomponents
  S : thread Slow;
  F : thread Fast;
end Slow_Fast.Impl;

process implementation Slower_Faster.Impl
subcomponents
  S : thread Slow
    {Period => 2 sec;};
  F : thread Fast
    {Period => 10 ms;};
end Slower_Faster.Impl;

system Root end Root;

system implementation Root.Impl
subcomponents
  P1 : process Slow_Fast.Impl;
  P2 : process Slower_Faster.Impl;
end Root.Impl;
```

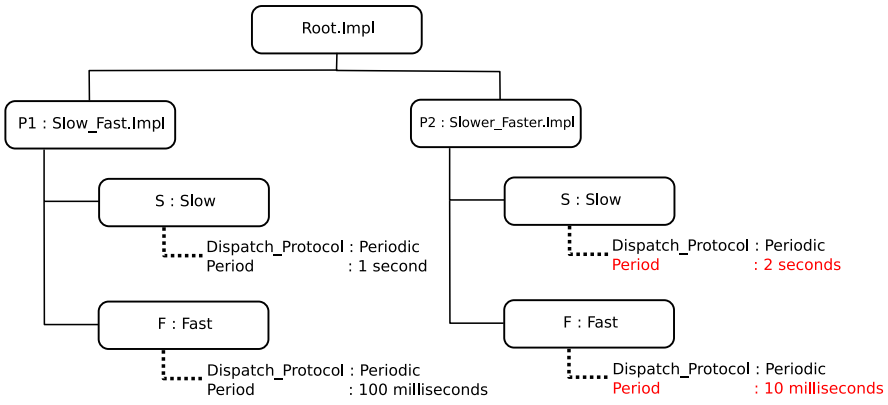

FIGURE 4.2 – Modèle d'instance de l'exemple 4.5

montre le modèle d'instance correspondant au modèle AADL 4.5. Il est clair que l'utilisation de ce modèle pour effectuer une analyse d'ordonnancement ou bien pour générer du code est beaucoup plus simple que l'utilisation directe du modèle AADL.

4.8 Avantages pour les systèmes TR²E

Dans la section 2.6, nous avons montré pourquoi les environnements de modélisation que nous avons présentés dans l'état de l'art sont limitées pour spécifier et produire des systèmes TR²E. Nous avons identifié trois problèmes dont souffrent ces implantations :

1. L'inadéquation aux systèmes TR²E critiques (section 2.6.1),
2. Le non passage à l'échelle (section 2.6.2),
3. L'automatisation partielle (section 2.6.3).

Par ailleurs, la conception des systèmes TR²E passe par des phases de prototypage avant d'obtenir le modèle final. Dans [Kordon and Luqi, 2002], deux approches de prototypage sont notées :

1. Dans l'approche "jetable" (*throw-away*), les prototypes sont construits pour valider un concept avant d'implanter le système réel. Cette approche est souvent utilisée pendant les phases d'acquisition d'un projet,
2. Dans l'approche "évolutive" (*evolutionary*), les prototypes deviennent progressivement le système final. Ils sont raffinés à plusieurs reprises. Le dernier prototype est le système final. La figure 4.3 illustre cette approche. Les raffinements peuvent être fournis à partir de différents niveaux du processus (modélisation, analyse, génération de code).

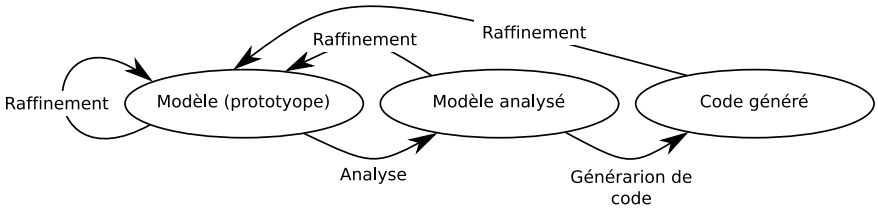

FIGURE 4.3 – Processus de développement évolutif

Vus les coûts des tests et de la validation des systèmes, nous considérons que l'approche "évolutive" devrait être appliquée pour les systèmes TR²E. En effet, elle permet de conserver les informations acquises sur les parties logicielle et matérielle de l'application. Le langage AADL adhère parfaitement à cette approche grâce aux différents mécanismes de raffinement qu'il supporte (extension, raffinement des éléments d'interface...).

Un processus de production basé sur le langage AADL permet de résoudre ces problèmes. En effet, AADL apporte deux bénéfices majeurs pour le prototypage et la conception des systèmes TR²E :

1. La nature concrète du langage AADL permet de spécifier des entités comme les processus légers ou les sous-programmes d'une manière précise. Par ailleurs, le mécanisme des propriétés permet de spécifier caractéristiques non architecturales essentielles des composants du domaine TR²E (les périodes et les échéances des processus légers, le nombre maximal de tâches par processus, la latence d'un bus...).

2. Le mécanisme des propriétés permet incorporer dans le modèle AADL des informations liées au déploiement et à la configuration du système TR^{2}E. Un outil exploitant convenablement ces informations en plus des informations fournies par le modèle, peut piloter un processus complètement automatique de production d'applications TR^{2}E. Toutefois, les ensembles de propriétés standards de AADL 1.0 sont insuffisants pour décrire les caractéristiques des systèmes TR^{2}E. Une partie de notre travail a consisté à enrichir ces ensembles en introduisant de nouvelles propriétés (voir la section 4.9).

Ainsi, les points évoqués ci-dessus montrent que le langage AADL permet de surpasser les limitations principales que nous avons notées dans la plupart des outils présentés dans l'état de l'art de ce mémoire. Nous pouvons dire que le langage AADL est un excellent candidat pour prototyper et concevoir des systèmes TR^{2}E.

Cependant, Le langage AADL étant un langage généraliste, il autorise aussi des constructions qui ne sont pas conformes aux exigences des systèmes critiques. Nous avons été amenés à définir un profil du langage qui permet de décrire des applications TR^{2}E. Dans la prochaine section, nous décrivons les restrictions supplémentaires que nous avons introduits au langage AADL pour avoir des applications conformes aux recommandations pour les TR^{2}E critiques.

4.9 Restrictions supplémentaires pour le langage

Dans cette section, nous présentons des règles et des restrictions supplémentaires qui doivent être respectées par l'utilisateur lors de la modélisation d'une application TR^{2}E. Seul un sous-ensemble du langage AADL est autorisé pour modéliser de telles applications. Le recours à ce sous-ensemble est motivé par des contraintes architecturales et des contraintes non fonctionnelles. En effet, nous voulons que le code produit soit conforme au profil Ravenscar ainsi qu'aux recommandations de développement pour les systèmes critiques (vus dans la section 2.5.1).

Contraintes architecturales Notre but est d'interdire les constructions qui nuisent à l'analysabilité statique de l'application ou qui rendent les analyses statique beaucoup trop pessimistes. Les aspects architecturaux que nous interdisons sont :

1. l'utilisation de l'invocation distante de sous-programmes (RPC synchrone) car elle rend le calcul du pire temps d'exécution dans un nœud dépendant du comportement d'un autre nœud. L'analyse statique de pire temps d'exécution, utilisant par exemple une méthode holistique [Tindell, 1993], devient extrêmement pessimiste,
2. le contrôle des périodes et des priorités des processus légers en utilisant les modes opérationnels. En effet, ceci rend l'analyse statique d'ordonnancement trop complexe car il faut l'effectuer pour toutes les combinaisons possibles de modes. Ceci n'est pas supporté par les outils d'analyse que nous utilisons et dépasse le cadre de notre étude, nous interdisons par conséquent l'utilisation de ces modes pour contrôler des propriétés qui pourraient avoir un impact sur l'ordonnancement de l'application. Toutefois, l'utilisation des modes est autorisée pour contrôler des aspects qui ne compromettent pas l'analysabilité statique de l'application comme le comportement des sous-programmes,
3. la présence, dans le modèle d'éléments d'interface qui sont non connectés. En effet, ceci indique souvent une erreur de conception de la part de l'utilisateur et conduit à la génération de code inutile (code *mort*).

Contraintes non fonctionnelles Les aspects non-fonctionnels d'un modèle AADL sont exprimés à l'aide des propriétés. Notre but est d'interdire l'utilisation des propriétés qui compromettent le caractère statique de l'application. Ainsi, dans un système embarqué où les ressources en mémoire sont généralement limitées :

1. nous interdisons l'utilisation des types de donnée de taille non bornée. Ceci permet de connaître, au moment de l'analyse, la taille exacte de toutes les données et de leur allouer statiquement la quantité de mémoire nécessaire. Cette restriction est étendue aussi pour la taille de pile des processus légers. Pour les mêmes raisons évoquées plus hauts, il faut que cette taille soit spécifiée statiquement,
2. nous restreignons la définition de processus légers aux seuls périodiques, sporadiques et hybrides. En effet, le profil Ravenscar impose que tous les processus légers soient cycliques, que le temps entre deux déclenchements successifs soit borné inférieurement par une constante connue lors de l'analyse et enfin interdit la création dynamique de tâches.

Ci après nous expliquons, pour chacune des entités d'une application répartie, les restrictions que nous posons. Nous donnons aussi des guides pour modéliser chacune de ces entités et nous illustrons ces guides par des exemples. La figure 4.4 montre l'architecture de l'exemple WORKLOAD MANAGER. Nous utilisons les entités de cet exemple pour illustrer la plupart des constructions dans la suite de cette section. Ce même exemple sera repris dans le chapitre 8 pour valider nos solutions et donner des mesures de performance. Cet exemple est inspiré de l'étude de cas utilisée pour illustrer le pouvoir d'expression du profil Ravenscar dans [Burns *et al.*, 2004, Section 7]. Nous avons adapté cet exemple au langage AADL.

Il s'agit d'un processus léger périodique (**Regular Producer**) qui effectue, à chaque période une quantité donnée de travail. Lorsque la quantité de travail demandée dépasse une limite (préalablement établie), la charge supplémentaire est déléguée à un processus léger sporadique de moindre priorité (**On Call Producer**). Le système reçoit, de temps en temps des interruptions extérieures qui devront être traitées. Ces interruptions sont reçues par un processus léger hautement prioritaire (**External Interrupt Server**) qui reçoit les interruptions et les enregistre dans un tampon afin d'être traitées ultérieurement. Lorsque une condition particulière est vérifiée, le processus léger (**Regular Producer**) réveille un autre processus léger de très faible priorité (**Activation Log Reader**) pour effectuer le traitement de la dernière interruption reçue.

Nous avons complété cet exemple pour illustrer l'envoi des interruptions extérieures. Pour cela, un second processus (INTERRUPTION SIMULATOR) abrite un processus léger périodique (**External Event Source**). Ce dernier envoie les interruptions au processus WORKLOAD MANAGER et remplace les interruptions matérielles dans l'exemple original. Les deux processus s'exécutent chacun sur un processeur LEON2 et sont liés par un bus SPACEWIRE.

4.9.1 Modélisation d'une application répartie

Pour modéliser une application TR^{2}E en AADL, le composant **system** est le plus approprié. En effet, comme le précise le standard, il s'agit d'un composant hybride utilisé pour introduire une hiérarchie entre les composants et les organiser. Un système AADL ne correspond pas à une entité concrète. Ce sont les outils qui interprètent les modèles AADL qui donnent à un système sa vraie signification. Tous les composants de l'application répartie (matériels et logiciels) sont rassemblés hiérarchiquement dans un ou plusieurs systèmes. Le système racine représentera l'application répartie. L'exemple de code 4.6 montre un exemple de système décrivant l'exemple WORKLOAD MANAGER décrit plus haut.

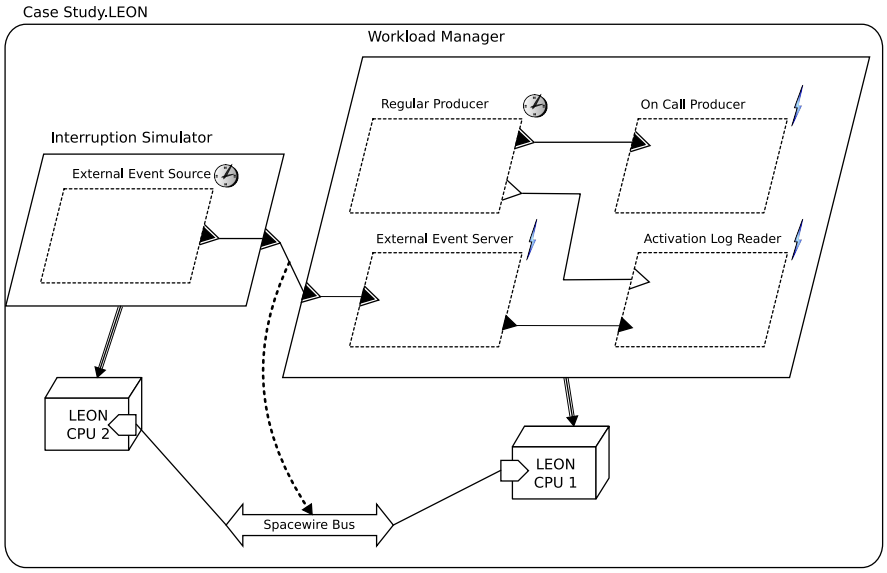

FIGURE 4.4 – Exemple WORKLOAD MANAGER

Exemple 4.6 – Modèle d'une application répartie

```
system Case_Study
end Case_Study;

system implementation Case_Study.LEON
subcomponents
  -- Noeuds
  WoM : process Workload_Manager.Impl {
     -- Propriétés du noeud WoM
    };
  InS : process Interrupt_Simulator.Impl {
     -- Propriétés du noeud InS
    };

  -- Processeurs
  CPU_1 : processor The_Processor;
  CPU_2 : processor The_Processor;

  -- Bus
  B     : bus Ethernet_Bus;
connections
  -- ...
properties
  -- ...
end Case_Study.LEON;
```

Pour chaque nœud de l'application répartie (généralement un processus), nous instancions un sous-composant AADL dans le système lui correspondant. Le système contient aussi les instances de processeurs ainsi que les bus utilisés dans l'application. Nous utilisons les propriétés AADL pour lier les processus aux différentes plates-formes de l'application répartie et les connexions au différents bus (voir 4.9.3 pour plus de détails). Le composant contient aussi une clause qui permet de connecter les éléments d'interface des différents processus et pour lier les connexions aux bus (voir 4.9.5 pour plus de détails).

4.9.2 Modélisation des nœuds des applications réparties

Pour décrire un nœud d'une application répartie, nous utilisons le composant **process** qui permet de modéliser un processus. L'exemple de code 4.7 montre le type et l'implantation du processus modélisant le processus **Worlkload_Manager** de l'exemple WORKLOAD MANAGER. Le processus correspondant à chaque nœud de l'application répartie est instancié sous forme de sous-composant dans le système décrivant l'application entière. Ainsi le processus **Worlkload_Manager** est instancié sous la forme du sous-composant WoM dans le système global.

Si le processus doit communiquer avec ses semblables, il doit avoir une interface. En AADL l'interface d'un composant est généralement un ensemble de ports déclarés dans la section FEATURES.

Exemple 4.7 – Modèle d'un nœud de l'exemple

```
process Workload_Manager
features
  External_Interrupt_Depository : in event data port Interrupt_Counter;
end Workload_Manager;

process implementation Workload_Manager.Impl
subcomponents
  Regular_Producer : thread Regular_Producer;
  On_Call_Producer : thread On_Call_Producer;
  External_Event_Server : thread External_Event_Server;
  Activation_Log_Reader : thread Activation_Log_Reader;
connections
  event data port External_Interrupt_Depository
    -> External_Event_Server.External_Interrupt_Depository;
  event data port Regular_Producer.Additional_Workload
    -> On_Call_Producer.Additional_Workload_Depository;
  event port Regular_Producer.Handle_External_Interrupt
    -> Activation_Log_Reader.Signal;
  data port External_Event_Server.External_Interrupt
    -> Activation_Log_Reader.External_Interrupt_Depository;
end Workload_Manager.Impl;
```

L'implantation d'un processus doit contenir au moins un sous-composant **thread** pour que le processus soit "actif". Elle contient aussi une clause qui permet de connecter les interfaces des processus légers contenus dans le processus à l'interface du processus lui même. La section 4.9.5 présente plus en détails les connections AADL.

Pour garantir l'ordonnancement de chacun des nœuds de l'application répartie indépendamment des autres nœuds, nous imposons que les communications entre ces nœuds soient asynchrones. Des études effectuées dans le passé [Waldo *et al.*, 1994] ainsi que dans le présent [Vinoski, 2008] ont montré que les appels synchrones de procédures augmentent la latence et obligent l'utilisateur à gérer une nouvelle catégorie d'erreurs : les erreurs partielles. Ce sont des erreurs non fatales mais qui compromettent le bon fonctionnement de l'application (connexion interrompue momentanément...). Elle doivent par conséquent être traitées. En interdisant ce type

d'appel et en nous restreignant aux seuls envois asynchrones de messages, nous obtenons une borne réduite du temps de réponse des processus légers. Des analyses d'ordonnancements moins pessimistes sont alors possibles.

La nature même du langage AADL fait en sorte que cette restriction ne limite pas l'expressivité du langage lors des modélisations. En effet, le mécanisme d'envoi de messages asynchrones est supporté de façon native dans le langage par le biais des ports, des paramètres et des connections. La communication entre les différents nœuds est effectuée par des envois d'information (donnée ou événement) à partir du port d'un nœud vers le port d'un autre nœud. Par ailleurs, nous interdisons l'utilisation des sous-programmes serveurs, une construction utilisée dans AADL pour implanter les invocations distantes de sous-programmes.

4.9.3 Modélisation des hôtes

En AADL, une même catégorie de composants permet de modéliser à la fois le processeur et le système d'exploitation sous-jacent : il s'agit du composant **processor**. Les caractéristiques d'un processeur sont spécifiées grâce aux propriétés AADL (architecture, système d'exploitation...). Dans une application répartie, les différents processeurs doivent être connectés par l'intermédiaire de bus AADL. Les propriétés des bus permettent de spécifier le type de communication entre les différents nœuds (protocole, couche transport...). La modélisation des composants matériels nous permet d'incorporer des informations de déploiement et de configuration. Ces informations sont utilisées par les outils d'analyse ainsi que par les générateurs de code.

Exemple 4.8 – Ensemble de propriétés **Deployment**

```
property set Deployment is
  Allowed_Transport_APIs : type enumeration
    (BSD_Sockets,
     SpaceWire);
  -- API de transport supportées
  Transport_API : Allowed_Transport_APIs applies to (bus);

  Port_Number : aadlinteger applies to (process);
  -- Numéro de port d'un processus (Ethernet)

  Process_ID : aadlinteger applies to (process);
  -- Identificateur d'un processus (SpaceWire)
  Channel_Address : aadlinteger applies to (process);
  -- Adresse d'un canal de communication (SpaceWire)

  Protocol_Type : type enumeration (iiop, diop);
  -- Protocoles supportés
  Protocol : Protocol_Type applies to (system);

  Allowed_Execution_Platform : type enumeration
   (Native,          -- Plates-formes natives (GNU/Linux, Solaris, Windows...)
    LEON_RTEMS,      -- Pour les cartes LEON2 ou tsim-leon (RTEMS)
    LEON_ORK,        -- Pour les cartes LEON2 ou tsim-leon (ORK)
    ERC32_ORK,       -- Pour les cartes ERC32 ou tsim-erc32 (ORK)
    ARM_DSLINUX,     -- Pour la Nintendo DS (tm) (DSLinux)
    ARM_N770);       -- Pour le Nokia N770
  -- Plates-formes supportées

  Execution_Platform : Allowed_Execution_Platform applies to (processor);
  -- The execution platform of a distributed application
end Deployment;
```

L'exemple de code 4.9 montre la description de la partie matérielle de l'exemple WORKLOAD MANAGER : un processeur et un bus. Pour spécifier les propriétés des différents processeurs et des bus, nous utilisons des propriétés standards ou bien d'autres que nous avons définies. Ainsi, pour préciser le mode d'ordonnancement du processeur, nous utilisons la propriétés standard `Scheduling_Protocol`. Nous avons défini un nouvel ensemble de propriétés, **Deployment** décrit par l'exemple 4.8 pour préciser les informations de déploiement d'une application TR^{2}E. Pour préciser la plate-forme d'exécution d'un processus ou encore la couche basse de transport d'un bus, nous utilisons respectivement les deux propriétés `Execution_Platform` et `Transport_API` que nous avons ajoutées. Enfin, certains outils utilisés pour effectuer les analyses d'ordonnancement disposent de leurs propres ensembles de propriétés. Ainsi, pour préciser que l'ordonnanceur associé au composant **The_Processor** est préemptif, nous utilisons la propriété `Preemptive_Scheduler` spécifique à l'outil CHEDDAR [Singhoff *et al.*, 2004]. Bien entendu, les nouvelles propriétés définies doivent être interprétées par les outils d'analyse ou de génération de code pour produire l'effet désiré.

Exemple 4.9 – Modèle de l'architecture matérielle

```
processor The_Processor
features
  SPW : requires bus access The_Bus;
properties
  Deployment::Execution_Platform            => LEON_ORK;
  Cheddar_Properties::Preemptive_Scheduler  => true;
  Scheduling_Protocol                       =>
    POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL;
    -- Équivalent à FIFO WITHIN PRIORITIES
end The_Processor;

bus The_Bus
properties
  Deployment::Transport_API => Spacewire;
end The_Bus;
```

Pour lier un nœud donné à la plate-forme sur laquelle il s'exécute, nous utilisons la propriété standard `Actual_Processor_Binding` comme le montre l'exemple de code 4.10. Il s'agit de la liaison entre les nœuds WoM et InS de l'exemple WORKLOAD MANAGER. La clause PROPERTIES du système **Case_Study.LEON** montre que le nœud s'exécute sur le processeur CPU_1 tandis que le nœud InS s'exécute sur le processeur CPU_2.

4.9.4 Modélisation des processus légers

Pour modéliser les entités actives d'exécution (les processus légers), nous utilisons le composant AADL **thread**. La section FEATURES d'un composant **thread** décrit son interface (l'ensemble des ports qui sont connectés à d'autres composants). La section PROPERTIES liste les différentes caractéristiques telles que sa priorité, sa nature (périodique, sporadique...) et sa période. Le standard AADL offre plusieurs propriétés pour spécifier ces caractéristiques. La section CALLS de l'implantation d'un composant **thread** contient la séquence de sous-programmes AADL qu'il appelle au cours de son exécution (voir 4.9.6 pour plus de détails sur la modélisation de sous-programmes en AADL). Le comportement du processus léger est également spécifié par l'intermédiaire de la propriété standard `Compute_Entrypoint` qui indique le nom du sous-programme à exécuter. Cette propriété est appliquée au composant ou à un de ses "in event [data] ports" auquel cas le comportement dépend du port qui a reçu l'événement. Enfin, la clause CONNECTIONS contient les connexions des paramètres des différents sous-programmes appelés.

Exemple 4.10 – Liaison avec un processeur

```
system implementation Case_Study.LEON
subcomponents
  -- Noeuds
  WoM : process Workload_Manager.Impl {
    -- Propriétés du noeud WoM
    };
  InS : process Interrupt_Simulator.Impl {
    -- Propriétés du noeud InS
    };

  -- Processeurs
  CPU_1 : processor The_Processor;
  CPU_2 : processor The_Processor;

  -- Bus
  B     : bus Ethernet_Bus;
connections
  -- ...
properties
  Actual_Processor_Binding => reference CPU_1 applies to WoM;
  Actual_Processor_Binding => reference CPU_2 applies to InS;
end Case_Study.LEON;
```

Exemple 4.11 – Modèle d'un processus léger en AADL

```
thread Regular_Producer
features
  Additional_Workload        : out event data port Workload;
  -- Travail supplémentaire: délégué à On_Call_Producer

  Handle_External_Interrupt  : out event port;
  -- Pour réveiller Activation_Log_Reader
properties
  Compute_Entrypoint                  => "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;
```

L'exemple de code 4.11 montre le modèle du composant **Regular_Producer** qui appartient au nœud WoM de l'exemple WORKLOAD MANAGER. D'après l'exemple, il s'agit d'un processus léger périodique de période 1000 millisecondes et d'échéance égale à 500 millisecondes. Ce composant possède deux ports en sortie : un port pour déléguer le travail supplémentaire au composant **On_Call_Producer** (`Additional_Workload`) et un autre pour réveiller le composant **Activation_Log_Reader** (`Handle_External_Interrupt`). Le comportement de ce processus léger est implanté dans le sous-programme `Work.Regular_Producer_Operation` fourni par l'utilisateur.

Le modèle 4.11 donne aussi des propriétés utiles pour l'analyse d'ordonnancement du système. Ainsi, la propriété du processus léger est spécifiée par l'intermédiaire de la propriété `Fixed_Priority` spécifique à CHEDDAR. Son pire temps d'exécution est spécifié par l'intermédiaire de la propriétés standard `Compute_Execution_Time`.

En addition de la sémantique imposée aux composants **thread** par le standard, nous avons ajouté des restrictions supplémentaires pour garantir l'analysabilité statique d'ordonnancement de notre système :

1. Nous voulons rendre possibles les analyses d'ordonnancement de type RMA (Rate Monotonic Analysis), RTA (Response Time Analysis) ou holistique sur le modèle AADL. Pour ce faire, tous les processus légers dans le système doivent être *cycliques*. Le déclenchement de chaque cycle doit être effectué soit par un événement temporel, auquel cas, le processus léger est dit *périodique* soit par un événement extérieur (au sens AADL du terme) reçu par le processus léger sur un de ses ports. Dans ce cas, un temps minimal doit être spécifié entre la réception de deux événements successifs et le processus léger est dit *sporadique*. Nous permettons aussi la définition de processus légers *hybrides* qui sont un mélange des périodiques et sporadiques (se déclenchent à la réception d'un événement temporel et aussi un événement extérieur). Il s'agit d'une nouvelle catégorie introduite par la nouvelle version du standard AADL AADLv2),

2. Nous voulons que les interactions entre les processus légers ne conduisent pas à un interblocage du système ni à une inversion de priorité de durée non bornée. Ainsi, elles doivent être effectuées par l'intermédiaire de données partagées. De telles interactions surviennent quand un processus léger envoie (resp. reçoit) un message à (resp. de la part d') un autre, ou bien quand ces processus légers accèdent en lecture ou en écriture à une donnée partagée grâce au mécanisme `data access` d'AADL. Dans le cas où la donnée partagée entre plusieurs composants **thread** est accessible en écriture par au moins un composant, l'accès à cette donnée doit être protégé par un verrou logiciel pour éviter de corrompre sa valeur en cas d'accès concurrents. Cette protection introduit éventuellement une inversion de priorité (un processus léger de haute priorité bloqué par un autre de basse priorité). Le profil Ravenscar recommande de réduire au minimum la durée de cette inversion et de connaître sa valeur statiquement. Nous imposons que le mécanisme de protection des données utilise le protocole PCP qui plafonne artificiellement les priorités des processus légers accédant à une donnée protégée. Ceci garantit l'absence d'interblocage et des blocages chaînés,

3. Nous ne voulons pas que l'utilisation des modes opérationnels compromette l'analysabilité statique de l'application. Ainsi, nous les autorisons uniquement pour contrôler le comportement des processus légers (séquences d'appel et propriété `Compute_Entrypoint`).

4.9.5 Modélisation des connections

Une connexion entre deux entités AADL (processus lourds, processus légers, sous-programmes...) est modélisée en utilisant les deux constructions suivantes :

1. les éléments d'interface du composant : chaque élément doit avoir un sens selon qu'il envoie ou reçoit de l'information (**in**, **out** ou les deux à la fois),

2. la section CONNECTIONS dans une implantation d'un composant : elle rassemble les liaisons entre les interfaces de les sous-composants et aussi celles entre l'interface propre au composant et les interfaces de ses sous-composants.

Nous avons ajouté des règles sémantiques supplémentaires pour renforcer la cohérence des systèmes modélisés. Ainsi, tous les éléments d'interfaces doivent être connectés. De plus, les connections entre les nœuds d'une application répartie sont obligatoirement associées à un bus. Ceci permet, lors de la génération de code, de sélectionner la couche basse de transport correspondante pour atteindre une destination donnée. Nous imposons aussi que les processeurs soient connectés aux bus de données. Ceci renforce la cohérence de l'application et permet de détecter les erreurs de modélisation de l'utilisateur. Ainsi un utilisateur ne peut associer une connexion à un bus si ce bus n'est pas accessible par le processeur du nœud en cours.

Exemple 4.12 – Modèle d'une connexion AADL

```
system implementation Case_Study.LEON
subcomponents
  -- Noeuds
  WoM : process Workload_Manager.Impl {
    -- Propriétés du noeud WoM
    };
  InS : process Interrupt_Simulator.Impl {
    -- Propriétés du noeud InS
    };

  -- Processeurs
  CPU_1 : processor The_Processor;
  CPU_2 : processor The_Processor;

  -- Bus
  B     : bus Ethernet_Bus;
connections
  -- Connexions des bus
  bus access B -> CPU_1.ETH;
  bus access B -> CPU_2.ETH;
  -- Connexions des noeuds
  event data port InS.External_Interrupt -> WoM.External_Interrupt_Depository
    {Actual_Connection_Binding => reference B;};
properties
  Actual_Processor_Binding => reference CPU_1 applies to WoM;
  Actual_Processor_Binding => reference CPU_2 applies to InS;
end Case_Study.LEON;
```

L'exemple de code 4.12 montre les connexions au niveau du système global de l'exemple WORKLOAD MANAGER. La clause CONNECTIONS montre les connexions entre les processeurs LEON2 et le bus SPACEWIRE. Elle montre aussi la connexion entre les deux nœuds de l'application et l'association de cette connexion au bus SPACEWIRE.

4.9.6 Modélisation des sous-programmes

Pour décrire un sous-programme AADL, nous utilisons le composant **subprogram**. Les paramètres du sous-programme sont spécifiés dans la section FEATURES du composant. Si le comportement du sous programme se résume à l'appel d'autres sous-programmes AADL, il suffit d'ajouter une section CALLS à l'implantation de ce sous-programme et d'y mettre les appels correspondants et de connecter les paramètres de ces appels entre eux et avec les paramètres du sous-programme appelant. Si l'implantation du sous-programme est fournie par l'utilisateur, nous utilisons les propriétés AADL standards. L'exemple de code 4.13 montre le modèle d'un sous programme **Do_Ping_Spg**. Nous pouvons voir que l'implantation de ce sous-programme est écrite en Ada et qu'elle s'appelle `Ping.Do_Ping_Spg`.

Exemple 4.13 – Modèle d'un sous-programme AADL

```
subprogram Do_Ping_Spg
features
  Data_Source : out parameter Simple_Type;
properties
  source_language => Ada95;
  source_name     => "Ping.Do_Ping_Spg";
end Do_Ping_Spg;
```

Les sous-programmes sont généralement appelés par les processus légers ou par d'autres sous-programmes en utilisant la section CALLS de l'implantation de l'appelant. De plus, les sous-programmes AADL peuvent avoir des **out event ports** ou des **out event data ports** pour permettre de déclencher des événements ou des événements-données sur les ports des composants **thread** auxquels ils sont connectés.

4.9.7 Modélisation des données

Les données modélisent les informations échangées entre les différentes entités de l'application répartie. Nous les décrivons en utilisant les composants **data**. Nous avons défini un ensemble de propriétés pour ces composants pour spécifier la nature de la donnée. Cet ensemble sera intégré sous forme d'annexe à la prochaine version du standard, AADLv2. L'exemple 4.14 décrit cet ensemble de propriétés.

AADL permet de spécifier plusieurs types de données. Nous les divisons en quatre familles principales.

Types simples

Les types simples qui sont décrits en AADL sont :

- Les booléens,
- Les entiers,
- Les réels (à virgule fixe ou flottante),
- Les caractères (simples ou étendus).

L'exemple de code 4.15 illustre quelques exemples de types simples de base. Nous utilisons l'ensemble de propriétés **Data_Model** pour spécifier la nature des données.

Exemple 4.14 – Ensemble de propriétés **Data_Model**

```
property set Data_Model is
  Base_Type : list of classifier (data) applies to (data);
  -- Type de base d'un composant

  Code_Set : aadlinteger applies to (data);
  -- Pour les caractères étendus

  Data_Digits : aadlinteger applies to (data);
  Data_Scale : aadlinteger applies to (data);
  -- Nombres à virgule fixe

  Data_Representation : enumeration
    (Array, Boolean, Character, Enum, Float,
     Fixed, Integer, String, Struct, Union)
  applies to (data);
  -- Propriété principale pour la définition des types de données

  Dimension : list of aadlinteger applies to (data);
  -- Dimensions d'un tableau

  Element_Names : list of aadlstring applies to (data);
  -- Noms des champs d'une strauctur e

  Enumerators : list of aadlstring applies to (data);
  -- Liste des enumérateurs d'un type énuméré

  IEEE754_Precision : enumeration (Simple, Double) applies to (data);
  -- Précision des nombres à virgule flottante

  Initial_Value : list of aadlstring applies to (data, port, parameter);
  -- Valeur initiale pour un type

  Integer_Range : range of aadlinteger applies to (data, port, parameter);
  -- Intervalle pour un type entier

  Measurement_Unit : aadlstring applies to (data, port, parameter);
  -- Unité d'une donnée

  Number_Representation : enumeration (Signed, Unsigned) applies to (data);
  -- Spécifie si le type est signé ou non

  Real_Range: range of aadlreal applies to (data, port, parameter);
  -- Intervalle pour un type réel

end Data_Model;
```

Exemple 4.15 – Type de données simples

```
-- Boolean type

data Boolean_Data
properties
  Data_Model::Data_Representation => Boolean;
end Boolean_Data;

-- Integer type

data Integer_Data
properties
  Data_Model::Data_Representation => Integer;
end Integer_Data;

-- Fixed point type

data Fixed_Point_Type
properties
  Data_Model::Data_Representation => Fixed;

  Data_Model::Data_Digits => 10;
  -- The total number of digits is 10

  Data_Model::Data_Scale => 4;
  -- The precision is 10**(-4)
end Fixed_Point_Type;

-- Character type

data Character_Data
properties
  Data_Model::Data_Representation => Character;
end Character_Data;
```

Types complexes

Les types complexes sont des types dont la définition est plus élaborée que les types simples. Ce sont :

- Les chaînes de caractères bornées (simples ou étendues),
- Les tableaux de longueur fixe,
- Les structures de données.

Nous nous intéressons principalement aux systèmes TR^{2}E dans lesquels la présence d'un type de taille inconnue à la compilation est potentiellement dangereuse car elle nécessite d'allouer dynamiquement de la mémoire au cours de l'exécution de l'application. Ainsi, les chaînes de caractères doivent avoir une longueur maximale spécifiée dans le modèle. Il en est de même pour les tableaux, il doivent avoir une taille fixe pour éviter toute allocation dynamique lors de l'exécution de l'application.

Les mêmes raisons évoquées plus haut nous mènent à interdire la définition des types pointeurs pour éviter de définir des structures de données récursives car ceci rend non bornée la taille de la donnée. Par ailleurs, un type de donnée de taille non bornée rend l'estimation du temps d'emballage et de déballage impossible à effectuer statiquement. L'exemple de code 4.16 illustre quelques exemples de types de données complexes modélisés avec AADL.

Exemple 4.16 – Type de données complexes

```
-- Chaine de caractères bornée

data String_Data
properties
  Data_Model::Data_Representation  => String;
  Data_Model::Max_Length           => 42;
end String_Data;

-- Tableau de taille fixe

data Data_Array
properties
  Data_Model::Data_Representation  => Array;
  Data_Model::Base_Type            => (data Integer_Data)
  Data_Model::Dimensions           => (42);
end Data_Array;

-- Structure de donnée

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

data implementation Data_Structure.i;
subcomponents
  Component_1 : data String_Data;
  Component_2 : data Data_Array;
end Data_Structure.i;
```

Types externes

Les types externes sont introduits pour utiliser dans les modèles AADL des types de données décrits par d'autres formalismes, comme ASN.1 [asn, 2002] par exemple. L'exemple de code 4.17 illustre un exemple de donnée AADL dont l'implantation est donnée en ASN.1.

Exemple 4.17 – Type de données ASN.1

```
data   ASN1_Type
properties
    -- Nom du fichier source ASN.1:
    Source_Text => ("my_types.asn");
    Source_Language => ASN1;
    -- Nom du type de donnée dans le fichier source:
    Type_Source_Name => "My_Type";
end T_POS;
```

Types de données avec accesseurs

Il existe des types de données qui requièrent des sous-programmes pour accéder à la donnée proprement-dite. Il s'agit notamment des données protégées pour lesquelles l'accès est fourni exclusivement par l'intermédiaire de sous-programmes garantissant la bonne gestion de la concurrence en utilisant des verrous logiciels. Ceci garantit qu'un seul processus léger accède à la donnée. Nous modélisons ces données en ajoutant aux composants **data** des sous-programmes dans la section FEATURES. Le caractère protégé ou non protégé de la donnée est spécifié en utilisant des propriétés AADL ainsi que le protocole utilisé pour géré l'inversion de priorité en cas de concurrence.

L'exemple de code 4.18 illustre une donnée protégée, `Protected_Object.Impl`. Cette donnée possède un champ unique qui est d'un type simple. Il est notable que la définition des sous-programmes qui sont utilisés comme accesseurs est légèrement différente des sous-programmes AADL classiques. En effet, chacun de ces sous programmes doit avoir accès aux parties *internes* de la donnée protégée. On utilise le mécanisme **requires data access** AADL pour ce faire. Nous précisons aussi les droits d'accès à la donnée pour chacun des sous-programmes par l'intermédiaire de la propriété `Required_Access`).

4.10 Synthèse

Dans ce chapitre, nous avons introduit le langage AADL. Nous utilisons ce formalisme dans la suite de ce mémoire pour atteindre les objectifs de notre travail de thèse. Nous avons décrit les différentes spécificités de ce langage et nous nous sommes concentrés sur celles qui nous aideront à spécifier, configurer et déployer les systèmes TR^{2}E. Le lecteur intéressé par ce langage trouvera plus de détails et d'exemples d'utilisation de ce langage dans [Feiler *et al.*, 2006].

Tout au long de notre travail de thèse, le langage AADL a poursuivi son évolution. La deuxième version de ce standard, AADLv2 devrait apporter davantage de fonctionnalités et rendre plus souple la description de systèmes TR^{2}E notamment par l'intermédiaire de nouvelles catégories de composants. En effet, deux nouveaux composants matériels seront ajoutés à AADLv2, les processeurs virtuels et les bus virtuels. Les premiers permettent de modéliser l'isolation spatiale et temporelle entre les partitions d'une application répartie critique. Les seconds permettent de décrire plus précisément les protocoles de communication. En ce qui concerne le prototypage, les composants abstraits ainsi que l'introduction des accès aux sous-programmes dans les éléments d'interface faciliteront le développement et le raffinement des modèles tout au long de leur conception.

De nombreuses améliorations ont été proposées par l'équipe de recherche dans laquelle nous travaillons, suite à nos travaux. Nous avons contribué en particulier aux aspects de la modélisation des données et de la génération de code grâce à l'avancement de notre équipe sur

Exemple 4.18 – Type de données protégées

```
-- Data type of object field

data Field_Type
properties
  Data_Model::Data_Type => Integer;
end Field_Type;

-- Protected data type

data Protected_Object
features
  Update : subprogram Protected_Update;
  Read   : subprogram Protected_Read;
properties
  Concurrency_Control_Protocol => Priority_Ceiling;
end Protected_Object;

-- The implementation of the protected object

data implementation Protected_Object.Impl
subcomponents
  Field : data Field_Type;
end Protected_Object.Impl;

-- Subprograms

subprogram Protected_Update
features
  this : requires data access Protected_Object.Impl
  {required_access => access Read_Write;}; -- Mandatory
  P    : in parameter Field_Type;
properties
  source_language => Ada95;
  source_name     => "Repository";
end Protected_Update;

subprogram Protected_Read
features
  this : requires data access Protected_Object.Impl
  {required_access => access Read_Only;}; -- Mandatory
  P    : out parameter Field_Type;
properties
  source_language => Ada95;
  source_name     => "Repository";
end Protected_Read;
```

ces deux domaines. Cependant, cette nouvelle version de AADL ne pourra être utilisée dans le cadre de nos travaux parce qu'elle n'est pas encore publiée[3]. Nous nous fondons uniquement sur la première version du standard, AADL 1.0 dans tout le reste de ce mémoire.

Après avoir constaté que le langage AADL est généraliste, nous avons défini des restrictions additionnelles aux modèles pour les rendre conformes aux recommandations des systèmes critiques. Nous avons aussi donné un ensemble de directives sur la manière de modéliser les différents composants d'une application TR^{2}E.

Le langage AADL constitue l'épine dorsale de tout notre processus de production. Il permet, en particulier grâce aux propriétés spécifiques au déploiement et la configuration que nous avons introduites, de rendre automatique la conception d'applications TR^{2}E à partir de leurs modèles. Le prochain chapitre présentera le processus global de production des systèmes TR^{2}E que nous proposons.

3. Le standard AADLv2 sera publié en janvier 2009

Chapitre 5

Processus de Production Automatique de Systèmes TR^{2}E

SOMMAIRE

5.1 Introduction

Dans les chapitres précédents, nous avons décrit l'architecture d'un intergiciel dédié à une application TR^{2}E (chapitre 3). Nous avons, pour ceci, classé les composants intergiciels en deux familles, les composants faiblement personnalisables (formant le noyau de l'intergiciel minimal) et les composants fortement personnalisables (produits automatiquement). Pour générer les composants de la seconde famille, nous avons choisi de modéliser les applications TR^{2}E à l'aide du

langage de description d'architecture AADL. Dans le chapitre 4, nous avons présenté ce langage ainsi que les avantages qu'il apporte. Nous avons défini des restrictions supplémentaires pour ce langage afin de permettre la description de systèmes TR²E statiquement analysables. Nous

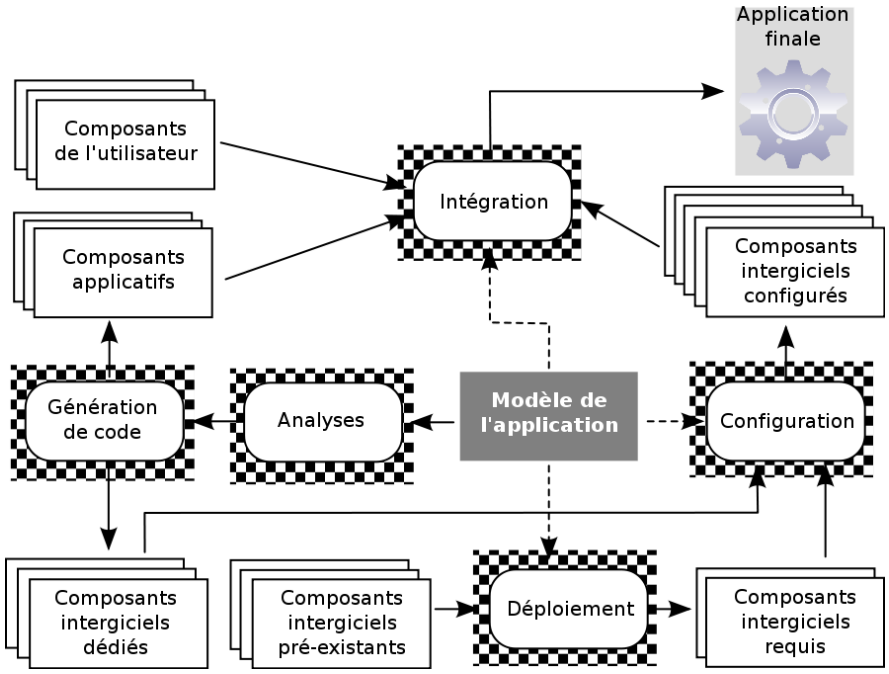

FIGURE 5.1 – Nouveau processus de conception d'une application TR²E

Dans ce chapitre, nous décrivons le processus global de production d'applications réparties. La figure 5.1 met en évidence les étapes de ce processus auxquelles nous nous intéressons dans ce chapitre. La figure 5.2 Détaille chacune de ces étapes. Le point de départ est un modèle en AADL décrivant une application TR²E. La majeur partie de ce modèle est écrite par l'utilisateur en respectant les restrictions à ce langage que nous avons décrites dans la section 4.9. Pour spécifier les propriétés non architecturale des composants décrits, le développeur utilise les propriétés standards du langage AADL. En cas d'absence de propriétés standards spécifiant une caractéristique particulière (priorité d'une tâche, catégorie d'un type de donnée...), le développeur utilise un ensemble de propriétés qui complète les ensembles standards. Nous avons défini dans la section 4.9 un ensemble de propriétés qui permet de modéliser les types de données. La nouvelle version du standard, AADLv2 intégrera cet ensemble sous forme d'une annexe. Les autres propriétés manquantes (priorité...) ont aussi été intégrées dans AADLv2. Toutes ces propriétés enrichissent l'expressivité du modèle et permettent de spécifier de nombreuses caractéristiques des composants :

- Caractéristiques fonctionnelles comme les noms des implantations des sous-programmes AADL donnés sous forme de fichiers sources ou de bibliothèques,
- Caractéristiques non fonctionnelles comme les périodes des tâches ou leur échéances,
- Caractéristiques architecturales comme l'association entre les processus et les processeurs sur lesquels ils sont exécutés.

Comme le montre la figure 5.2, le processus de production que nous proposons est constitué de quatre étapes principales.

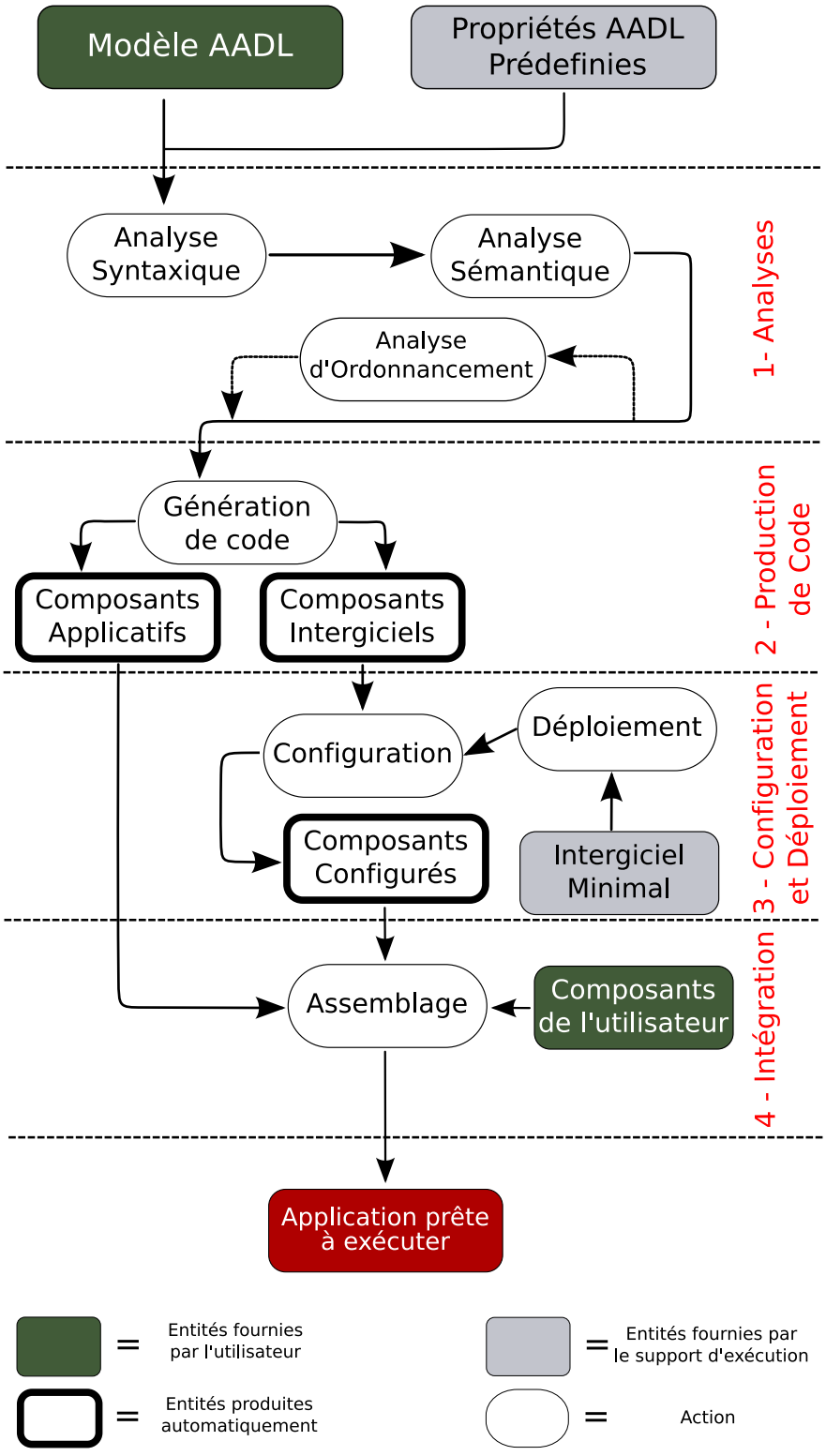

FIGURE 5.2 – Processus de production détaillé

1) L'analyse Une fois l'application TR²E modélisée en AADL, son modèle subit une suite d'analyses afin de garantir que le modèle est cohérent. Ces analyses consistent à vérifier la syntaxe et la sémantique du modèle vis-à-vis du standard AADL. Nous vérifions aussi que le modèle est créé en utilisant le sous ensemble du langage AADL que nous avons défini dans la section 4.9 et qu'il respecte les restrictions données dans cette même section. Ceci permet par la suite d'avoir un code généré conforme au profil Ravenscar (voir 2.5.1). Des analyses optionnelles effectuées par des outils externes sont aussi possibles (comme l'analyse d'ordonnançabilité par exemple). Ces analyses effectuées sur le modèle AADL renforcent la sûreté de fonctionnement de l'application.

2) La production de code Après la phase d'analyse, le modèle AADL est utilisé pour produire automatiquement deux familles de composants de l'application répartie :

1. les composants applicatifs qui constituent généralement les types de données ainsi que les sous-programmes utilisés par l'utilisateur. Ces composants sont produits automatiquement à partir du modèle AADL, notamment à partir des composants **data** et **subprogram**. Les produire automatiquement, avec des conventions de nommage précises et connues par l'utilisateur a un double bénéfice. Tout d'abord, il permet d'avoir une traçabilité du code tout au long du processus de développement puisqu'il est facile de retrouver les entités de code sources qui ont été générées à partir d'une entité AADL : elles ont généralement le même nom ou des noms très proches. Ensuite, ceci permet d'utiliser les entités générées automatiquement dans d'autres endroits du code généré mais aussi dans le code écrit par l'utilisateur. Les règles de transformation que nous utilisons sont similaires aux règles de projection des descriptions IDL de CORBA vers un langage de programmation spécifique,
2. les composants intergiciels fortement personnalisables en fonction des propriétés de l'application et que nous avons décrits dans la section 3.4. Ces composants sont produits automatiquement en fonction de la topologie de l'application et en particulier l'interaction entre les composants de types **process** et **thread**. Leur optimisation en fonction de la plate-forme est effectuée en analysant les composants matériels présents dans le modèle (**processus**, **bus**...).

3) Le déploiement et la configuration Cette étape consiste à sélectionner les services de l'intergiciel minimal qui sont utilisés par un nœud donné. La sélection est effectuée automatiquement grâce aux informations de déploiement extraites en particulier des composants matériels du modèle AADL (caractéristiques des composants **processor** et **bus** par exemple). Ces services sélectionnés, ainsi que ceux qui sont générés automatiquement, sont configurés en fonction des propriétés du nœud. À l'issue de ces deux étapes est produit un ensemble de services intergiciels optimisés et configurés aux besoins de chacun des nœuds de l'application.

4) L'intégration Cette phase consiste à rassembler les composants applicatifs produits pendant la phase de génération de code, les composants intergiciels configurés lors de la phase de déploiement et de configuration et les composants fournis par l'utilisateur. Cette édition des liens permet de produire les exécutables correspondant au nœuds de l'application répartie. À l'issue de cette phase, nous disposons d'une application prête à être exécutée.

Dans la suite de ce chapitre, nous détaillons chacune de ces quatre étapes. La section 5.2 présente les analyses que nous effectuons sur les modèles AADL pour valider leur bon fonctionnement. Dans les sections 5.3 et 5.4, nous donnons les règles de transformation des modèles

AADL pour produire respectivement les composants applicatifs et intergiciels. La section 5.5 décrit la manière dont les composants intergiciels sont déployés et configurés. Dans la section 5.6, nous présentons comment tous les composants logiciels sont intégrés automatiquement pour former une application répartie prête à être exécutée. Enfin, la section 5.7 conclut le chapitre.

5.2 Analyses

La phase d'analyse permet de vérifier statiquement le respect des règles et les propriétés du langage. Plusieurs analyses sont effectuées sur les modèles AADL. Nous vérifions tout d'abord la syntaxe et la sémantique du modèle vis-à-vis du standard mais aussi vis-à-vis des restrictions additionnelles que nous avons introduites pour garantir le respect des recommandations pour les systèmes TR^{2}E. Ensuite nous vérifions la cohérence des propriétés spécifiées par l'utilisateur. Enfin, nous effectuons une analyse avancée d'ordonnancement.

5.2.1 Analyse syntaxique et sémantique

Il s'agit, lors de ces analyses, de valider que le modèle AADL est conforme au standard et aux restrictions additionnelles présentées dans la section 4.9. Il s'agit d'une opération complètement automatisée qui réalise, l'une après l'autre, les trois tâches suivantes :

1. Vérification de la syntaxe du modèle AADL pour garantir qu'il respecte la grammaire du standard. Cette analyse est effectuée sur l'arbre syntaxique du modèle AADL,
2. Vérification de règles sémantiques élémentaires données par le standard (*legality rules*). Ces règles permettent de garantir que le modèle décrit un système réalisable. Par exemple, il est interdit pour un processus lourd de contenir directement des appels à des sous-programmes car il ne représente pas une entité active. Il faut qu'un processus contienne des processus légers qui, eux, contiennent les appels aux sous-programmes. Cette analyse est aussi effectuée sur l'arbre syntaxique du modèle. Si elle se déroule avec succès, alors le modèle AADL peut être instancié,
3. La vérification des restrictions additionnelles que nous avons spécifiées s'effectue en dernier lieu sur l'arbre d'instance du modèle AADL. Elle permettent notamment de garantir le caractère statique de l'application, que ce soit du point de vue de l'ordonnancement que celui de l'utilisation des ressources.

La figure 5.3 montre le profil AADL pour application TR^{2}E que nous sommes capables d'analyser statiquement et pour laquelle nous générons du code. Le composant de plus haut niveau de toute application répartie doit être un système. Ce système contient obligatoirement un processeur ou plus et un processus ou plus. Si l'application est répartie et que les nœuds s'exécutent sur des processeurs différents, le système doit contenir au moins un bus de donnée qui lie ces processeurs.

Chaque processus lourd doit contenir au moins un processus léger (*thread*). Il peut contenir optionnellement des données partagées. Les processus légers peuvent contenir des éléments d'interfaces : de ports de type événement, donnée... Ils peuvent appeler un ou plusieurs sous-programmes. Les sous-programmes AADL peuvent avoir soit des paramètres soit des ports en sortie et peuvent appeler d'autres sous-programmes.

Une telle topologie découle naturellement des restrictions que nous avons définies pour avoir un profil de AADL qui soit utilisable dans le contexte des systèmes TR^{2}E. L'obligation d'avoir des composant matériels dans le modèle (processus, bus) a été introduite pour forcer l'utilisateur

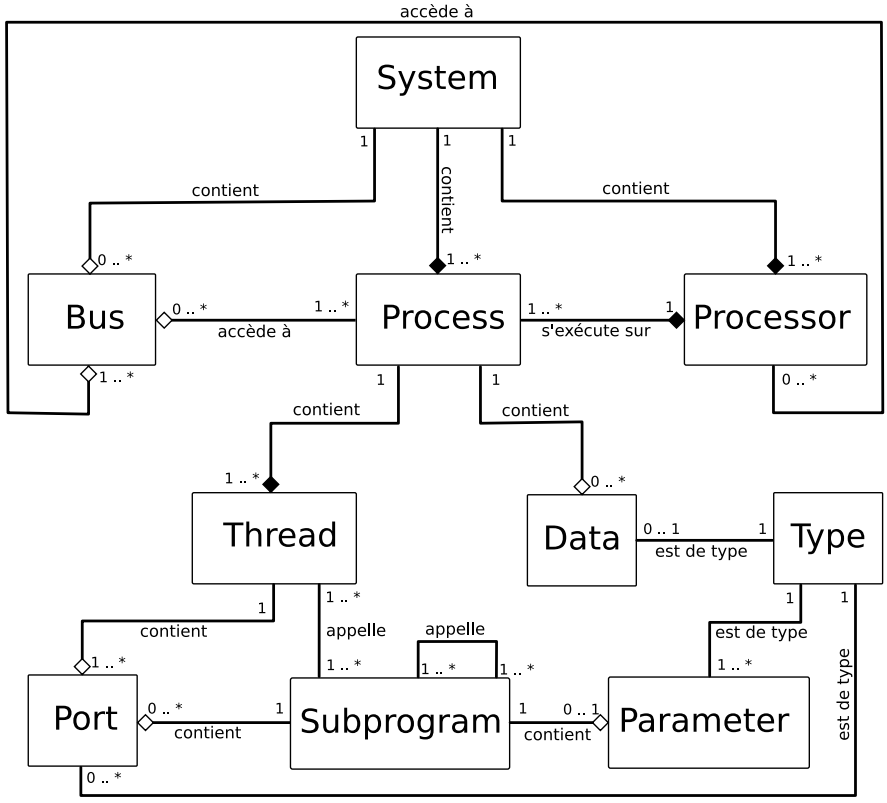

FIGURE 5.3 – Architecture générique d'une application

à incorporer des informations de déploiement dans son modèle. Ces informations servent à sélectionner et à configurer les composants intergiciels mais aussi à choisir la bonne instance du compilateur pour intégrer tous les composants et produire l'application prête à être exécutée. Un modèle d'instance qui vérifie cette topologie et qui est conforme aux restrictions données dans la section 4.9 possède un degré minimal de cohérence et est prêt pour subir l'analyse suivante.

5.2.2 Cohérence des spécifications

Lors de cette analyse, nous vérifions que les spécifications données par l'utilisateur sont cohérentes et permettent de produire un système réel opérationnel. Nous vérifions particulièrement que les entités logicielles modélisées ne requièrent pas l'utilisation de plus de ressources matérielles que celles disponibles dans le système. Il s'agit d'une série d'analyses simples qui permettent de s'assurer que l'application n'est pas *trivialement irréaliste*. En particulier :

- Nous vérifions que chacun des processus légers ne manipule pas des données dont la taille est plus grande que la taille de sa pile. En particulier la taille potentielle des piles des sous-programmes doit être inférieure à la taille de la pile du processus léger qui les appelle.
- Nous vérifions que la somme de la taille des piles de tous les processus légers ainsi que la taille des données partagées est inférieure à la taille de la mémoire disponible pour le nœud qui les contient. Il est évident que la taille calculée sera inférieure à la taille réelle de l'exécutable. Nous ne testons ici que la condition nécessaire.
- Nous vérifions que si une connexion logique existe entre deux nœuds de l'application qui s'exécutent sur deux processeurs différent, alors nécessairement, le support matériel pour cette connexion logique existe (un ou plusieurs bus). Il faut aussi que l'utilisateur associe explicitement dans le modèle une telle connexion à un bus AADL.
- Nous vérifions que la somme des bandes passantes des connexions logiques associés à un bus est inférieure à la bande passante totale du bus.
- Nous vérifions que les données comportementales spécifiées par l'utilisateur sont correctes et qu'elles n'entrent pas en conflit les unes avec les autres.

D'autres travaux dans notre équipe explorent permettent de spécifier et de vérifier les contraintes d'un modèle AADL à l'aide d'un langage de contrainte spécifique [Gilles and Hugues, 2008b]. Les contraintes sont données dans les clauses d'annexes des composants AADL.

Notons que nous ne mettons pas les analyses avancées d'ordonnancement dans cette classe d'analyses bien que l'échec d'ordonnancement soit lui aussi une tentative d'utilisation de ressources matérielles non disponibles.

5.2.3 Analyses statiques avancées

Après avoir construit un modèle AADL valide, cohérent et se prêtant à des analyses statiques, nous effectuons les analyses avancées. Il est important de noter que notre but n'est pas d'implanter les analyses mais de profiter (quand cela est possible) de l'existence d'outils tiers pour créer des interfaces vers ces outils pour qu'ils analysent nos modèles AADL.

Pour avoir un système multi-tâches fiable qui fonctionne correctement dans un domaine critique, il faut que ce système vérifie un ensemble de conditions. Par exemple, le projet ASSERT dans le cadre duquel s'est déroulée une partie de ce travail de thèse a fixé trois conditions nécessaires au bon fonctionnement en se basant sur l'expérience de l'Agence Spatiale Européenne dans la conception de systèmes TR^{2}E :

1. Il faut que l'ensemble des tâches qui constituent ce système soit ordonnançable : toutes les tâches doivent satisfaire leur échéances. Pour garantir cette condition, une analyse d'ordonnancement doit être faite sur le modèle AADL de l'application. Concrètement, des règles de transformation sont définies pour convertir les modèles AADL en des structures analysables par CHEDDAR, un analyseur d'ordonnancement temps-réel [Singhoff *et al.*, 2004],
2. Il faut aussi que le fonctionnement du système ne comporte pas d'interblocage dû à une interconnexion erronée entre les nœuds ou les processus légers. Pour ce faire, il faut effectuer de la vérification de modèle soit directement sur AADL soit en le transformant en un autre formalisme (réseau de Petri, par exemple),
3. Enfin, il faut que certaines pannes soient tolérées et traitées (rupture momentanée de la connexion...). Le mécanisme des modes opérationnels dans AADL nous permet d'implanter cette tolérance aux pannes. Ainsi, un processus léger passe d'un mode nominal vers un mode d'urgence s'il reçoit un événement adéquat.

À l'issue de ces analyses, nous commençons la génération de code. Deux types de composants sont générés automatiquement : les composants applicatifs et les composants intergiciels.

5.3 Génération de Composants Applicatifs

Dans cette partie, nous donnons les règles de transformation des différents composants AADL vers des constructions d'un langage de programmation impératif. Nous voulons que ces transformations soient les plus générales possibles. Nous donnerons donc un algorithme générique qui permet de parcourir l'entité AADL correspondante et de produire les constructions voulues. Nous donnons aussi quelques exemples d'entités AADL suivie de leur transformées respectives. Pour chaque langage de programmation, l'ensemble des algorithmes seront implantés en tenant compte des spécificités de chaque langage. La séparation entre les différents services intergiciels ainsi que celle entre les différents composants d'un même service rendent l'architecture de notre intergiciel simple. Ceci facilite la tâche de définition des règles de transformation vers un langage impératif.

Il faut noter ici que les algorithmes que nous présentons ne précisent pas nécessairement les constructions exactes qui seront générées. Par exemple pour ordonner la génération d'un type entier, nous invoquons une primitive générique qui s'appelle **Define_Integer_Type** en lui passant les paramètres nécessaires pour la définition (le nom et la taille du type...) :

GÉNÉRATION D'UN TYPE ENTIER ()
1 $Define\_Integer\_Type(< Nom\_Du\_Type >, < Taille\_Du\_Type >)$

Ainsi pour le langage Ada, si nous désirons définir un type entier de taille $32\ bits$, cette primitive donne lieu à la génération du code suivant :

```
type <Nom_Du_Type> is new Interfaces.Integer_32;
```

Tandis que pour le langage C, nous aurons la construction suivante :

```
typedef int_32 <Nom_Du_Type>;
```

Les différences entre les deux constructions ci-dessus ne sont fondamentales. Cependant, il existe des situations où le code Ada est fondamentalement différent du code C. Par exemple,

si nous voulons créer un fil d'exécution qui effectue, à chaque seconde, l'appel à un sous-programme **Hello**, dans notre algorithme nous mettrons :

PERIODIC_THREAD ()
1 $Create\_Periodic\_Thread(< Nom\_Du\_Thread >, Hello, 1Sec...)$

En Ada, les fils d'exécution sont supportés intrinsèquement grâce à la notion de *tâche*. La construction du fil d'exécution sera effectuée de la manière suivante :

```
task <Nom_Du_Thread> is
   pragma Priority (...);
end <Nom_Du_Thread>;

task body <Nom_Du_Thread> is
   Period      : constant Ada.Real_Time.Time_Span := Seconds (1);
   Next_Start : Ada.Real_Time.Time;
begin
   Next_Start := Ada.Real_Time.Clock + Period;

   loop
      Hello;

      delay until Next_Start;
      Next_Start := Next_Start + Task_Period
   end loop;
end <Nom_Du_Thread>;
```

Par contre, le langage C, ne supporte pas les processus légers de manière intrinsèque. Il faut encapsuler le comportement périodique dans un sous-programme et passer par des appels à des bibliothèques comme POSIX pour créer un processus léger qui exécute le sous-programme. Nous voyons d'après l'exemple ci-dessous que l'absence de ce support intrinsèque rend le code C beaucoup plus compliqué que le code en Ada.

```
pthread_mutex_t mutex;
pthread_cond_t  cond;

void delay_until (struct timespec *next_start)
{
  pthread_mutex_lock (mutex);

  pthread_cond_timedwait (&cond, &mutex, &next_start);
  /* Attente avec un temps d'expiration */

  pthread_mutex_unlock (&mutex);
}

void wrapper (void) {

  struct timespec period;
  struct timeval next_start;
  struct timeval now;

  /* Initialisation du verrou et de la variable conditionnelle */
  pthread_mutex_init (& mutex, NULL);
  pthread_cond_init (& cond, NULL);

  period.tv_sec  = 1;   /* Une seconde */
  period.tv_nsec = 0;

  gettimeofday (&now);
  next_start.tv_sec = now.tv_sec + period.ts;
  next_start.tv_nsec = now.tv_usec * 1000 + period.tv_nsec;

  while (1) {
```

```
      hello ();

      /* Attente de la prochaine occurrence */
      delay_until (&next_start);

      /* Mise à jour de la prochaine occurrence de la période */
      next_start.tv_sec  = next_start.tv_sec  + period.ts;
      next_start.tv_nsec = next_start.tv_nsec + period.tv_nsec;
   }
}

/* Création du fil d'exécution */
pthread_create (&tid, NULL, wrapper, NULL);
```

Il faut noter aussi que nous disposons de routines qui permettent d'extraire des composants AADL les valeurs des propriétés ou encore pour la liste des sous-composants par exemple. Ainsi, pour obtenir la période d'un processus léger AADL **T**, nous utilisons la valeur de la propriété standard **Period** associée à ce processus léger à l'aide de la routine **Get_Property_Value** :

GET_PROPERTY_VALUE ()
1 $Get\_Property\_Value(T, \text{“}Period\text{”})$

5.3.1 Transformation des données

Les types de données sont déduits des instances de composants AADL de type **data**. Ces composants sont instanciés à plusieurs endroits du modèle :

- dans les éléments d'interface de type `data` ou `event data` qui se trouvent dans les processus lourds, les processus légers ou encore les sous-programmes,
- dans les paramètres des sous-programmes et
- lors de la déclaration de variables partagées entre plusieurs entités du modèles.

Ces déclarations sont transformées en définitions de types. Les types définis sont utilisés pour définir les différentes instances. Le caractère asynchrone que nous avons imposé aux communications de nos applications réparties nous oblige à définir une valeur “par défaut” pour chacun de ces types de données définis. Cette valeur est utilisée au moment où les données ne sont pas encore disponibles (à l'initialisation de l'application par exemple ou lors d'une panne partielle qui rend la communication momentanément indisponible). Ainsi, une déclaration d'une constante de ce type ayant une valeur “par défaut” pour ce type est produite pour chaque déclaration de composant **data**. Ce mécanisme est utile uniquement dans les processus légers périodiques et hybrides car se sont les seuls à pouvoir être réveillé par un événement temporel sans avoir reçu aucune donnée valide.

Comme le montre la règle de transformation 5.1, le point de départ est une déclaration de type ou d'implantation d'un composant `data` qui est caractérisée par son nom, son type, ses éventuels sous-composants, ses éventuels accesseurs (dans la section FEATURES) et de son éventuel protocole pour gérer la concurrence s'il s'agit d'un type de donnée avec accesseurs. La déclaration AADL de ce composant doit être transformée en une définition de type dans le langage de programmation choisi. Par convention, le nom de ce type est le même que le nom du composant AADL sauf si ce dernier ne respecte pas les règles de nommage avec le langage ; auquel cas, les instances de projection doivent fournir des moyens pour convertir les noms des composants en noms compatibles avec le langage de programmation (similaires aux règles de projection de l'IDL CORBA vers les différents langages de programmation). Par exemple, le langage Ada interdit de définir des identificateurs qui contiennent des occurrences successives du

Algorithme 5.1 Transformation des types de données AADL

```
TRANSFORM_DATA (D : AADL_Data_Instance)
 1  Category ← GET_PROPERTY_VALUE (D, "Data_Model :: Data_Representation")
 2  switch
 3    case Category = Data_Integer :
 4         DEFINE_INTEGER_TYPE (MAP_IDENTIFIER (NAME (D)))
 5    case Category = Data_Boolean :
 6         DEFINE_BOOLEAN_TYPE (MAP_IDENTIFIER (NAME (D)))
 7    case Category = Data_Float :
 8         if Float_Supported then
 9           DEFINE_FLOAT_TYPE (MAP_IDENTIFIER (NAME (D))
10         else
11           error "Floating point types are not supported !"
12    case Category = Data_Fixed :
13         if Fixed_Supported then
14           Scale ← GET_PROPERTY_VALUE (D, "Data_Model :: Data_Scale")
15           Total ← GET_PROPERTY_VALUE (D, "Data_Model :: Data_Total")
16           DEFINE_FIXED_TYPE (MAP_IDENTIFIER (NAME (D)), Scale, Total)
17         else
18           error "Fixed point types are not supported !"
19    case Category = Data_Character :
20         DEFINE_CHARACTER_TYPE (MAP_IDENTIFIER (NAME (D)))
21    case Category = Data_String :
22         Max_Length ← GET_PROPERTY_VALUE (D, "Data_Model :: Max_Length")
23         DEFINE_BNOUNDED_STRING_TYPE (MAP_IDENTIFIER (NAME (D)), Max_Length)
24    case Category = Data_Array :
25         Length ← GET_PROPERTY_VALUE (D, "Data_Model :: Dimensions")
26         DEFINE_ARRAY_TYPE (MAP_IDENTIFIER (NAME (D)),
27                            GET_PROPERTY_VALUE (D, "Data_Model :: Base_Type"),
28                            Max_Length)
29    case Category = Data_Structure :
30         Field_List ← Empty_List
31         for each S in SUBCOMPONENTS (D) do
32           TRANSFORM_DATA (S)
33           Field ← CREATE_FIELD (NAME (S), GET_MAPPED_TYPE (S))
34           APPEND (Field, Field_List)
35         DEFINE_STRUCTURE_TYPE (MAP_IDENTIFIER (NAME (D)),
36                                Field_List)
37    case Category = Data_With_Accessors :
38         Field_List, Accessor_List ← Empty_List
39         for each S in SUBCOMPONENTS (D) do
40           TRANSFORM_DATA (S)
41           Field ← CREATE_FIELD (NAME (S), GET_MAPPED_TYPE (S))
42           APPEND (Field, Field_List)
43         for each A in FEATURES (D) do
44           TRANSFORM_SUBPROGRAM (A)
45           Accessor ← CREATE_ACCESSOR (NAME (A),
46                                       GET_MAPPED_SUBPROGRAM (A))
47           APPEND (Accessor, Accessor_List)
48         if GET_PROPERTY_VALUE (D, "Concurrency_Control_Protocol") = Priority_Ceiling then
49           DEFINE_PROTECTED_TYPE (MAP_IDENTIFIER (NAME (D)), Field_List, Accessor_List)
50         else
51           DEFINE_UNPROTECTED_TYPE (MAP_IDENTIFIER (NAME (D)), Field_List, Accessor_List)
52    case default :
53         error "Cannot map this data component instance !"
```

caractère "_". Les identificateurs venant du modèle AADL et qui ne respectent pas cette restriction sont modifiés en insérant un caractère différent de "_" entre deux occurrences successives de ce dernier.

Exemple 5.1 – Types de données en AADL

```
data Integer_Type
properties
  Data_Model::Data_Representation => Integer;
end Integer_Type;

data Array_Of_Integers
properties
  Data_Model::Data_Representation => Array;
  Data_Model::Dimension => (10);
  Data_Model::Base_Type => (data Integer_Type);
end Array_Of_Integers;
```

Exemple 5.2 – Code Ada généré

```
-- Transformation du composant AADL
-- Integer_Type.

type Integer_Type is new Integer;

-- Transformation du composant AADL
-- Array_Of_Integers.

type Array_Of_Integers is array
  (1 .. 10) of Integer_Type;
```

L'extrait de code 5.1 montre un exemple en AADL d'un type de donnée entier ainsi qu'un type tableau qui contient 10 entiers. L'extrait de code 5.2 montre une transformation en langage Ada de ces deux composants. En l'absence d'informations supplémentaires de la part de l'utilisateur (taille du type entier, intervalle...), le type est simplement une extension du type standard **Integer**. L'utilisateur précise davantage la définition d'un type en utilisant les propriétés fournies dans l'ensemble de propriétés prédéclaré **Data_Model**. Nous avons défini cet ensemble de propriétés pour pallier le manque de spécification dans la première version du standard AADL. Cet ensemble fera partie des fichiers de propriétés "annexes" de la prochaine version AADLv2.

La nature du type défini dépend fortement de la valeur de la propriété **Data_Model : :Data_-Representation**. Si le composant AADL représente une structure de donnée, il sera transformé en un type enregistrement avec des champs dont les types correspondent aux projections de leur déclarations respectives.

En cas de présence de sous-programmes accesseurs, leurs implantations doivent être conformes au protocole de gestion de concurrence du composant. Nous voulons que l'accès aux données soit protégé contre la concurrence et que l'inversion de priorité due à cette protection soit bornée. Nous voulons aussi garantir l'absence d'interblocage que pourrait causer cette protection. Une solution est d'utiliser un des protocoles d'héritage de priorités. Les recommandations du profil Ravenscar décrites dans la section 2.5.1 imposent l'utilisation du protocole de plafonnement de priorité PCP [Sha *et al.*, 1990] par ce qu'il garantit l'absence d'interblocage mais aussi parce qu'il réduit considérablement le temps de blocage d'une tâche (causé par l'inversion de priorité).

Peu de langages de programmation impératifs disposent de manière intrinsèque des constructions nécessaires pour ces variables protégées (comme en Ada par exemple). Pour les autres langages de programmation (comme le langage C), nous sommes obligés d'utiliser des bibliothèques système pour créer et initialiser les verrous logiciels nécessaires à la protection des données. Ceci rend le code C beaucoup plus compliqué que le code Ada et augmente la taille du code de l'intergiciel minimal rendant sa vérification et sa certification plus coûteuses.

La règle de transformation 5.2 illustre la manière de calculer et déclarer la constante correspondant à un type AADL particulier. Chaque type possède une valeur conventionnelle utilisée pour initialiser les variables de ce type. Les types de données avec accesseurs ne donnent pas lieu à une telle définition de constante car les variables de ce type ne doivent pas être transmises à travers le réseau et sont soumises à des règles d'initialisation spéciales (initialisation des verrous logiciels...). Les types structures de données ont pour valeur par défaut, une structure contenant récursivement les valeurs par défaut de tous les champs.

Algorithme 5.2 Déclaration des constantes pour les types AADL

```
COMPUTE_DEFAULT_VALUE (D : AADL_Data_Instance)
 1  Category ← GET_PROPERTY_VALUE (D, "Data_Model :: Data_Representation")
 2  switch
 3    case Category = Data_Integer :
 4         Literal ← Make_Literal(0)
 5    case Category = Data_Float or Category = Data_Fixed :
 6         Literal ← Make_Literal(0.0)
 7    case Category = Data_Boolean :
 8         Literal ← Make_Literal(false )
 9    case Category = Data_Character :
10         Literal ← Make_Literal(< space >)
11    case Category = Data_String :
12         Literal ← Make_Literal(< empty_string >)
13    case Category = Data_Array :
14         Length ← GET_PROPERTY_VALUE (D, "Data_Model :: Dimensions")
15         Element_Literal← GET_DEFAULT_VALUE
16                          (GET_PROPERTY_VALUE (D, "Data_Model :: Base_Type"))
17         Literal ← Make_Literal(Element_Literal, Length)
18    case Category = Data_Structure :
19         Field_List ← Empty_List
20         for each S in SUBCOMPONENTS (D) do
21           Field ← GET_DEFAULT_VALUE (S)
22           APPEND (Field, Field_List)
23         Literal ← Make_Literal(Field_List)
24    case Category = Data_With_Accessors :
25         NIL
26    case default :
27         error "Cannot map this data component instance !"
28  if Category ≠ Data_With_Accessors then
29    DEFINE_CONSTANT (MAP_CONSTANT_IDENTIFIER (NAME (D)),
30                     GET_MAPPED_TYPE (D),
31                     Literal)
```

Exemple 5.3 – Exemples de types AADL

```
data Component_Type
properties
  Data_Model::Data_Representation => Integer;
end Component_Type;

-- Une structure de donnée contenant trois
-- Entiers
data Record_Type
properties
  Data_Model::Data_Representation => Struct;
end Record_Type;

data implementation Record_Type.Impl
subcomponents
  X : data Component_Type;
  Y : data Component_Type;
  Z : data Component_Type;
end Record_Type.Impl;
```

Exemple 5.4 – Code Ada généré

```
-- Valeur par défaut pour le type
-- Component_Type.

Component_Type_Default_Value : constant
    Component_Type := 0;



-- Valeur par défaut pour le type
-- Record_Type_Impl.

Record_Type_Impl_Default_Value : constant
    Record_Type_Impl :=
      (X => Component_Type_Default_Value,
       Y => Component_Type_Default_Value,
       Z => Component_Type_Default_Value);
```

L'exemple de code 5.3 montre un exemple d'un type structure de données modélisé en AADL. Il s'agit d'une structure de trois entiers (représentant une coordonnée dans l'espace par exemple). La constante par défaut générée pour cette structure est une structure ayant pour chaque champ la valeur par défaut associée à son type. L'exemple de code 5.4 montre la génération en Ada d'une telle constante.

5.3.2 Transformation des sous-programmes

Les sous-programmes AADL sont des entités destinées à abriter le code de l'utilisateur. La section FEATURES contient éventuellement des paramètres pour échanger les données avec l'extérieur. En AADL, les sous-programmes sont des entités passives qui doivent être appelées par des processus légers pour s'exécuter. Ainsi, les composants sous-programmes se transforment naturellement en sous-programmes du langage de programmation impératif choisi pour la génération de code.

Le comportement du sous-programme produit dépend de la nature du composant AADL. Nous classons les sous-programmes AADL en trois catégories principales selon les valeurs des propriétés standards `Source_Language` et la présence ou non de la clause CALLS dans l'implantation du composant. Il s'agit des sous-programmes *opaques* implantés dans un langage de programmation particulier, des sous-programmes *à séquence pure d'appel* et des sous-programmes *vides*.

Grâce à la présence de port en sorties, les sous-programmes AADL (toutes catégories confondues) deviennent capables de déclencher des événements et d'événements-données du coté de leur appelant. Ceci permet de contrôler l'envoi d'événements à partir d'un processus léger par l'intermédiaire du code de l'utilisateur qui utilise les méthodes définies par le standard AADL pour déclencher les événements. La transformée de tels sous-programmes doit prendre en compte le fait qu'un même sous-programme (ayant des ports en sortie) est invoqué par plusieurs processus légers sur le même nœud. Autrement dit, cette transformée ne doit pas être compromise par la concurrence. Nous avons opté pour l'utilisation d'un paramètre "opaque" additionnel qui représente l'état des ports d'un sous-programme à chaque appel de manière indépendante et garantir ainsi la sécurité vis-à-vis de la concurrence. Nous reviendrons plus en détail sur cette catégorie de sous-programmes lors de la transformation des processus légers. En effet, le code généré pour les deux cas est très similaire.

Algorithme 5.3 Déduction de la catégorie d'un sous-programme AADL

```
GET_SUBPROGRAM_KIND (S : Subprogram_Instance)
 1  Language ← Get_Property_Value(S, "Source_Language")
 2  Source_Name ← Get_Property_Value(S, "Source_Name")
 3  Source_Files ← Get_Property_Value(S, "Source_Text")
 4  Subprogram_Kind ← Unknown
 5  switch
 6    case Language_Ada :
 7         if Source_Name ≠ "" or LENGTH (Source_Files) > 0 then
 8           if HAS_NO_CALL_SEQ (S) then
 9             return Ada_95_Subprogram
10    case Language_C :
11         if Source_Name ≠ "" or LENGTH (Source_Files) > 0 then
12           if HAS_NO_CALL_SEQ (S) then
13             return C_Subprogram
14    case < Other_Supported_Language >:
15         ...
16    case No_Language :
17         if Source_Name ≠ "" and LENGTH (Source_Files) = 0 then
18           if HAS_NO_CALL_SEQ (S) then
19             return Empty_Subprogram
20           else
21             if LENGTH (CALLS (S)) = 1 then
22               return Pure_Call_Sequence_Subprogram
```

L'algorithme 5.3 indique comment la catégorie d'un sous-programme est déterminé à partir de ses caractéristiques. Ensuite selon cette catégorie trouvée, le générateur de code produira le code correspondant. Dans la suite, nous expliquerons pour chaque catégorie de sous-programme le comportement du générateur.

Transformation des sous-programmes opaques

Il s'agit de la manière la plus simple pour implanter un sous-programme en AADL. En effet, toute la partie comportementale est à la charge de l'utilisateur et le modèle AADL contient uniquement la définition de l'interface du sous-programme. Pour ces sous-programmes, les trois propriétés `Source_Language`, `Source_Name` et `Source_Text` sont définies et la clause CALLS est absente. Le modèle AADL indique le langage de programmation, le nom et le fichier source de son implantation. Ces composants donnent lieu à la génération d'un sous-programme squelette qui effectue un appel à l'implantation indiquée. Dans le cas de compilateurs comme le compilateur Ada GNAT où le nom d'un fichier est déduit du nom de l'unité de compilation qu'il contient, la fourniture de `Source_Text` est optionnelle.

Exemple 5.5 – Sous-programme opaque

```
subprogram Do_Ping_Spg
features
  Data_Source : out parameter Opaque_Type;

properties
  source_language => Ada95;
  source_name     => "Ping.Do_Ping_Spg";
end Do_Ping_Spg;
```

Exemple 5.6 – Code Ada généré

```
with Ping;

procedure Do_Ping_Spg_Impl
  (Data_Source : out Opaque_Type)
  renames Ping.Do_Ping_Spg;
```

L'exemple 5.5 montre le modèle AADL d'un sous-programme opaque implanté en Ada. Ce sous-programme sera transformé vers une simple enveloppe de code qui se contente d'appeler l'implantation **Ping.Do_Ping_Spg** fournie par l'utilisateur comme l'illustre l'exemple de code 5.6.

Transformation des sous-programmes à séquence pure d'appel

Dans ce cas, le sous-programme contient une et une seule séquence d'appel dans sa clause CALLS. Aucune des trois propriétés indiquées plus haut ne doit être définie pour ce type de sous-programme. Le comportement du sous-programme est donc complètement défini par son modèle AADL : une suite d'appels à d'autres sous-programmes. Ce type de composant donne lieu à la génération d'un sous-programme qui effectue la séquence d'appels vers les transformées respectives des composants appelés. Bien entendu, il est possible de connecter les sous-programmes appelés. Le code généré doit fournir toutes les variables intermédiaires nécessaires pour garantir le bon flux de données spécifié par ces connexions.

Exemple 5.7 – Sous-programme à séquence d'appel

```
subprogram Internal_Sender
features
  Output : out parameter Message;
properties -- ...
end Internal_Sender;

subprogram Internal_Transmitter
features
  Input  : in  parameter Message;
  Output : out parameter Message;
properties -- ...
end Internal_transmitter;

subprogram Sender
features
  Output : out parameter Message;
end Sender;
subprogram implementation Sender.impl
calls { Send : subprogram Internal_Sender;
  Transmit : subprogram Internal_transmitter;};
connections
  C_1 : parameter Send.Output -> Transmit.Input;
  C_2 : parameter Transmit.Output -> Output;
end Sender.impl;
```

Exemple 5.8 – Code Ada généré

```
procedure Internal_Sender
  (Output : out Message)
is
  --  ...
end Internal_Sender;

procedure Internal_Transmitter
  (Input  :     Message;
   Output : out Message)
is
  --  ...
end Internal_Sender;

procedure Sender_Impl
  (Output : out Message)
is
   C_1_Var : Message;
   -- Corresponds to the
   -- C_1 connection.
begin
   Internal_Sender (C_1_Var);
   Internal_Transmitter (C_1_Var,
                         Output);
end Sender_Impl;
```

Le modèle 5.7 présente un sous-programme AADL **Sender.Impl** dont le comportement consiste à appeler le sous-programme **Internal_Sender** puis le sous-programme **Internal_Transmitter**. On ne s'intéresse pas à la manière dont ces deux derniers sont implantés. L'exemple de code 5.8 montre un exemple de code Ada généré à partir de ce modèle. Le sous-programme **Sender.Impl** donne lieu à un sous-programme appelé **Sender_Impl** pour respecter les règles de nommage Ada qui interdisent le caractère '.' dans les identificateurs. Celui-ci appelle les deux sous-programmes produits à partir de **Internal_Sender** et **Internal_Transmitter**. Le générateur de code traite automatiquement la génération de variables intermédiaires correspondant aux connexions liant les sous-programmes appelés. Dans notre ce cas, une variable intermédiaire est générée : **C_1_Var**. Elle correspond à la connexion **C_1** qui lie le paramètre en sortie **Output** de **Internal_Sender** au paramètre en entrée **Input** de **Internal_Transmitter**.

Transformation des sous-programmes vides

Les sous-programmes vides sont des sous-programmes pour lesquels l'utilisateur n'a spécifié aucune caractéristique comportementale. Ils donnent lieu à la production d'un sous-programme qui lève une erreur fatale à l'exécution. Ce genre de sous-programme paraît inutile au premier abord. Cependant, nous les utilisons pour tester les modèles lors des premières phases du développement. Nous voulons effectuer des analyses et voir si notre modèle conduit à une génération de code correcte. Après, nous *remplissons* ces sous-programmes en spécifiant leur comportements respectifs.

5.3.3 Transformation des fils d'exécution

Les composants **thread** sont transformés en des constructions du langage qui permettent de construire un fil d'exécution. Peu de langages de programmation offrent des constructions intrinsèques pour créer de tels fils d'exécution (comme le langage Ada par exemple par l'intermédiaire des tâches). D'autres langages ont recours à des appels à de bibliothèques systèmes pour créer ces fils d'exécution (comme la norme POSIX [IEEE, 1994] utilisée par de nombreux langages).

Comme nous l'avons précisé dans la section 4.9.4, seuls trois types de processus légers sont acceptés pour garantir une analyse d'ordonnançabilité statique de l'application (périodique, sporadique et hybride). Pour chacun de ces trois types, un patron de conception est construit dans le langage impératif choisi. Il décrit le comportement du processus léger d'après sa catégorie mais aussi d'après la sémantique que le standard AADL associe aux composants **thread** [SAE, 2004, Section 5.3]. Ce patron doit être instancié pour chaque tâche dans l'application en fonction des caractéristiques et des propriétés du composant **thread** qui lui correspondent. Les paramètres nécessaires à l'instanciation de ce patron sont :

1. La liste des éléments d'interface du composant. Cette liste est requise pour les tâches sporadique et hybride uniquement,
2. Un identificateur pour la tâche,
3. La période de la tâche (pour les tâches périodiques et hybrides) ou le temps minimal entre deux déclenchements (pour les tâches sporadiques),
4. L'échéance de la tâche,
5. La priorité de la tâche,
6. La taille de la pile de la tâche,
7. Pour les tâches sporadiques et hybrides, un sous-programme qui effectue l'attente d'un événement extérieur. Un appel à ce sous-programme bloque la tâche jusqu'à l'arrivée d'un événement et retourne l'élément d'interface qui l'a reçu,
8. Le sous-programme qui effectue le travail cyclique de la tâche,
9. Un éventuel sous-programme qui initialise le processus léger.

Certains paramètres sont déduits directement du modèle AADL par simple lecture de propriétés (période, échéance, priorité [4], taille de la pile et sous-programme servant à l'initialisation). Les autres paramètres doivent être générés automatiquement en analysant le composant thread. Dans la suite, nous donnons pour chacun des constituants d'un composant **thread**, les règles de sa transformation vers un langage de programmation impératif.

4. Si on utilise RMS pour ordonnancer un ensemble de tâches, les priorités des tâches doivent être déduites de leurs périodes respectives. Dans un premier temps, nous supposons que l'utilisateur donne des périodes cohérentes avec RMS dans son modèle AADL et nous effectuons un ordonnancement de type *FIFO within priorities*.

Transformation des éléments d'interface

Algorithme 5.4 Construction de l'énumération représentant les ports d'un thread AADL

```
PRODUCE_PORT_ENUMERATION (T : Thread_Instance)
1  Protocol ← Get_Property_Value(T, "Dispatch_Protocol")
2  Enumeration_List ← Empty_List
3  for each P in PORTS (T) do
4    Enumerator ← MAP_ENUMERATOR (NAME (P))
5    APPEND (Enumerator, Enumeration_List)
6  DEFINE_ENUMERATION (MAP_ENUMERATION_IDENTIFIER (NAME (T)),
7                      Enumeration_List)
```

L'ensemble des éléments d'interface d'un composant **thread** est transformé en une énumération. Nous avons choisi cette transformation pour être capables d'une part de créer des tableaux indexés par les différents ports d'un composants **thread** et d'autre part de pouvoir utiliser les énumérateurs dans des structures conditionnelles à choix multiples (*switch case*). Ceci permet un temps constant d'exécution indépendant du nombre de ports dans un composant **thread**.

L'algorithme 5.4 montre comment est construite l'énumération représentant les ports d'un processus léger donné. Bien entendu, nous supposons que durant cette phase la catégorie du processus léger a déjà été vérifiée et que cet algorithme ne sera pas appliqué sur un processus léger périodique.

Exemple 5.9 – Déclaration d'un thread AADL

```
thread A_Thread
features
  Input_1   : in event data port Integer;
  Input_2   : in data port Message;
  Input_3   : in event port;
  Output_1  : out event data port Integer;
  Output_2  : out data port Message;
  Output_3  : out event port;
end A_Thread;
```

Exemple 5.10 – Code Ada pour les ports du thread

```
type A_Thread_Port_Type is
  (Input_1,
   Input_2,
   Input_3,
   Output_1,
   Output_2,
   Output_3);
```

L'exemple de code 5.9 montre un modèle AADL d'une tâche sporadique ayant un ensemble de ports en entrée et en sortie. Cet ensemble de ports est traduit, en Ada par exemple vers une définition d'un type énuméré comme le montre l'exemple de code.

Notons que cette même règle de transformation est appliquée aux sous-programmes qui possèdent des ports dans leur interface. Elle donne lieu à une énumération qui sera utilisée par les routines d'envoi d'événements.

Transformation de la partie comportementale

La construction du sous-programme qui effectue le travail cyclique d'une tâche dépend fortement de la manière dont le comportement de cette tâche a été spécifié par l'utilisateur. En effet, nous autorisons trois manières différentes pour spécifier le comportement d'un composant **thread**.

Threads avec une séquence d'appel Le comportement de ces tâches est assez simple et très similaire à celui des sous-programmes à séquence pure d'appel. Elles exécutent à chaque occurrence la liste des sous-programmes appelés dans leur séquence d'appel. Cette manière

d'implanter un composant **thread** est très adaptée aux tâches périodiques car tout le comportement est géré automatiquement par le générateur de code et l'utilisateur n'a pas à manipuler les ports du processus léger pour envoyer ou recevoir des informations.

Le sous-programme généré est similaire à celui d'un sous-programme AADL qui appelle lui même d'autres sous-programmes (donné dans les exemples 5.7 et 5.8). Le générateur de code produit les variables intermédiaires nécessaires pour la connexion entre les sous-programmes appelés. Il produit la collecte des données reçues dans les ports en entrée du composant **thread** et les appels des sous-programmes dans le bon ordre. Enfin, le générateur produit l'envoi des éventuelles données ou événements sur les ports en sortie du composant.

Threads avec un point d'entrée par port Dans ce cas chacun des ports événements (ou événement donnée) en entrée du composant **thread** se trouve associé à un sous-programme par l'intermédiaire de la propriété **Compute_Entrypoint** appliquée à chacun des ports. Le composant **thread** décrit dans l'exemple 5.9 fait partie de cette catégorie. Ainsi, à chaque déclenchement du processus léger, le point d'entrée correspondant au port qui a reçu l'ordre de déclenchement est exécuté. Cette manière d'implanter est adaptée aux processus légers sporadiques et hybrides. Pour les processus légers hybrides, l'utilisateur doit définir, en plus des points d'entrée des ports, un point d'entrée pour le processus léger lui même pour effectuer le comportement périodique. Cette approche ne doit pas être utilisée pour les processus légers périodiques. En effet, ils ne doivent pas avoir de ports de type événement en entrée.

Le générateur produit un sous-programme qui possède un paramètre dénotant le port qui a déclenché le thread. Ensuite dans le sous-programme produit, un test est effectué sur la valeur de ce port et le point d'appel correspondant est appelé.

Exemple 5.11 – Implantation d'un thread AADL

```
thread A_Thread
features
  Input_1   : in event data port Integer
    {Compute_Entrypoint => "Pkg.On_Input_1";};
  Input_2   : in data port Message;
  Input_3   : in event port
    {Compute_Entrypoint => "Pkg.On_Input_3";};
  Output_1  : out event data port Integer;
  Output_2  : out data port Message;
  Output_3  : out event port;
properties
  Dispatch_Protocol => Sporadic;
  -- ...
end A_Thread;
```

Exemple 5.12 – Le "travail" du thread en Ada

```
with Pkg;

procedure A_Thread_Job
  (Port : A_Thread_Port_Type) is
begin
   case Port is
      when Input_1 => Pkg.On_Input_1
         (Get_Value (Input_1));
      when Input_3 => Pkg.On_Input_2;
      when others  => raise Program_Error;
   end case;
   -- Affectation des valeurs pour les
   -- ports en sortie et déclenchement de
   -- l'envoi sur ces ports.
end A_Thread_Job;
```

L'exemple de code 5.11 reprend le composant **A_Thread** décrit dans le modèle 5.9. Il lui ajoute la propriété **Dispatch_Protocol** pour préciser qu'il s'agit d'une tâche sporadique et associe un point d'entrée à chaque port en entrée. Puisque ce composant possède un point d'entrée par port, le comportement du sous-programme généré exécute le point d'entrée correspondant à l'événement reçu. L'utilisation d'énumération pour représenter les ports nous permet de trouver le comportement voulu en temps constant. Comme le montre l'exemple de code 5.12, chacun des points d'entrée pour les ports de type événement-donnée possède un paramètre dans sa signature. Ceci permet au code de l'utilisateur de retrouver la valeur de la donnée reçue par le port. La lecture de la donnée s'effectue à l'aide de la routine **Get_Value** que nous verrons plus loin dans cette section.

Threads avec un seul point d'entrée Dans ce cas, le point d'entrée est spécifié par l'intermédiaire de la propriété standard **Compute_Entrypoint** associé au composant **thread** lui-même. Il s'agit de l'approche la moins automatisée pour implanter une tâche. Il s'agit de permettre à l'utilisateur de gérer lui même tout le comportement lors d'une occurrence (gestion des files d'attentes des ports, dialogue avec les routines de l'intergiciel pour envoyer des informations...).

Cette méthode de conception est utilisée dans des cas rares où l'utilisateur désire effectuer des comportements avancées sur les files d'attentes des éléments d'interface comme l'extraction de plus d'un élément par exemple ou encore la lecture d'une file d'attente différente de celle du port qui vient de provoquer le déclenchement.

Transformation des modes opérationnels

Comme nous l'avons précisé dans la section 4.9.4, les modes opérationnels d'un composant **thread** ne doivent contrôler que le comportement du thread (la séquence d'appel ou bien la valeur de la propriété **Compute_Entrypoint** associée au composant). Ainsi, il ne compromettent ni compliquent l'analysabilité statique de l'application. Dans ce cas, d'une manière similaire à celle utilisée pour transformer les éléments d'interface, les modes sont transformés en une énumération. La machine à états qui pilote le changement de modes est transformé en une machine à état dans le langage de programmation impératif. Elle est placée au début du sous-programme généré pour effectuer le comportement cyclique. Ainsi au début de son exécution, la valeur du mode est calculée en fonction de événements reçus et de la valeur courante. Ensuite le sous-programme appelle effectue le traitement en fonction du mode trouvé.

Exemple 5.13 – Modes dans thread AADL

```
thread A_Thread
features
  Go_Lazy    : in event port;
  Go_Normal  : in event port;
  Go_Crazy   : in event port;
properties Dispatch_Protocol => Sporadic;
end A_Thread;
thread implementation A_Thread.Impl
modes
  Normal : initial mode;
  Crazy  : mode;
  Lazy   : mode;
  Normal, Lazy  -[Go_Crazy ]-> Crazy;
  Normal, Crazy -[Go_Lazy  ]-> Lazy;
  Crazy, Lazy   -[Go_Normal]-> Normal;
properties
  Compute_Entrypoint => "Pkg.Normal_Handler"
    in modes (Normal);
  Compute_Entrypoint => "Pkg.Crazy_Handler"
    in modes (Crazy);
  Compute_Entrypoint => "Pkg.Lazy_Handler"
    in modes (Lazy);
end A_Thread.Impl;
```

Exemple 5.14 – Le "travail" du thread en Ada

```
type Mode_Type is (Normal, Crazy, Lazy);
Mode : Mode_Type := Normal;
procedure A_Thread_Job
  (Port : A_Thread_Port_Type) is
begin
  case Port is
    when Work_Normally =>
      case Mode is
        when Crazy | Lazy => Mode := Normal;
        when others       => null;
      end case;

    when -- ...

    when others => null;
  end case;
  -- Appel du point d'entrée correspodant
  case Mode is
    when Normal => Pkg.Normal_Handler (Port);
    when Crazy  => Pkg.Crazy_Handler (Port);
    when Lazy   => Pkg.Lazy_Handler (Port);
  end case;
end A_Thread_Job;
```

L'exemple 5.13 montre un modèle AADL d'un composant **thread** sporadique possédant trois modes opérationnels. Le composant possède un seul point d'entrée. Le changement de mode est décrit par la machine à état donnée dans la section MODES du composant **A_Thread.Impl**. Selon la valeur du mode courant, le processus léger exécute un des sous-programmes spécifiés dans la section PROPERTIES du même composant. Le code Ada produit pour gérer le changement de modes est présenté dans l'exemple de code 5.14. Nous y remarquons notamment, l'énumération représentant les modes (**Mode_Type**), la variable globale représentant le mode courant et

initialisée à **Normal** comme précisé dans le modèle AADL. De plus, le code généré implante la machine à état qui pilote le changement de mode et effectue le comportement voulu. Là aussi, l'utilisation d'une énumération pour représenter les modes opérationnels permet d'effectuer les calculs et les transitions en un temps constant.

Instanciation du processus léger

Comme le montre la règle 5.5, un composant AADL **thread** est transformé en un couple formé de :

1. l'instance du patron de conception pour créer un fil d'exécution,
2. un ensemble de méthodes qui permet l'accès en lecture et/ou en écriture à l'interface du composant afin d'envoyer et recevoir de l'information. Il s'agit de l'implantation du service intergiciel d'interaction (voir 3.3.3) qui simplifie la communication à travers les ports du composant.

Exemple 5.15 – Instance d'un thread sporadique en Ada

```
package A_Thread_Task is new Sporadic_Task
  (Port_Type                => Receiver_Port_Type,
   Entity                   => SC_2_Receiver_ID,
   Task_Period              => Ada.Real_Time.Milliseconds (20),
   Task_Deadline            => Ada.Real_Time.Milliseconds (20),
   Task_Priority            => System.Default_Priority,
   Task_Stack_Size          => 64_000,
   Job                      => A_Thread_Job,
   Wait_For_Incoming_Events => Wait_For_Incoming_Events);
```

L'exemple de code 5.15 montre l'instance générée pour **A_Thread.Impl** en Ada. Le générateur de code affecte des valeurs par défaut aux paramètres optionnels qui ne sont pas fournis par l'utilisateur. Ainsi, l'échéance du processus léger prend une valeur égale à celle de la période et sa priorité est fixée à la priorité par défaut du système.

Le sous-programme **Wait_For_Incoming_Events** qui est le dernier paramètre de l'instanciation de **A_Thread.Impl** fait partie des méthodes du services d'interaction produit automatiquement. Ce service constitue une couche au-dessus du service d'interaction de l'intergiciel minimal. Il fournit les routines suivantes qui permettent d'accéder en lecture et en écriture aux éléments d'interface d'un composant **thread**. Les noms de ces routines ainsi que leurs comportement sont spécifié par le standard AADL :

- **Send_Output** envoie explicitement les événements et les données à travers le réseau s'il s'agit d'une communication distante. Si la communication est locale, cette méthode effectue la livraison directe vers la destination. Elle prend en paramètre une référence vers le port à traiter.
- **Put_Value** marque un port comme contenant une information (donnée ou événement) prête à être envoyée. L'envoi de la donnée sera effectué par un appel à **Send_Output**. Elle prend comme paramètres une référence vers le port en question ainsi que la valeur de la donnée à envoyer s'il s'agit d'un port donnée ou événement-donnée.
- **Get_Value** renvoie la valeur reçue à la tête de la file d'attente d'un port événement ou événement-donnée ou tout simplement la valeur d'un port de type donnée. Elle ne provoque pas de consommation : plusieurs appels successifs à cette méthode retournent le même résultat.

Algorithme 5.5 Transformation d'un thread AADL

```
TRANSFORM_THREAD (T : Thread_Instance)
 1  Protocol ← GET_PROPERTY_VALUE (T, "Dispatch_Protocol")
 2  Period ← GET_PROPERTY_VALUE (T, "Period")
 3  Deadline ← GET_PROPERTY_VALUE (T, "Deadline")
 4  if Deadline = NIL then
 5    Deadline ← Period
 6  Priority ← GET_PROPERTY_VALUE (T, "Priority")
 7  if Priority = NIL then
 8    Priority ← Default_System_Priority
 9  Stack_Size ← GET_PROPERTY_VALUE (T, "Storage_Size")
10  Initialize_Entrypoint ← GET_PROPERTY_VALUE (T, "Initialize_Entrypoint")
11  if Initialize_Entrypoint = NIL then
12    Initialize_Entrypoint ← Null_Subprogram
13  CREATE_THREAD_JOB (T)
14  switch
15    case Protocol = Periodic :
16        Instantiate_Periodic_Thread(MAP_IDENTIFIER (NAME (T)),
17                                    Period,
18                                    Deadline,
19                                    Priority,
20                                    Stack_Size,
21                                    GET_THREAD_JOB (T),
22                                    Initialize_Entrypoint)
23    case Protocol = Periodic :
24        PRODUCE_PORT_ENUMERATION (T)
25        Instantiate_Sporadic_Thread(GET_PORT_ENUMERATION (T),
26                                    MAP_IDENTIFIER (NAME (T)),
27                                    Period,
28                                    Deadline,
29                                    Priority,
30                                    Stack_Size,
31                                    GET_THREAD_JOB (T),
32                                    Initialize_Entrypoint)
33    case Protocol = Hybrid :
34        PRODUCE_PORT_ENUMERATION (T)
35        Instantiate_hybrid_Thread(GET_PORT_ENUMERATION (T),
36                                  MAP_IDENTIFIER (NAME (T)),
37                                  Period,
38                                  Deadline,
39                                  Priority,
40                                  Stack_Size,
41                                  GET_THREAD_JOB (T),
42                                  Initialize_Entrypoint)
43    case default :
44        error "Cannot transform this thread"
45  INSTANTIATE_INTERROGATORS (T, FEATURES (T))
```

- **Get_Count** retourne le nombre de messages non consommés dans la file d'attente d'un port événement ou événement-donnée
- **Next_Value** consomme le message à la tête de la file d'attente d'un port événement ou événement-donnée.
- **Wait_For_Incoming_Events** concerne les processus légers sporadiques. Elle attend jusqu'à l'arrivée d'événements, et retourne le port sur lequel l'événement est arrivé.

Le comportement étant générique et faiblement personnalisable en fonction des propriétés d'une application, elles sont rassemblées sous forme d'un archétype qui fait partie de l'intergiciel minimal (c.f. la section 3.3.3). Il s'agit du service d'interrogation de l'intergiciel minimal. Pour chaque processus léger, une instance de ces routines est créée. Nous créons une instance pour chaque processus léger afin de pouvoir la paramétrer statiquement. L'instance est paramétrée par les entités suivantes, générées pour chaque processus léger ayant des ports :

- Un type énuméré qui liste les ports du composant **thread**,
- Un type structure de donnée variable (comme les *unions* en C) qui est piloté par le type énuméré cité plus haut. Pour chaque port donnée ou événement donnée, cette structure contiendra un champ supplémentaire ayant le type de la donnée. Pour les ports de type événement pur, cette structure est vide. Nous appellerons ce type l'*interface du thread* et nous y accéderons dans les prochains algorithmes à l'aide de la routine **Get_Interface_Type**.
- Un identificateur pour le processus léger,
- Une table indiquant pour chaque port, sa catégorie et son sens (in data port, ou data port, in event port...),
- Une table indiquant pour chaque port événement en entrée, la taille de sa file d'attente (spécifiée à l'aide de la propriété AADL "Queue_Size"),
- Une table indiquant pour chaque port en sortie, la liste de ses destinations,
- Une routine *Marshall* qui emballe une information de type *interface du thread* dans un tampon de communication.
- Une routine qui retourne la date de la prochaine échéance du processus léger.

L'exemple de code 5.16 montre comment ces interrogateurs sont instanciés dans le cas de Ada par exemple. Des entités supplémentaires doivent être fournies au paquetage générique pour compléter l'instanciation comme les différents types. Le nouveau paquetage instancié **A_Thread_Interrogators** fournit les sous-programmes cités plus haut.

Pour les sous-programmes AADL possédant des ports, un ensemble d'interrogateurs est créé de manière similaire.

5.3.4 Transformation des instances de données partagées

Les instances des données protégées sont les sous-composants de types **data** qui sont déclarés dans les implantations des composants **process**. Il s'agit de données partagées entre les processus légers du processus.

Comme le montre la transformation 5.6, un sous-composant **data** est caractérisé par son nom et le type du composant qu'il instancie. Il est transformé en une déclaration de variable partagée (protégée ou non, selon la spécification de l'utilisateur). Ainsi les processus légers ou les sous-programmes qui partagent l'accès à cette variable y accèdent en utilisant l'ensemble des méthodes prévues.

Exemple 5.16 – Instance des “interrogateurs” pour **A_Thread_Impl**

```
type A_Thread_Interface (Port : A_Thread_Port_Type) is record
   case Port is
      when Input_1  => Input_1_DATA : Integer;
      when Input_2  => Input_2_DATA : Message;
      when Input_3  => null;
      when Output_1 => Output_1_DATA : Integer;
      when Output_2 => Output_2_DATA : Message;
      when Output_3 => null;
   end case;
end record;

type A_Thread_Integer_Array   is array (A_Thread_Port_Type) of Integer;
type A_Thread_Port_Kind_Array is array (A_Thread_Port_Type) of Port_Kind;
-- Des types pour définir les tableaux ci-dessous

A_Thread_Port_Kinds : constant A_Thread_Port_Kind_Array :=
   (Input_1  => In_Event_Data_Port,
    Input_2  => In_Data_Port,
    Input_3  => In_Event_Port,
    Output_1 => Out_Event_Data_Port,
    Output_2 => Out_Data_Port,
    Output_3 => Out_Event_Port);

A_Thread_FIFO_Sizes : constant A_Thread_Integer_Array :=
  (Input_1  => 16, -- Valeur par défaut pour un port d'évènement
   Input_2  => 1,  -- Port donnée
   Input_3  => 16, -- Valeur par défaut pour un port d'évènement
   Output_1 => -1, -- Valeur conventionnelle pour les ports en sortie
   Output_2 => -1, -- Valeur conventionnelle pour les ports en sortie
   Output_3 => -1); -- Valeur conventionnelle pour les ports en sortie

type A_Thread_Address_Array is array (A_Thread_Port_Type) of System.Address;

A_Thread_N_Destinations : constant A_Thread_Integer_Array :=
  (Input_1  => -1, -- Valeur conventionnelle pour les ports en entrée
   Input_2  => -1, -- Valeur conventionnelle pour les ports en entrée
   Input_3  => -1, -- Valeur conventionnelle pour les ports en entrée
   Output_1 => <Nombre De Destinations>, -- Déduite de la topologie
   Output_2 => <Nombre De Destinations>, -- Déduite de la topologie
   Output_3 => <Nombre De Destinations>); -- Déduite de la topologie

A_Thread_Destinations : constant A_Thread_Address_Array :=
   (Input_1  => System.Null_Address,
    Input_2  => System.Null_Address,
    Input_3  => System.Null_Address,
    Output_1 => <Tableau Des Ports Destination De Output_1>'Address,
    Output_2 => <Tableau Des Ports Destination De Output_2>'Address,
    Output_3 => <Tableau Des Ports Destination De Output_3>'Address);

-- Instance du paquetage générique

package A_Thread_Interrogators is new Thread_Interrogators
  (Port_Type             => A_Thread_Port_Type,
   Integer_Array         => A_Thread_Integer_Array,
   Port_Kind_Array       => A_Thread_Port_Kind_Array,
   Address_Array         => A_Thread_Address_Array,
   Thread_Interface_Type => A_Thread_Interface,
   Current_Entity        => A_Thread_Impl_ID,
   Thread_Port_Kinds     => A_Thread_Port_Kinds,
   Thread_Fifo_Sizes     => A_Thread_FIFO_Sizes,
   N_Destinations        => A_Thread_N_Destinations,
   Destinations          => A_Thread_Destinations,
   Marshall              => Marshall,
   Next_Deadline         => A_Thread_Task.Next_Deadline);
```

Algorithme 5.6 Déclaration d'une donnée protégée AADL

```
DECLARE_SHARED_DATA (S : Data_Subcomponent_Instance)
1  D ← GET_CORRESPONDING_INSTANCE (S)
2  Category ← GET_PROPERTY_VALUE (D, "Data_Model :: Data_Representation")
3  if Category ≠ Data_With_Accessors then
4     error "Only data with accessors can be shared."
5  Declate_Variable(MAP_IDENTIFIER (NAME (S)), GET_MAPPED_TYPE (D)
```

5.4 Génération de Composants Intergiciels

Dans cette partie nous montrons plus en détail comment les composants intergiciels fortement personnalisables sont construits à partir du modèle de l'application.

Lors de la phase de transformation des entités AADL, une partie des services intergiciels se trouvent produits implicitement : il s'agit des services d'*activation*, d'*exécution*, de *typage* et d'*interaction*. Les services restants demandent du travail supplémentaire pour être produits : il s'agit des services d'*adressage*, de *liaison*, de *représentation avancée* et de *la couche haute de transport*.

5.4.1 Activation et Exécution

Les services d'*activation* et d'*exécution* sont produits statiquement puisque toutes les tâches qui forment un nœud particulier sont créées au début de l'application et reste en fonction pendant toute la durée d'exécution de l'application. De plus le sous-programme représentant le comportement de la tâche est produit automatiquement. Ce sous-programme est appelé automatiquement chaque fois que la tâche est déclenchée. Dans le cas de tâches sporadiques ou hybrides la couche de transport invoque la routine `Deliver` qui affecte la valeur correcte au paramètre de ce sous-programme indiquant le port qui a provoqué le déclenchement. Le fait que ce sous-programme agisse en fonction de l'événement reçu par la tâche assure le comportement voulu.

Ces deux services bénéficient fortement de l'utilisation des types énumérés pour implanter les différents constituants d'une tâche (élément d'interface, modes...). Ceci permet d'effectuer automatiquement le comportement voulu en un temps constant.

5.4.2 Typage

Le service de *typage* est assuré par la génération automatique des types de données nécessaires à pour chaque nœud. Comme nous l'avons expliqué dans la section 5.3, tous les éléments d'interface de types donnée ou événement donnée ainsi que toutes les instances de données partagées donnent lieu à une définition d'un type de donnée.

5.4.3 Interaction

Le service d'*interaction* est assuré par la production de l'ensemble de méthodes qui permettent l'accès à l'interface du composant **thread**. Ces méthodes d'interaction sont invoquées par le code généré ou par le code de l'utilisateur et garantissent la bonne transmission des informations (destination multiples, connexions retardées...). La manière dont ces méthodes sont produites a été vue en détail dans la section 5.3.3.

5.4.4 Adressage et liaison

Il s'agit de produire un service qui, pour chaque nœud, retrouve en temps constant toutes les informations pour "atteindre" les nœuds auxquels il est connecté.

Une table de nommage/adressage est générée pour permettre à chaque nœud de l'application de communiquer avec les autres nœuds de l'application. Cette table donne, pour chaque **port** en entrée du nœud courant les informations nécessaires à l'initialisation d'un point de réception de messages. Pour chacun des ports distants auxquels le nœud courant est connecté, cette table donne les informations de transport nécessaires pour les atteindre. Cette table est aussi utilisée pour initialiser la couche basse de transport. Ceci constitue l'implantation des services d'adressage et de liaison.

Algorithme 5.7 Construction d'une table de nommage pour un processus AADL

```
DECLARE_NAMING_TABLE (Node : Process_Instance)
 1  for each P in PORTS (Node) do
 2    if IS_IN (P) then
 3      Bus ← GET_BOUND_BUS (P)
 4      MAYBE_INSTANTIATE_NEW_NAMING_TABLE (Bus)
 5      ADD_NEW_ELEMENT_ASSOCIATION (GET_CORRESPONDING_TABLE (Bus),
 6                                   MAP_ELEMENT_NAME (NAME (P)),
 7                                   GET_PROPERTY_VALUE (Bus, "Transport_Information"))
 8    else
 9      for each D in DESTINATIONS (P) do
10        Bus ← GET_BOUND_BUS (D)
11        INSTANTIATE_NEW_NAMING_TABLE (Bus)
12        ADD_NEW_ELEMENT_ASSOCIATION (GET_CORRESPONDING_TABLE (Bus),
13                                     MAP_ELEMENT_NAME (NAME (D)),
14                                     GET_PROPERTY_VALUE (Bus, "Transport_Information"))
```

L'algorithme 5.7 indique la manière dont nous construisons les tables de nommage. Nous déclarons une table de nommage par instance de service de transport utilisé par le nœud. Ainsi, si un nœud utilise à la fois Ethernet et SPACEWIRE, deux tables de nommage seront construites. L'algorithme est une boucle sur tous les ports d'un processus AADL donné. Pour les ports en entrée, nous sommes sûrs qu'il possède une seule source. Ainsi nous récupérons le bus de la connexion qui le lie à sa source à l'aide de la routine **Get_Bound_Bus**. Ensuite, nous instancions une table de nommage correspondant à ce bus (*si ce n'est déjà fait*). Enfin nous ajoutons une nouvelle entrée à cette table qui associe le port à l'information de la couche basse de transport utilisée par le bus. Pour les ports en sortie, il est possible en AADL qu'ils soient connectés à plusieurs destinations et que les connexions utilisent des couches de transport différentes. Par conséquent nous effectuons le traitement sur chacune des destinations des ports en sortie. A la fin de cet algorithme, nous disposons d'autant de tables de nommage/adressages qu'il y en a de couche de transports.

Toutes ces tables sont indexées par des énumérations qui représentent les ports. Ceci rend constant le temps de recherche de l'information de transport correspondant à un port particulier.

5.4.5 Couche haute de transport

Pour chaque nœud de l'application répartie, la couche haute de transport est générée automatiquement. Elle offre trois routines :

SEND permet d'envoyer un message de la part d'un port source vers un port destination. Cette routine est invoquée par la couche protocole invoquée par la méthode **Send_Output** des interrogateurs. Elle prend en paramètre les ports source et destination et le message construit par la couche transport. Elle modifie le message en lui ajoutant des informations spécifiques à la couche transport (emballage de la source, et de la destination...). Ensuite, selon la nature de la connexion entre la source et la destination, elle effectue soit une livraison directe du message vers le destinataire s'il s'agit d'une communication locale, soit un appel à la routine SEND de la couche basse de transport qui régit la connexion entre les deux ports source et destination.

DELIVER cause la livraison d'un message vers la destination. Cette routine est invoquée soit par la haute couche de transport s'il s'agit d'une livraison locale, soit par l'entité qui gère la réception dans la basse couche de transport. Il prend comme paramètres une référence vers le processus léger récepteur ainsi que le message à délivrer. L'effet de cette routine est la modification des structures internes du service *interaction* implanté dans les interrogateurs en déposant les nouvelles données ou événements reçus par un processus léger particulier. Elle cause, d'une façon indirecte, le déclenchement des processus légers sporadiques et hybrides qui reçoivent de nouveaux événements.

INIT effectue l'initialisation de toutes les couches basses de transports supportées dans le nœud. L'analyse du modèle AADL de l'application nous permet de déterminer les couches basses de transports nécessaires pour chaque nœud de l'application et les autres nœuds avec lesquels il communique. Pour des couches de transport comme SPACEWIRE [ESTEC, 2003] par exemple, il est nécessaire d'ajouter une tâche qui effectue la réception des messages venant des autres nœuds. Cette tâche est créée au moment de l'initialisation de l'application.

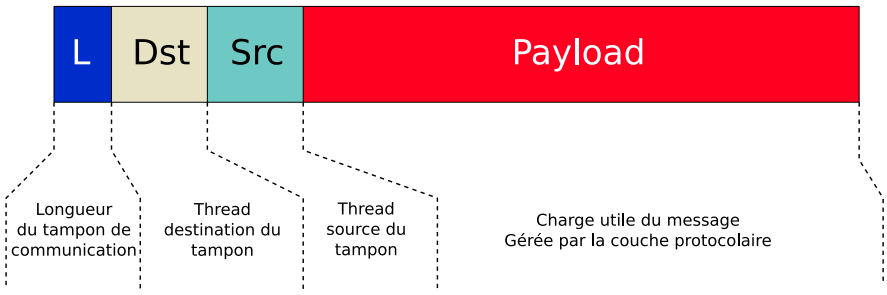

FIGURE 5.4 – Structure d'un tampon de communication

La figure 5.4 montre la structure d'un tampon de communication. À l'entête du tampon, on trouve sa longueur. Ceci sert à la couche de transport destination pour traiter le tampon uniquement. En effet, puisque nous travaillons sur des systèmes critiques, nous nous interdisons l'allocation dynamique des tampons de communication à la réception de message. Nous disposons d'un tampon global pré-alloué qui possède une longueur maximale. Les messages reçus sont stockés dans ce tampon, et leur taille est déterminée en lisant la première partie du message. Après la longueur du message nous trouvons les identificateurs des processus légers récepteur émetteur du message. La donnée du processus léger récepteur sert à la couche haute de transport à savoir vers quelle routine de livraison interne le message sera délivré. La donnée du processus léger émetteur est une information mise à la disposition du destinataire pour savoir d'où vient son message. Enfin, le reste du tampon représente la charge utile délivrée à la

couche protocolaire de la destination puis à la couche applicative. Nous verrons plus en détail la structure de cette portion du message quand nous présenterons le service de *représentation* plus loin dans cette section.

5.4.6 Représentation avancée

Ce service est étroitement lié au service de typage. Il permet de garantir la cohérence des données lors de leur envoi à travers le réseau. Une analyse des interfaces des composants **thread** dans l'application répartie nous permet de déterminer les types de données qui seront transmis (localement ou à distance). Pour chacun de ces types, deux sous-programmes sont générés automatiquement :

Marshall : cette méthode encode une information d'un type donné dans un tampon de communication.

Unmarshall : cette méthode décode une information d'un type donné depuis un tampon de communication.

La manière d'encoder une donnée dans un tampon peut être simple (copie de la mémoire occupée de la donnée dans le tampon) si les nœuds sont homogènes. Elle peut être complexe pour pallier l'hétérogénéité des architectures (petits boutistes *vs.* gros boutistes...) et des protocoles sophistiqués d'encodage et de décodage des données sont utilisés dans ce cas (CDR...).

Algorithme 5.8 Instanciation des *Marshallers* pour les types de données AADL

```
DATA_TYPE_MARSHALLERS (Node : Process_Instance)
1  for each T in THREADS (Node) do
2    for each P in PORTS (T) do
3      if IS_DATA (P)or IS_EVENT_DATA (P) then
4        INSTANTIATE_MARSHALLER (GET_CORRESPONDING_TYPE (P))
5    CREATE_MARSHALLER (GET_INTERFACE_TYPE (T))
```

Comme le montre l'algorithme 5.8, pour chacun des types de donnée utilisés dans les ports, nous instancions les routines *Marshall* et *Unmarshall* correspondant. Ensuite pour chaque thread, nous générons deux routines *Marshall* et *Unmarshall* qui effectuent l'emballage et le déballage de son type-interface dans un tampon de communication. Ces deux sous-programmes sont plus compliqués que ceux qui sont instanciés automatiquement parce que nous n'insérons pas la structure de données correspondant à l'interface dans un message. En effet, un processus léger qui reçoit un message s'attend à ce que celui ci soit conforme à son interface et non pas à celle de l'expéditeur du message. Pour résoudre ce problème, le sous-programme *Marshall* décompose la structure de donnée et insère d'abord le port correspondant à la **destination** du message. Ensuite il insère l'éventuelle donnée associée au port. Quant à *Unmarshall*, il retire le port qui indique le port destination. Enfin, selon la valeur de ce port il extrait (ou non) la donnée associée afin de construire une nouvelle structure de type *interface de thread* destination.

La figure 5.5 montre la structure d'un message échangé entre deux processus légers. Ce message représente la charge utile du tampon de communication vu dans la figure 5.4. Le premier champ du message représente le port destinataire du message. La valeur de ce port extraite par le service représentation d'entité destination pilote la façon dont les données suivantes sont extraites du message. Si la connexion qui a acheminé le message est une connexion AADL retardée alors le deuxième champ du message représente l'instant auquel le message sera disponible à la destination comme prévu par le standard AADL. Cet instant est calculé relativement

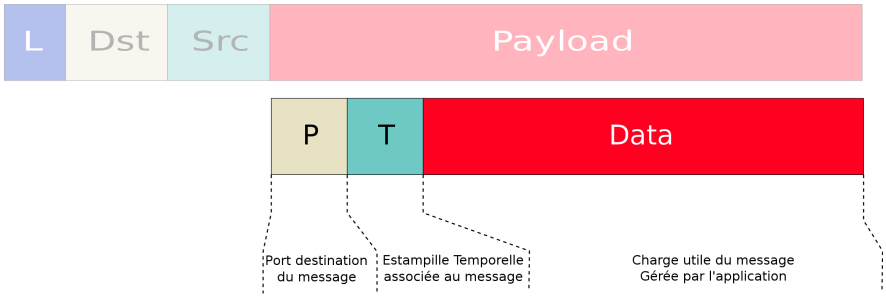

FIGURE 5.5 – Structure d'un message envoyé par un processus léger

à l'instant de la fin de l'initialisation de l'application répartie effectué par les différents nœuds (voir la section 3.3.5). Si le port destination est un port donnée ou événement-donnée, alors le dernier champ du message représente la valeur de la donnée envoyée. Cette valeur sera extraite à l'aide de la routine *Unmarshall* qui a été produit pour le type correspondant.

5.5 Déploiement et configuration des composants intergiciels

Dans cette section, nous expliquons comment sont sélectionnés les composants de l'intergiciel minimal dont l'application a besoin (déploiement). Nous expliquons aussi comment ces composants ainsi que ceux produits automatiquement sont paramétrés en fonction des caractéristiques de l'application (configuration).

5.5.1 Déploiement

La sélection des composants de l'intergiciel minimal se fait lors du parcourt de l'arbre d'instance du modèle AADL de l'application. En particulier, lors du passage par les composants matériels, leur propriétés sont analysées pour savoir exactement de quel composant de l'intergiciel minimal ils ont besoin. Nous utilisons l'ensemble de propriétés de déploiement que nous avons défini dans la section 4.9.

Bus

Les propriétés d'un bus AADL permettent de sélectionner les versions correctes suivantes des services de l'intergiciel minimal :

- La couche basse de transport : en effet parmi les propriétés additionnelles relative au déploiement que nous avons ajoutées. Une propriété `Transport_API` qui permet de préciser quel mécanisme de bas niveau de transport utilise le bus (Ethernet, SPACEWIRE).
- Le protocole : en effet, le bus contient aussi des informations sur le protocole utilisé. La propriété `Protocol` que nous avons introduite permet de préciser le protocole utiliser pour échanger les données à travers les bus (trivial...).
- La représentation élémentaire : généralement, chaque protocole utilise est associé à un service de représentation. Par exemple, le protocole trivial qui consiste à envoyer les données dans le réseau par une simple copie ne nécessite pas un service de représentation évoluée. Il suffit de convertir les données en des tampons d'octet sans les modifier. Par contre pour un protocole comme GIOP, nous utilisons la représentation CDR spécifiée par CORBA.

Processeurs

Les processeurs permettent de préciser la plate-forme sur laquelle le nœud s'exécute. Ceci permet de raffiner la sélection des services. Par exemple si un même service est implanté différemment pour deux plate-forme différentes, la nature du processeur permet d'indiquer quelle version choisir. La plate-forme d'un processeur est spécifiée à l'aide de la propriété **Execution_Platform** que nous avons introduite.

5.5.2 Configuration

La configuration des services de l'intergiciel est effectuée statiquement et très simplement grâce aux différentes énumération et constantes que nous produisons en transformant les entités AADL. Dans cette partie nous donnons ces énumérations et constantes et nous précisons quels services elles permettent de configurer.

Nœuds

Sur chaque nœud de l'application, nous produisons une énumération contenant la liste des nœuds auxquels il est connecté. Nous produisons aussi une constante qui a le même type énuméré et qui permet au nœud de s'identifier lui même.

Cette énumération est utilisée pour configurer la couche basse de transport en ouvrant des canaux de communication uniquement pour les nœuds réellement connectés.

Exemple 5.17 – Nœuds d'une application TR²E (en Ada)

```
type Node_Type is
  (WoM_K,
   InS_K);
```

L'exemple 5.17 montre un exemple de code Ada de l'énumération représentant les nœuds de l'exemple décrit par la figure 4.4 page 72.

Processus légers

Sur chaque nœud de l'application, nous produisons une énumération contenant la liste des composants **threads** de ce nœud ainsi que les composants **threads** des autres nœuds et qui communiquent avec le nœud courant. Nous produisant aussi une table qui associe ces composants **threads** avec les nœuds auxquels ils appartiennent.

Cette énumération ainsi que cette table permettent de compléter l'initialisation des couches basses de transport en ouvrant le nombre correct de canaux de communication vers chaque nœud distant.

L'exemple 5.18 montre un exemple de code Ada de l'énumération représentant les processus légers de l'exemple décrit par la figure 4.4 page 72. Nous y trouvons aussi la table de correspondance qui indique à quel nœud appartient chaque processus léger.

Taille des messages

Sur chaque nœud, nous analysons toutes les données qui y sont manipulées et nous calculons la taille maximale d'un message envoyé ou reçu. Par la suite nous générons une constante

Exemple 5.18 – Processus légers d'une application TR^{2}E (en Ada)

```
type Entity_Type is
  (WoM_Regular_Producer_K,
   WoM_On_Call_Producer_K,
   WoM_External_Event_Server_K,
   WoM_Activation_Log_Reader_K,
   InS_External_Event_Source_K);

Entity_Table : constant array (Entity_Type'Range) of Node_Type :=
  (WoM_Regular_Producer_K        => WoM_K,
   WoM_On_Call_Producer_K        => WoM_K,
   WoM_External_Event_Server_K   => WoM_K,
   WoM_Activation_Log_Reader_K   => WoM_K,
   InS_External_Event_Source_K   => InS_K);
```

qui représente cette taille. Cette constante permet d'initialiser les services de protocole et de représentation élémentaire en allouant statiquement les tampons de communication.

Notons que l'évaluation de cette constante ne prend pas en compte uniquement les données transmises. Elle ajoute aussi la taille des entêtes des messages pour avoir la taille du tampons de communication échangé.

5.6 Intégration de la chaîne de compilation

Après avoir généré le code pour les composants applicatifs et intergiciels de l'application répartie, la dernière phase consiste à intégrer ce code aux composants déjà existants de l'intergiciel minimal ainsi qu'à ceux de l'utilisateur. Ensuite, le code est compilé pour avoir un nombre d'exécutables (égal au nombre des nœuds dans l'application) prêts à être téléchargés et exécutés dans leurs emplacements respectifs.

Rendre cette phase automatique parait simple au premier abord. Mais une analyse poussée du problème montre que cela demande une quantité de travail non négligeable comparée aux autres phases du processus (analyses, génération de code...) :

- Certains compilateurs de langage de programmation requièrent des fichiers d'accompagnement des fichiers sources pour compiler ceux-ci (comme le compilateur Ada GNAT et les fichiers projets ou encore les Makefiles...).
- L'intégration des composants de l'intergiciel minimal requiert que le bon compilateur soit sélectionné selon le langage de programmation choisi, que la version du compilateur croisée convenable soit sélectionnée selon l'architecture du nœud en question donnée dans le modèle de l'application.
- L'intégration de composants pré-compilés et/ou produits par des outils tiers requiert l'ajout d'options lors de l'invocation du compilateur.

La figure 5.6 décrit le processus global de compilation. Après avoir servi à la génération des composants applicatifs et intergiciels, le modèle AADL est aussi utilisé pour piloter la phase de configuration de l'intergiciel. Cette phase consiste à sélectionner les composants de l'intergiciel minimal dont l'application a besoin et de les intégrer aux composants générés automatiquement pour produire un intergiciel configuré et dédié à l'application répartie en cours. Ceci est effectué notamment grâce à l'incorporation de la description des composants matériels dans le modèle.

Le modèle AADL donne lieu aussi à la génération de fichiers de supports (Makefiles, fichiers projet...). Ces fichiers pilotent la phase de compilation qui prend en entrée le code de l'utilisateur

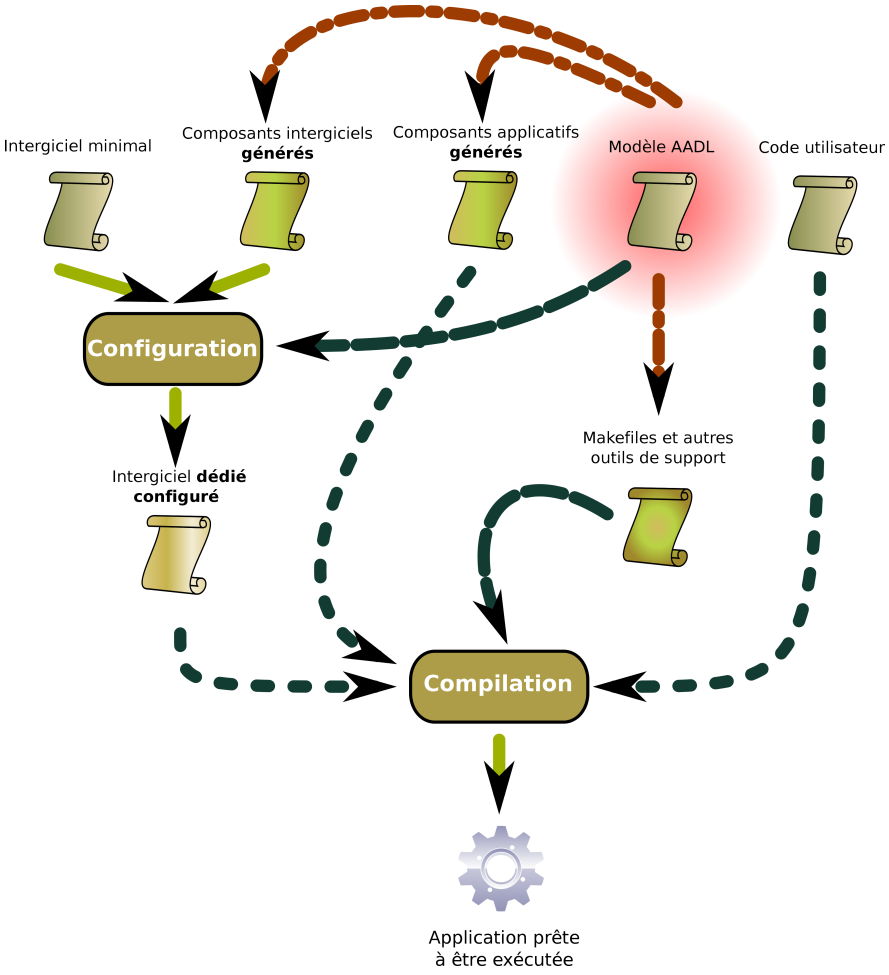

FIGURE 5.6 – Processus global de compilation

(y compris le code produit par des outils tiers) ainsi que le code de l'intergiciel configuré. À l'issue de la phase de compilation, nous disposons d'une application répartie prête à être exécutée.

5.7 Synthèse

Dans ce chapitre, nous avons présenté les différentes étapes d'un processus automatique de production de systèmes TR^{2}E critiques à partir de modèles AADL. En partant d'un modèle AADL écrit selon les restrictions que nous avons spécifiées dans le chapitre précédant, nous avons présenté les différentes analyses qu'il est possible d'effectuer sur ce modèle pour garantir le fonctionnement correct du système. Nous avons montré trois sortes d'analyse qu'il est possible d'effectuer sur un modèle AADL. Elles varient des analyses sémantiques simples aux analyses avancées d'ordonnancement.

La partie la plus importante de ce processus est la génération automatique de code qui, à partir du modèle AADL de l'application, produit les différents composants applicatifs et intergiciels qui seront intégrés avec les composants déjà existants pour former les différents nœuds de l'application. Nous avons présenté les règles de transformation des différents composants AADL vers un langage de programmation impératif générique. Nous avons ensuite montré comment les composants intergiciels sont déployés et configurés en fonction des caractéristiques de l'application TR^{2}E en cours.

Ceci achève la première partie de ce mémoire qui constitue l'étude théorique. La prochaine partie est consacrée à l'implantation d'un intergiciel minimal (dont l'architecture a été introduite au chapitre 3) ainsi qu'à l'instanciation du processus de production présenté dans ce chapitre.

Deuxième partie

Mise en Œuvre et Expérimentations

Chapitre 6

Conception d'un Intergiciel Minimal pour les Systèmes TR^{2}E

SOMMAIRE

6.1 Introduction

Dans le chapitre 3, nous avons présenté l'architecture générique d'un intergiciel dédié aux applications TR^{2}E. Cette architecture est formée d'un ensemble de services intergiciels canoniques. Nous avons vu que certains services et composants de l'application répartie sont faiblement personnalisables en fonction des propriétés de l'application répartie. Ces services ne devraient pas être générés automatiquement. D'autres services sont fortement personnalisables. Il sont, quant à eux, produits d'une manière automatique à partir des caractéristiques de l'application. Nous avons conclu que les services faiblement personnalisables constituent le noyau de l'intergiciel minimal. Le reste des composants et services sont générés automatiquement et interagissent avec cet intergiciel minimal.

Dans ce chapitre, nous introduisons notre implantation de cet intergiciel minimal et présenter les avantages qu'il apporte pour les systèmes TR^{2}E (section 6.2). Nous décrivons ensuite en détail l'architecture l'intergiciel minimal à travers ses instances en Ada et en C (section 6.3). Dans la section 6.4, nous nous intéresserons de plus près aux mécanismes mis en place pour gérer l'envoi et la réception de messages entre les nœuds ou les tâches d'une application. Nous

montrons comment ce mécanisme offre un déterminisme et une sûreté de fonctionnement dans un environnement concurrent. Ce mécanisme est le composant central (et le plus complexe) de l'intergiciel minimal. Enfin nous concluons ce chapitre par une synthèse globale.

6.2 PolyORB-HI

Dans cette section, nous précisons la manière dont notre intergiciel minimal supporte les constructions du langage AADL. Ensuite nous donnons les avantages offerts par nos implantations de cet intergiciel en termes d'empreinte mémoire et de configurations automatique et statique.

L'intergiciel minimal se nomme PolyORB-HI en raison de son architecture fortement influencée par celle de l'intergiciel schizophrène PolyORB [Quinot, 2003]. L'architecture schizophrène définit (outre la garantie de l'interopérabilité entre différents standards de répartition qui n'est pas l'objectif de notre travail) la séparation des préoccupations dans un intergiciel. Ainsi, les parties qui s'occupent de la construction, l'envoi et la réception des messages proprement dits, constituent la couche de communication de bas niveau. Les instances de la communication de haut niveau (*personnalité* selon la terminologie des intergiciels schizophrènes) viendrons ce greffer sur cette couche de communication. La figure 6.1 illustre cette architecture. Elle montre les personnalités applicatives et protocolaires de l'intergiciel schizophrène PolyORB. Les composants génériques de l'intergiciel s'occupant de la gestion de la concurrence et de l'acheminement des messages entre les interfaces des différentes tâches constituent la couche applicative. Les instances de tâches ainsi que celles des outils d'interrogation des interfaces requis par le standard AADL viendront se greffer sur cette couche applicative.

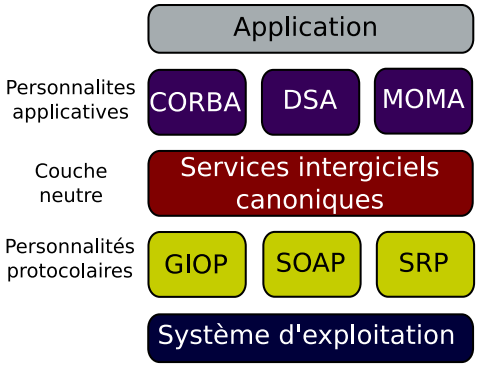

Figure 6.1 – Architecture schizophrène

PolyORB-HI supporte toutes les constructions AADL dont nous avons besoin pour produire des applications TR²E. Il nécessite une taille mémoire extrêmement faible. Enfin, sa configuration se fait d'une manière automatique à partir du modèle AADL : toutes les ressources sont calculées et allouées statiquement (ou bien au temps d'initialisation de l'application) sans aucune intervention requise de la part de l'utilisateur.

6.2.1 Support des constructions AADL

Pour produire des applications TR²E, l'intergiciel doit supporter des constructions clef : (1) les processus légers, (2) les éléments d'interface et (3) les données partagées.

Les processus légers

L'intergiciel doit fournir les outils nécessaires à la création des différents types de tâches supportés dans les systèmes TR^{2}E (périodiques, sporadiques...). Il s'agit du service intergiciel de *parallélisme* décrit dans la section 3.3.1. Concrètement, ce support est constitué d'entités génériques de l'intergiciel minimal instanciées pour chaque composant **thread** du type défini dans le modèle AADL. Pour les langages de programmation qui ne supportent pas la généricité, des fonctions de création des processus légers doivent être fournies par l'intergiciel minimal et sont appelées pour effectuer la création au moment de l'initialisation de l'application et uniquement à ce moment là. Ceci remplace le travail effectué automatiquement par le compilateur pour des langages comme Ada.

Pour être conforme aux restrictions additionnelles des systèmes TR^{2}E que nous avons définies dans la section 4.9, et pour garantir une analysabilité statique d'ordonnancement, seules trois catégories de tâches sont supportées : (1) les tâches périodiques, (2) les tâches sporadiques, et (3) les tâches hybrides. En effet, pour prévoir de manière statique le pire temps d'exécution d'une tâche et ainsi effectuer une analyse d'ordonnancement, il faut que les déclenchements de la tâche soient séparés par un temps minimal connu à l'avance. Dans le guide de programmation Ravenscar [Burns *et al.*, 2004], les auteurs excluent les tâches apériodiques non sporadiques car elles ne donnent aucune information sur le temps minimal séparant deux déclenchements successifs et peuvent théoriquement requérir toute la puissance de calcul de l'application privant ainsi les autres tâches (moins prioritaires) de s'exécuter. Les tâches sporadiques et périodiques garantissent ce temps minimal (voir leurs définitions plus bas dans cette section). La troisième catégorie, les tâches hybrides, constitue une combinaison de tâche périodique et de tâche sporadique. Elle offre par conséquent la même garantie de temps minimal entre deux déclenchements.

Dans la suite nous détaillons le comportement de ces tâches. La manière dont ce comportement est associé à un fil d'exécution indépendant est fortement dépendante du langage de programmation choisi. Nous nous y intéresserons quand nous présenterons les différentes instances de POLYORB-HI.

Les tâches périodiques Les tâches périodiques se réveillent à la réception d'un signal à des intervalles de temps fixes.

Algorithme 6.1 Algorithme d'une tâche périodique

```
PERIODIC-THREAD-BEHAVIOR (Period, Deadline, Init, Job)
1  WAIT_SYSTEM_INITIALIZATION ()
2  INIT_THREAD ()
3  Next_Start ← System_Start + Period
4  Next_Deadline ← System_Start + Deadline
5  while True do
6    JOB ()
7    SYSTEM_DELAY_UNTIL (Next_Start)
8    Next_Start ← Next_Start + Period
9    Next_Deadline ← Next_Start + Deadline
```

L'algorithme 6.1 montre le comportement générique d'une tâche périodique : au tout début de l'exécution de l'application répartie, un mécanisme d'initialisation garantit le déblocage de toutes les tâches en même temps et, surtout, après la fin de la création des canaux de communications (étape 1 de l'algorithme). Cette phase d'initialisation spécifie la valeur d'une constante globale

à toutes les tâches du nœud qui constitue l'instant de début global (**System_Start**). Ensuite si l'utilisateur a spécifié une routine d'initialisation pour la tâche (initialisation des variables du code de l'utilisateur...), celle-ci sera exécutée une seule fois à l'extérieur de la boucle principale de la tâche (étape 2). Enfin la tâche initialise les premières valeurs du prochain temps de déclenchement, ainsi que la prochaine échéance (étapes 3 et 4). Enfin, il entre dans sa boucle infinie. À chaque itération, la tâche effectue le travail demandé (étape 6). Puis, il rend la main à l'ordonnanceur jusqu'à l'instant de son prochain réveil (étape 7) où la tâche met à jour les valeurs du prochain temps de déclenchement ainsi que la prochaine échéance (étapes 8 et 9). La valeur de la prochaine échéance est rendue visible pour l'utilisateur ainsi que pour les autres composants de l'intergiciel. Elle est utilisée notamment pour la gestion des connections retardées en AADL.

Cet algorithme respecte les recommandations données par [Burns *et al.*, 2004] et peut donc être transcrit facilement vers du code Ada respectant le profil Ravenscar.

Les tâches sporadiques Les tâches sporadiques se réveillent à la réception d'un signal extérieur avec un temps minimal entre deux réceptions consécutives.

Algorithme 6.2 Algorithme d'une tâche sporadique

```
SPORADIC-THREAD-BEHAVIOR (Min_Interarrival_Time, Deadline, Wait_Event, Init, Job)
1  WAIT_SYSTEM_INITIALIZATION ()
2  THREAD_INIT ()
3  while True do
4    WAIT_EVENT (Port)
5    Next_Start ← SYSTEM_CLOCK () + Min_Interarrival_Time
6    Next_Deadline ← SYSTEM_CLOCK () + Deadline
7    JOB (Port)
8    SYSTEM_DELAY_UNTIL (Next_Start)
```

L'algorithme 6.2 montre le comportement générique d'une tâche sporadique : au tout début de l'exécution de la tâche, le même mécanisme d'initialisation que pour les tâches périodique est effectué (étape 1). Ensuite, si la tâche possède une routine d'initialisation celle-ci sera exécutée une seule fois à l'extérieur de la boucle principale de la tâche (étape 2). Après, la tâche entre dans la boucle infinie principale. Au début de chaque itération, la tâche se bloque en attendant l'arrivée d'un événement à l'aide de la routine bloquante **Wait_Event** (étape 4). À l'arrivée d'un événement, la tâche est réveillée et notifiée de la nature de l'événement (le paramètre en sortie *Port* de la routine **Wait_Event**).

Avant d'effectuer son travail, la tâche calcule la valeur du prochain temps de réveil en fonction du temps minimal entre deux événements (étape 5). La valeur calculée sert à suspendre la tâche après la fin de son travail pour renforcer le caractère sporadique. La tâche calcule aussi la valeur de la prochaine échéance (étape 6). Elle effectue ensuite le travail demandé en précisant le port qui a provoqué le déclenchement (étape 7). Enfin la tâche est suspendue pour garantir l'écoulement du temps minimal entre deux événements (étape 8). Nous supposons que la routine système **System_Delay_Until** est implantée de sorte qu'elle ne provoque aucune attente si on lui donne un instant dans le passé. Ainsi, si le travail effectué par la tâche sporadique a duré lui même plus que le temps minimal entre l'arrivée de deux événements, aucune attente n'est effectuée.

Cet algorithme respecte, lui aussi, les recommandations données par [Burns *et al.*, 2004] et peut donc être transcrit facilement vers du code Ada respectant le profil Ravenscar.

Les tâches hybrides Les tâches hybrides combinent à la fois le comportement d'une tâche sporadique et celui d'une tâche périodique. Imaginons par exemple le cas d'une tâche qui reçoit sporadiquement des événements depuis un capteur et qui, par ailleurs, doit effectuer un comportement périodique (envoi d'un rapport, calcul d'une moyenne...). Nous pouvons combiner ces deux comportements sans pour autant avoir besoin de deux tâches.

Au premier abord, l'implantation de cette catégorie de tâches semble incompatible avec les recommandations du profil Ravenscar car elle contient une violation apparente qui consiste à attendre sur deux événements différents en parallèle : un événement temporel et un autre événement extérieur (voir la section 2.5.1). Cette double attente est interdite dans les systèmes critiques car elle rend le flux d'exécution difficilement prévisible lors d'une analyse statique [Burns *et al.*, 2004].

Cependant cette violation n'est qu'apparente. En effet, ramenons le problème des tâches hybrides à un problème de tâche sporadique que nous avons déjà résolu. Pour chaque système contenant au moins une tâche hybride, nous effectuons les modifications suivantes :

1. Pour chaque tâche hybride nous ajoutons un nouveau port de type événement en entrée que nous appelons **Period_Event**. Ce port sert à recevoir un événement extérieur à chaque fois que la période de la tâche hybride arrive. Ainsi la boucle de contrôle d'une tâche hybride sera identique à celle d'une tâche sporadique et une seule attente est effectuée à chaque itération.

2. Une tâche supplémentaire de forte priorité (le maximum des priorités de l'ensemble des tâches hybride du nœud) est ajoutée au système. Nous l'appelons le "gestionnaire de tâches hybrides". Son unique travail consiste à envoyer des événements aux tâches hybrides pour signaler l'arrivée de leurs période.

Cette manière d'implanter les tâches hybrides ne remet pas en question les recommandations du profil Ravenscar. Chaque tâche hybride est transformée en une tâche sporadique. La différence avec les tâches sporadiques classiques est que la tâche hybride reçoit un événement extérieur supplémentaire, émis par le gestionnaire à des intervalles de temps réguliers.

Algorithme 6.3 Algorithme du gestionnaire des tâches hybrides

```
HYBRID-THREAD-DRIVER (Hybrid_Thread_Info_List)
 1  WAIT_SYSTEM_INITIALIZATION ()
 2  Previous_Start ← System_Start
 3  while True do
 4    Earliest_Next_Start ← SYSTEM_LAST_TIME
 5    for each H in Hybrid_Thread_Info_List do
 6      if H.Next_Dispatch ≤ Previous_Start then
 7        H.Next_Dispatch ← H.Next_Dispatch + H.Period
 8      if H.Next_Dispatch ≤ Earliest_Next_Start then
 9        Earliest_Next_Start ← H.Next_Dispatch
10    Previous_Start ← Earliest_Next_Start
11    for each H in Hybrid_Thread_Info_List do
12      if H.Next_Dispatch ≤ Previous_Start then
13        H.Eligible ← True
14    SYSTEM_DELAY_UNTIL (Previous_Start)
15    for each H in Hybrid_Thread_Info_List do
16      if H.Eligible then
17        H.Eligible ← False
18        SEND_EVENT (H.Period_Event)
```

Ainsi, nous arrivons à implanter les tâches hybrides tout en respectant les recommandations du profil Ravenscar. Nous estimons que ce coût est acceptable car le traitement effectué par cette tâche supplémentaire est simple. En effet, comme le montre l'algorithme 6.3, le gestionnaire des tâches hybrides reçoit en entrée/sortie une table. Cette table contient la liste des tâches hybrides dans le nœud, leurs périodes, leurs nouveaux ports événements servant à les déclencher en cas d'arrivée de période, leurs "prochains" instants de déclenchement et enfin un drapeau booléen indiquant si la tâche hybride doit être déclenchée. Le travail de ce processus léger est une boucle infinie dont chaque itération se résume aux trois actions suivantes :

1. Pour chaque tâche hybride, remettre à jour le temps de son prochain déclenchement périodique et déduire l'instant d'activation le plus proche d'une tâche hybride,
2. Marquer comme éligibles les tâches hybrides qui doivent être activées à l'instant le plus proche,
3. Se suspendre jusqu'à l'instant d'activation le plus proche,
4. Déclencher toutes les tâches hybrides éligibles en envoyant des événements sur leur port supplémentaire et les marquer comme non éligibles.

À chaque itération, le gestionnaire effectue un nombre très limité d'actions simples. La durée de ce traitement est parfaitement calculable de manière statique. Nous verrons dans le chapitre 8 qu'il s'agit d'un temps négligeable.

Les éléments d'interface

L'intergiciel minimal fournit un ensemble d'outils requis par le standard AADL pour permettre aux différentes entités de l'application répartie ainsi qu'au code fourni par l'utilisateur de lire et d'écrire dans les éléments d'interface des processus légers AADL. Les différentes actions de lecture et d'écriture doivent être déterministes. Pour chaque tâche dans l'application répartie, et pour chaque sous-programme contenant des ports, une instance de ces interrogateurs est créée. Elle permet d'envoyer et de recevoir des messages sur les ports sans se préoccuper de l'identité de la source ou de la destination.

Les interrogateurs simplifient l'implantation des opérations d'écriture et de lecture d'informations vers les interfaces des tâches (ou des sous-programmes). Une entité invoque une opération d'écriture d'un événement sur un port en sortie le fait d'une manière identique que le port soit connecté à une destination locale (dans le même nœud) ou à une destination distante (dans un nœud différent). Elle ne considère pas non plus si le port en question est connecté à une seule destination ou à plusieurs. Tous ces aspects sont pris en compte automatiquement lors de l'appel aux routines d'envoi ou de réception. Les interrogateurs sont invoqués dans le code fourni par l'utilisateur mais aussi dans d'autres endroits du code généré.

Comme nous l'avons précisé dans la section 5.3, le standard AADL spécifie les noms et les comportements de ces routines d'interrogation. Toutefois, la définition de ces routines n'est pas complète. Seul leur nom et leur comportement attendu sont donnés. La signature exacte de ces routines reste dépendante du langage de programmation utilisé. Nous présentons dans la section 6.3 les différents choix que nous avons eus à faire pour étendre les définitions du standard AADL pour implanter ces routines en Ada et en C. Dans la section 6.4, nous montrons le déterminisme et l'absence d'interblocage dans l'ensemble de ces routines.

Les données partagées

Dans le cas où le langage de programmation ne le fait pas intrinsèquement, l'intergiciel garantit la protection des données protégées entre les différentes entités de l'application. Ceci est effectué en fournissant des routines de verrouillage et de déverrouillage utilisés dans le code généré pour implanter les méthodes d'accès aux données. Ces méthodes d'accès sont elles-mêmes utilisées dans le code généré ou bien par le code de l'utilisateur.

Chaque instance d'une donnée partagée n'est accessible qu'au travers des méthodes qu'elle expose. La sûreté de fonctionnement vis-à-vis de la concurrence est garantie par les routines de verrouillage de l'intergiciel.

Par ailleurs, comme nous l'avons précisé dans la section 4.9.4, ces données partagées constituent l'unique moyen d'interaction entre les différents processus légers du nœud. L'inversion de priorité qu'elles sont susceptibles de causer doit être bornée et la borne doit être statique. Pour cela, nous avons choisi d'utiliser le protocole PCP de protection des données qui respecte ces deux conditions.

6.2.2 Faible empreinte mémoire

Le fait de ne rassembler dans l'intergiciel minimal que les composants faiblement personnalisables rend sa taille extrêmement réduite. Cependant, deux autres éléments contribuent également à la réduction de l'empreinte mémoire totale de l'application répartie : (1) tous les composants de l'intergiciel minimal ne sont pas nécessairement inclus dans tous les nœuds d'une application TR^{2}E particulière et (2) les composants générés automatiquement sont fortement personnalisés pour le nœud qui les contient. En effet, ces composants générés contiennent juste le code nécessaire pour le nœud et sont donc plus petits que leurs équivalents censés être utilisables dans tous les cas de figure supportés par l'intergiciel.

Sélection des composants de l'intergiciel minimal

Les composants de l'intergiciel minimal ne sont pas tous inclus dans l'application répartie. Par exemple, dans un nœud, on ne trouve que les couches basses de transport effectivement utilisées. En particulier, une application monolithique ne contient aucun composant de la couche de transport. De même, dans un nœud particulier, on ne trouve que les archétypes de types de tâches utilisées.

Ceci ne peut être réalisé que si les différents composants de l'intergiciel minimal sont suffisamment indépendants les uns des autres pour être sélectionnés de manière individuelle. Nous verrons dans la section 6.3 comment cela est effectué.

Optimisation des composants produits automatiquement

Un autre aspect qui contribue considérablement à la réduction de la taille mémoire de l'application est l'optimisation des composants intergiciels produits automatiquement en fonction des propriétés de chaque nœud. Ainsi, on détermine de manière statique le nombre de tâches nécessaires au modèle AADL. La taille exacte des différents tampons de communication utilisés dans les échanges d'information est déduite lors de l'analyse des types de données et des signatures des sous-programmes AADL.

6.2.3 Configuration automatique et statique

Dans cette section, nous montrons comment les différents composants de l'intergiciel sont configurés à la fois automatiquement et statiquement. Nous précisons les avantages de ce type de configuration ainsi que les contraintes qu'ils imposent en terme de développement logiciel.

Configuration automatique

Nous avons montré dans la section 4.8 que le langage AADL permet d'incorporer une partie de l'information de déploiement et de configuration de l'application TR^{2}E. Par une analyse du modèle d'une application, nous déduisons de façon automatique tous les paramètres utilisés pour la configuration des composants de l'intergiciel minimal. Les composants intergiciels générés sont, eux aussi, configurés statiquement.

Les composants de l'intergiciel générés ainsi que ceux de l'intergiciel minimal sélectionnés automatiquement pour faire partie du nœud sont paramétrés selon les propriétés de ce nœud. Plusieurs paramètres sont ainsi déterminés. Parmi les plus importants, nous trouvons :

- La déclaration du nombre requis de tâches pour assurer l'exécutions des différentes actions (périodiques, sporadiques...)
- L'allocation de la mémoire requise pour l'exécution des tâches ainsi que pour les données
- L'ouverture de canaux de communication requis pour envoyer ou recevoir des informations,
- L'allocation des tampons de communication destinés à envoyer et à recevoir les données.

Configuration statique

La configuration des composants de l'intergiciel est effectuée de manière statique. Tous les paramètres sont spécifiés soit à la compilation de l'application répartie, soit lors de l'initialisation de l'application. Aucun paramètre de configuration n'est spécifié après la phase d'initialisation de l'application.

Ainsi, pour reprendre les exemples de paramètres cités dans le paragraphe précédant :

- Le nombre exact de tâches de chaque catégorie est déterminé pour chaque nœud de l'application,
- Pour chaque instance de tâche créée, la taille exacte de mémoire est allouée statiquement pour sa pile. De la même manière, toutes les données partagées sont créées statiquement,
- Les couches basses de transport utilisent les informations sur la topologie de l'application, extraites du modèle AADL (table de nommage...) pour ouvrir le nombre exact de canaux de communication nécessaires,
- Une analyse des différents types de données et des signatures des sous-programmes dans un nœud donné permet de déduire la taille maximale d'un tampon de communication. Ces tampons sont crées dans les piles des fonctions de communication au lieu de les allouer dynamiquement.

En contre partie de cette configuration statique, les composants de l'intergiciel minimal ne sont compilés qu'en présence des composants qui sont générés automatiquement et qui sont utilisés pour les configurer. Ceci oblige la distribution de l'intergiciel minimal sous forme de code source plutôt que sous forme de bibliothèque binaire. Mais devant les gains que nous obtenons en empreinte mémoire, cela ne représente pas une pénalité importante.

6.3 Architecture détaillée de POLYORB-HI

Dans cette partie, nous décrivons l'architecture détaillée de notre intergiciel minimal. Il s'agit de l'implantation de l'architecture d'un intergiciel dédié aux applicationTR^{2}E que nous avons définie dans le chapitre 3. Nous présentons particulièrement les choix que nous faits pour que les implantations de cette architecture respectent les recommandations du profils Ravenscar et celles des systèmes critiques (voire 2.5.1).

Deux versions de cet intergiciel existent : une version écrite en Ada et une autre en C. Ces deux instances présentent les mêmes fonctionnalités et permettent ainsi de concevoir des applications TR^{2}E aussi bien en Ada qu'en C.

6.3.1 POLYORB-HI Ada

La première version de POLYORB-HI que nous avons développée est écrite en Ada. Elle est formée d'un ensemble de paquetages qui représentent les services de l'intergiciel minimal. Tous ces paquetages sont conformes au profil Ravenscar ainsi qu'à un ensemble de restrictions pour les systèmes critiques.

Exemple 6.1 – Restrictions supportées dans POLYORB-HI

```
-- Les restrictions Ada supportées dans PolyORB-HI Ada

-- Restrictions présentes dans le standards Ada 2005

pragma Restrictions (No_Allocators);                          -- H.4 (7)
pragma Restrictions (No_Unchecked_Deallocation);              -- J.10 (5)
pragma Restrictions (No_Floating_Point);                      -- H.4 (14)
pragma Restrictions (No_Access_Subprograms);                  -- H.4 (17)
pragma Restrictions (No_Unchecked_Access);                    -- H.4 (18)
pragma Restrictions (No_Dispatch);                            -- H.4 (19)
pragma Restrictions (No_IO);                                  -- H.4 (20)
pragma Restrictions (No_Recursion);                           -- H.4 (22)
pragma Restrictions (No_Implementation_Attributes);           -- 13.12 (2)
pragma Restrictions (No_Obsolescent_Features);                -- 13.12 (4)
pragma Restrictions (No_Dependence => Ada.Finalization);      -- 13.12.1 (12)

-- Restrictions spécifiques au compilateur Ada, GNAT

pragma Restrictions (No_Streams);                             -- GNAT specific
pragma Restrictions (No_Direct_Boolean_Operators);            -- GNAT specific
pragma Restrictions (No_Exception_Handlers);                  -- GNAT specific
```

L'exemple de code 6.1 montre les restrictions supportées par la version Ada de POLYORB-HI. Le premier lot de restrictions représente des restrictions standards. Chacune d'elles est accompagnée du numéro de paragraphe qui l'introduit dans le manuel de référence de Ada 2005 [Working Group, 2005]. On trouve notamment :

– *No_Allocators* et *No_Unchecked_Deallocation* interdisent toutes sortes d'allocation ou désallocation dynamiques de mémoire dans le code de l'intergiciel et celui de l'utilisateur. Ceci garantit l'absence de *fuite* de mémoire dans l'application. En effet les seules données qui sont allouées dynamiquement sont les données variables locales des différents sous-programmes. Celles ci sont allouées automatiquement dans les piles respectives des sous-programmes et sont désallouées proprement par le système à la fin de l'appel. L'architecture schizophrène de l'intergiciel a dû être profondément revue. En effet, toute l'architecture de POLYORB se base sur des usines à objets incompatibles avec ces restrictions. Pour

pallier ce problème, il a fallu analyser finement les informations fournies par le modèle AADL pour allouer statiquement toutes les entités à l'application,

- La restriction *No_Floating_Point* interdit l'utilisation des nombres réels à virgule flottante. En effet, la précision offerte par ces nombres est variable selon la distance qui les sépare de zéro. De tels types compromettent le bon fonctionnement des systèmes critiques s'ils sont utilisés pour mesurer des données comme le temps par exemple. De plus la plupart des systèmes embarqués ne disposent pas d'une unité de traitement pour les nombres à virgule flottante. Le support natif de Ada pour les nombres décimaux à virgules fixes nous a été d'une très grande utilité pour respecter cette restriction. Il faut noter toutefois que l'utilisation des nombres à virgule flottante peut être demandée explicitement dans certains systèmes. Dans ce cas, l'utilisateur désactive cette restriction, mais des analyses supplémentaires devraient être effectuées sur le code pour garantir son bon fonctionnement [Monniaux, 2008],
- La restriction *No_Access_Subprogram* interdit l'utilisation de pointeurs sur sous-programmes. Elle s'applique notamment à l'intergiciel. Cette restriction permet de connaître le flux d'exécution à la compilation et rend ainsi possible la mise en œuvre de tests comme les tests de couverture par exemple ou les analyses de pire temps d'exécution (*WCET*). Le respect de cette restriction nous a poussé à revoir en profondeur l'architecture schizophrène. En effet, l'architecture schizophrène utilise fortement les mécanismes d'indirection dont le mécanisme de "hook"[5]. Il lie les différents services intergiciels sans que ceux-ci aient à se connaître précisément. L'utilisation des constructions génériques Ada a permis de pallier ce problème comme nous le verrons plus loin. Cependant, les entités génériques Ada introduisent une surcharge dans la taille mémoire. Il convient de les utiliser avec modération. Nous avons ainsi limité leur nombre ainsi que celui de leurs instances dans le code généré,
- La restriction *No_Unchecked_Access* interdit l'accès aux adresses mémoires des entités Ada. Ceci compromet une analysabilité statique du flux de données. L'architecture schizophrène utilise fortement de mécanisme pour transmettre d'une manière efficace les données entre les différentes couches de l'intergiciel et lors de l'encodage et du décodage dans les tampons de communication,
- La restriction *No_Dispatch* interdit l'utilisation de l'aiguillage dynamique dans les constructions de l'intergiciel pour les raisons évoquées dans la section 2.6.1. Le respect de cette restriction a, lui aussi, requis une adaptation de l'implantation de l'intergiciel. L'utilisation du mécanisme orienté-objet dans cette architecture provient du fait que les descriptions d'interfaces de CORBA ne permettaient pas de connaître toutes informations au moment de la compilation. Ces informations sont complétées à l'exécution de l'application et l'aiguillage dynamique permet de sélectionner les services voulus. En AADL, la situation est différente. La totalité des informations sur les services intergiciels et leur configurations sont déduites du modèle AADL à la génération de l'application. La génération automatique de code permet de configurer statiquement ces services : il n'y a plus besoin de l'aiguillage dynamique,
- La restriction *No_IO* interdit l'utilisation du mécanisme d'entrée-sortie de base du compilateur (**Ada.Text_IO**). Il existe des plates-formes embarquées ne disposent pas de systèmes de fichiers. Ceci rend le code de l'intergiciel non portable entre les plates-formes natives

5. Un hook est généralement un point d'entrée logiciel (pointeur sur un sous-programme, script, exécutable...), qui est personnalisable par l'utilisateur et qui est exécuté par un code non modifiable par cet utilisateur. Dans notre contexte, il s'agit d'un pointeur sur un sous-programme invoqué par un composant de l'intergiciel minimal mais dont la valeur (le sous-programme) est spécifiée à la génération de code

(utilisées pour les premiers tests) et les plates-formes embarquées. L'utilisation des mécanismes d'entrée sortie est importante pendant les premières phases de développement d'une application. Elle permet d'analyser les traces d'exécution des nœuds. Pour respecter les recommandations du profil Ravenscar, nous avons développé, pour chaque plate-forme supportée, un mécanisme d'entrée sortie approprié (sortie standard, port série...). Les informations extraites du modèle AADL permettent de sélectionner le composant convenable,

- La restriction *No_Recursion* interdit l'utilisation de la récursivité. Elle compromet les analyses statiques de pire temps d'exécution. Dans l'architecture schizophrène, la récursion est utilisée particulièrement dans les couches qui emballent et déballent les données de taille et de profondeur non bornées (structure de tableaux de structure...). Ce type non borné de données n'est pas autorisé (voire la section 4.9.7). Toutes les routines d'emballages et de déballage sont générées statiquement et ne sont pas récursives,
- La restriction *No_Implementation_Attributes* interdit l'utilisation d'attributs Ada qui ne sont pas standards afin de renforcer la portabilité du code. En particulier, l'utilisation de l'attribut **Unrestricted_Access** spécifique au compilateur GNAT est interdite. En effet, son utilisation à la place de l'attribut standard **Access** annule toutes les vérifications d'accessibilité effectuées normalement par le compilateur GNAT [Fernàndez-Marina, 2008a; Fernàndez-Marina, 2008b],
- La restriction *No_Obsolescent_Features* interdit les constructions obsolètes dans Ada 2005. En effet, de nouvelles constructions plus simples et plus lisibles ont été introduites. Ceci permet d'améliorer la qualité du code de l'intergiciel ainsi que celui de l'utilisateur et de le rendre plus lisible en vue d'éventuelles certifications,
- La restriction *No_Dependence* sur le paquetage **Ada.Finalization** interdit l'utilisation des types "contrôlés" Ada. D'une part parce que ces types introduisent des constructions et des contrôles implicites qui compromettent le calcul du temps d'exécution. D'autre part, parce que l'allocation dynamique de mémoire, motivation principale pour l'utilisation de ces types, a été interdite.

Le second lot de restrictions concerne trois autres restrictions spécifiques au compilateur GNAT que nous utilisons pour construire les applications Ada :

- La restriction *No_Streams* interdit l'utilisation des streams Ada et réduit l'ensemble des moyens communications possibles aux seules couches de transports de l'intergiciel. Ceci interdit toute communication "à l'insu" de l'intergiciel et le code utilisateur ne peut contenir des appels à des routines de communications autres que ceux de l'intergiciel. Ceci oblige le développeur à modéliser toutes les communications possible en AADL afin de générer les canaux de communication automatiquement et permet de les prendre en compte lors des analyses statiques de l'application,
- La restriction *No_Direct Boolean Operators* force l'utilisation d'une forme spéciale des opérateurs booléens pour optimiser et rendre plus sûres les structures conditionnelles.
- La restriction *No_Exception_Handlers* interdit l'interception des exceptions Ada. Ceci rendrait plus complexe l'analyse de flux d'exécution.

Notre objectif ne vise pas à respecter toutes les restrictions présentes dans l'annexe H du standard Ada 2005 ni toutes les restrictions supplémentaires offertes par le compilateur GNAT. Par exemple, nous n'utilisons pas la restriction standard *No_Fixed_Point* qui interdit l'utilisation des nombres décimaux à virgule fixe ainsi que toutes les formes d'attentes temporelles. Cette restriction rend l'implantation des tâches périodiques et sporadiques impossible. Notre objectif est de sélectionner un ensemble de restrictions que les systèmes TR^{2}E doivent respecter. Cet ensemble a été choisi en nous basant sur l'expérience dans la conception de systèmesTR^{2}E

de plusieurs acteurs avec lesquels nous avons interagi durant le projet Européen ASSERT et particulièrement l'Agence Spatiale Européenne.

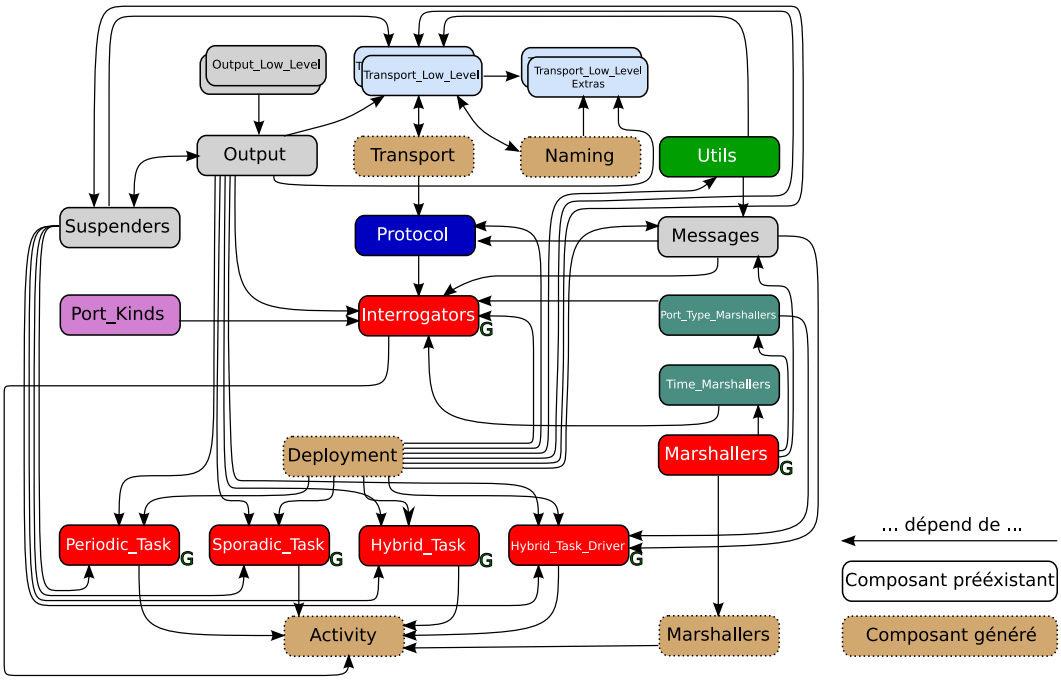

FIGURE 6.2 – Architecture de POLYORB-HI-Ada

La figure 6.2 décrit les interactions qui existent entre les différents composants de POLYORB-HI-Ada ainsi que les liens existants entre ces composants et ceux générés automatiquement. Les composants qui sont générés automatiquement sont représentés par des rectangles en trait interrompu dans la figure. Dans la suite, nous décrivons les composants de POLYORB-HI-Ada.

PolyORB_HI.Thread_Interrogators

Ce paquetage générique constitue le composant central de l'intergiciel minimal. Il permet pour chaque composant **thread** AADL, d'instancier dans le composant **Activity** le mécanisme d'envoi et de réception des informations d'une manière transparente. Ce paquetage offre les routines nécessaires pour :

- Envoyer une information (donnée, événement ou événement-donnée) sur un port (en invoquant la routine **Send** de la couche protocolaire),
- Lire une information reçue sur un port donné. La donnée est déposée par la couche haute de transport,
- Déterminer l'expéditeur d'une information,
- Déterminer le nombre restant de messages dans la file d'attente d'un port de type événement,
- Se bloquer en attente de nouveaux événements. La routine qui assure cette fonctionnalité est passée en paramètre des paquetages génériques des différents types de tâches.

L'implantation de ce paquetage garantit que toutes les actions citées ci-dessus se font d'une manière déterministe (constante pour la majorité et, au pire cas, en $O(N)$ où N est la taille de la donnée traitée). La section 6.4 décrit en détail ces fonctionnalités et montre leur déterminisme.

PolyORB_HI.Messages

Ce paquetage contient les routines nécessaires pour lire et écrire les messages qui sont échangés entre les entités d'une application répartie. La structure de paquetage est inspirée de celle du paquetage standard **Ada.Streams** avec quelques modifications pour respecter les restrictions ci-dessus. Le type racine représentant un message, **Message_Type** est une structure de donnée de taille bornée. La taille maximale d'un message est paramétrée par une constante déduite automatiquement du modèle AADL et générée dans le composant **Deployment** (**Max_Payload_Size**). Ceci explique la dépendance de **PolyORB_HI.Messages** envers ce composant. Une variable de type message est utilisée dans plusieurs envois et réceptions sans pour autant impliquer une allocation dynamique de mémoire. Le pire cas des opérations de lecture et d'écriture sur les messages se font en un temps proportionnel à la taille de la donnée lue ou écrite.

Enfin, les entités émettrices et réceptrices d'un message sont représentées par des énumérations. Ces énumérations ne sont pas nécessairement les mêmes d'un nœud à un autre, nous utilisons les routines du paquetage **PolyORB_HI.Utils** pour garantir la cohérence des valeurs emballées dans un messages (voir plus loin pour plus de détails sur **PolyORB_HI.Utils**).

PolyORB_HI.Marshallers_G

Ce paquetage générique permet de créer des routines d'emballage et de déballage d'un type de donnée dans un message. Une instance de ce paquetage est créée pour chaque type de donnée transférable dans des messages. Dans un premier temps l'emballage des données dans les messages est effectuée d'une façon très simple (par copie mémoire). Dans le cas d'une application répartie, ceci requiert que les différents nœuds appartiennent à des plates-formes homogènes. Grâce à cet emballage, le temps d'exécution est $O(N)$ où N est la taille de la donnée à emballer ou à déballer ce qui donne une bonne prévision du pire temps d'exécution de ces routine lors des analyses.

Deux instances particulières de **PolyORB_HI.Marshallers_G** font partie de l'intergiciel minimal : une pour emballer des données temporelles (**PolyORB_HI.Time_Marshallers**) et une seconde pour emballer les énumérations correspondants aux ports sources ou destinations d'un message (**PolyORB_HI.Port_Type_Marshallers**). Elles sont utilisées dans toutes les applications réparties et il est inutile de les générer automatiquement à chaque fois.

PolyORB_HI.[...]_Task

Nous supportons trois types de tâches dans POLYORB-HI. Il s'agit uniquement des types de tâches qui respectent les recommandations des systèmes critiques (voie la section 4.9.4). Nous avons explicité le comportement de ces tâches dans la section 6.2.1. Pour chacun de ces trois types, un paquetage générique existe et est instancié autant de fois qu'il y a de tâches de ce type. Parmi les propriétés des tâches qui servent à instancier un de ces paquetages nous trouvons :

- Le type énuméré correspondant aux différents ports de la tâche (généré automatiquement),
- L'énumérateur qui représente la tâche à instancier qui est généré automatiquement.
- La période de la tâche (pour les processus légers périodiques et hybrides) ou le temps minimal entre deux déclenchements (pour les tâches sporadiques) (déduite du modèle),
- L'échéance de la tâche (déduite du modèle),
- La priorité de la tâche (déduite du modèle),

- La taille de la pile de la tâche (déduite du modèle),
- Pour les tâches sporadiques et hybrides, un sous programme qui effectue l'attente d'un événement extérieur. Un appel à ce sous-programme bloque la tâche jusqu'à l'arrivée d'un événement et retourne le port qui l'a reçu (généré automatiquement),
- Le sous-programme qui effectue le travail cyclique de la tâche. Pour les tâches périodiques, il s'agit d'un sous-programme à signature vide. Pour les tâches sporadiques et hybrides il prend en paramètre le port qui déclenchent la tâche. (généré automatiquement),
- Un éventuel sous-programme qui est appelé après l'initialisation de la tâche (fourni par l'utilisateur).

Ces composants génériques dépendent tous des composants suivants de l'intergiciel minimal :

- **PolyORB_HI.Suspenders**, pour garantir l'activation de toutes les tâches au même instant, et surtout, après la fin de la phase d'initialisation au début de l'exécution de l'application répartie.
- **PolyORB_HI.Output**, pour avoir une trace d'exécution de la tâche pendant les premières phases de développement.

Ils dépendent aussi du composant **Deployment** qui est généré automatiquement. Ce composant contient la représentation de la topologie de l'application et permet de fournir les identificateurs de chaque processus léger.

Toutes les instances de tâches sont créées dans le composant **Activity** qui est produit automatiquement.

PolyORB_HI.Hybrid_Task_Driver

Comme nous l'avons expliqué dans la section 6.2.1, lorsqu'un nœud contient au moins une tâche hybride, une tâche supplémentaire est ajouté. Elle a pour rôle de réveiller les tâches hybrides lorsque leur période survient. Cette tâche-gestionnaire est implantée dans le paquetage **PolyORB_HI.Hybrid_Task_Driver**. Elle est paramétrée par les éléments suivants :

- L'ensemble des tâches hybrides du système. Il s'agit d'un tableau contenant des structures de donnée indiquant pour chaque thread hybride, son identifiant, sa période ainsi que le port événement sur lequel il reçoit la notification périodique,
- La priorité, égale au maximum des priorités des tâches hybrides du nœud en cours,
- La taille de pile,
- La procédure **Deliver** de la couche haute transport qui permet de délivrer directement les événements périodiques aux tâches hybrides. Nous sommes sûrs que la communication entre le gestionnaire et les tâches hybrides est locale.

L'instance de ce composant est créée dans le composant **Activity**.

PolyORB_HI.Protocols

Ce paquetage permet d'effectuer l'envoi des messages d'une entité de l'application vers une autre. Il fournit un sous-programme, *Send*, qui prend comme paramètres l'entité expéditrice, l'entité réceptrice (fournis par le composant **Deployment**) ainsi que le message à envoyer (**PolyORB_HI.Messages**).

PolyORB_HI.Suspenders

Ce paquetage offre un ensemble d'objets Ada appelés des *Points de Suspension*. Nous les utilisons pour suspendre toutes les tâches d'une application répartie en attendant que tous les composants intergiciels soient proprement initialisés (canaux de communication ouverts...). L'attente sur ces points de suspension est implantée dans les paquetages génériques décrivant les types de tâches. Ce paquetage utilise le nombre total de tâche dans un nœud donné (information fournie par le composant **Deployment**). Il fournit aussi une routine qui permet de débloquer toutes les tâches à la fin de l'initialisation. Cette routine est appelé à la fin de l'initialisation de tous les nœuds d'une application répartie. Ainsi, les tâches commencent leur exécution après la fin du régime transitoire de l'application répartie.

Ce paquetage fournit une routine supplémentaire qui bloque indéfiniment la tâche d'environnement du nœud en cours. En effet, cette tâche sert uniquement à effectuer l'initialisation du nœud. Cette tâche ne peut être supprimée parce que :

- Le profil Ravenscar interdit la destruction de tâches,
- La plupart des plates-formes considèrent cette tâche comme le *parent* de tous les autres tâches du nœud. Sa destruction induirait la destruction de toutes ses filles.

PolyORB_HI.Transport_Low_Level_[...]

Chaque basse couche de transport est implantée dans un paquetage indépendant. Par exemple les transports basés sur les sockets BSD se trouvent dans le paquetage **PolyORB_HI.Transport_Low_Level_Sockets**. Chaque nœud de l'application répartie inclue les basses couches de transports qui sont utilisées pour ses communications. Lors de l'envoi d'un message, la couche haute de transport qui est générée automatiquement gère la sélection de la couche basse correspondante en fonction de la destination du message. Les envois des messages sont effectués de façon asynchrone et déterministe (en $O(N)$ au pire cas).

Il existent des situations particulières où des couches basses de transport requièrent l'inclusion d'entités supplémentaires. Par exemples, la couche de transport basé sur SpaceWire [ESTEC, 2003] a besoin d'une tâche "démon" pour gérer l'envoi des messages. Cette tâche n'est donc nécessaire que sur les nœuds qui envoient des messages à d'autre nœuds. Pour optimiser au maximum l'empreinte mémoire d'un nœud, nous mettons les composants supplémentaire d'une couches basse de transports dans un paquetage séparé qui ne sera inclus que pour les nœuds qui en ont besoin. Ces fonctionnalités supplémentaires sont implantées séparément dans le paquetage **PolyORB_HI.Transport_Low_Level_Extra_[...]**.

6.3.2 PolyORB-HI C

La version écrite en C de PolyORB-HI a été développée après la version en Ada dans le cadre d'un stage de Master que nous avons encadré [Delange, 2007]. Elle a été très fortement influencée par celle-ci. En particulier, la découpe des composants de l'intergiciel dans la version C est très similaire à celle de la version Ada.

Cependant, les choix de conception que nous avons pris dans cette version ne sont pas tous similaires aux choix adoptés pour implanter la version Ada. Ceci provient des différences fondamentales entre les deux langages de programmation. Dans cette section nous exposons ces différences.

Absence du profil Ravenscar

Contrairement au langage Ada, il n'existe pas de profil de concurrence qui permet d'utiliser le langage C pour les systèmes TR²E. En effet le langage C ne supporte pas la concurrence d'une façon intrinsèque. Il utilise des interfaces de programmation (comme POSIX) pour créer les tâches et pour protéger les accès aux données.

Pour avoir du code source C analysable statiquement, il ne suffit pas de restreindre le langage à un sous ensemble. Il faut aussi restreindre l'usage de l'interface POSIX ou tout autre moyen utilisé pour gérer la concurrence. Pour ce faire nous nous sommes inspirés des recommandations du profil Ravenscar pour Ada [Burns *et al.*, 2004] afin de définir un profil semblable pour le langage C. Au lieu d'utiliser directement les routines de la norme POSIX, l'intergiciel minimal expose une interface logicielle qui masque ces routines et force l'utilisation de constructions d'entités conformes aux recommandations de Ravenscar.

Par exemple, la création d'une tâche n'utilise pas la fonction **pthread_create**. Nous utilisons des patrons de conception qui nous permettent de créer uniquement des tâches périodique ou sporadique comme recommandé par Ravenscar. Seuls ces patrons de conception sont utilisés pour créer des tâches que ce soit dans les composants de l'intergiciel minimal ou ceux produits automatiquement.

Dans la version Ada, un code non conforme au profil Ravenscar ne compile pas. Les erreurs dans l'intergiciel minimal ou bien dans les composants produits sont détectés automatiquement. Dans la version en C, le compilateur ne vérifie pas la conformité du code avec le profil défini. Dans le cas C, seuls l'effort de conception et de relecture du code ou l'utilisation d'outils tiers comme ASTRÉE [Cousot *et al.*, 2005] garantissent la conformité de l'intergiciel minimal et des composants générés automatiquement aux recommandations du profil Ravenscar.

Absence des entités génériques

Contrairement au langage Ada, le langage C ne dispose pas de constructions génériques. Les archétypes des différents types de tâches que nous supportons ainsi que les archétypes des interrogateurs sont donc implantés d'une manière différente.

Pour instancier une tâche périodique, nous faisons appel à une routine de l'intergiciel minimal qui initialise les différentes données nécessaires au bon fonctionnement de la tâche (période, priorité...), et crée la tâche en appelant les outils de l'interface POSIX.

Pour garantir le respect des recommandations du profil Ravenscar, aucune des routines de création de tâches ou de création d'interrogateurs n'est invoquée après l'initialisation de l'application répartie. Il s'agit des routines de l'interface dont nous avons parlées dans le paragraphe précédent.

Absence des objets protégés

Les objets protégés constituent un moyen pour échanger les données entre les tâches en Ada. Ils permettent d'accéder aux données en lecture et en écritures tout en protégeant contre les accès concurrents. Les objets protégés permettent aussi de synchroniser les tâches Ada.

En langage C, les objets protégés n'existent pas. Nous les remplaçons par l'utilisation du quadruplet suivant :

1. Une structure de donnée représentant les données à partager (les champs d'un objets protégé),
2. Un verrou logiciel pour prévenir les accès concurrents,

3. Une variable conditionnelle et un verrou pour émuler le fonctionnement des entrées des objets protégés,
4. Un ensemble de sous programmes pour accéder aux données d'une façon similaire aux méthodes d'un objet protégé (encapsulation et protection des variables).

Ce quadruplet s'inspire de la phase d'expansion du compilateur Ada GNAT pour les objets protégés [Miranda and Schonberg, 2004].

Conclusion

Malgré les manques dont souffre le langage C en terme de constructions et les manques dont souffre le compilateur (**GCC**) en terme de restrictions, nous somme parvenus à avoir une version de l'intergiciel minimal en langage C. Cette version reprend l'architecture de la version écrite en Ada.

La conception de cette version a demandé un travail supplémentaire parce que le compilateur C que nous utilisons ne permet pas de vérifier si les programmes sont écrits conformément aux recommandations du profil Ravenscar et à celles des systèmes critiques.

6.4 Gestion locale de la communication

Comme nous l'avons expliqué dans la section 5.3.3 page 103, le standard AADL définit un ensemble de routines qui permettent la gestion locale de la communication. Ces routines permettent d'accéder en lecture ou en écriture aux interfaces des tâches pour envoyer ou recevoir des données et des événements.

Dans cette partie nous expliquons les choix de conception que nous avons adoptés pour implanter ces routines. Nous présentons les contraintes imposées par les recommandations du profil Ravenscar qui ont fortement influencé ces choix. Ensuite nous décrivons en détail le mécanisme interne sur lequel repose l'implantation de ces routines. Enfin, nous montrons pour chacune de ces routines, l'absence d'interblocage et le déterminisme des actions.

6.4.1 Contraintes du profil Ravenscar

L'implantation des routines d'interrogation est la réalisation d'un mécanisme de communication entre tâches. En effet dans le cas d'un envoi local de messages, la communication est entre la tâche expéditeur est la tâche destinataire. Dans le cas d'un envoi réparti de messages, la communication est, sur le nœud destination, entre le tâche appartenant à la couche transport (qui gère la réception des messages depuis le réseau) et la tâche destination du message.

Les recommandations du profil Ravenscar limitent fortement les communications entre les tâches. Ces dernières sont restreintes uniquement aux objets protégées. En Ada, les rendez-vous entre tâches sont interdits par le profil Ravenscar. Seul les objets protégés à une seule entrée sont autorisés.

De plus, il est interdit pour une tâche de se bloquer en attente de plusieurs sources d'événements simultanément. Pour les tâches AADL contenant plus d'un port événement en entrée, il faut trouver concept pour permettre d'effectuer une seule attente pour recevoir plusieurs événements.

Ceci nous mène à une architecture basée sur une table constituée d'une concaténation de tableaux circulaires. Chacun de ces tableaux représente la file d'attente d'un port pour une tâche donné. Nous décrirons cette architecture dans la prochaine section.

6.4.2 Architecture interne

La recommandation du profil Ravenscar la plus contraignante pour les interrogateurs est l'interdiction d'attentes simultanées pour les tâches. En particulier, une tâche AADL qui possède plusieurs ports événements en entrée (ayant chacun sa propre file d'attente) doit effectuer une seule attente. Si une des files d'attente n'est pas vide (ceci veut dire qu'une autre tâche vient d'y déposer un nouveau message), le message nouvellement arrivé doit être récupéré de la file d'attente correspondante.

Nous réalisons ce comportement en utilisant une file d'attente unique pour tous les messages reçus par la tâche. Cette file d'attente est la concaténation de plusieurs tableaux circulaires représentant chacun la file spécifique à une port en entrée particulier. Ainsi chaque tâche n'effectue pas d'attente simultanée. Les messages reçus sont décodés pour connaître leur destination à l'intérieur de la tâche. La concaténation de plusieurs files d'attentes permet, contrairement à l'utilisation de plusieurs files d'attentes séparées, de renforcer le caractère statique de l'application tout en économisant la quantité de mémoire occupée par ces files (voir plus loin).

Pour garder un ordre chronologique global sur les messages de type événement reçus, nous utilisons un second tableau circulaire qui contient les ports ayant reçu un nouveau message.

Tableaux circulaires

La partie (a) de la figure 6.3 décrit la structure du tableau global. Il s'agit d'une concaténation de tableaux circulaires correspondant chacun à la file d'attente d'un des ports en entrée de la tâche. Pour chaque tableau circulaire, deux indices (Deb_n et Fin_n) indiquent le début et la fin des données valides dans ce tableau. Ces deux indices sont mis à jour chaque fois qu'un nouveau message est déposé (resp. consommé) dans (resp. de) la file d'attente.

La partie (b) de la figure 6.3 met la lumière sur un de ces tableau circulaires. Les ports de type événement (b_1) doivent avoir une file d'attente dont la longueur est spécifiée par l'utilisateur. Elle contient les messages reçus dans un ordre chronologique ou selon une autre politique supportée par le standard (priorité des messages...). Par conséquent la taille du tableau circulaire pour ces port est égale à la taille de leur file d'attente spécifiée dans le modèle AADL. Les ports de type donnée ne contiennent pas de file d'attente. Une nouvelle valeur qui arrive écrase la valeur précédente. Pour les ports de type donnée connectés par une connexion AADL retardée, un nouveau message n'est "délivrable" qu'après l'échéance de la tâche qui l'a envoyé. Avant l'arrivée de cette échéance l'ancienne valeur est délivré. Par conséquent, nous utilisons des singletons pour les ports de type donnée (b_3) connectés par des connexions AADL immédiate et des "paires" pour ceux connectés par des connexions retardées (b_2).

La partie (c) de la figure 6.3 décrit la structure d'un élément du tableau. Il s'agit d'une structure de données contenant l'expéditeur du message ainsi que le message sous la forme d'une donnée de type *interface de la tâche* produit automatiquement à partir de la tâche AADL. L'interface de la tâche est aussi 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 (voir la section 5.3.3).

L'équation (6.1) illustre la formule qui donne la taille du tableau pour une tâche T, notée $File\_Globale_T$. Il s'agit de la somme des tailles de toutes les files d'attentes des ports comme nous les avons explicitées plus haut. $InEventPort_T$ désigne l'ensemble des ports en entrée de type événement pur. $InEventDataP_T$ est l'ensemble ports en entrée de type événement donnée. $InDataP\_Imm_T$ et $InDataP\_Delay_T$ désignent respectivement les ensembles de ports

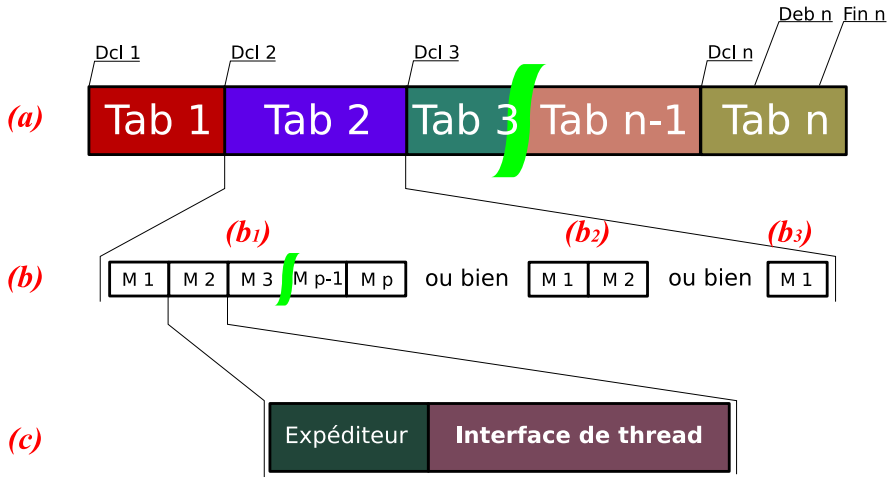

FIGURE 6.3 – Structure des tableaux circulaires

en entrée de type donnée pure connectés par des connexions immédiates et des connexions retardées.

$$\begin{aligned} File\_Globale_T = & \sum_{p \in InEventP_T \bigcup InEventDataP_T} Taille\_File(p) \\ & + Card(InDataP\_Imm_T) \\ & + 2 \times Card(InDataP\_Delay_T) \end{aligned} \tag{6.1}$$

L'accès à l'élément à la tête d'un tableau circulaire donné s'effectue grâce au deux indices qui délimitent les données valides dans le tableau ainsi qu'au décalage correspondant au tableau du port numéro n et dû à la concaténation de tous les tableaux circulaire (Dcl_n) comme expliqué par l'équation (6.2).

$$Tete\_File_n = Deb_n + Dcl_n - 1 \tag{6.2}$$

Pour un port en entrée de type événement ou événement donnée **P**, le décalage est égal à un plus la somme des tailles des files d'attente de tous les ports en entrée déclarés avant **P** dans la tâche comme précisé par l'équation (6.3). Cette information est déduite par à l'analyse du modèle AADL de la tâche. Elle sert à la génération statique de tous les décalages de tous les ports.

$$\begin{aligned} Dcl_1 &= 1 \\ Dcl_2 &= 1 + Taille\_File_1 \\ Dcl_3 &= 1 + Taille\_File_1 + Taille\_File_2 \\ &\dots \\ Dcl_n &= 1 + \sum_{p \leq n} Dcl_p \end{aligned} \tag{6.3}$$

De cette manière, nous conservons, pour chaque port de type événement en entrée, un ordre chronologique des messages reçus dans la file d'attente de ce port. Cependant cela ne

suffit pas pour avoir un ordre global sur les messages reçus sur plusieurs ports. Pour cela nous avons besoin d'un second tableau pour garder un historique des événements reçus.

Historique des événements reçus

Le paragraphe précédant a montré le mécanisme utilisé pour ordonner séparément les messages reçus sur chaque port de type événement ou événement donnée d'une tâche. La tâche effectue une seule attente pour recevoir les événements sur tous ses ports. Cet ordre partiel pour chacune des files des ports ne suffit pas pour savoir si un message dans la file d'attente d'un port P_1 est arrivé avant un message dans la file d'attente d'un port P_2. Nous avons besoin d'un ordre global pour délivrer à l'issue d'une attente le message le plus ancien [6].

Nous utilisons un tableau circulaire dont la taille est égale à la somme des tailles de toutes les files d'attentes des ports de types événement [donnée] en entrée. Ce tableau contient, dans l'ordre chronologique de réception, les identificateurs des ports ayant reçus des événements.

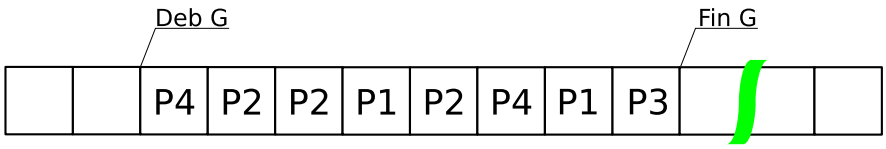

FIGURE 6.4 – Structure du tableau d'historique

La figure 6.4 illustre la structure du tableau d'historique. Comme nous pouvons le voir, un identificateur d'un port particulier apparaît éventuellement plusieurs fois dans le tableau. Le nombre maximal d'occurrences d'un port donné dans ce tableau est égal à la taille de la file d'attente de ce port. Ceci correspond à la situation ou la file d'attente est pleine et aucun événement n'a encore été consommé par la tâche. Le tableau donné dans la figure 6.4 correspond au cas d'une tâche possédant au moins 4 ports en entrée de type événements ou événement donnée. Cette tâche reçoit les messages suivant de la part d'autres tâches plus prioritaires (ce qui explique la non consommation de ces messages) :

1. Un message sur le port P_4, ensuite
2. Un message sur le port P_2, ensuite
3. Un second message sur le port P_2, ensuite
4. Un message sur le port P_1, ensuite
5. Un troisième message sur le port P_2, ensuite
6. Un second message sur le port P_4, ensuite
7. Un second message sur le port P_1, enfin
8. Un message sur le port P_3.

Le tableau d'historique est accompagné d'un couple d'index (Deb_H et Fin_H) qui pointent respectivement sur le port ayant reçus le plus ancien message et sur celui ayant reçu le plus récent. Ces deux index sont mis à jour à chaque ajout d'un nouvel événement et à chaque consommation d'événement.

6. Nous supposons ici que l'ordre utilisé est l'ordre chronologique. Le même problème et la même solution sont utilisés pour d'autres politiques de réception des messages.

6.4.3 Déterminisme et absence d'interblocage

Dans cette partie, nous montrons que les actions d'interrogation des interfaces des tâches sont réalisées soit en un temps constant soit en un temps proportionnel à la taille de la donnée manipulée. Ensuite nous montrons qu'il ne peut y avoir d'interblocage entre les tâches qui manipulent ces interrogateurs.

Déterminisme

Plusieurs opérations de bas niveau sont offertes par les interrogateurs pour manipuler les interfaces de tâches. Il s'agit notamment de routines pour :

- Lire un message de la file d'attente d'un port en entrée en le consommant (**Dequeue**) ou non (**Read_In**),
- Lire la valeur d'un port en sortie afin de l'envoyer (**Read_Out**),
- Stocker un message nouvellement reçu dans la file d'attente d'un port en entrée particulier (**Store_In**) et mettre à jour les différents indexes et les barrières qui débloquent les tâches en attentes,
- Stocker un message pour un envoi vers un port de sortie ((**Store_Out**),
- Connaître le nombre de messages en suspens dans la file d'un port en entrée (**Count**),
- Lire et écrire la valeur la plus récente d'un port (**Get_Most_Recent_Value**, **Set_Most_Recent_Value**).

Pour chacune de ces routines nous présentons l'algorithme de son fonctionnement et nous explicitons sa complexité et les structures de données nécessaires.

Il faut noter aussi que toutes ces routines sont protégées contre la concurrence. Cependant, nous utilisons le protocole de plafonnement de priorité (PCP, *Priority Ceiling Protocol* [Sha *et al.*, 1990]). L'utilisation de ce protocole garantit un temps de blocage borné pour les tâches et n'affecte donc pas le déterminisme.

Dequeue et Read_In L'algorithme 6.4 décrit le fonctionnement de la routine **Dequeue**. Les deux premières conditions concernent les ports dont la file d'attente est vide et ceux dont la file d'attente est de longueur nulle (spécifié par l'utilisateur). Dans ces cas, nous retournons le dernier message reçus sur ce port en utilisant la routine **Get_Most_Recent_Value** qui agit en un temps proportionnel à la taille du message comme nous le verrons plus bas dans sa description. Dans le dernier cas (file d'attente non vide), la routine retourne l'élément le plus ancien qui existe dans la file d'attente du port comme indiqué par l'équation (6.2). Ensuite, les indexes de la file sont mis ajour pour pointer vers le prochain élément reçu. Dans le cas où la file d'attente contient un seul élément ($Deb_P = Fin_P$), la file est marquée comme vide ($Empty_P = True$) et une variable indiquant le nombre de files d'attente vides est incrémentée. Cette variable ($N\_Empties$) est très utile pour mettre à jour la barrière globale qui indique qu'au moins une file d'attente des ports de la tâche n'est pas vide (Not_Empty). Ainsi nous ne parcourons pas toutes les files d'attentes des ports pour savoir si elles sont vides ou non.

À la différence de **Dequeue**, **Read_In** est une routine de lecture seule. Il n'y a pas de mise à jour des indexes ni des barrières. Nous déduisons facilement de l'algorithme 6.4 que les routines **Dequeue** et **Read_In** s'exécutent en un temps proportionnel à la taille de la donnée. La valeur maximale de cette taille est déduite du modèle AADL lors de son analyse et permet de borner précisément le pire temps d'exécution de ces routines.

Algorithme 6.4 La routine **Dequeue**

```
DEQUEUE (P : Port_Type)
 1  if Empty_P then
 2    return GET_MOST_RECENT_VALUE (P)
 3  else FIFO_Size_P = 0 then
 4    return GET_MOST_RECENT_VALUE (P)
 5  else
 6    if Deb_P = Fin_P then
 7      if IS_EVENT (P) then
 8        N_Empties ← N_Empties + 1
 9      Empty_P ← True
10    Result ← Global_Queue(Deb_P + Dcl_P − 1)
11    if Deb_P = FIFO_Size_P then
12      Deb_P ← 1
13    else
14      Deb_P ← Deb_P + 1
15    INCREMENT_HISTORY_FIRST (Deb_H)
16    Not_Empty ← N_Empties < N_Ports
17    return Result
```

Read_Out L'algorithme 6.5 montre le comportement de la routine **Read_Out**. Elle accède directement à une table indexée par les ports de la tâche et dans la quelle la routine **Store_Out** a déposé la valeur à envoyer. Pour économiser de la mémoire, cette table est la même que celle utilisée par la routine **Get_Most_Recent_Value** afin de déterminer la dernière valeur reçue pour un port en entrée. En effet, les positions dans cette table qui correspondent aux ports en entrée sont manipulées par **Get_Most_Recent_Value** et **Set_Most_Recent_Value** tandis que les positions correspondantes aux ports en sortie le sont par **Read_Out** et **Store_Out**.

Algorithme 6.5 La routine **Read_Out**

```
READ_OUT (P : Port_Type)
1  return Most_Recent_Value_Table_P
2  Value_Put_P ← False
```

La routine **Read_Out** marque le message lu comme "invalide". Cette action évite la réplication d'envois de message lorsqu'une nouvelle valeur n'a pas été déposée et de respecter ainsi la sémantique spécifiée par le standard AADL pour le routine standard **Send_Output** qui utilise cette fonction. Une table de booléens indexée par les ports de la tâche a été introduite ($Value\_Put$) ce qui permet une lecture et écriture en temps constant.

Nous constatons donc que le temps d'exécution de la routine **Read_Out** est lui aussi proportionnel à la taille du message lu.

Store_In La routine **Store_In** est utilisée pour déposer un message dans la file d'attente d'un port. Dans le cas d'un message provenant d'un autre nœud de l'application répartie, cette routine est appelée par la tâche de la couche transport qui gère la réception des messages. Sinon, cette routine est directement appelée par la tâche expéditeur du message.

L'algorithme 6.6 décrit le comportement de cette routine. Pour les ports en entrée de type donnée le comportement de cette routine se résume à l'invocation de **Set_Most__-Recent_Value** qui agit en un temps constant comme nous le verrons ci-après. Dans le cas d'un port en entrée de type événement ou événement-donnée, la routine met à jour les différents indexes et

Algorithme 6.6 La routine **Store_In**

```
STORE_IN (Ti : Thread_Interface; Ts : Time)
 1  P ← Ti.Port
 2  if not IS_EVENT (P) then
 3    SET_MOST_RECENT_VALUE (P, Ti, Ts)
 4  else
 5    if Is_Empty_P then
 6      N_Empties ← N_Empties − 1
 7      Is_Empty_P ← False
 8    if Fin_P = FIFO_Size_P then
 9      Fin_P ← 1
10    else
11      Fin_P ← Fin_P + 1
12    Global_Queue(Fin_P + Dcl_P − 1) ← TiSET_MOST_RECENT_VALUE (P, Ti, Ts)
13    INCREMENT_HISTORY_LAST (Fin_H)
14    Global_History_Queue(Fin_H) ← P
15    Not_Empty ← True
```

barrières pour préparer la nouvelle position où sera posé le message. Toutes ces mises à jour sont effectuées en un temps constant comme le montre les instructions 5 à 11 de l'algorithme. Ensuite le message est déposé à la fin de la file d'attente (en un temps proportionnel à sa taille) et une copie est utilisée par la routine **Set_Most_Recent_Value** pour spécifier la valeur la plus récente pour le port P. Enfin, l'historique des événements reçus par la tâche est mis à jour en lui ajoutant le port qui vient de recevoir le dernier événement (en un temps constant aussi). La barrière globale est mise à jour pour libérer la tâche si elle est bloquée.

L'algorithme 6.6 ne montre pas le comportement effectué par la routine lorsqu'un un nouveau message est reçu alors que la file d'attente est pleine. Le standard AADL spécifie trois politiques d'action possibles devant une telle situation (propriété standard `Overflow_Handling_Protocol`) :

1. Suppression du message le plus ancien. L'implantation s'exécute en un temps proportionnel à la taille du message. En effet, la file étant pleine, il suffit de placer le nouveau message à la place du plus ancien (position Deb_P), d'incrémenter Deb_P et Fin_P à la position suivante d'une manière circulaire. Cette politique est implantée actuellement dans POLYORB-HI (`DropOldest`),
2. Suppression du message le plus récent. Cette politique s'exécute en un temps proportionnel à la taille du message. Il suffit d'écraser la valeur de Fin_P et de ne mettre à jour aucun des index(`DropNewest`) ,
3. Levée d'erreur (`Error`).

La routine **Store_In** s'exécute dans un temps proportionnel à la taille du message.

Store_Out La routine **Store_Out** est utilisée par la tâche elle-même pour déposer un message qui sera envoyé à la fin de l'exécution du travail de la tâche. L'algorithme 6.7 illustre le comportement de cette routine : il s'agit de déposer le message dans la table prévue à cet effet ($Most\_Recent\_Value\_Table$) à la position correspondant au port, de placer l'estampille temporelle dans une seconde table et de marquer le message pour qu'il soit envoyé. Ce comportement est réalisé en un temps proportionnel à la taille de la donnée.

Algorithme 6.7 La routine **Store_Out**

```
STORE_OUT (Ti : Thread_Interface; Ts : Time)
1  P ← Ti.Port
2  Value_Put_P ← True
3  Most_Recent_Value_Table_P ← Ti
4  Time_Stamps_P ← Ts
```

Count Pour déterminer le nombre de messages restant dans la file d'attente d'un port en entrée, nous utilisons la routine **Count**. Comme le montre l'algorithme 6.8, cette routine agit en un temps constant.

Algorithme 6.8 La routine **Count**

```
COUNT (P : Port_Type)
1  if Is_Empty_P then
2    return 0
3  else FIFO_Size_P = 0 then
4    return 0
5  else
6    if Fin_P ≥ Deb_P then
7      return Fin_P − Deb_P + 1
8    else
9      return FIFO_Size_P − Deb_P − Fin_P + 1
```

Get_Most_Recent_Value et Set_Most_Recent_Value Ces deux routines permettent de lire et d'écrire la valeur la plus récente reçue pour un port en entrée. Elles sont utilisées pour tous les types de ports (donnée, événement...) et permettent d'avoir la valeur attendue dans le cas des ports de type donnée connectés par des connexions AADL retardée.

Algorithme 6.9 La routine **Get_Most_Recent_Value**

```
GET_MOST_RECENT_VALUE (P : Port_Type)
 1  if IS_EVENT (P) then
 2    return Most_Recent_Value_Table_P
 3  else
 4    if FIFO_Size_P = 1 then
 5      return Global_Table(Fin_P + Dcl_P − 1)
 6    else
 7      if Time_Stamps_P ≤ SYSTEM_CLOCK then
 8        return Global_Table(Fin_P + Dcl_P − 1)
 9      else
10        return Global_Table(Deb_P + Dcl_P − 1)
```

L'algorithme 6.9 décrit le comportement de la routine **Get_Most_Recent_Value**. Pour les ports en entrée de type événement ou événement-donnée, cette routine retourne la valeur stockée dans la table $Most\_Recent\_Value\_Table$. Ceci est effectué en un temps proportionnel à la taille de la donnée. Pour les ports de type donnée connectés par une connexion AADL immédiate (donc qui ont une longueur de file d'attente conventionnelle qui est égale à 1) la routine retourne la valeur correspondante au port dans la table globale des messages reçus. Pour les ports de

type donnée connectés par une connexion AADL retardée (qui ont une longueur de file d'attente conventionnelle qui est égale à 2) la routine compare la valeur de l'estampille temporelle de la valeur la plus récente reçue sur le port avec le temps courant. Dans le cas où cette valeur est inférieure, nous concluons que la valeur est visible par le récepteur et la valeur la plus récente est retournée. Dans le cas ou l'estampille contient un instant *futur*, ceci veut dire que la valeur la plus récente n'est pas censée encore être vue par le récepteur. La valeur la plus ancienne est alors retournée. Tout ce comportement est effectué dans un temps proportionnel à la taille du message comme le montre l'algorithme.

Le comportement de la routine **Set_Most_Recent_Value** est très similaire à celui décrit dans l'algorithme 6.9 à la seule différence que des écritures sont effectuées à la place des lectures. Pour les connections retardées, la valeur la plus ancienne est écrasée par la valeur déposée. Le temps d'exécution est aussi proportionnel à la taille du message.

Absence d'interblocage

Une application TR^{2}E est dans une situation d'interblocage lorsqu'un sous-ensemble S de ses tâches vérifie la propriété suivante : *chacune des tâches de S est en attente d'un événement qu'une autre tâche de S, elle même bloquée, ne peut lui fournir*. Dans [Coffman *et al.*, 1971], les auteurs établissent quatre conditions nécessaires pour avoir une possibilité d'interblocage dans un système. Un système qui ne réunit pas toutes ces conditions ne peut avoir d'interblocage :

1. Les tâches demandent l'accès exclusif aux ressources. L'accès se fait donc en exclusion mutuelle en utilisant des verrous logiciels par exemple,
2. Il existe des tâches qui possèdent des ressources et qui attendent la possession d'autres ressources,
3. Seule une tâche possédant une ressource peut décider de libérer cette ressource,
4. Une attente cyclique entre les tâches existe. Chacune des tâches impliquées dans cette attente cyclique possède déjà une ou plusieurs ressources requises d'autres et est elle même en attente d'une ressource possédée par une autre tâche impliqué dans le même cycle.

Pour éviter l'interblocage, il est donc suffisant de garantir l'absence d'au moins une de ces conditions tout au long de la vie de l'application répartie. Il se trouve que dans notre situation les seules ressources auxquelles une tâche requiert d'accéder en exclusion mutuelle sont ses propres interrogateurs et aussi ceux des tâches auxquelles elle est directement connectée. Les recommandations du profil Ravenscar interdisent à un processus léger en possession d'une ressource en exclusion mutuelle de demander un accès potentiellement bloquant à une autre ressource. Ceci peut être vérifié dans les algorithmes des routines décris plutôt. De plus, à aucun moment à l'intérieur de ces routines d'interrogation, un autre routine correspondant aux interrogateurs d'une autre tâche n'est appelé.

Par conséquent les conditions 2 et 4 pour avoir un interblocage ne sont pas vérifiées ce qui élimine toute possibilité d'interblocage dans nos applications.

6.5 Synthèse

Dans ce chapitre, nous avons décrit l'architecture et l'implantation de notre intergiciel minimal pour les systèmes TR^{2}E, POLYORB-HI. Deux versions de cette intergiciel existent. La version

Ada respecte le profil Ravenscar ainsi que les restrictions du langage Ada pour les systèmes critiques. La seconde, écrite en C, reprend les mêmes principes que celle en Ada. Puisqu'il n'existe pas de profil similaire à Ravenscar pour le langage C, nous avons repris les recommandations de ce profil et fait en sorte de contrôler l'utilisation faite de la bibliothèque POSIX.

Nous avons ensuite décrit en détail l'architecture du composant central de PolyORB-HI : les interrogateurs. Ce mécanisme gère la communication entre les tâches de l'application d'une façon transparente à l'utilisateur et d'une manière conforme au standard AADL. Nous avons montré que toutes les routines de ce composant sont déterministes et qu'il ne peut y avoir d'interblocage entre les tâches.

Le chapitre suivant explique la manière dont les composants restants de l'intergiciel produits automatiquement viennent se greffer sur l'intergiciel minimal. Nous y présentons notre suite d'outils Ocarina qui permet d'analyser les modèles AADL afin de produire les composants applicatifs et intergiciels dédiés à l'application TR^{2}E. Ces composants sont paramétrés avec ceux de l'intergiciel minimal et intégrés pour produire une application prête à être exécutée.

Chapitre 7

Génération, Déploiement et Configuration Automatiques de Systèmes TR²E

SOMMAIRE

7.1 Introduction

Dans le chapitre précédent, nous avons présenté notre l'intergiciel minimal pour les systèmes TR²E. Il s'agit de l'implantation de l'architecture intergicielle que nous avons définie dans le chapitre 3. Cet intergiciel rassemble les composants faiblement personnalisables. Le reste des composants sont générés automatiquement à partir du modèle AADL selon le processus de production défini dans le chapitre 5.

Dans ce chapitre, nous présentons la suite d'outils OCARINA [Vergnaud *et al.*, 2006] qui supporte ce processus. OCARINA est un logiciel libre écrit en Ada 2005 et distribué selon les termes de la licence GPLv2 modifiée pour le compilateur GNAT. Il permet de générer, déployer et configurer automatiquement les applications TR²E à partir de leur modèles AADL. Nous utilisons OCARINA pour analyser les modèles AADL afin de valider leur bon fonctionnement mais aussi afin de déterminer les ressources nécessaires à chaque nœud de l'application. Ensuite, nous

générons automatiquement le code conforme aux recommandations pour les systèmes TR²E présentées dans la section 2.5.1. Enfin nous intégrons tous les composants de l'application pour produire un ensemble de nœuds prêts à être exécutés.

OCARINA constitue le pilier principal pour le processus de production que nous proposons d'implanter. En effet, hormis les composants applicatifs qui servent à connecter le code de l'utilisateur à l'application, la majeure partie des composants intergiciels sont générés automatiquement par OCARINA. Grâce à l'analyse préalable du modèle AADL de l'application, ces composants sont optimisés aux besoins de l'application. Chacun des nœuds possède une instance de l'intergiciel qui est *dédiée* à ses besoins.

Tout d'abord, nous présentons l'architecture détaillée d'OCARINA (section 7.2). Ensuite nous décrivons nos principales contributions à cet outil : la génération automatique de code pour les différentes instances de l'intergiciel POLYORB-HI (section 7.3), le déploiement et la configuration automatiques des composants intergiciels (section 7.4) et l'implantation du processus de production automatique en intégrant les composants générés par OCARINA avec le reste des constituants d'une application TR²E (section 7.5). La section 7.6 conclut ce chapitre.

7.2 Ocarina : Architecture détaillée

OCARINA est un projet débuté en 2003 au sein de l'équipe S3 de TELECOM ParisTech où nous effectuons notre travail de thèse. Il s'agit d'une suite de bibliothèques logicielles pour manipuler les modèles AADL et effectuer plusieurs analyses et vérifications. OCARINA permet aussi de générer automatiquement du code à partir de ces modèles vers plusieurs supports d'exécution.

L'architecture d'OCARINA est très similaire à l'architecture des compilateurs modernes [Wilhelm and Maurer, 1995]. Comme le montre la figure 7.1 OCARINA est formée de trois bibliothèques principales :

1. une bibliothèque centrale offre les différentes routines pour construire et parcourir les arbres syntaxiques. Il s'agit particulièrement d'abstraction de bas niveau pour manipuler les fichiers, les nœuds des arbres syntaxiques et les chaînes de caractères (sous forme de code de hachage pour augmenter les performances),

2. un ensemble de parties frontales qui utilisent les routines de la bibliothèque centrale pour analyser la syntaxe et la sémantique de fichiers sources donnés dans le langage qu'elles prennent en charge. Pour l'instant, seule la partie frontale permettant d'analyser les fichiers sources AADL existe,

3. un ensemble de parties dorsales qui prennent comme entrée les arbres produits par la partie frontale. Leur rôle vise soit à produire automatiquement le code soit à vérifier le modèle. Au début de notre travail de thèse, une seule partie dorsale existait dans OCARINA. Elle implantait une partie des propositions de génération automatique de code Ada pour l'intergiciel schizophrène POLYORB issus des travaux de thèse de Thomas VERGNAUD [Vergnaud, 2006].

Il faut noter toutefois quelques différences entre OCARINA et les autres compilateurs (comme **GCC**), en particulier :

1. La motivation principale d'avoir plusieurs parties frontales n'est pas le support de multiples formalismes d'entrée (au même niveau que AADL). Le formalisme principal de la partie

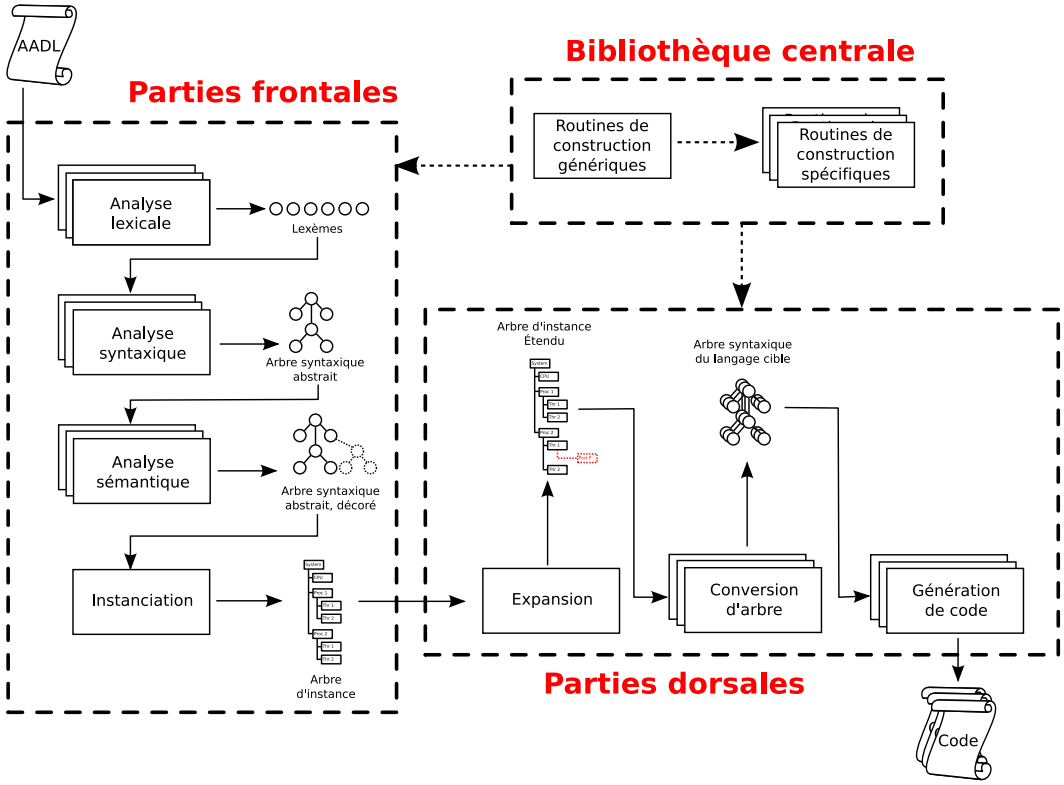

FIGURE 7.1 – Architecture globale d'OCARINA

frontale est toujours AADL. Cependant, le support de quelques annexes dans le futur (notamment l'annexe comportementale [SAE, 2008]) requiert le support de la syntaxe de ces annexes. Ce support sera implanté dans des parties frontales secondaires d'OCARINA,

2. OCARINA étant aussi un outil de déploiement et de configuration, les parties frontales n'effectuent pas uniquement la tâche de transformer le modèle AADL en un formalisme cible (code généré par exemple). Elles réalisent aussi la sélection automatique des composants de l'intergiciel minimal nécessaires pour chaque nœuds et l'intégration automatique des tous les composant intergiciels et applicatifs pour construire l'application finale. Dans ce sens, OCARINA peut être comparé à l'outil **GNATDIST** permettant la conception et le déploiement d'applications réparties selon le standard Ada [Working Group, 2005, Annexe E],
3. La sélection de la partie dorsale ne se fait pas comme pour les compilateurs classiques (une instance de compilateur supporte une seule partie dorsale). Ce sont les propriétés contenues dans le modèle AADL et relatives au déploiement de l'application qui pilotent le choix de la partie dorsale.

Dans la suite, nous décrivons chacun des trois constituants principaux d'OCARINA. Nous précisons aussi les contributions que nous avons implantées sur chaque constituant durant notre travail de thèse. Enfin nous donnons les bénéfices cette architecture.

7.2.1 Bibliothèque centrale

Pour que les parties frontales et les parties dorsales d'un compilateur puissent s'échanger des arbres syntaxiques, il faut qu'elles aient accès aux routines permettant de construire les

nœuds de cet arbre (du coté de la partie frontale) et de parcourir et lire ces même nœuds (du côté de la partie dorsale). La bibliothèque centrale d'OCARINA (**libocarina.a**) rassemble un ensemble de routines permettant de manipuler l'arbre d'un formalisme quelconque. Cette bibliothèque est complétée par un ensemble de routines spécifiques à chacun des formalismes supportés (AADL, C, Ada...). Ces dernières routines de manipulation font partie de la bibliothèque centrale et non pas de la partie frontale ou dorsale. En effet :

- un formalisme comme Ada est supporté dans plusieurs parties dorsales différentes (voire la section 7.2.3). Il est donc nécessaire que ces routines soient situées dans un niveau supérieur à celui des parties dorsales où elle sont utilisées,
- un formalisme comme AADL est supporté à la fois dans une partie frontale et une partie dorsale. Il est donc nécessaire que ces routines soient situés dans un niveau différent de celui des parties frontale et dorsale.

Pour ces raisons, nous avons choisi de mettre toutes les routines de manipulation d'arbres syntaxique dans la bibliothèque centrale d'OCARINA.

7.2.2 Partie frontale

Comme dans tout compilateur moderne, la partie frontale d'OCARINA effectue l'analyse syntaxique et sémantique des fichiers sources AADL et produit un *arbre syntaxique abstrait*. La figure 7.2 donne une vision globale de la partie frontale d'OCARINA :

1. un analyseur lexical transforme le fichier source AADL en une séquence d'éléments lexicaux appelés *lexèmes* (identificateur, séparateur, opérateur...),
2. un analyseur syntaxique pilote l'analyseur lexical et transforme la séquence de lexèmes en une structure arborescente appelée *arbre syntaxique abstrait*. Les erreurs de syntaxe sont détectées par cet analyseur,
3. un analyseur sémantique décore l'arbre syntaxique (ensemble d'opérations permettant de naviguer plus facilement). Par ailleurs, il permet la vérification de la sémantique du modèle,
4. l'instanciation (ne faisant pas partie des compilateurs classiques) consiste à produire un nouvel arbre syntaxique. Cet arbre résulte de l'application récursive, sur chaque composant, des propriétés et des sous-composants dont il hérite. La racine de cet arbre est un composant de type **system** comme nous l'avons expliqué dans la section 4.7.

Nous avons contribué à la partie frontale d'OCARINA en complétant le mécanisme d'instanciation pour supporter les données partagées ainsi que les modes opérationnels pour les composants de type **thread**.

Analyses lexicale, syntaxique et sémantique

Les analyseurs lexical et syntaxique vérifient que le modèle AADL est bien conforme à la grammaire donnée par le standard [SAE, 2004]. Ensuite l'analyseur sémantique vérifie la sémantique du modèle. Les analyses sémantiques varient de la simple analyse des règles de typage (compatibilité entre les extrémités des connexions, orientation des connexions...) jusqu'aux analyses plus complexes (évaluation des valeurs de constantes, vérification des intervalles des types numériques...).

Au terme de ces trois analyses, nous somme sûrs d'avoir un modèle "légal" du point de vue du langage AADL. Cependant, il est possible que ce modèle ne soit pas conforme à certaines règles avancées. Ceci sera le rôle de la phase d'instanciation.

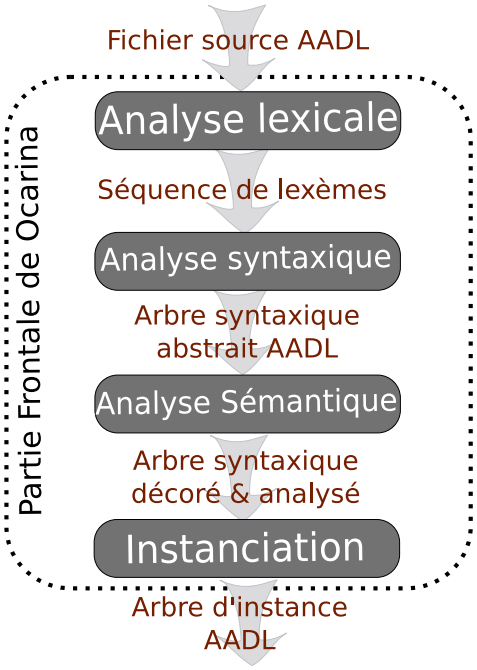

FIGURE 7.2 – Partie frontale de OCARINA

Instanciation

Comme requis par le standard AADL, un modèle doit être instancié. Un modèle d'instance est une structure arborescente représentant une vue hiérarchique du modèle qui décrit la topologie de l'application TR^{2}E. Le modèle d'instance est formé principalement d'un ensemble d'instances composants. Chaque instance correspond à un sous composant du modèle AADL original. Dans le cas où le sous-composant original correspond à une implantation qui contient elle-même des sous composants, l'instance contient des instances de composants correspondant sous-composants de cette implantation et ainsi de suite.

Cette phase d'instanciation permet, notamment, d'évaluer les valeurs correctes des propriétés d'une instance de composant. En effet, pour une instance particulière (sous-composant, appel à un sous programme...), la valeur d'une propriété donnée est déduite, dans un ordre décroissant de priorité :

1. à partir des associations de propriétés de l'instance elle-même,
2. à partir des associations de propriétés de l'implantation soit du composant qui correspond à l'instance, soit de l'un de ses parents si cette implantation est une extension,
3. à partir des associations de propriétés soit du type de composant correspondant à l'instance, soit de l'un de ses parents (de proche en proche) si ce type est une extension,
4. à partir de la valeur par défaut de la propriété,
5. la propriété est considérée comme indéfinie pour l'instance en cours.

Pendant la phase d'instanciation, il est possible de détecter des incohérences dans le modèle AADL. Par exemple, durant cette phase, nous renforçons le fait qu'un processus contienne au moins une tâche.

7.2.3 Parties dorsales

La partie dorsale prend comme entrée l'arbre syntaxique décoré (résultat de la partie frontale). Elle effectue l'expansion de cet arbre en transformant les constructions complexes en des constructions plus simples. Enfin l'arbre étendu est parcouru pour produire du code. La différence entre OCARINA et les compilateurs classiques réside dans les parties dorsales qui prennent en entrée l'arbre d'instance. Cet arbre est le résultat de la résolution des valeurs de propriétés et de connexions dans le modèle AADL (voir la section 4.7). Il se prête à l'analyse et à la génération de code plus simplement que l'arbre syntaxique AADL.

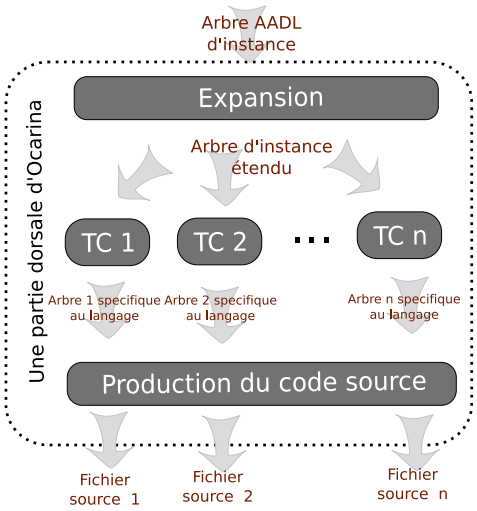

FIGURE 7.3 – Une partie dorsale d'OCARINA

Expansion

Après la phase d'instanciation, nous avons un arbre qui décrit la topologie de l'application TR²E. Mais avant de passer à la génération de code, certaines constructions de cet arbre doivent être simplifiées et d'autres doivent être ajoutées : il s'agit de l'expansion.

Après cette phase d'expansion, nous avons un arbre d'instance qui est plus simple et plus riche que l'arbre original. Par exemple, les instances de tâches hybrides se voient ajouter un port de type événement en entrée supplémentaire afin d'implanter leur communication avec le pilote de tâches hybrides (voir la section 6.2.1). Ce nouvel arbre permet de produire du code ou d'effectuer de nouvelles analyses.

Durant cette phase, plusieurs types d'erreur de haut niveau peuvent être détectées (types de données non supportés par la partie dorsale, types de tâches non supportés, connections manquantes...). En effet, ces analyses ne peuvent pas être effectuées dans les phases précédentes.

Nous avons réimplanté l'architecture des parties dorsale d'OCARINA pour qu'elles correspondent à la figure 7.3. Nous avons aussi ajouté les parties dorsales générant le code Ada pour l'intergiciel POLYORB-HI et supervisé le développement de celle générant le code C [Delange *et al.*, 2008].

Arbres intermédiaires

Dans ce paragraphe, nous décrivons la stratégie que nous avons adopté dans les parties dorsales pour générer du code à partir des modèles AADL. Deux objectifs principaux ont motivé la structure des partie dorsales actuelles d'OCARINA : (1) ces parties doivent être flexibles et facilement maintenables et (2) l'implantation d'optimisation (sur le code généré) doit être relativement simple. La structure des parties dorsales doit aussi prendre en compte que AADL, à la différence des langages de programmation classiques, produit plusieurs fichiers sources à partir d'une seule spécification AADL.

Pour atteindre ces objectifs, nous avons introduit dans OCARINA une nouvelle phase. Cette phase consiste à transformer l'arbre AADL d'instance en un arbre syntaxique du langage cible. Puis cet arbre est traversé pour produire du code source.

La figure 7.3 montre l'emplacement de cette nouvelle phase ($TC_1..TC_n$) dans une des parties dorsales d'OCARINA. Pour chaque langage de programmation supporté (Ada, C) un ensemble de transformations est ajouté. Elles convertissent l'arbre d'instance étendu en un ensemble d'arbres syntaxiques du langage de programmation choisi (un par fichier source généré). Cette transformation nous permet de gagner en flexibilité : nous pouvons ajouter des nœuds dans l'arbre syntaxique cible dans un ordre différent de celui de leur transformation en code source. Nous pouvons même supprimer ou écraser des nœuds de cet arbre. Cette manière de généré de code a aussi été testée avec succès lors de notre travail de MASTER [Zalila, 2005] pour concevoir un compilateur IDL optimisé pour l'intergiciel schizophrène POLYORB [Zalila *et al.*, 2006]. Nous utilisons cette approche au lieu de l'approche classique qui consiste à parcourir le modèle AADL et de produire le code "à la volée". En effet cette dernière approche oblige le parcours des modèles AADL dans le même ordre de la génération des entités du langage de programmation cible et rend le code des générateurs peu maintenable et non flexible.

Nous avons ainsi construit plusieurs parties dorsales pour produire du code à partir de modèles AADL :

- Une partie dorsale Ada vers l'intergiciel POLYORB,
- Une partie dorsale Ada vers l'intergiciel minimal POLYORB-HI-Ada,
- Une partie dorsale C pour l'intergiciel minimal POLYORB-HI-C,

D'autres parties dorsales (générateurs ou analyses) sont réalisés dans le cadre d'autres travaux de thèse :

- Une partie dorsale réseaux de Petri afin d'effectuer des vérifications formelles sur les modèles AADL,
- Une partie dorsale pour les systèmes critiques selon la norme ARINC 653.

Dans la suite, nous nous intéressons à la génération de code Ada et C pour l'intergiciel minimal POLYORB-HI.

7.2.4 Bénéfices de l'architecture

Cette architecture offre plusieurs avantages. Elle permet de piloter plus simplement le processus de production tout en ayant la possibilité d'étendre ce processus en ajoutant des parties dorsales ou en enrichissant les parties déjà existantes. L'organisation de OCARINA en bibliothèques permet aussi à d'autres outils d'utiliser les routines de manipulation des modèles AADL, notamment l'outil d'analyse d'ordonnancement CHEDDAR [Singhoff *et al.*, 2004]. Nous l'utilisons pour effectuer l'analyse statique d'ordonnancement sur les modèles AADL. Il utilise d'ailleurs les routines de la partie frontale et de la bibliothèque centrale pour transformer les modèles AADL en une représentation interne pour être analysée.

Dans la suite de ce chapitre, nous nous intéressons uniquement à la génération de code Ada et C pour l'intergiciel minimal PolyORB-HI.

7.3 Génération de code pour PolyORB-HI

La génération de code pour les systèmes TR²E repose sur des patrons de conception pour les systèmes critiques qui sont empruntés en partie à partir de travaux précédents [Bordin and Vardanega, 2005] sur la génération de code compatible avec le profil Ravenscar. Pour adapter ces patrons à la répartition, nous utilisons le concept de *Broker* [Buschmann *et al.*, 1996] que nous avons contraint en enlevant tous les comportements *dynamiques* incompatibles avec les exigences des systèmes TR²E.

7.3.1 Génération de code Ada

Le modèle AADL d'une application TR²E est analysé par Ocarina et la partie dorsale générant le code Ada pour l'intergiciel PolyORB-HI permet de produire, pour chaque nœud de l'application, deux familles de composants qui y sont dédiés : les composants applicatifs et les composants intergiciels. Chacun des composants générés est conforme au profil Ravenscar ainsi qu'à l'ensemble de restrictions pour les systèmes critiques présentées dans la section 6.3.1. Les règles de transformation qui permettent de créer ces composant ont été expliquées en détail dans les sections 5.3 et 5.4. Nous présentons dans cette section l'architecture du générateur de code Ada et précisons les fonctionnalités les plus pertinentes.

Composants applicatifs

Les composants applicatifs sont les composants destinés à lier le code fourni par l'utilisateur avec le code de l'intergiciel. Il s'agit des transformées des types de données présents dans le modèle AADL, des données partagées, des transformées des sous-programmes et des instances d'interrogateurs permettant d'accéder aux interfaces des tâches.

Types de données Toutes les transformées des types de données utilisées par un nœud donné sont rassemblées dans le paquetage **PolyORB_HI_Generated.Types**. Ceci permet une localisation simple de ces types quand ils sont utilisés dans le code de l'utilisateur. Le code 7.1 montre le paquetage **PolyORB_HI_Generated.Types** produit pour le nœud **Workload_Manager** de l'exemple décrit dans la section 4.9.

Exemple 7.1 – Paquetage **PolyORB_HI_Generated.Types**

```
package PolyORB_HI_Generated.Types is

   type Workload is new Standard.Integer;
   Workload_Default_Value : Workload := 0;

   type Interrupt_Counter is new Standard.Integer;
   Interrupt_Counter_Default_Value : Interrupt_Counter := 0;

end PolyORB_HI_Generated.Types;
```

Données partagées Les données partagées entre les tâches doivent être protégées contre les accès concurrents et ne doivent être accessibles que par l'intermédiaire des méthodes qu'elles exposent. Par conséquent, il ne faut pas les déclarer dans une spécification de paquetage Ada qui les rendraient visibles à tous les composants de l'application. D'autre part, les sous-programmes implantant le comportement des tâches ont besoin d'accéder à ces données. Ces sous programmes sont implantés dans le corps du paquetage **PolyORB_HI_Generated.Activity** ; d'où la nécessité de la présence de la déclaration de ces données partagées dans le corps du paquetage **PolyORB_HI_Generated.Activity**.

Sous-programmes Chacun des sous-programmes utilisé dans un nœud est implanté par un sous-programme Ada du paquetage **PolyORB_HI_Generated.Subprograms**. Ceci permet d'homogénéiser les appels effectués par les tâches vers les sous programmes AADL et de simplifier l'accès de l'utilisateur vers ces sous-programmes. Selon la nature du sous-programme AADL, l'implantation en Ada varie du simple renommage d'un sous programmes existant à l'implantation complète (dans le cas d'un sous-programmes appelant d'autres sous-programmes).

Instances de interrogateurs Ces instances implantent le service intergiciel d'interaction. Il s'agit d'un service fortement configurable qui instancie le service d'interrogation de l'intergiciel minimal. Les sections 3.3 et 3.4 définissent respectivement ces deux ensembles de services. La section 6.4 décrit l'implantation du service d'interrogation.

De la même manière que pour les données protégées, l'utilisation des sous-programmes implantant le service intergiciel d'interaction avec les éléments d'interface des tâches nous a poussés à les déclarer dans le corps du paquetage **PolyORB_HI_Generated.Activity**. Nous minimisons ainsi les dépendances entre les paquetages Ada formant le nœud. Les sous-programmes qui correspondent aux routines spécifiées par le standard AADL sont ensuite exportés dans la spécification du paquetage **PolyORB_HI_Generated.Activity** pour les rendre visibles du code de l'utilisateur.

Programme principal Il est nécessaire d'avoir un sous-programme principal dans la plupart des langages de programmation. Pour chaque nœud de l'application répartie, nous produisons un sous-programme principal qui inclut les composants restants nécessaires et effectue leur initialisation. Ensuite, le processus léger exécutant ce sous-programme se met en suspension indéfinie pour permettre au reste des processus léger de s'exécuter.

Composants intergiciels

Pour compléter l'intergiciel minimal, les composants restants des services intergiciels fortement personnalisables sont générés automatiquement. Ces composants sont dédiés aux caractéristiques du nœud en cours. Il s'agit des composants suivant :

- les instances de tâches. Pour chaque composants AADL de type **thread**, une instance du paquetage générique correspondant à sa catégorie est crée. Nous avons décrit l'implantation de ces paquetages dans la section 6.3.1. Toutes ces instances sont créées dans le paquetage **PolyORB_HI_Generated.Activity**,
- les routines d'emballage et de déballage avancées correspondant aux types de données présent dans le modèle AADL. Il s'agit de sous-programmes utilisant des instances du paquetage générique **PolyORB_HI.Marshallers_G** présenté dans la section 6.3.1. Ce paquetage offre les mécanismes de représentation élémentaire. Les sous-programmes

générés implante le service de représentation avancée. Ils appartiennent au paquetage **PolyORB_HI_Generated.Marshallers**,
- les tables statiques de nommage correspondant à chacune des couches de transport supportés. Ces tables sont produites dans le paquetage **PolyORB_HI_Generated.Naming**,
- la couche haute de transport dont les routines permettent de sélectionner correctement la couche basse de transport selon la source et la destination d'un message. Elle est produite dans le paquetage **PolyORB_HI_Generated.Transport**.

Instances de tâches Ces instances sont créées dans la spécification du paquetage **PolyORB_HI_Generated.Activity**. Les sous-programmes implantant le comportement d'une itération des tâches sont implantés dans le corps de ce paquetage. Si la tâches contient des modes opérationnels qui pilotent son comportement, ceux ci sont déclarés dans le corps du même paquetage. Ainsi, ils ne sont pas visibles de l'utilisateur et peuvent être utilisés et modifiés par le sous-programme implantant le travail de la tâches.

Déploiement Nous voulons que chaque nœud de l'application répartie possède toutes les informations nécessaires concernant la topologie de l'application répartie. En particulier, chaque nœud doit "connaître" les autres nœuds auxquels il est connecté. Ses tâches doivent connaître les autres tâches avec lesquelles elles communiquent et les éléments d'interfaces avec qui elles échangent des informations. Nous voulons aussi que l'accès aux information cités ci-dessus soit effectué d'une manière déterministe et que ces informations soit codées statiquement.

Pour réaliser cet objectif, nous produisons, pour chaque nœud de l'application répartie, un paquetage Ada contenant les informations sur la topologie de l'application nécessaire au tâche. Concrètement, il s'agit de trois types énumérés représentant respectivement :

1. Les autres nœuds de l'application connectés au nœud en question, en plus du nœud lui-même,
2. Toutes les tâches de l'application répartie connectées (à travers les nœuds qui les contiennent) au nœud en question, en plus des tâches du nœud lui même,
3. Tous les ports de tâches qui sont connectés au nœud courant, en plus des ports des tâches du nœud lui même. dernier.

Une constante du premier type énuméré est aussi produite pour permette à chaque nœud de connaître son identificateur.

Enfin, nous générons deux tables permettant :

1. l'association des énumérateurs correspondant aux tâches à ceux correspondant aux nœuds qui les contiennent,
2. l'association des énumérateurs correspondant aux ports à ceux des tâches qui les contiennent.

Ce paquetage contient aussi une constante représentant la taille maximale du plus grand type de donnée utilisé dans le nœud en cours. Cette constante est utilisée pour allouer statiquement les tampons de communication servant à échanger les messages.

Le code 7.2 montre le paquetage **PolyORB_HI_Generated.Deployment** produit pour le nœud **Workload_Manager** de l'exemple décrit dans la section 4.9.

Cette manière d'implanter le paquetage **PolyORB_HI_Generated.Deployment** optimise fortement la structure de l'application et réduit le nombre d'énumérateurs et surtout la taille des différentes tables qui sont indexées par ces énumérateurs. Cependant, chaque nœud contient

Exemple 7.2 – Paquetage **PolyORB_HI_Generated.Deployment**

```
package PolyORB_HI_Generated.Deployment is

   type Node_Type is (WoM_K, InS_K);
   -- Noeuds de l'application

   for Node_Type use (WoM_K => 1,
                      InS_K => 2);
   -- Clauses de représentation pour garantir la cohérence des données
   -- transmises.

   My_Node : constant Node_Type := WoM_K;
   -- Noeud courant

   type Entity_Type is (WoM_Regular_Producer_K,
                        WoM_On_Call_Producer_K,
                        -- ...
                        InS_External_Event_Source_K);
   -- Processus légers de l'application

   for Entity_Type use (WoM_Regular_Producer_K       => 1,
                        WoM_On_Call_Producer_K       => 2,
                        -- ...
                        InS_External_Event_Source_K => 5);

   Entity_Table : constant array (Entity_Type'Range) of Node_Type
     := (WoM_Regular_Producer_K       => WoM_K,
         WoM_On_Call_Producer_K       => WoM_K,
         -- ...
         InS_External_Event_Source_K => InS_K);
   -- Correspondance entre les processus légers et les noeuds

   type Port_Type is (WoM_Regular_Producer_Additional_Workload_K,
                      WoM_Regular_Producer_Handle_External_Interrupt_K,
                      WoM_On_Call_Producer_Additional_Workload_Depository_K,
                      -- ...
                      InS_External_Event_Source_External_Interrupt_K);
   -- Ports de l'application

   for Port_Type use
      (WoM_Regular_Producer_Additional_Workload_K            => 1,
       WoM_Regular_Producer_Handle_External_Interrupt_K      => 2,
       WoM_On_Call_Producer_Additional_Workload_Depository_K => 3,
       -- ...
       InS_External_Event_Source_External_Interrupt_K        => 8);

   Port_Table : constant array (Port_Type'Range) of Entity_Type
     := (WoM_Regular_Producer_Additional_Workload_K
                                  =>      WoM_Regular_Producer_K,
         WoM_Regular_Producer_Handle_External_Interrupt_K
                                  =>      WoM_Regular_Producer_K,
         WoM_On_Call_Producer_Additional_Workload_Depository_K
                                  =>      WoM_On_Call_Producer_K,
         -- ...
         InS_External_Event_Source_External_Interrupt_K
                                  => InS_External_Event_Source_K);
   -- Correspondance entre les ports et les processus légers

   Max_Payload_Size : constant Standard.Integer := 112;
   -- Taille maximale d'un message échangé par le noeud courant

end PolyORB_HI_Generated.Deployment;
```

un ensemble d'énumérateurs différent des autres nœuds. Ceci introduit des incohérences quand il envoie les énumérateurs des ports aux autres nœuds car leurs indexes ne sont pas nécessairement les mêmes. Pour pallier ce problème, nous utilisons les clauses de représentation Ada [Working Group, 2005, Section 13.4]. Ainsi les énumérateurs correspondant à la même entité AADL et déclarés sur des nœuds différents ont des valeurs homogènes.

Emballage et déballage de données Tous les routines d'emballage et de déballage de données sont rassemblées dans un paquetage nommé **PolyORB_HI_Generated.Marshallers**. La spécification de ce paquetage expose les différentes routines **Marshall** et **Unmarshall** utilisées dans le nœud. Le corps contient les instances nécessaires du paquetage **PolyORB_HI.-Marshallers_G** et les implantations des fonctions qui sont déclarées dans la spécification.

Table de nommage La table de nommage implante les services intergiciels de l'adressage et de liaison. Elle permettent de trouver en un temps constant les ressources de communications nécessaires pour atteindre une entité de l'application répartie.

Pour chaque couche basse de transport utilisée dans le nœud, une table indexée par les ports (3^{e} type énuméré dans le paquetage **PolyORB_HI_Generated.Deployment**) permet d'associer les ports aux informations de transport permettant de les atteindre. Ces tables sont déclarées dans le paquetage **PolyORB_HI_Generated.Naming** et sont utilisées pour initialiser les couches basses de transport en ouvrant exactement le nombre requis de canaux de communication lors de l'initialisation de l'application répartie.

Exemple 7.3 – Paquetage **PolyORB_HI_Generated.Naming**

```
with Interfaces.C;                          use Interfaces.C;
with SOIF_MTS_Wrapper_Ada.Structures;       use SOIF_MTS_Wrapper_Ada.Structures;
with PolyORB_HI_Generated.Deployment;       use PolyORB_HI_Generated.Deployment;

package PolyORB_HI_Generated.Naming is

  WoM_Node_Name : char_array := "WoM_K" & nul;
  InS_Node_Name : char_array := "InS_K" & nul;

  -- Table de nommage
  SOIF_MTS_Naming_Entry : array (Node_Type'Range)
    of aliased SOIF_MTS_Naming_Entry_Type :=
    (WoM_K => (Name    => WoM_Node_Name (WoM_Node_Name'First)'Access,
               Address => (Pid     => SOIF_MTS_ASSERT_V2_SPACEWIRE_PID,
                           Los     => SOIF_MTS_NON_GUARANTEED_DELIVERY,
                           Address => 2,
                           Proc_Id => 1200)),
     InS_K => (Name    => InS_Node_Name (InS_Node_Name'First)'Access,
               Address => (Pid     => SOIF_MTS_ASSERT_V2_SPACEWIRE_PID,
                           Los     => SOIF_MTS_NON_GUARANTEED_DELIVERY,
                           Address => 3,
                           Proc_Id => 1201)));

  SOIF_MTS_Naming_Store : SOIF_MTS_Naming_Store_Type :=
    (Size  => 2,
     Store => SOIF_MTS_Naming_Entry (SOIF_MTS_Naming_Entry'First)'Access);
  pragma Export (C, SOIF_MTS_Naming_Store, "SOIF_MTS_Naming_store");
  -- Export de la table de nommage vers le pilote Spacewire qui est
  -- développé en utilisant le langage C.

end PolyORB_HI_Generated.Naming;
```

Le code 7.3 montre le paquetage **PolyORB_HI_Generated.Naming** produit pour le nœud **Workload_Manager** de l'exemple décrit dans la section 4.9. Les valeurs des différentes information sur les couches de transports sont déduite à la fois du bus AADL utilisé et des propriétés spécifiques données pour les sous-composants AADL correspondant aux nœuds (numéro de canal, adresse...).

Couche haute de transport Cette couche permet de sélectionner automatiquement et en un temps constant la couche basse de transport nécessaire ou le mécanisme de livraison local pour échanger un message avec une entité distante ou locale.

La couche haute de transport contient les routines SEND et DELIVER permettant d'envoyer ou de recevoir un message. L'implantation de la routine SEND utilise la table de nommage pour invoquer les routines de la basse couche de transport qui est utilisée entre la source et la destination du message. Dans le cas d'une communication locale, la routine SEND invoque directement la routine DELIVER. L'implantation de la routine DELIVER utilisent les interrogateurs pour placer le message reçu dans la file d'attente de la tâche destinataire correspondant.

Le code 7.4 montre le paquetage **PolyORB_HI_Generated.Transport** produit pour le nœud **Workload_Manager** de l'exemple décrit dans la section 4.9. L'exemple de code ne montre pas l'implantation des routines interne de livraison et d'envoi. Ces routines utilisent le service intergiciel d'interaction ainsi que la couche basse de transport correspondant aux entités qu'ils gèrent.

7.3.2 Génération de code C

Les composants applicatifs et intergiciels générés pour la version en langage C de POLY-ORB-HI sont très similaires à ceux de la version en Ada. La différence majeure réside dans l'absence de surcharge de sous-programmes en C. Par exemple nous ne pouvons pas produire plusieurs sous-programmes et les appeler **Marshall**. Pour résoudre ce problème, deux solutions sont possibles :

- créer un seul sous programme avec le nom voulu qui encapsule les appels aux sous programmes surchargés (qui deviennent internes et qui ont chacun, son propre nom). La sélection se fait par l'intermédiaire d'un paramètre formel supplémentaire passé au sous-programme. Cette solution n'est applicable que pour les sous-programmes qui ont des signatures identiques (ou très similaire) mais qui existent dans des paquetages différents dans le cas Ada.
- Créer des sous programmes avec des noms différents mais qui sont facilement identifiables par l'utilisateur.

Nous avons choisi la deuxième solution car elle permet d'éviter les conversions de types parfois dangereuses dans le contexte de systèmes critiques.

Tous les composants générés dans le langage C sont conformes aux recommandations du profil Ravenscar que nous avons adaptées au langage C.

7.4 Déploiement et Configuration Automatiques

Dans cette section nous montrons comment le déploiement et la configuration d'une application TR^{2}E sont effectuées automatiquement.

Exemple 7.4 – Paquetage **PolyORB_HI_Generated.Transport**

```
with PolyORB_HI.Port_Type_Marshallers; use PolyORB_HI.Port_Type_Marshallers;
with PolyORB_HI_Generated.Activity;    use PolyORB_HI_Generated.Activity;
with PolyORB_HI_Generated.Marshallers; use PolyORB_HI_Generated.Marshallers;
with Ada.Real_Time;                    use Ada.Real_Time;
with PolyORB_HI.Time_Marshallers;      use PolyORB_HI.Time_Marshallers;

package body PolyORB_HI_Generated.Transport is

   procedure On_Call_Producer_Deliver (Port :        Port_Type;
                                       From :        Entity_Type;
                                       Msg  : in out Message_Type);

   procedure External_Event_Server_Deliver (Port :        Port_Type;
                                            From :        Entity_Type;
                                            Msg  : in out Message_Type);

   procedure Activation_Log_Reader_Deliver (Port :        Port_Type;
                                            From :        Entity_Type;
                                            Msg  : in out Message_Type);

   procedure Deliver (Entity : Entity_Type; Message : Stream_Element_Array) is
      Msg  : Message_Type;
      Port : Port_Type;
   begin
      Write (Msg, Message ((Message'First + Header_Size) .. Message'Last));
      Unmarshall (Port, Msg);
      case Entity is
         when WoM_On_Call_Producer_K =>
            On_Call_Producer_Deliver (Port, Sender (Message), Msg);
         when WoM_External_Event_Server_K =>
            External_Event_Server_Deliver (Port, Sender (Message), Msg);
         when WoM_Activation_Log_Reader_K =>
            Activation_Log_Reader_Deliver (Port, Sender (Message), Msg);
         when others =>
            raise Program_Error;
      end case;
   end Deliver;

   procedure Regular_Producer_Send (Entity  : Entity_Type;
                                    Message : Stream_Element_Array);

   procedure External_Event_Server_Send (Entity  : Entity_Type;
                                         Message : Stream_Element_Array);

   procedure Send (From, Entity : Entity_Type; Message : Message_Type) is
      Msg : constant Stream_Element_Array := Encapsulate
        (Message, From, Entity);
   begin
      case From is
         when WoM_Regular_Producer_K =>
            Regular_Producer_Send (Entity, Msg);
         when WoM_External_Event_Server_K =>
            External_Event_Server_Send (Entity, Msg);
         when others =>
            raise Program_Error;
     end case;
   end Send;

   -- Implantation des routines internes de livraison...

   -- Implantation des routines internes d'envoi. Ils utilisent la
   -- couche basse de transport correspondant en cas d'envoi distant
   -- ou bien un des routines de livraison internes ci-dessus en cas
   -- d'envoi local...

end PolyORB_HI_Generated.Transport;
```

7.4.1 Déploiement

Le déploiement (voir la définition 2.3 page 24) consiste à sélectionner les composants dont l'application a besoin. Grâce à notre stratégie de génération de code seuls les composants intergiciels utilisés par un nœud sont inclus dans son code. Ainsi, la génération automatique de la couche de transport permet d'inclure uniquement les composants des couches basses de transports utilisés par le nœud. En particulier, dans le cas d'une application monolithique, aucune couche de transport n'est incluse.

7.4.2 Configuration

La configuration (voir la définition 2.4 page 24) consiste à paramétrer les composants sélectionnés lors du déploiement. Grâce aux paquetage **PolyORB_HI_Generated.Deployment** et **PolyORB_HI_Generated.Naming**, la plupart des composants de l'intergiciel minimal et ceux produits automatiquement sont paramétrés statiquement. Ainsi, le nombre exact de tâches dont le nœud a besoin est créé, le nombre exact de canaux de communication est ouvert, la taille maximale des tampons de communication est déduite et allouée statiquement.

7.5 Chaîne de production automatique

Nous voulons que les composants de l'intergiciel minimal, ceux générés automatiquement ainsi que les composants fournis par l'utilisateur soient intégrés automatiquement pour former l'application TR^{2}E finale. Nous voulons aussi supporter que le code de l'utilisateur soit lui même généré par d'autres outils tiers (SDL, LUSTRE, ASN.1...).

Les parties dorsales que nous avons décrites dans la section 7.3 ne produisent pas uniquement du code source. Elle génèrent aussi des fichiers de support permettant d'intégrer les composants de l'utilisateur et les composants de l'intergiciel. Ces fichiers de support sont utilisés pour compiler les nœuds de l'application répartie et produire des binaires prêts à être exécutés. Dans le cas Ada, ces fichiers de support sont des *Makefile* et des fichiers projets du compilateur GNAT. Dans le cas C, il s'agit uniquement de *Makefile*.

Comme le montre la figure 7.4, le code généré est lié automatiquement avec le code utilisateur de plusieurs manières :

1. **Code source :** l'utilisateur donne l'implantation source (en Ada ou en C) des sous-programmes. Dans ce cas, le code généré inclut des appels vers les unités de compilation correspondantes. Les fichiers de support assurent la compilation automatique des fichiers de l'utilisateur,
2. **Fichiers objet ou bibliothèques :** l'utilisateur indique dans le modèle AADL que l'implantation d'un sous-programme donné est fournie par un ensemble de fichiers objet ou librairies. Dans ce cas, le code généré inclut les appels vers les unités de compilation correspondantes. Les fichiers de support effectuent automatiquement l'édition de lien du nœud avec les fichiers objets et les bibliothèques données,
3. **Langage spécialisé de haut niveau (LUSTRE, SDL, ASN.1...) :** dans ce cas, le code généré inclut les appels aux unités de compilation qui doivent être générées par des outils tiers à partir des spécifications du langage de haut niveau. Ensuite, les fichiers de support effectuent tout d'abord l'invocation de ces outils tiers pour effectuer la génération de code. Enfin, ils compilent les fichiers générés par ces outils tiers et effectuent l'édition des liens du nœud en cours.

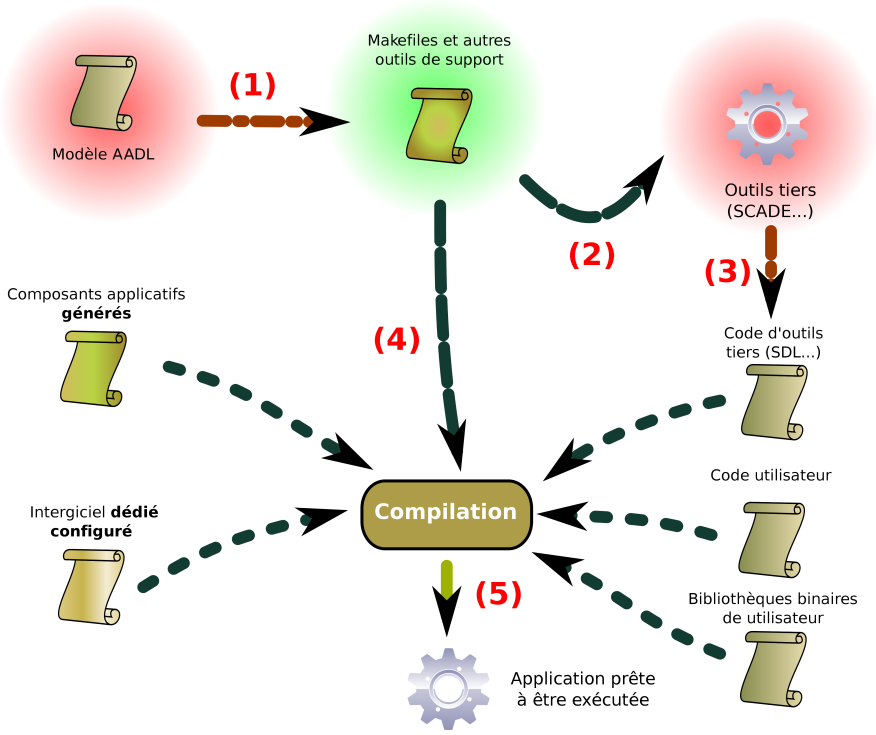

FIGURE 7.4 – Intégration du code de l'utilisateur

Ceci permet un développement rapide et une conception flexible des systèmes TR²E et ne contraint pas les implantations de l'utilisateur à une seule catégorie.

7.6 Synthèse

Dans ce chapitre, nous avons présenté notre suite outil OCARINA qui permet d'analyser les modèles AADL des applications TR²E. OCARINA possède une architecture similaire à un compilateur moderne et permet à parti d'un modèle AADL de produire automatiquement du code vers plusieurs support d'exécution ou bien de transformer les modèles AADL vers des formalismes destinées à la vérification (comme les réseaux de Petri). Aussi, OCARINA offre des bibliothèques logicielles aux outils tiers pour manipuler les modèles AADL. Par exemple, l'analyseur d'ordonnancement CHEDDAR que nous utilisons pour effectuer des analyses statiques d'ordonnançabilité sur nos modèles AADL utilise les bibliothèques d'OCARINA pour manipuler ces modèles.

Nous avons ensuite décrit les deux parties dorsales que nous avons définies pour produire du code vers les deux versions de l'intergiciel POLYORB-HI. Les composants générés par ces deux parties dorsales sont configurés automatiquement et optimisés en fonction des propriétés de chaque nœud de l'application TR²E. Nous avons aussi montré comment le comportement de ces deux parties dorsales ne se résume pas à la génération de code. La production de fichiers de support permet d'intégrer les composants produits automatiquement avec les composants de l'utilisateur et ceux produit par d'autres outils.

Nous avons spécifié des règles claires et précises de transformation des entités AADL vers des constructions des langages de programmation Ada et C. Ceci a conduit à confier la tâche

de la rédaction de l'annexe de génération de code de la prochaine version du standard AADL (AADLv2) à l'équipe de recherche à laquelle nous appartenons.

Le prochain chapitre validera nos choix de conception sur un exemple complexe et permettra de vérifier l'adéquation des solutions que nous avons apportées au problème des systèmes TR^2E. Nous y donnerons aussi des mesures de performances.

Chapitre 8

Validation et Analyse de Performances

SOMMAIRE

8.1 Introduction

Dans les chapitres précédents, nous avons défini un processus automatique pour l'analyse et la production des applications TR^{2}E critiques à partir de leur description d'architecture en AADL. Nous avons aussi défini l'architecture d'intergiciel fondée sur l'architecture schizophrène. Cette architecture couplée au processus de production, permet de générer automatiquement un intergiciel dédié aux besoins spécifiques de chaque application. Nous avons ensuite présenté notre implantation de cet intergiciel, POLYORB-HI. Enfin, nous avons présenté les contributions effectuées sur la suite d'outils OCARINA pour implanter le processus automatique d'analyse et de production.

Dans ce chapitre, nous montrons notre approche répond à la problématique que nous nous étions fixée. Pour ce faire, nous nous sommes basés sur un exemple classique d'application critique.

Dans la section 8.2 nous présentons l'étude de cas d'origine. Ensuite, nous décrivons l'adaptation en langage AADL de cette étude. Nous expliquons comment nous sommes passés d'une

application monolithique à une application répartie. Dans la section 8.3, nous donnons les résultats de l'analyse d'ordonnancement effectués sur le modèle par l'outil CHEDDAR. Nous donnons les résultats en terme d'exécution et d'empreinte mémoire du code Ada généré pour cette étude dans la section 8.4. La section 8.5 conclut ce chapitre.

8.2 Études de cas

Plusieurs études de cas ont été réalisées pour prouver le bon fonctionnement de notre processus de production. En particulier, une étude cas présentant une application répartie du domaine spatial a été élaborée pour effectuer la démonstration finale du projet européen ASSERT. Nous avons présenté cette étude et donné les résultats de nos expérimentations dans certaines de nos publications [Zalila *et al.*, 2008; Hugues *et al.*, 2008]. Dans cette section nous présentons une nouvelle étude de cas illustrant les résultats de notre travail de thèse.

Cette nouvelle étude est un exemple classique issu du document présentant les recommandations pour le profil Ravenscar [Burns *et al.*, 2004]. Elle permet d'illustrer les capacités d'expression du profil pour construire des systèmes temps-réel. Nous présentons l'étude d'origine. Ensuite nous expliquons comment nous l'avons transformée en une application répartie et adaptée au langage AADL. Nous donnerons à la fin de cette section les avantages de l'utilisation du langage AADL.

8.2.1 Présentation de l'étude de cas d'origine

L'exemple que nous avons choisi décrit un système de gestion de charge de travail. Il s'agit d'un processus périodique pouvant gérer des charges de travail variables. Les travaux réguliers sont effectués par un processus léger périodique de haute priorité. Les travaux supplémentaires (irréguliers) sont délégués à un processus léger sporadique de moindre priorité. Par ailleurs, le système est capable de recevoir des interruptions venant de l'extérieur. Ces interruptions sont reçues par un processus léger de très haute priorité et sont enregistrées dans un tampon spécifique. Le traitement de ces interruptions est délégué à un processus léger sporadique de très faible priorité qui est réveillé de temps en temps par le processus périodique principal.

Ainsi, l'exemple est formé des entités suivantes :

– Quatre processus légers que nous donnons ci après par ordre décroisant de priorité :

  1. Un processus léger sporadique (*External Event Server*) effectue la réception des interruptions extérieures et leur enregistrement dans un tampons spécifique,
  2. Un processus léger périodique (*Regular Producer*) effectue la charge de travail régulière. Il délègue, sous des conditions spécifiques, la charge de travail supplémentaire et le traitement des interruptions extérieures à d'autres processus légers,
  3. Un processus léger sporadique (*On Call Producer*) effectue la charge de travail supplémentaire,
  4. Un processus de travail sporadique (*Activation Log Reader*) effectue une quantité de travail correspondant au traitement de la dernière interruption reçue.

– Trois données partagées et protégées :

  1. Un tampon (*Request Buffer*) reçoit les ordres de travaux supplémentaires. Il est rempli par *Regular Producer* et consulté par *On Call Producer*,

2. Une file d'attente (*Event Queue*) des interruptions extérieures. Elle est remplie par le périphérique émettant les interruptions et consultée par *External Event Server*,
3. Un journal (*Activation Log*) des interruptions à traiter. Il est rempli par *External Event Server* et consulté par *Activation Log Reader*.

– Une entité passive (*Production Workload*). Elle abrite l'opération *Small Whetstone* [Curnow *et al.*, 1976] qui traite les travaux demandés par un des processus légers.

Nom	Protocole	Période	Échéance	WCET	Priorité
Regular_Producer	Périodique	1000 ms	500 ms	498 ms	7
On_Call_Producer	Sporadique	1000 ms	800 ms	250 ms	5
Activation_Log_Reader	Sporadique	1000 ms	1000 ms	125 ms	3
External_Event_Server	Sporadique	5000 ms	100 ms	2 ms	11

TABLE 8.1 – Caractéristiques des processus légers

La table 8.1 résume les caractéristiques de chacun des processus légers. Nous avons déduit les valeurs de pire temps d'exécution à partir du code Ada décrivant leurs comportements fourni par [Burns *et al.*, 2004]. Pour les processus légers sporadiques, la colonne **Période** indique le temps minimal entre deux déclenchements.

Dans la prochaine section, nous illustrons comment nous avons adapté cet exemple au langage AADL et comment nous l'avons rendu réparti.

8.2.2 Adaptation au langage AADL

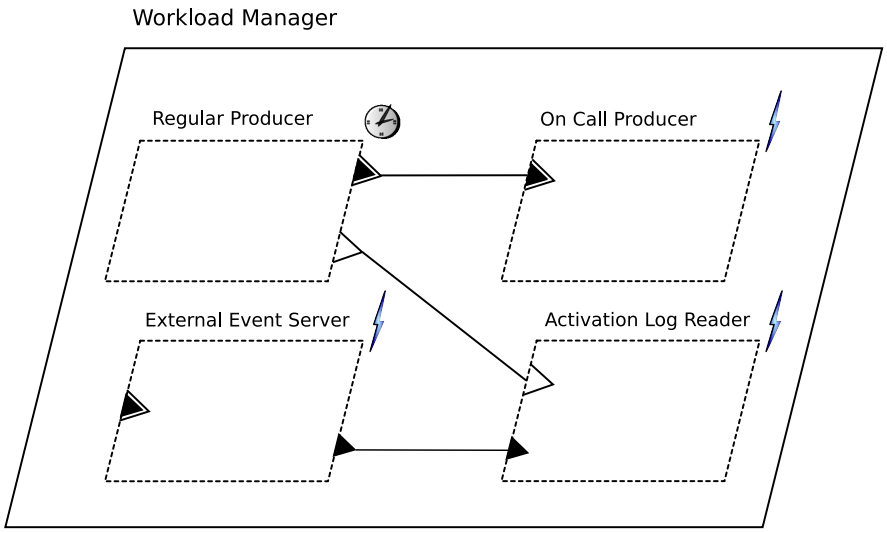

FIGURE 8.1 – Adaptation en AADL de l'exemple du guide Ravenscar

L'adaptation de l'exemple décrit dans la section précédente vers le langage AADL a été une tâche facile. La figure 8.1 donne une description du modèle du processus. Nous avons donné dans la section 4.9 plusieurs exemples des différents constituants de ce modèle AADL. Dans la suite nous expliquons la manière dont nous avons adapté cet exemple vers le langage AADL. Ensuite nous décrivons le passage d'une application monolithique à une application répartie.

Le modèle obtenu couvre bien les fonctionnalités du langage AADL utilisées pour spécifier des systèmes TR^{2}E.

Composants actifs

Les composants actifs se traduisent naturellement en AADL par des composants **thread**. Nous disposons de toutes les propriétés nécessaires pour spécifier la nature, la période et la propriété d'un composant **thread**. Par ailleurs les deux types de composants **thread** présents dans l'exemple sont supportés par notre intergiciel et notre générateur de code.

Composants passifs

Le comportement de chacun de ces composants est spécifié par l'intermédiaire de la propriété standard `Compute_Entrypoint`. Nous associons cette propriété à chacun des ports en entrée de type événement ou bien directement au composant **thread** selon la nature de ce dernier. Nous avons repris exactement le même comportement associé aux tâches de l'exemple initial.

Nous avons adapté l'opération effectuant la charge de travail pour qu'elle utilise des nombres à virgule fixe au lieu des nombres à virgule flottante afin de respecter les restrictions de POLYORB-HI.

Objets partagés

La seule différence entre le modèle AADL et la description de l'exemple donné dans [Burns *et al.*, 2004] réside dans la manière de spécifier des données partagées. En effet, nous n'avons pas eu besoin de spécifier explicitement les composants *Request Buffer*, *Event Queue* et *Activation Log*. Les éléments d'interfaces des différents composants **thread** on pleinement joué le rôle de ces objets.

La file d'attente de l'objet *Activation Log* est de taille égale à un comme nous l'avons constaté dans le code fourni par le guide Ravenscar. De plus, une écriture dans *Activation Log* par *External Event Server* ne provoque pas le déclenchement de *Activation Log Reader*. Par conséquent, nous avons choisi de représenter cet objet par un port de type donnée.

Passage vers une application répartie

Comme le montre la figure 8.1, le processus **Workload_Manager** reçoit des interruptions extérieures. Pour simuler l'envoi de ces interruptions nous avons ajouté un second processus (**Interruption_Simulator**) qui envoie de manière aléatoire des messages à **Workload_Manager**. Chacun des deux processus s'exécute sur un processeur de type LEON2. Ils sont reliés par l'intermédiaire d'un bus SPACEWIRE. Ceci permet également d'évaluer la partie communication distante dans notre étude de cas.

La figure 8.2 rappelle la nouvelle architecture de l'étude de cas.

8.2.3 Bénéfices de l'utilisation du langage AADL

L'utilisation du modèle AADL a permis de ne plus avoir à gérer les objets partagés explicitement. Ces derniers sont spécifiés par l'intermédiaire des différents éléments d'interface des composants **thread**. De plus, comme nous l'avons précisé dans le chapitre 5, l'utilisateur ne

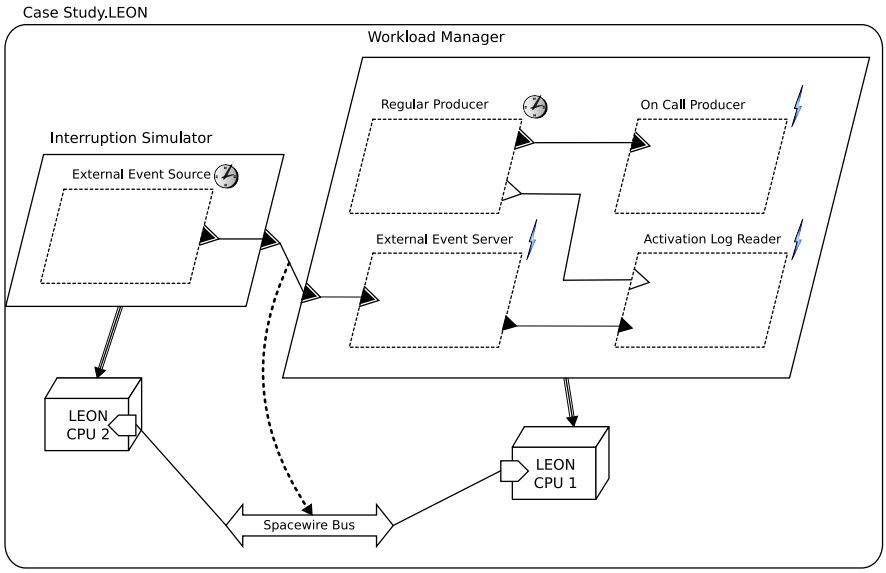

FIGURE 8.2 – Passage vers une application répartie

code pas les différentes caractéristiques des tâches de l'application. Il n'a pas non plus à gérer la communication entre ces tâches. Tout est pris en charge automatiquement par le code généré avec OCARINA.

Par ailleurs, le modèle AADL subit une analyse automatique qui vérifie le respect des échéances des tâches. Ce point fait l'objet de la prochaine section.

8.3 Tests d'ordonnancement

Pour analyser le modèle AADL de notre application, nous utilisons l'outil CHEDDAR [Singhoff *et al.*, 2004]. CHEDDAR utilise les bibliothèques de notre suite d'outils OCARINA pour convertir les modèles AADL en une représentation interne. Il permet de proposer des modifications sur des modèles non ordonnançables pour les rendre ordonnançables.

Exemple 8.1 – Modèle du composant **On_Call_Producer**

```
thread On_Call_Producer
features
  Additional_Workload_Depository : in event data port Workload
    {Queue_Size          => 5;
     Compute_Entrypoint => "Work.On_Call_Producer_Operation";};
  -- Pour recevoir la charge de travail supplémentaire de la part de
  -- Reagular_Producer. Cette taille est donnée par le guide
  -- Ravenscar.
properties
  Dispatch_Protocol                  => Sporadic;
  Period                             => 1000 ms;
  Deadline                           => 800 ms;
  Compute_Execution_Time             => 0 ms .. 250 ms;
  Cheddar_Properties::Fixed_Priority => 5;
end On_Call_Producer;
```

En plus des propriétés standards (`Period`, `Deadline`...), CHEDDAR spécifie un ensemble de propriétés spécifiques (`Cheddar_Properties`) pour décrire les aspects temps-réel des compo-

sants **thread** (`Fixed_Priority...`) et **processor** (`Preemptive_Scheduler...`). Ainsi, nous avons associé à chaque composant **thread** les propriétés indiquant ses caractéristiques temps-réel. L'exemple 8.1 montre les caractéristiques du composant **On_Call_Producer**.

Si OCARINA invoque l'outil CHEDDAR sur notre modèle AADL afin de vérifier le respect des échéances des tâches, nous obtenons le résultat suivant :

```
[...]

Scheduling feasibility, Processor case_study.leon.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 task response time :

- Bound on task response time :  (see [2], page 3, equation 4).
    case_study.leon.wom.activation_log_reader => 875
    case_study.leon.wom.on_call_producer => 750
    case_study.leon.wom.regular_producer => 500
    case_study.leon.wom.external_event_server => 2
- All task deadlines will be met : the task set is schedulable.

[...]
```

CHEDDAR effectue en premier une analyse basée sur le taux d'utilisation du processeur **CPU_1**. Le taux d'utilisation du processeur étant égal à 0.87340 dans le pire cas, nous ne pouvons rien conclure quant à l'ordonnançabilité des tâches par une analyse de type RMA [Sha *et al.*, 1993]. CHEDDAR passe alors à une analyse basée sur le temps de réponse de chacune des tâches. Cette analyse permet de conclure que toutes les tâches respectent leurs échéances et que notre système est ordonnançable. Nous pouvons générer le code pour notre application.

8.4 Résultats avec POLYORB-HI Ada

Dans cette section nous présentons les résultats en termes de génération de code et d'exécution de l'étude de cas. Nous donnons aussi différentes statistiques sur les tailles des sources et des binaires.

8.4.1 Génération de code

À partir de notre modèle AADL, nous avons utilisé OCARINA pour produire un ensemble de fichiers sources Ada pour les deux nœuds de l'application. Ces fichiers implantent les com-

posants intergiciels et applicatifs dédiés à chacun des nœuds. Nous donnons ci-après la liste des fichiers produits pour chacun des nœuds. Il est important de noter que chacun des nœuds **Workload_Manager** et **Interruption_Simulation** dispose d'un ensemble différent de fichiers générés :

- `polyorb_hi_generated-activity.ad[s|b]` : contiennent la spécification et le corps du paquetage `PolyORB_HI_Generated.Activity`. Ils implantent les services intergiciels d'exécution et d'interaction,
- `polyorb_hi_generated-deployment.ads` : contient la spécification du paquetage `PolyORB_HI_Generated.Deployment` que nous avons décrit dans la section 7.3,
- `polyorb_hi_generated-marshallers.ad[s|b]` : contiennent la spécification et le corps du paquetage `PolyORB_HI_Generated.Marshallers`. Ils implantent le service intergiciel de représentation avancée,
- `polyorb_hi_generated-naming.ads` : contient la spécification du paquetage `PolyORB_HI_Generated.Naming`. Il contient l'implantation des services intergiciel d'adressage et de liaison,
- `polyorb_hi_generated-transport.ad[s|b]` : contiennent la spécification et le corps du paquetage `PolyORB_HI_Generated.Transport`. Ils implantent le service de la couche haute de transport,
- `polyorb_hi_generated-types.ads` : contient la spécification du paquetage `PolyORB_HI_Generated.Types`. Il implante le service intergiciel de typage,
- `wom.adb` : contient le sous-programme principal du nœud.

Un ensemble de fichiers de support (Makefile, fichiers de projets pour le compilateur GNAT) ont aussi été produits. Ils permettent de sélectionner les paquetages de l'intergiciel minimal nécessaires à chacun des nœuds de l'application. Nous remarquons que deux fichiers de l'intergiciel minimal sont sélectionnés pour le nœud **Workload_Manager** et ne le sont pas pour le nœud **Interruption_Simulation**. Il s'agit des fichiers suivant :

- `polyorb_hi-sporadic_task.ad[s|b]` : contiennent la spécification et le corps du paquetage `PolyORB_HI.Sporadic_Task`. En effet, le nœud **Interruption_Simulation** contient uniquement une tâche périodique,
- `polyorb_hi-transport_low_level-extras_mts.ad[s|b]` : contiennent la spécification et le corps du paquetage `PolyORB_HI.Transport_Low_Level.Extra` spécifique à la couche SPACEWIRE. Ce paquetage contient des routines nécessaires à la réception de messages sur un bus SPACEWIRE. Ce paquetage n'est pas sélectionné pour le nœud **Interruption_Simulation** puisque ce dernier effectue l'envoi des signaux jouant le rôle d'interruptions et ne reçoit rien.

Enfin, les fichiers de support produits automatiquement sont utilisés pour compiler et assembler le code généré et celui de l'utilisateur. Nous compilons le code généré en utilisant le compilateur Ada GNAT pour l'architecture LEON2. Pour optimiser l'empreinte mémoire les exécutables générés, nous supprimons toutes les vérifications du compilateur et nous forçons la recompilation de tout le support d'exécution de GNAT (options `-a -f -gnatp` de la ligne de commande). Le résultat de cette compilation est un ensemble de deux binaires représentant chacun un nœud de l'application.

8.4.2 Exécution

Nous avons exécuté notre application TR^{2}E en utilisant le simulateur de processeur LEON2 **Tsim** [Geisler, 2008]. Nous avons utilisé un module d'entrée/sortie développé par l'entreprise Sci-

Sys[7]. Il permet de simuler le bus de communication SPACEWIRE en utilisant **Tsim**. Ci-dessous, nous donnons les traces d'exécution de chacun des nœuds de l'application.

8.4.3 Trace d'exécution de Interruption_Simulation

Nous avons choisi d'envoyer des interruptions de manière pseudo aléatoire avec un temps d'arrivée minimal entre deux interruptions supérieur ou égal à 5 secondes. Ceci vise à respecter les contraintes de sporadicité du processus léger **External_Interrupt_Server** qui gère la réception des interruptions.

```
tsim> go
[Executing 'go'...]
resuming at 0x40000000
[ 0.00] External Events: starting
[ 0.00] External Events: send an new event: 1.
[ 15.00] External Events: send an new event: 2.
[ 25.00] External Events: send an new event: 3.
[ 35.00] External Events: send an new event: 4.
[ 40.00] External Events: send an new event: 5.
```

8.4.4 Trace d'exécution de Workload_Manager

Dans la trace ci-dessous, nous donnons une partie de la trace d'exécution du nœud **Regular_Producer**. Nous remarquons que tous les travaux supplémentaires sont exécutés. Toutes les interruptions externes ont été traitées par la tâche **Activation_Log_Reader** ce qui montre que le dimensionnement du système utilisant une file d'attente de taille 1 pour les interruptions est correct puisqu'aucune interruption n'a été perdue.

```
tsim> go
[Executing 'go'...]
resuming at 0x40000000
[ 0.00] Regular Producer: doing some work.
[ 0.48] Regular Producer: end of cyclic activation
[ 1.00] Regular Producer: doing some work.
[ 1.48] Sending extra work to  'On_Call_Producer': 250
[ 1.48] Regular Producer: end of cyclic activation
[ 1.48] On Call Producer: doing some work.
[ 1.74] On Call Producer: end of sporadic activation.
[ 2.00] Regular Producer: doing some work.
[ 2.48] Signaling 'Activation Log Reader'
[ 2.48] Regular Producer: end of cyclic activation
[ 2.48] Activation Log Reader: no new interrupts.
[ 3.00] Regular Producer: doing some work.
[ 3.48] Regular Producer: end of cyclic activation
[ 3.80] External Event Server: received an external interrupt
[ 3.80] External Event Server: end of sporadic activation.
[ 4.00] Regular Producer: doing some work.
[ 4.48] Regular Producer: end of cyclic activation
[ 5.00] Regular Producer: doing some work.
[ 5.48] Signaling 'Activation Log Reader'
```

7. SciSys est un des participants industriels au projet ASSERT. http://www.scisys.co.uk

```
[ 5.48] Regular Producer: end of cyclic activation
[ 5.48] Activation Log Reader: do some work.
[ 5.60] Read external new interruption: 1. Arrived at [ 3.90]
[ 5.60] Activation Log Reader: end of parameterless sporadic activation.
[ 6.00] Regular Producer: doing some work.
[ 6.48] Sending extra work to  'On_Call_Producer': 250
[ 6.48] Regular Producer: end of cyclic activation
[ 6.48] On Call Producer: doing some work.
[ 6.74] On Call Producer: end of sporadic activation.
[ 7.00] Regular Producer: doing some work.
[ 7.48] Regular Producer: end of cyclic activation
[ 8.00] Regular Producer: doing some work.
[ 8.48] Signaling 'Activation Log Reader'
[ 8.48] Regular Producer: end of cyclic activation
[ 8.48] Activation Log Reader: do some work.
[ 8.60] Activation Log Reader: no new interrupts.
[ 8.60] Activation Log Reader: end of parameterless sporadic activation.
[ 9.00] Regular Producer: doing some work.
[ 9.48] Regular Producer: end of cyclic activation
```

8.4.5 Empreintes mémoire

Dans cette section nous donnons différentes statistiques sur les tailles des fichiers sources et les empreintes mémoire des binaires de l'étude de cas. Les empreintes mémoire sont mesurées en utilisant la commande `size` des `binutils`[8].

	Workload_Manager	**Interruption_Simulator**
Code de l'utilisateur	188	38
Code généré automatiquement	1860	596
Code de l'intergiciel minimal	1512	1398
Total	3560	2032

TABLE 8.2 – Tailles des fichiers sources (en ligne de code)

Dans la table 8.2, nous donnons la taille des différents fichiers sources qui forment l'application. Nous remarquons que la taille du code écrit par l'utilisateur est très petite par rapport à la taille globale de l'application (2% pour **Interruption_Simulator** et 5% pour **Workload_Manager**.

Il est à noter que la taille des composants générés automatiquement est proche de celle des composants de l'intergiciel minimal car l'application est relativement petite. La taille totale de tous les fichiers de l'intergiciel minimal POLYORB-HI-Ada est inférieure à 2000 lignes. Pour les applications réparties contenant des dizaines de composants, le code généré sera beaucoup plus grand que celui du code de l'intergiciel minimal.

La table 8.3 donne l'empreinte mémoire des binaires de notre étude de cas. Nous avons une empreinte totale de $2217\ Ko$ pour le nœud **Workload_Manager**. Une grande partie de cette empreinte est due à la taille des piles des tâches du nœud. En effet, le compilateur GNAT *for* LEON impose une taille de pile minimale de $100\ Ko$ pour chaque tâche. Dans notre cas, nous avons les quatre tâches du modèle AADL, la tâche qui reçoit les messages dans la couche basse de transport et une tâche supplémentaire spécifique à SPACEWIRE. Ceci implique l'utilisation

8. www.gnu.org/software/binutils/

	Workload_Manager	Interruption_Simulator
Code de l'utilisateur	4	1
Code généré automatiquement	482	123
Code de l'intergiciel minimal	245	143
Support d'exécution du compilateur	475	467
Total des fichiers objets	**1215**	**734**
Empreinte de l'exécutable	**2217**	**1745**

TABLE 8.3 – Empreintes mémoire des binaires (en kilo-octet) pour LEON2

$600\ Ko$ de l'empreinte mémoire par les piles de différentes tâches. Une autre partie du code est due aux pilotes de communication SPACEWIRE et du noyau ORK [de la Puente *et al.*, 2000].

Nous constatons que la taille des composants intergiciels est différente pour chacun des nœuds de l'application. Le nœud **Interruption_Simulator** possède des composants intergiciels de plus petite taille que ceux dans **Workload_Manager** (36%). Ceci est obtenu grâce à la personnalisation des composants aux besoins spécifiques de ce nœud.

8.4.6 Comparaison avec POLYORB

Nous avons utilisé la partie dorsale **POLYORB-QOS-Ada** de OCARINA pour construire le même exemple pour l'intergiciel schizophrène POLYORB [Quinot, 2003]. Puisque nous ne pouvons pas tester cet intergiciel sur l'architecture LEON2, nous avons recompilé notre exemple pour l'architecture native (GNU/LINUX). Les nœuds utilisent les Sockets BSD pour communiquer. Le tableau 8.4 donne les empreintes mémoire des exécutables construit en utilisant l'intergiciel POLYORB (première ligne) et intergiciel POLYORB-HI (seconde ligne).

	Workload_Manager	Interruption_Simulator
POLYORB	2026	1994
POLYORB-HI	579	527

TABLE 8.4 – Empreintes mémoire (en kilo-octet) sur GNU/LINUX

Nous remarquons que l'empreinte mémoire obtenue avec POLYORB-HI est considérablement plus réduite que celle obtenu avec l'intergiciel POLYORB (26%-28%). Ceci s'explique par le fait que POLYORB, contrairement à POLYORB-HI, se configure dynamiquement et que ses composants ne sont pas dédiés à l'application.

8.4.7 Exemple local

Pour pouvoir comparer notre étude ce cas avec l'exemple d'origine, nous avons modifié le modèle AADL en enlevant le second nœud **Interruption_Simulator**, le second processeur **CPU_2** et le bus SPACEWIRE. Nous obtenons un modèle fonctionnellement équivalent au modèle de [Burns *et al.*, 2004].

OCARINA détecte automatiquement le caractère monolithique de l'application et n'intègre pas de couche basse de transport dans le code généré. Nous obtenons ainsi un nœud **Workload_Manager** considérablement plus petit que celui pour le cas réparti. Le tableau 8.5 donne le nombre de ligne de code ainsi que les tailles des différents binaires pour notre exemple ainsi que

pour l'exemple d'origine. Nous remarquons que l'empreinte mémoire de notre exemple est très proche de celle de l'exemple initial (+6%). Ce surplus en taille mémoire est acceptable puisque la quasi totalité du code est générée automatiquement grâce à notre processus de production contrairement à l'exemple du guide Ravenscar où tout le code est écrit manuellement.

	Exemple AADL	Exemple Ravenscar
Nombre de lignes de code	**3352**	**488**
Empreinte de l'exécutable	**1535**	**1441**

TABLE 8.5 – Nombre de ligne de code et empreintes mémoire des binaires du nœud **Workload_Manager** (en kilo-octet)

Nous pouvons conclure que le code généré pour notre exemple est fortement personnalisé aux propriétés de l'application.

8.5 Synthèse

Dans ce chapitre, nous avons présenté les résultats expérimentaux de notre travail de thèse. Nous avons repris une étude de cas classique du domaine temps-réel critique et nous l'avons adoptée au langage AADL et rendue répartie. Nous avons pu effectuer une analyse statique d'ordonnancement sur le modèle AADL en utilisant l'outil CHEDDAR. Ensuite nous avons produit du code et exécuté notre application avec succès.

Nous avons donné des mesures sur la taille des fichiers sources et l'empreinte mémoire des binaires de notre étude de cas et conclu que le code fourni pas l'utilisateur est beaucoup plus petit que le code produit automatiquement ou encore celui de l'intergiciel minimal. Les tailles trouvées sont raisonnables pour des systèmes TR^{2}E. L'empreinte mémoire des exécutables obtenus est considérablement plus réduite que celle de POLYORB-QOS.

Ceci nous permet de dire que nous avons accompli les objectifs de notre travaux de thèse qui consistent à construire automatiquement des intergiciels dédiés aux besoins des applications réparties.

Chapitre 9

Conclusions et Perspectives

SOMMAIRE

Dans les chapitres précédents de ce mémoire, nous avons présenté la problématique de notre travail de thèse : analyser automatiquement les applications TR^{2}E et construire un intergiciel dédié à ces applications. Ensuite, nous avons proposé et implanté des solutions à cette problématique. Pour illustrer ces solutions, nous avons présenté une étude de cas. Dans ce chapitre, nous rappelons les réalisations de ce travail de thèse. Ensuite, nous présentons les conclusions et les perspectives pour étendre ce travail.

9.1 Rappel des contributions et des résultats

La problématique principale de notre travail de thèse consistait à définir un processus de production nous permettant d'analyser automatiquement les applications TR^{2}E critiques. La complexité de tels système est toujours croissante et leur production manuelle est source d'erreur. Nous voulons également produire une grande partie du code qui soit personnalisée pour les besoins de ces applications.

Nouvelle architecture intergicielle

La solution que nous avons proposée consistait tout d'abord à définir une nouvelle architecture intergicielle fondée sur l'architecture schizophrène (chapitre 3). Cette architecture propose une nouvelle découpe des services intergiciels en les divisant en deux familles principales : les services fortement personnalisables en fonction des besoins de l'application et les services faiblement personnalisables.

Les services intergiciels fortement personnalisables sont produits automatiquement à partir d'une description l'application TR^{2}E. Ils sont optimisés sans qu'il n'y ait de risque d'erreur dû au développement manuel par l'utilisateur. Les services faiblement personnalisables font partie d'une bibliothèque intergicielle minimale. Un sous-ensemble de ces derniers services est sélectionné pour être intégré dans l'application TR^{2}E.

Nous avons ensuite implanté cette nouvelle architecture intergicielle. Le chapitre 6 décrit POLYORB-HI, l'intergiciel pour les systèmes TR^{2}E que nous avons réalisé durant ce travail de

thèse. Cet intergiciel respecte les recommandations du profil Ravenscar et pour les systèmes critiques (voir 2.5.1). Deux versions de cet intergiciel ont été développées : une en Ada et une autre en C. Une grande partie des composants de cet intergiciel est produite automatiquement en fonction des propriétés de l'application. Ceci nous a mené à choisir un formalisme adéquat de description des applications réparties et à définir un processus automatique pour produire certains composants.

Processus automatique de production

Nous avons choisi AADL 1.0, un langage de description d'architecture, pour décrire les applications TR^{2}E. Le chapitre 4 décrit les aspects du langage AADL utiles pour notre travail de thèse. Nous avons ensuite défini un sous-ensemble de ce langage et ajouté des règles de sémantiques supplémentaires pour garantir le respect du profils Ravenscar et des restrictions liées aux systèmes critiques (section 4.9).

Nous avons ensuite proposé un processus automatique de production. Ce processus, décrit dans le chapitre 5 effectue l'analyse automatique des applications TR^{2}E à partir de leur modèles AADL. Il génère un ensemble de composants applicatifs et une grande partie des composants intergiciels en les optimisant en fonction des besoins de l'application. Nous avons défini les règles de transformation des modèles AADL vers ces composants dans les sections 5.3 et 5.4. Ce processus permet aussi de déployer et configurer les composants de l'intergiciel minimal. Enfin, tous les composants de l'application TR^{2}E (applicatifs, intergiciels et fournis par l'utilisateur) sont rassemblés automatiquement et compilés pour produire l'application finale.

Pour démontrer la faisabilité de cette approche, nous avons enrichi la suite d'outils OCARINA qui manipule les modèles AADL. Nous avons implanté les générateurs de code Ada et C pour l'intergiciel POLYORB-HI. Nous avons aussi implanté les modules qui permettent de déployer et de configurer statiquement les composants de l'intergiciel à partir du modèle AADL. OCARINA permet enfin d'intégrer les composants de l'utilisateur avec ceux générés automatiquement pour les compiler. Cette tâche est effectuée grâce à un ensemble de fichiers de supports (Makefiles...) dont nous avons implanté la production à partir du modèle AADL.

9.2 Conclusions

Pour valider les solutions que nous avons réalisés, nous avons mis en œuvre des études de cas pour valider leur adéquation au problématiques posées au début de ce mémoire. L'avancement de nos travaux ainsi que les résultats satisfaisant des études de cas nous ont permis aussi de contribuer à la future version du standard AADL.

Expérimentations

Plusieurs études de cas ont été réalisées durant notre travail de thèse. Il s'agit, pour la plupart, d'exemples d'applications TR^{2}E du domaine de l'espace proposés par les partenaires industriels du projet Européen ASSERT. Pour ce mémoire, nous repris un exemple classique d'application Ravenscar critique que nous avons étendue pour introduire des fonctions de répartition. Nous avons ensuite adapté cet exemple, issu de [Burns *et al.*, 2004], pour le langage AADL et nous l'avons testé à l'aide d'un simulateur pour la plate-forme LEON2/SPACEWIRE.

Le chapitre 8 décrit en détails cette étude de cas. Il donne aussi les résultats de l'analyse statique d'ordonnançabilité ainsi que les résultats expérimentaux de l'exécution des nœuds de

l'application TR^{2}E. Nous avons donné à la section 8.4.5 différents résultats sur les tailles des sources et des binaires de l'application. Ceci à permis de mettre en évidence le fait que l'instance de l'intergiciel pour chacun des nœuds de l'application était personnalisé en fonction de ses propriétés. Par conséquent les empreintes mémoire des nœuds étaient faibles.

Le succès de ces études de cas et l'avancement réalisé dans l'implantation des solutions proposées nous a permis de contribuer significativement au développement du standard AADLv2 qui sera publié au début de l'année 2009.

Contributions au standards AADL

L'équipe S3 de TELECOM ParisTech s'est vue confier la rédaction de deux annexes de la future version du standard AADL. Il s'agit de l'annexe de modélisations des types de données et celle des règles de transformation de code pour les langages Ada et C depuis le langage AADL.

L'annexe de modélisation des donnée sera inspirée de l'ensemble de propriété spécifique à notre suite d'outils OCARINA qui offre les mêmes fonctionnalité. L'annexe de génération de code, sera inspirée des travaux présentés dans les sections 5.3 et 5.4 de ce mémoire.

9.3 Perspectives

Les travaux présentés dans ce mémoire permettent d'automatiser le processus d'analyse, de production et d'optimisation des applications réparties à partir de leur modèle AADL. Nous nous sommes concentrés sur les couches intergicielle et applicative pour effectuer les optimisations. Cependant d'autres couches de l'application devraient elles aussi être optimisées pour garantir un exécutable dédié au besoins d'une applications TR^{2}E. La future version du standard, AADLv2 permettra la mise en œuvre de telles optimisations. Par ailleurs, des analyses supplémentaires pourraient être effectuées pour renforcer la garantie de bon fonctionnement de l'application.

Passage à AADLv2

Des travaux sont en cours au sein de l'équipe S3 de TELECOM ParisTechpour supporter la nouvelle version du standard AADL dans la suite d'outils OCARINA [Lasnier, 2008]. La nouvelle version du standard offre plusieurs amélioration de la syntaxe et la sémantique des entités existantes (modes opérationnels, connexions, processus légers, éléments d'interfaces...). Elle introduit deux nouvelles catégories de composants : les processeurs virtuels et les bus virtuels. Ces deux composants permettent de raffiner la spécification de la partie matérielle de l'application.

Intégration de la sécurité et la sûreté de fonctionnement

La sécurité et la sûreté de fonctionnement sont des aspects très importants pour les applications critiques. Leur intégration dans la construction d'intergiciels peut être effectuée grâce aux nouvelles fonctionnalités offertes par AADLv2. Des travaux sont en cours dans notre équipe pour intégrer ces deux aspects dans un nouveau noyau. Il s'agit d'intégrer la notion de partitionnement (ARINC 653 [ARINC, 2003]...) au sein du noyau du système et de piloter sa configuration et optimisation à partir du modèle AADL [Delange, 2008].

Optimisation des modèles AADL

Les travaux de ce mémoire ont permis de produire automatiquement des composants applicatifs et intergiciel optimisés selon les propriétés du modèle AADL. Cependant, le modèle AADL lui même devrait être optimisé en fonction des besoins de l'application. Il s'agit par exemple de fusionner les composants **thread** harmoniques ou bien ceux ayant des contraintes de précédence en un seul afin d'optimiser le nombre de tâches et de verrous logiciels dans l'application. L'implantation de ces optimisations constitue une partie des travaux de thèse en cours d'Olivier GILLES au sein de notre équipe de recherche.

Analyses supplémentaires

L'architecture de l'outil OCARINA permet d'incorporer d'autres outils d'analyse pour vérifier le bon fonctionnement des systèmes à partir des modèles AADL. Des travaux sont en cours dans notre équipe pour implanter une nouvelle analyse du pire temps d'exécution [Gilles and Hugues, 2008a] et aussi pour transformer les modèle AADL en des réseaux de Petri pour effectuer de la vérification formelle [Renault *et al.*, 2008].

Le code généré peut, lui aussi, être analysé et vérifié automatiquement. SPARK [Barnes, 2003] est un langage d'annotation pour le code Ada qui permet de spécifier des caractéristiques du code généré (pré-condition, post-condition...) et de les démontrer. Ceci pourra servir à la certification du code généré.

La production d'intergiciels dédiés aux applications TR^{2}E à l'aide d'un processus automatique de développement simplifie et rend plus sûre la conception de ces applications. Nous contribuons ainsi à la thématique de recherche *"usines à intergiciel"* dans le domaine des systèmes temps-réel embarqués.

Bibliographie

[Allen, 1997] Robert John Allen. *A formal approach to software architecture*. PhD thesis, Pittsburgh, PA, USA, 1997. Chair-David Garlan.

[ARINC, 2003] ARINC. 653-1 Avionics Application Software Standard Interface, 2003.

[Armstrong, 1996] Joe Armstrong. Erlang - a survey of the language and its industrial applications. In *In INAP'96 - The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, 1996.

[asn, 2002] REC. X680-X.683, ISO/IEC : Abstract Syntax Notation (ASN.1). Technical report, ITU-T, 2002.

[AUTOSAR Gbr, 2006] AUTOSAR Gbr. Technical Overview. Technical report, 2006.

[Barnes, 2003] John Barnes. *High Integrity Software : The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Barnes, 2008] John Barnes. *Safe and Secure Software, an invitation to Ada 2005*. AdaCore, April 2008.

[Bordin and Vardanega, 2005] Matteo Bordin and Tullio Vardanega. Automated Model-Based Generation of Ravenscar-Compliant Source Code. In *ECRTS '05 : Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 59–67, Washington, DC, USA, 2005. IEEE Computer Society.

[Bruneton *et al.*, 2002] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proceedings of the International Workshop on Component-Oriented Programming (WCOP)*, Malaga, Spain, JUN 2002.

[Bruneton *et al.*, 2006] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java : Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, 36(11-12) :1257–1284, 2006.

[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, 2004.

[Buschmann *et al.*, 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture : A System of Patterns*. John Wiley & Sons, New York, 1996.

[Coffman *et al.*, 1971] E. G. Coffman, M. Elphick, and A. Shoshani. System Deadlocks. *ACM Comput. Surv.*, 3(2) :67–78, 1971.

[Comar and Porter, 1994] Cyrille Comar and Brett Porter. Ada 9X tagged types and their implementation in GNAT. In *TRI-Ada '94 : Proceedings of the conference on TRI-Ada '94*, pages 71–81, New York, NY, USA, 1994. ACM.

[Coupaye *et al.*, 2007] T. Coupaye, V. Quéma, L. Seinturier, and J.-B. Stefani. *Intergiciel et Construction d'Applications Réparties*, chapter Le système de composants Fractal. January 2007.

[Cousot *et al.*, 2005] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyzer. In *ESOP'05*, 2005.

[Curnow *et al.*, 1976] H J Curnow, B A Wichmann, and Tij Si. A Synthetic Benchmark. *The Computer Journal*, 19 :43–49, 1976.

[de la Puente *et al.*, 2000] Juan Antonio de la Puente, José F. Ruiz, and Juan Zamorano. An Open Ravenscar Real-Time Kernel for GNAT. In *Ada-Europe '00 : Proceedings of the 5th Ada-Europe International Conference on Reliable Software Technologies*, pages 5–15, London, UK, 2000. Springer-Verlag.

[Delange *et al.*, 2008] J. Delange, J. Hugues, L. Pautet, and B. Zalila. Code Generation Strategies from AADL Architectural Descriptions Targeting the High Integrity Domain. In *4th European Congress ERTS*, Toulouse, France, jan 2008.

[Delange, 2007] Julien Delange. Génération de Code C pour Applications Critiques. Master's thesis, Université Pierre & Marie Curie, Paris VI, sep 2007.

[Delange, 2008] Julien Delange. Design and implementation of POK, a partitioned kernel and runtime. Technical report, TELECOM ParisTech, 2008.

[Demathieu *et al.*, 2008] Sébastien Demathieu, Frédéric Thomas, Charles André, Sébastien Gérard, and François Terrier. First Experiments Using the UML Profile for MARTE. In *11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC'08)*, pages 50–57, Orlando, Florida, USA, 2008. IEEE Computer Society.

[Déplanche and Faucou, 2006] Anne-Marie Déplanche and Sébastien Faucou. *Systèmes temps réel 1 : Techniques de description et de vérification*, chapter Description d'architectures pour le temps réel : l'approche AADL. Hermes, Lavoisier, 2006.

[Dumant *et al.*, 1998] B. Dumant, F. Horn, F. Dang Tran, and J-B. Stefani. Jonathan : an open distributed processing environment in java. In *IFIP Int'l Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 175–190, Londres, 1998. Springer Verlag.

[Eisler, 2006] M. Eisler. XDR : External Data Representation Standard. RFC 4506 (Standard), may 2006.

[ESTEC, 2003] ESA ESTEC. ECSS-E-50-12A SpaceWire - Links, nodes, routers and networks. Technical report, European Space Agency, 2003.

[ESA, 2007] European Space Agency ESA. ASSERT : Automated proof-based System and Software Engineering for Real-Time systems. http ://www.assert-project.net, 2007.

[FAA, 2004] FAA. *Handbook for Object-Oriented Technology in Aviation (OOTiA)*. FAA, October 2004.

[Feiler *et al.*, 2006] Peter H. Feiler, David P. Gluch, and John J. Hudak. The Architecture Analysis & Design Language (AADL) : An Introduction. Technical report, 2006. CMU/SEI-2006-TN-011.

[Fernàndez-Marina, 2008a] Ramòn Fernàndez-Marina. Gem #33 : Accessibility Checks (Part I : Ada95). http://www.adacore.com/2008/04/28/gem-33/, apr 2008.

[Fernàndez-Marina, 2008b] Ramòn Fernàndez-Marina. Gem #41 : Accessibility Checks (Part II : Ada2005). http://www.adacore.com/2008/06/30/gem-41/, jun 2008.

[Fredriksson, 2002] Lars-Berno Fredriksson. CAN for Critical Embedded Automotive Networks. *IEEE Micro*, 22(4) :28–35, 2002.

[Garlan *et al.*, 1997] David Garlan, Robert T. Monroe, and David Wile. Acme : An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[Gasperoni, 2006] Franco Gasperoni. Safety, security, and object-oriented programming. *SIGBED Rev.*, 3(4) :15–26, 2006.

[Geisler, 2008] Research Geisler. TSIM2 Simulator User's Manual. http ://www.gaisler.com/doc/tsim-2.0.10.pdf, 2008.

[Gilles and Hugues, 2008a] O. Gilles and J. Hugues. Applying WCET analysis at architectural level. In *Worst-Case Execution Time (WCET'08)*, pages 113–122, Prague, Czech Republic, jul 2008.

[Gilles and Hugues, 2008b] O. Gilles and J. Hugues. Validating requirements at model-level. In *Ingénierie Dirigée par les modèles (IDM'08)*, pages 35–49, Mulhouse, France, jun 2008.

[Gokhale *et al.*, 2002] Anirudda Gokhale, Balachandran Natarjan, Douglas C. Schmidt, Andrey Nechypurenko, Nanbor Wang, Jeff Gray, Sandeep Neema, Ted Bapty, and Jeff Parsons. CoSMIC : An MDA Generative Tool for Distributed Real-time and Embdedded Component Middleware and Applications. In *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, WA, November 2002.

[Gomez, 1998] Claude Gomez. *Engineering and Scientific Computing with Scilab*. Birkhauser Boston, 1998.

[Gorappa *et al.*, 2005] S. Gorappa, J. A. Colmenares, H. Jafarpour, and R. Klefstad. Tool-based Configuration of Real-time CORBA Middleware for Embedded Systems. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC'05)*, Seattle, USA, May 2005.

[Gosling and Bollella, 2000] James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co. Inc., 2000.

[Grandpierre and Sorel, 2003] T. Grandpierre and Y. Sorel. From Algorithm and Architecture Specification to Automatic Generation of Distributed Real-Time Executives : a Seamless Flow of Graphs Transformations. In *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003.

[Grandpierre *et al.*, 1999] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized Rapid Prototyping for Real-time Embedded Heterogeneous Multiprocessors. In *Proceedings of the seventh international workshop on Hardware/software codesign*, pages 74–78, New York, NY, USA, 1999. ACM.

[Hugues *et al.*, 2005] Jérôme Hugues, Fabrice Kordon, and Laurent Pautet. Revisiting COTS middleware for DRE systems. In *Proceedings of the 8th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'05)*, pages 72–79, Seattle, WA, USA, May 2005. IEEE.

[Hugues *et al.*, 2008] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. *ACM Transactions in Embedded Computing Systems (TECS)*, 7(4) :1–25, jul 2008.

[Hugues, 2005] Jérôme Hugues. *Architectures et Services des Intergiciels Temps Réel*. PhD thesis, École Nationale Supérieure des Télécommunications, sep 2005.

[IEEE, 1994] Institute of Electrical and Electronics Engineers, Inc. Staff IEEE. *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) : System Application Program Interface (API), Amendment 1 : Realtime Extension (C Language), IEEE Std 1003.1b-1993*. IEEE Standards Office, New York, NY, USA, 1994.

[IEEE, 2000] IEEE. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2000*, Jan 2000.

[Kavimandan *et al.*, 2007] Amogh Kavimandan, Krishnakumar Balasubramanian, Nishanth Shankaran, Aniruddha Gokhale, and Douglas C. Schmidt. Quicker : A model-driven qos mapping tool for qos-enabled component middleware. In *ISORC '07 : Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 62–70, Washington, DC, USA, 2007. IEEE Computer Society.

[Kermarrec *et al.*, 1995] Yvon Kermarrec, Laurent Pautet, and Samuel Tardieu. GARLIC : Generic Ada Reusable Library for Interpartition Communication. In *Proceedings of the conference on TRI-Ada '95*, pages 263–269, New York, NY, USA, 1995. ACM.

[Kermarrec *et al.*, 1996] Y. Kermarrec, L. Nana, and L. Pautet. GNATDIST : A configuration language for distributed ada 95 applications. In *Proceedings of TRI-Ada'96*, pages 63–72. acm-press, 1996.

[Kordon and Luqi, 2002] Fabrice Kordon and Luqi. An Introduction to Rapid System Prototyping. *IEEE Transactions on Software Engineering*, 28(9) :817–821, 2002.

[Kordon and Pautet, 2005] F. Kordon and L. Pautet. Toward next-generation toward next-generation middleware ? *IEEE Distributed Systems Online*, 5(1), 2005.

[Krishnamurthy *et al.*, 2004] Yamuna Krishnamurthy, Irfan Pyarali, Christopher Gill, Louis Mgeta, Yuanfang Zhang, Stephen Torri, and Douglas C. Schmidt. The Design and Implementation of Real-Time CORBA 2.0 : Dynamic Scheduling in TAO. In *RTAS '04 : Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 121, Washington, DC, USA, 2004. IEEE Computer Society.

[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*, pages 131–140, New York, NY, USA, 2002. ACM.

[Lam and Shankar, 1994] S. S. Lam and A. U. Shankar. A Theory of Interfaces and Modules - I : Composition Theorem. *IEEE Trans. Softw. Eng.*, 20(1) :55–71, 1994.

[Lasnier, 2008] Gilles Lasnier. Étude et Support du Standard AADLv2 dans Ocarina. Master's thesis, Université Pierre & Marie Curie, Paris VI, sep 2008.

[Lu *et al.*, 2003] Tao Lu, Emre Turkay, Aniruddha Gokhale, and Douglas C. Schmidt. CoSMIC : An MDA Tool suite for Application Deployment and Configuration,. In *Proceedings of the OOPSLA 2003 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Anaheim, CA, October 2003.

[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, 1995.

[Mead and Conway, 1979] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1979.

[Medvidovic and Taylor, 2000] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Software Engineering*, 26(1) :70–93, 2000.

[Miranda and Schonberg, 2004] Javier Miranda and Edmond Schonberg. *GNAT : The GNU Ada Compiler*. jun 2004. http://www.adacore.com/gap-static/GNAT_Book/gnat-book.pdf.

[Monniaux, 2008] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3) :1–41, 2008.

[Nielsen *et al.*, 2007] Henrik Frystyk Nielsen, Marc Hadley, Anish Karmarkar, Noah Mendelsohn, Yves Lafon, Martin Gudgin, and Jean-Jacques Moreau. SOAP version 1.2 part 1 : Messaging framework (second edition). W3C recommendation, W3C, apr 2007. http ://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[OMG, 2003] OMG. *MDA Guide Version 1.0.1*. OMG, June 2003. omg/2003-06-01.

[OMG, 2004] OMG. *Common Object Request Broker Architecture : Core Specification, Version 3.0.3*. OMG, March 2004. OMG Technical Document formal/04-03-12.

[OMG, 2005] OMG. *Real-time CORBA Specification Version 1.2*. OMG, January 2005. OMG Technical Document formal/05-01-04.

[OMG, 2006a] OMG. *CORBA Component Model Specification Version 4.0*. OMG, avr 2006. OMG Technical Document formal/06-04-01.

[OMG, 2006b] OMG. *Deployment and Configuration of Component-based Distributed Applications Specification, Version 4.0*. OMG, April 2006. OMG Technical Document formal/06-04-02.

[OMG, 2007a] OMG. *Data Distribution Service for Real-time Systems Version 1.2*. OMG, January 2007. OMG Technical Document formal/07-01-01.

[OMG, 2007b] OMG. *OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2*. OMG, November 2007. OMG Technical Document formal/2007-11-04.

[OMG, 2007c] OMG. UML Profile for MARTE, Beta 1, ptc/07-08-04. http ://www.omg.org/cgi-bin/doc ?ptc/2007-08-04, 2007.

[OMG, 2008] OMG. *Common Object Request Broker Architecture (CORBA) for embedded Specification*. OMG, February 2008. OMG Technical Document ptc/2008-02-02.

[Panday *et al.*, 2001] Arjun Panday, Damien Couderc, and Simon Marichalar. AIL : description of a global electronic architecture at the vehicle scale. In *DATE'01 : Proceedings of the conference on Design, automation and test in Europe*, page 112, Piscataway, NJ, USA, 2001. IEEE Press.

[Pautet and Kordon, 2004] Laurent Pautet and Fabrice Kordon. Des vertus de la schizophrénie pour le prototypage d'applications à composants interopérables. *TSI*, 23(10), 2004.

[Pautet and Tardieu, 2000] Laurent Pautet and Samuel Tardieu. GLADE : A Framework for Building Large Object-Oriented Real-Time Distributed Systems. In *ISORC '00 : Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 244, Washington, DC, USA, 2000. IEEE Computer Society.

[Pautet, 2002] 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, jan 2002.

[Pimentel and University, 2006] Juan Pimentel and Kettering University. Designing and Implementing Real-time and Dependable, Embedded Control Applications Using FlexCAN. *IEEE 32nd Annual Conference on Industrial Electronics, IECON 2006*, pages 3650–3655, 2006.

[Pimentel, 2007] Juan R. Pimentel. An Incremental Approach to Task and Message Scheduling for AUTOSAR Based Distributed Automotive Appqlications. In *SEAS '07 : Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*, page 1, Washington, DC, USA, 2007. IEEE Computer Society.

[Puaut, 2002] I. Puaut. Real-time performance of dynamic memory allocation algorithms. *Real-Time Systems, 2002. Proceedings. 14th Euromicro Conference on*, pages 41–49, 2002.

[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, mar 2003.

[Raman *et al.*, 2005] Krishna Raman, Yue Zhang, Mark Panahi, Juan A. Colmenares, Raymond Klefstad, and Trevor Harmon. RTZen : Highly Predictable, Real-Time Java Middleware for Distributed and Embedded Systems. In Gustavo Alonso, editor, *Middleware*, volume 3790 of *Lecture Notes in Computer Science*, pages 225–248. Springer, 2005.

[Renault *et al.*, 2008] Xavier Renault, Jérôme Hugues, and Fabrice Kordon. Toward a Global Approach to Generate Petri Net Specification from Component Model. Technical report, Université Pierre & Marie Curie, 2008.

[Research, 2004] Gaisler Research. LEON2 Processor User's Manual, XST Edition, Version 1.0.23. may 2004.

[RTCA and EUROCAE, 1992] RTCA and EUROCAE. *DO-178B, Software Considerations in Airborne Systems and Equipment Certification*, 1992.

[SAE, 2004] SAE. *Architecture Analysis & Design Language (AS5506)*, September 2004. available at http://www.sae.org.

[SAE, 2008] SAE. *Architecture Analysis & Design Language : Annex Behavior, Draft V2.6*, May 2008.

[Sangiovanni-Vincentelli and Natale, 2007] Alberto Sangiovanni-Vincentelli and Marco Di Natale. Embedded System Design for Automotive Applications. *Computer*, 40(10) :42–51, 2007.

[Schmidt and Cleeland, 2000] D. Schmidt and C. Cleeland. Applying a Pattern Language to Develop Extensible ORB Middleware. In *Design Patterns in Communications (L. Rising, ed.), Cambridge University Press, 2000.*, 2000.

[Schmidt *et al.*, 1998] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the tao real-time object request broker. volume 21, pages 294–324, 1998.

[Schmidt *et al.*, 2000] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture – Volume 2 – Patterns for Concurrent and Networked Objects*. Wiley & Sons, New York, NY, USA, 2000.

[Schreiner and Goschka, 2007] Dietmar Schreiner and Karl M. Goschka. A Component Model for the AUTOSAR Virtual Function Bus. In *COMPSAC'07 : Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 2- (COMPSAC 2007)*, pages 635–641, Washington, DC, USA, 2007. IEEE Computer Society.

[Sha *et al.*, 1990] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9) :1175–1185, 1990.

[Sha *et al.*, 1993] L. Sha, M. H. Klein, and J. B. Goodenough. Rate Monotonic Analysis for Real-Time Systems. *Computer*, 26(3) :73–74, 1993.

[Shankaran *et al.*, 2007] Nishanth Shankaran, Douglas C. Schmidt, Xenofon D. Koutsoukos, Yingming Chen, and Chenyang Lu. Design and performance evaluation of configurable component middleware for end-to-end adaptation of distributed real-time embedded systems. In *ISORC '07 : Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 291–298, Washington, DC, USA, 2007. IEEE Computer Society.

[Singhoff *et al.*, 2004] F. Singhoff, J. Legrand, L. Nana Tchamnda, and L. Marcé. Cheddar : a Flexible Real Time Scheduling Framework. *ACM Ada Letters journal, 24(4) :1-8, ACM Press. Also published in the proceedings of the ACM SIGADA International Conference, Atlanta, 15-18 November, 2004*, November 2004.

[Sorel, 2004] Y. Sorel. SynDEx : System-Level CAD Software for Optimizing Distributed Real-Time Embedded Systems. *Journal ERCIM News*, 59 :68–69, October 2004.

[Sorel, 2005] Y. Sorel. From modeling/simulation with Scilab/Scicos to optimized distributed embedded real-time implementation with SynDEx. In *Proceedings of the International Workshop On Scilab and Open Source Software Engineering, SOSSE'05*, Wuhan, China, October 2005.

[Sriplakich *et al.*, 2008] Prawee Sriplakich, Xavier Blanc, and Marie-Pierre Gervals. Collaborative software engineering on large-scale models : requirements and experience in ModelBus. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 674–681, New York, NY, USA, 2008. ACM.

[SUN, 1988] SUN. RPC : remote procedure call protocol specification Version 2 ; RFC1058. *Internet Request for Comments*, (1057), 1988.

[SUN, 2006] SUN. Java Software Development Kit Version 1.6, 2006. http://java.sun.com/javase/6.

[Sztipanovits and Karsai, 1997] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *Computer*, 30(4) :110–111, 1997.

[The ASSERT Consotium, 2005] The ASSERT Consotium. The ASSERT Project Technical Annex, Annex1 I1R1 051118. Technical report, European Space Agency, 2005.

[Tindell, 1993] Ken Tindell. Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. Technical report, Univerity of York, 1993.

[Urueña and Zamorano, 2007] Santiago Urueña and Juan Zamorano. Building high-integrity distributed systems with Ravenscar restrictions. *Ada Lett.*, XXVII(2) :29–36, 2007.

[Vergnaud *et al.*, 2005] T. Vergnaud, L. Pautet, and F. Kordon. Using the AADL to describe distributed applications from middleware to software components. In *Reliable Software Technologies (RST'05)*, York, UK, jun 2005.

[Vergnaud *et al.*, 2006] T. Vergnaud, B. Zalila, and J. Hugues. Ocarina : a Compiler for the AADL. Technical report, École Nationale Supérieure des Télécommunications, jun 2006.

[Vergnaud, 2006] Thomas Vergnaud. *Modélisation des Systèmes Temps-réel Répartis Embarqués pour la Génération Automatique d'Application Formellement Vérifiées*. PhD thesis, École Nationale Supérieure des Télécommunications, dec 2006.

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

[Vinoski, 2002] Steve Vinoski. Middleware "Dark Matter". *IEEE Internet Computing*, 6(5) :92–95, 2002.

[Vinoski, 2008] Steve Vinoski. Convenience Over Correctness. *IEEE Internet Computing*, 12(4) :89–92, 2008.

[Waldo *et al.*, 1994] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. Technical report, Mountain View, CA, USA, 1994.

[Wilhelm and Maurer, 1995] Renhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[Working Group, 2005] Ada Working Group. *Ada Reference Manual*. ISO/IEC, 2005. Available at http://www.adaic.com/standards/05rm/RM-Final.pdf.

[Zalila *et al.*, 2006] B. Zalila, J. Hugues, and L. Pautet. An Improved IDL Compiler for Optimizing CORBA Applications. *ACM SIGAda Ada Letters*, XXVI(3) :21 – 27, dec 2006.

[Zalila *et al.*, 2008] B. Zalila, L. Pautet, and J. Hugues. Towards Automatic Middleware Generation. In *11th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'08)*, pages 221–228, Orlando, Florida, USA, may 2008.

[Zalila, 2005] Bechir Zalila. Optimisation, Déterminisme et Asynchronisme de Souches et Squelettes CORBA pour Systèmes Répartis Temps-réel. Master's thesis, Université Pierre & Marie Curie, Paris VI, sep 2005.

[Zelesnik, 1996] Gregory Zelesnik. The UNICON Language Reference Manual. Technical report, Pittsburg, 1996.

[Zomaya, 1996] Albert Y. H. Zomaya, editor. *Parallel and distributed computing handbook*. McGraw-Hill, Inc., New York, NY, USA, 1996.