Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées

Thomas Vergnaud

École doctorale d'informatique, télécommunications et électronique de Paris

Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées

Thèse de doctorat soutenue le 1er décembre 2006 par

Thomas VERGNAUD

pour obtenir le grade de

docteur de l'École nationale supérieure des télécommunications
– spécialité informatique & réseaux –

Composition du jury :

rapporteurs :

Yvon KERMARREC, professeur à l'École nationale supérieure des télécommunications de Bretagne
Jean-François PRADAT-PEYRE, maître de conférence au Conservatoire national des arts et métiers, habilité à diriger des recherches

examinateurs :

Marie-Pierre GERVAIS, professeur à l'université Paris 10 – Nanterre
Frank SINGHOFF, maître de conférence à l'université de Bretagne occidentale

directeurs de thèse :

Laurent PAUTET, professeur à l'École nationale supérieure des télécommunications de Paris
Fabrice KORDON, professeur à l'université Paris 6 – Pierre & Marie Curie

Telle est la nature de l'esprit humain, telles sont les limites de sa science propre, qu'il n'y a jamais lieu à faire des découvertes toutes nouvelles, mais seulement à éclaircir, vérifier, distinguer dans leur propre source certains faits de sens intime, faits simples, liés à notre existence, aussi anciens qu'elle, aussi évidents, mais qui s'y trouvent enveloppés avec diverses impressions hétérogènes qui les rendent vagues et obscurs.

Maine DE BIRAN, *in* Essai sur les fondements de la psychologie

Résumé – Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées

La construction d'une application répartie fait en général intervenir une couche logicielle particulière, appelée intergiciel, qui prend en charge la transmission des données entre les différents nœuds de l'application. L'usage d'un intergiciel pouvant être adapté aux conditions particulières d'une application donnée s'avère être un bon compromis entre performances et coût de développement.

La conception d'applications pour les systèmes embarqués temps-réel implique la prise en compte de certaines contraintes spécifiques à ce domaine, que ce soit en terme fiabilité ou de dimensions à la fois temporelles et spatiales. Ces contraintes doivent notamment être respectées par l'intergiciel.

L'objet de ces travaux est la description des applications temps-réel réparties embarquées en vue de configurer automatiquement l'intergiciel adéquat. L'étude se focalise sur la définition d'un processus de conception permettant d'intégrer les phases de description, de vérification et de génération de l'application complète.

Pour cela, nous nous reposons sur le langage de description d'architecture AADL. Nous l'exploitons comme passerelle entre la phase de description de l'architecture applicative, les formalismes de vérification, la génération du code exécutable et la configuration de l'exécutif réparti. Nous montrons comment spécifier un exécutif pour AADL afin de produire automatiquement le code applicatif et l'intergiciel pour une application répartie. Nous montrons également comment exploiter ces spécifications pour produire un réseau de Petri afin d'étudier l'intégrité des flux d'exécution dans l'architecture.

Afin de valider notre processus de conception, nous avons conçu et développé Ocarina, un compilateur pour AADL qui utilise l'intergiciel schizophrène PolyORB comme exécutif.

Abstract – Modelling Distributed Real-Time Embedded Systems for the Automatic Generation of Formally Verified Applications

Building distributed applications usually relies on a particular software layer, called middleware, that manages data transmission between application nodes. Using a middleware that can be adapted to the particular requirements of a given application provides a good trade-off between execution performance and development cost.

Applications for embedded real-time systems must take specific parameters into account, such as reliability concerns or temporal and spatial constraints. As part of the application, the middleware has to respect these constraints also.

This work focuses on the description of distributed embedded real-time applications in order to automatically configure the associated middleware. We define a design process that integrates different steps for the description, verification and generation of the complete application.

We use the Architecture Analysis & Design Language (AADL) as a backbone between the application description, the verification formalisms, the code generation and the runtime configuration. We show how to specify an AADL runtime to automatically generate the application code and the middleware for a given distributed application. We also show how to generate Petri nets from AADL descriptions an how it helps in the verification of the integrity of the execution flows in architectures.

To validate this design process, we implemented Ocarina, an AADL compiler that uses PolyORB, a schizophrenic, highly tailorable middleware, as a runtime.

Table des matières

Avant-propos

Les travaux présentés dans cette thèse couvrent différents domaines : modélisation, méthodes formelles et génération de code. Ils ne traitent pas directement des systèmes embarqués ni des applications temps-réel ; les travaux dans ces domaines sont déjà nombreux. Il s'agit plutôt d'étudier la description des communications dans le cadre de tels systèmes, afin de produire automatiquement des applications réparties.

Je dois tout d'abord exprimer ma reconnaissance à mes directeurs de thèse Laurent PAUTET et Fabrice KORDON. Ils ont su m'orienter tout en me laissant une grande autonomie dans mes recherches. Ils y ont parfois cru plus que moi. J'ai beaucoup appris à leur contact, tant au point de vue scientifique qu'en matière d'organisation ou tout simplement sur le plan humain. Qu'ils en soient ici remerciés.

Je remercie également mes rapporteurs Yvon KERMARREC et Jean-François PRADAT-PEYRE pour avoir lu attentivement mon mémoire de thèse et apporté des remarques pertinentes pour son amélioration, ainsi que Marie-Pierre GERVAIS et Frank SINGHOFF, qui ont bien voulu être mes examinateurs.

Merci à Bruce LEWIS et Peter FEILER, du comité de standardisation d'AADL. Ils ont toujours été à l'écoute de mes questions puis de mes propositions relatives au langage.

Je tiens aussi à remercier Minh VO, David ABAD-GARCIA et Olivier GILLES pour leur contribution au développement d'Ocarina. Le projet n'aurait pas avancé aussi vite sans leur aide.

Merci à Lucile DENŒUD et Éric VARADARADJOU, mes collègues du bureau C255 à l'ENST. J'ai beaucoup apprécié leur compagnie, les blagues, les projections d'objet divers à travers la pièce ou les discussions interminables pour savoir si une table communique – même lorsqu'il n'y a pas de table. Merci aussi à tous ceux qui sont passés par ce bureau, et qui y résident encore pour certains : Martina LYČKOVA, Guilhem PAROUX, Ludovic MARTIN, Hoa HA DUONG, Alexandre SZTYKGOLD… Aller au bureau a toujours été un plaisir.

Merci aussi, naturellement, à mes collègues de l'ENST : Irfan HAMID, Khaled BARBARIA, Bechir ZALILA, Jérôme HUGUES, et plus particulièrement Isabelle PERSEIL, qui s'est proposée pour relire mon mémoire et m'a aidé à l'enrichir. J'ai beaucoup apprécié de travailler avec eux. Toutes mes pensées vont aussi aux autres membres du département informatique & réseaux ; la liste est trop longue pour tous les citer, mais je ne les oublie pas.

Merci enfin aux chercheurs et doctorants du laboratoire d'informatique de Paris 6 : Emmanuel PAVIOT-ADET, Xavier RENAUD Alexandre HAMEZ, Lom HILLAH, Alban LINARD, Mathieu BOUILLAGUET et tous les autres pour leur aide sur les phases d'expérimentation avec les réseaux de Petri, leur bonne humeur et tout simplement pour leur présence.

CHAPITRE PREMIER

Introduction

LE BON FONCTIONNEMENT d'une application répartie repose sur l'utilisation d'une infrastructure de communication adaptée. Cette structure prend habituellement la forme d'une couche logicielle intercalée entre le système d'exploitation et l'application – d'où son nom d'*intergiciel*.

I-1 Adaptabilité des intergiciels

La mise en place de la couche de communication doit se faire en adéquation avec les besoins fonctionnels de l'application. Il est techniquement possible d'utiliser un intergiciel quelconque pour faire fonctionner tout type d'application ; cependant, les performances d'un système reposant sur un intergiciel inadapté seront vraisemblablement médiocres, soit parce que l'intergiciel met en place des mécanismes inutilement complexes pour assurer un service qui se révèle simple, soit au contraire parce qu'il ne fournit pas tous les services attendus, qui doivent donc être pris en charge par l'application [Mullender, 1993].

L'une des problématiques majeures de l'étude des intergiciels est donc de pouvoir concevoir une infrastructure de communication qui corresponde aux caractéristiques de l'application.

Différents modèles de répartition ont été développés afin de répondre aux besoins spécifiques des différentes classes d'application. Par exemple, les objets distribués et leur évolution en modèle à composants permettent un niveau d'abstraction assez élevé, bien adapté à des domaines d'application comme les systèmes d'information ; le passage de message, au contraire, met en place des concepts de plus bas niveau et permet une plus grande souplesse dans les communications, et est donc employé dans des domaines comme le calcul parallèle.

Différentes implantations d'intergiciels spécialisés ont été conçus pour répondre aux différents besoins identifiés. Par exemple, plusieurs intergiciels basés sur le passage de messages existent : les différentes implantations de MPI [Gropp et Lusk, 1999 ; Graham et al., 2005] sont destinées au calcul parallèle, tandis que JMS [Kleijnen et Raju, 2003] est plus axé vers les systèmes d'information. Ces différentes implantations cantonnent l'utilisation de l'intergiciel à un domaine d'application précis. La spécialisation d'un intergiciel va alors de pair avec la restriction de son domaine d'utilisation. Pour des besoins applicatifs très particuliers, cette approche nécessite de concevoir un intergiciel sur mesure, ce qui s'avère trop coûteux.

Afin de limiter les coûts de développement d'un intergiciel correspondant aux besoins d'une application donnée, diverses solutions ont été proposées [Pautet, 2001]. Les intergiciels configurables se basent sur un modèle de distribution donné et permettent une sélection de paramètres afin d'en affiner l'implantation. L'approche suivie par les intergiciels génériques étend ces concepts afin de faciliter la création d'un intergiciel personnalisé pour un modèle de distribution donné.

L'architecture d'intergiciel schizophrène [Pautet, 2001 ; Quinot, 2003] a été proposée comme généralisation de ces différentes approches. Elle repose sur une structure en trois couches (applicative, neutre et protocolaire) permettant la création d'intergiciels extrêmement configurables. La couche neutre constitue l'élément central de l'intergiciel ; elle met en place une structure canonique d'intergiciel basée sur un assemblage de composants, permettant ainsi une grande configurabilité. La couche applicative permet de personnaliser l'intergiciel afin de présenter aux applications des interfaces adaptées. La couche protocolaire peut gérer différents protocoles afin de garantir une interopérabilité optimale avec les autres nœuds de l'application répartie. Un tel intergiciel est capable de prendre en charge simultanément plusieurs personnalités – d'où l'appellation « schizophrène ».

Les travaux autour de l'architecture schizophrène ont été concrétisés par le projet PolyORB [Quinot, 2003 ; Hugues, 2005], qui consiste est une infrastructure appliquant les principes de la schizophrénie. PolyORB permet de construire un intergiciel adapté à une grande variétés d'applications en limitant le coût de développement.

Dans le cadre de systèmes embarqués temps-réel, la mise en place d'un intergiciel de communication doit aussi répondre à des impératif non fonctionnels tels que les dimensions de l'application (temps d'exécution, taille mémoire) ou la fiabilité.

La prise en compte de ces critères est relativement délicate, étant donné la complexité intrinsèque des intergiciels. De fait, si le fonctionnement des applications fait éventuellement l'objet de modélisations formelles, ce n'est pas le cas des intergiciels sous-jacents, dont la vérification se limite souvent à l'exécution de tests sur l'ensemble constitué par l'application et son intergiciel. Cette approche, si elle peut permettre de détecter certains problèmes, ne garantit pas le fonctionnement de l'intergiciel et le respect des contraintes demandées par l'application ; la gestion des communications peut donc perturber le fonctionnement de l'application de façon imprévue.

Les recherches sur l'architecture schizophrène ont été prolongées afin d'exploiter l'organisation canonique proposée par l'architecture schizophrène. Les travaux ont notamment été orientés vers la rationalisation de l'architecture afin de permettre la construction d'un intergiciel vérifiable, pouvant être utilisé pour un environnement temps-réel [Hugues, 2005].

I-2 Vers une démarche systématique pour la construction d'intergiciels adaptés

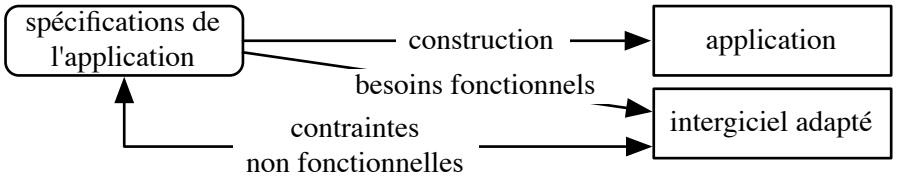


FIG. I.1 – Extraction des paramètres de l'application pour construire une infrastructure de communication adaptée

La mise en place d'un intergiciel pour un système temps-réel réparti embarqué (TR^{2}E) nécessite la prise en compte de nombreux facteurs. La conception manuelle de ce genre d'intergiciel constitue une tâche complexe, et donc coûteuse. Il est nécessaire de définir un processus de conception permettant d'assister la conception et la réalisation de tels intergiciels, comme nous

l'illustrons sur la figure I.1. Un tel processus permet de déduire la structure d'un intergiciel adapté en fonction des paramètres fonctionnels, tels que le modèle de distribution adéquat ou la politique d'ordonnancement. Il permet également la prise en compte des contraintes exprimées sur l'application, telles que les temps d'exécution ; en retour, il doit être possible de corriger les contraintes non fonctionnelles s'il n'est pas possible de les respecter dans l'implantation finale.

Ce travail de thèse s'attache à décrire une méthodologie pour la construction d'une application et de l'intergiciel de communication correspondant ; nous nous plaçons plus particulièrement dans le cadre de systèmes temps-réel répartis embarqués. Nous nous appuyons pour cela sur les travaux mené autour de l'architecture d'intergiciel schizophrène ; nous en reprenons les principes pour constituer un intergiciel adapté aux caractéristiques de l'application considérée.

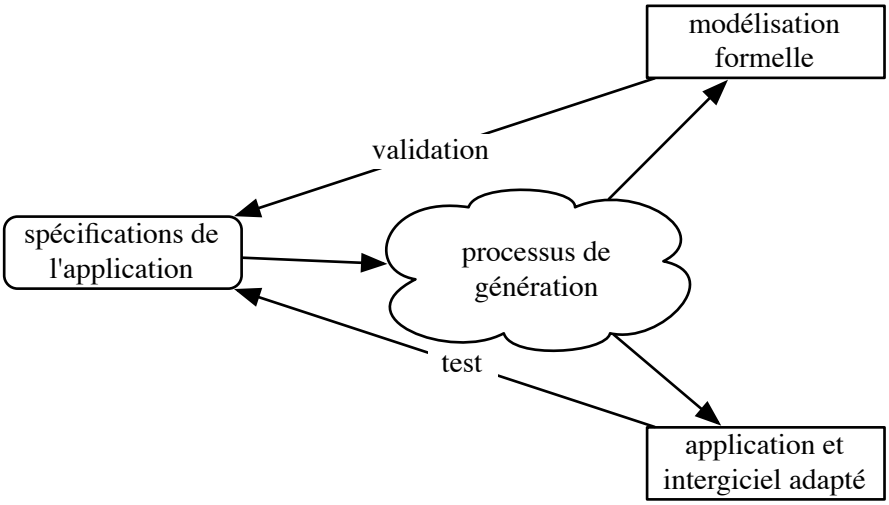


FIG. I.2 – Mise en place d'un processus pour la configuration et la génération d'un intergiciel adapté

Notre proposons un processus de génération permettant de produire automatiquement une application répartie vérifiée. Cette problématique peut se décomposer en plusieurs étapes, résumées par la figure I.2.

La première étape consiste à extraire les caractéristiques de l'application que nous devons prendre en compte pour spécifier l'intergiciel. Il est pour cela nécessaire de recourir à un formalisme permettant de décrire l'application elle-même, afin d'en exprimer toutes les caractéristiques, tant fonctionnelles (notamment les modèles de répartition à mettre en œuvre) que non fonctionnelles (dimensions temporelles et spatiales). L'usage d'un langage de description d'architecture apporte une solution pertinente à ce problème ; de tels languages rendent en effet possible la description tous les aspects de l'application. Notre démarche s'appuie sur AADL, un langage destiné à la description des systèmes embarqués temps-réel. AADL se distingue notamment par sa capacité à rassembler au sein d'une même notation l'ensemble des informations concernant l'organisation de l'application et son déploiement. Il permet donc de spécifier à la fois les applications et leur intergiciel.

La description de l'application est ensuite exploitée afin d'en déduire une configuration adéquate pour l'intergiciel de communication. Il est alors possible d'interpréter la description afin d'en extraire une description formelle propre à l'analyse ou la vérification comportementale ; nous pouvons alors vérifier formellement l'ensemble formé par l'application et l'intergiciel qui sera

créé. Dans le cadre de nos travaux, nous nous focalisons sur les réseaux de Petri. En fonction des résultats des analyses formelles, cette étape peut conduire à un raffinement de la description de l'application afin de préciser la structure de l'application.

Tout au long du processus de conception, il doit être possible d'exploiter la description de l'application selon différents aspects ; ceci permet de s'assurer de la validité de l'architecture. Nous décrivons des règles de génération pour différents langages de programmation, plus particulièrement Ada, C et Java. Nous présentons également des règles de traduction pour la production d'un réseau de Petri, de sorte qu'il soit possible d'analyser certaines propriétés comportementales. Le système ainsi obtenu peut être testé afin de s'assurer du respect des contraintes non fonctionnelles spécifiées dans la modélisation initiale.

I-3 Organisation de la thèse

Notre thèse est présentée comme suit. Dans le chapitre II, nous étudions d'abord différentes approches utilisées pour la configuration des intergiciels. La configuration des intergiciels peut être abordée sous différents aspects, qui peuvent éventuellement être complémentaires. Nous montrons comment les langages de description d'architecture constituent une bonne solution à notre problématique. Le chapitre III conclut notre état de l'art par une étude détaillée du langage AADL. Nous montrons comment ce langage peut être exploité pour décrire précisément l'organisation d'une application ; une telle précision de description permet d'extraire toutes les informations nécessaires à la mise en place d'un intergiciel adapté. Nous présentons également les évolutions de syntaxe que nous avons proposées pour faciliter la description de l'assemblage des applications et les communications.

Le chapitre IV présente la façon dont nous exploitons AADL dans le cadre de nos travaux. Nous étudions tout d'abord la pertinence d'AADL comme langage de configuration pour décrire efficacement une application et son déploiement. Nous montrons ainsi comment AADL peut être intégré au sein d'outils existants pour fournir un langage de description portable entre différents outils.

Nous montrons ensuite comment mettre en place un processus de conception complet pour la vérification et la production automatique d'une application et de l'intergiciel associé. La mise en place d'un tel processus implique l'identification des constructions architecturales décrites en AADL et la spécification d'un exécutif. Nous décrivons un ensemble de directives architecturales permettant de guider l'utilisateur dans la description de l'application et nous donnons leur signification en terme de spécification comportementale.

Les chapitres V et VI détaillent la façon dont nous traduisons une description AADL en une application exécutable. Nous y exploitons les concepts introduits dans le chapitre IV pour traduire les constructions AADL en langage de programmation et configurer l'intergiciel correspondant. Nous nous basons pour cela sur l'architecture d'intergiciel schizophrène et son implantation de référence, PolyORB.

Le chapitre VII présente finalement l'exploitation que nous pouvons faire d'une description AADL pour vérifier formellement l'architecture correspondante. Nous utilisons le *model checking* pour valider la structure de l'application vis-à-vis des flux d'exécution. Les constructions formelles que nous déduisons de la description AADL se basent sur les mêmes spécifications d'exécutif que celles que nous avons définies pour la génération de code. Nous pouvons ainsi analyser le comportement de l'application qui sera générée.

Le chapitre VIII constitue une mise en application de notre méthodologie. Nous présentons

Ocarina, un compilateur AADL que nous avons développé pour générer une description formelle ou du code source à partir des directives architecturales décrites dans le chapitre IV. Nous comparons les performances d'une application construite automatiquement à partir de sa description en AADL et d'une implantation classique basée sur CORBA. Nous considérons également une étude de cas de taille réelle dont nous tirons une modélisation formelle afin d'évaluer la viabilité de notre approche.

CHAPITRE II

Description de la configuration d'une application répartie

LA RELATION ENTRE L'APPLICATION et l'intergiciel peut être envisagée de plusieurs façon différentes. Compte tenu de la très grande diversité des recherches menées autour des intergiciels et de la modélisation des applications, nous ne nous attachons pas à présenter un panorama exhaustif des travaux relatifs à ces domaines. Nous présentons une synthèse des différentes approches qui peuvent être appliquées pour la configuration des intergiciels ; nous en dégageons une évolution dans la démarche de construction des intergiciels, et la relation qui peut être établie avec le processus de construction des applications.

II-1 Adaptation des intergiciels

Dans le cadre de notre étude, nous définissons l'intergiciel comme étant une couche intermédiaire située sur chaque nœud, entre le système d'exploitation centralisé et la partie applicative. Le rôle de l'intergiciel est de permettre à la partie applicative de communiquer avec ses pairs sur les autres nœuds.

II-1.1 Relation entre l'intergiciel et l'application

L'intergiciel tient un rôle central dans les performances de l'application globale. Le choix de l'intergiciel à utiliser dépend donc de l'application ; il doit être en adéquation avec les besoins de celle-ci. Nous pouvons dénombrer trois axes d'adaptation de l'intergiciel :

- capacité de l'intergiciel à s'interfacer avec l'application ;
- prise en compte des informations de localisation de chaque nœud ;
- configuration des mécanismes internes de l'intergiciel vis-à-vis de la qualité de service attendue.

À ces trois aspects s'ajoute la nécessité de pouvoir garantir un certain nombre de propriétés telles que l'absence de blocages ou de famines dans le traitement des communications. Le processus de construction de l'intergiciel doit donc pouvoir intégrer des possibilités d'analyse.

II-1.2 Description des caractéristiques de l'application

L'adaptation de l'intergiciel selon ces différents aspects nécessite l'utilisation d'un formalisme de description pour l'application. Ce formalisme doit permettre de décrire les différentes caractéristiques de l'application ayant un impact sur la configuration de l'intergiciel.

Dans les sections suivantes, nous étudions différentes solutions pour la description d'une application répartie. Nous montrons l'évolution de ces solutions, depuis les langages de description d'interface, centrés sur les intergiciels, jusqu'aux langages de modélisation, centrés sur les applications. Nous évaluons l'efficacité de chaque approches vis-à-vis des critères que nous avons énoncés.

II-2 Approches centrées sur l'intergiciel

Pour la mise en place d'une application répartie, de nombreuses solutions consistent à accorder à l'intergiciel une place centrale. La couche de communication influe alors sur la façon dont l'application est construite.

II-2.1 Bibliothèque de communication

Nous pouvons considérer que l'un des intergiciels les plus simples est l'interface *socket*, permettant à un programme d'utiliser une pile de protocoles de type Internet. Une bibliothèque de *sockets* fournit une interface permettant d'effectuer des opérations primitives, telles que l'écoute ou l'envoi de messages (paquets), le calcul de l'adresse IP d'un nœud à partir de son nom (DNS), etc. Les *sockets* se caractérisent notamment par le fait que toutes les opérations évoluées – notamment la transmission d'un type de donnée déterminé – sont laissées au concepteur de l'application.

L'application doit s'adapter à l'interface des *sockets* en ce qui concerne le format des données à transférer ; notamment, l'utilisateur doit prendre en compte les éventuelles différences de représentation des différents nœuds. La localisation des nœuds est également laissée à l'initiative de l'application, même si un service d'annuaire est en général disponible (DNS, *Domain Name Service*). De part son caractère minimal, une *socket* ne fournit aucune capacité d'adaptation ; là encore, l'application effectue toutes les opérations de communication et d'exécution.

Bien que primitive, cette approche permet d'obtenir des très bonnes performances à l'exécution, dans la mesure où l'utilisateur possède un contrôle quasi-total sur le déroulement des communications. Ce principe est d'ailleurs repris dans certains intergiciels plus évolués, tels que MPI (Message Passing Interface).

MPI [MPI, 2003] définit une interface d'intergiciel pour le calcul parallèle. Il s'agit donc d'une spécification d'intergiciel pour un usage hautement spécialisé, où la rapidité d'exécution est primordiale [Graham et al., 2005]. Pour cela, une implantation de MPI fournit un certain nombre de primitives pour la transmission de types de données prédéfinis. La mise en place de mécanismes pour la transmission de types de données personnalisés est laissée aux soin de l'application elle-même. Du fait de son champ d'application extrêmement spécialisé, une implantation de MPI est systématiquement optimisée pour la rapidité de transmission, et ne fournit pas de service supplémentaire tel qu'une gestion d'ordonnancement, etc. La localisation des nœuds est prise en charge par l'intergiciel, en général sous forme de fichier de configuration décrivant l'emplacement des différents nœuds sur le réseau ; la bibliothèque MPI s'occupe alors de déployer les nœuds sur les machines indiquées.

II-2.2 Adaptation de l'intergiciel aux interfaces de l'application

L'utilisation d'un intergiciel se limitant à une bibliothèque de communication figée permet d'obtenir de très bonnes performances en terme d'exécution. Cependant, ces avantages s'accompagnent d'un coût très important au niveau de la conception de l'application. Celle-ci doit en

effet assurer toutes les opérations de transmission évoluées, telles que la manipulation de types de données particuliers, la mise en place du paradigme client/serveur, etc.

Afin de faciliter le développement de l'application, il est nécessaire de recourir à un intergiciel dont l'interface avec l'application peut être adaptée en fonction des types de données à transporter.

II-2.2.1 RPC

L'un des outils de configuration les plus simples mettant en place cette approche est rpcgen, associé à l'intergiciel d'invocation de procédure distante RPC [Bloomer, 1992]. RPC (*Remote Procedure Call*) est une spécification d'intergiciel basé sur le paradigme client/serveur ; son implantation par Sun Microsystem prend en charge essentiellement les langages C et C++. Il définit un protocole de communication, ainsi qu'une représentation commune des données transmises, facilitant ainsi la mise en place d'applications réparties mettant en jeu des nœuds de différentes architectures matérielles [Srinivasan, 1995a,b].

Il est en théorie possible d'utiliser la bibliothèque de communication RPC directement à partir de l'application. Du fait de la complexité des services à utiliser, il s'agit néanmoins d'une manipulation complexe. L'implantation RPC de Sun fournit donc un outil, rpcgen, permettant la génération de la couche d'adaptation entre l'intergiciel et l'application. L'utilisateur décrit les interfaces des éléments applicatifs à répartir à l'aide d'un langage spécialisé appelé RPCL. Le listing II.1 donne un exemple de description d'une méthode distante effectuant une addition.

```
1  struct parametres {
2    int a;
3    int b;
4  };
5
6  program CALCUL_PROG {
7    version CALCUL_V1 {
8      int addition (parametres) = 1;
9    } = 1;
10 } = 0x222222FF;
```

Listing II.1 – description RPCL d'une application

Nous devons définir un programme (`CALCUL_PROG`) et une version (`CALCUL_V1`), auxquels nous associons respectivement une référence et un nombre ; ces valeurs sont utilisées par le gestionnaire RPC pour établir une correspondance entre les appels du client et les services offerts par le serveur. Nous pouvons noter que rpcl n'autorise qu'un seul paramètre par fonction, ce qui nous oblige à définir une structure de donnée (`parametres`). À partir d'une telle description, rpcgen produit deux séries de fichiers :

- les souches, matérialisant les subrogés locaux des entités distantes, utilisés par le client ;
- les squelettes, correspondant aux implantations des entités sur le serveur.

Les souches permettent de présenter à l'application une interface de haut niveau ; elles prennent en charge la conversion et la sérialisation des données à transmettre, ainsi que les mécanismes de transmission. Les squelettes effectuent les opérations inverses, et appellent le code applicatif que l'utilisateur associe aux procédures distantes considérées.

Le langage rpcl ne décrit que les interfaces de procédures ; il ne permet pas de décrire l'emplacement des nœuds de l'application. Il ne fait donc que faciliter l'écriture du client et du serveur.

II-2.2.2 CORBA

Les principes de RPC ont été étendus à la programmation orientée objet, notamment à travers les spécifications de CORBA [Vinoski, 1993]. CORBA (*Common Object Request Broker Architecture*) est un ensemble de spécifications issus des travaux de l'OMG (*Object Management Group*) qui reprend les mêmes principes de souche et de squelette que RPC, ainsi que l'utilisation d'un langage de description des interfaces (IDL, *Interface Description Language*).

```
1 interface Calcul {
2   short addition (in short a, in short b);
3 };
```

Listing II.2 – description IDL d'une application

Le listing II.2 illustre la déclaration d'un objet réparti offrant une méthode permettant d'effectuer des additions. Nous voyons que l'approche d'IDL est relativement proche de celle de RPCL ; l'interface remplace le programme, la notion de version disparaît et les aspects orientés objets sont ajoutés. Par rapport à RPCL, IDL supprime l'obligation d'associer un nombre aux méthodes ; il permet également de définir des méthodes ayant plusieurs paramètres.

Dans sa conception, CORBA ambitionne de fournir une solution standard pour la mise en place de systèmes répartis. Contrairement à RPC, qui se focalise sur le langage C, CORBA vise donc à faciliter l'interopérabilité entre des nœuds applicatifs implantés dans des langages de programmation différents. Pour cela, CORBA définit des règles de transformation entre IDL et les langages d'implantation tels que Java, C++, Ada, Python...afin d'intégrer un large éventail d'applications. L'exploitation d'IDL rejoint néanmoins celle de RPCL : la description des interfaces est transformée en un ensemble de souches et de squelettes.

Un intergiciel CORBA prend en charge la localisation des nœuds de l'application. À son lancement, un serveur enregistre auprès de l'ORB les objets qu'il met à la disposition des clients. Pour chaque objet, l'ORB crée une référence que le serveur doit faire parvenir aux clients. Une référence rassemble toutes les informations nécessaires pour l'identification d'un objet : emplacement du nœud sur le réseau, identificateur de l'objet sur ce nœud, etc.

Les références peuvent être transmises aux clients sous forme de chaîne de caractères. CORBA définit principalement deux codages de références : les IOR (*Interoperable Object Reference*) permettent de coder une référence sous forme d'une chaîne de caractère opaque ; les CorbaLoc fournissent une fonctionnalité équivalente avec une présentation similaire aux URL. Il est nécessaire de fournir au client l'IOR ou la CorbaLoc de l'objet serveur ; l'ORB du client peut alors établir la communication avec l'ORB du serveur.

Les spécifications CORBA définissent également un service de nommage (CosNaming). Un tel service correspond à un objet rassemblant les références des différents objets du système réparti. Chaque entité du système doit donc disposer de l'IOR du service de nommage ; elle s'adressera ensuite à ce service pour déclarer ses objets serveurs ou demander la référence d'un objet dont elle connaît le nom.

Dans un cas comme dans l'autre, la localisation des nœuds n'est connue qu'au moment de l'exécution, et n'est donc pas consignée dans la description IDL. L'intervention de l'utilisateur est donc nécessaire.

Les spécifications initiales de CORBA visaient la mise en place de systèmes d'information. Plusieurs sous-spécifications de CORBA ont été définies afin de répondre à des domaines d'application particuliers ayant des contraintes spécifiques, comme les systèmes embarqués (Minimum CORBA) ou temps-réel (RT-CORBA)...

Il existe de nombreuses implantations des spécifications CORBA ; peu d'entre elles permettent l'adaptation de l'intergiciel aux besoins particuliers de l'application en terme d'efficacité d'exécution.

TAO

Dans certains domaines d'application, comme les systèmes temps-réel, des contraintes particulières s'appliquent à l'infrastructure de communication ; il s'agit typiquement d'avoir une notion de priorité dans les requêtes, d'assurer une politique d'ordonnancement déterminée, etc. Un intergiciel dit *configurable* permet une certaine souplesse d'implantation tout en restant attaché à une spécification d'intergiciel donnée. Une telle approche permet d'obtenir de bonnes performances pour des applications conçues pour le modèle de distribution considéré [Gill et al., 2004].

TAO [Pyarali et al., 2002] est un exemple typique d'intergiciel configurable, qui implante les spécifications de CORBA dédiées au temps-réel. TAO fournit une architecture basée sur un assemblage de patrons de conceptions ; les différents composants de l'intergiciel peuvent être affinés afin d'adapter certains paramètres de fonctionnement, tels que la politique d'ordonnancement. Plusieurs mécanismes ont été développés pour faciliter la configuration de TAO ; nous les mentionnons dans les sections suivantes.

PolyORB

L'architecture d'intergiciel schizophrène [Pautet, 2001] reprend les principes des intergiciel génériques dans une forme étendue ; elle repose sur un ensemble de services canoniques qui peuvent être configurés en fonction des paramètres de l'application [Hugues et al., 2004]. Autour des composants canoniques sont mise en place des personnalités protocolaires ou applicatives qui permettent d'instancier l'intergiciel pour qu'il se comporte selon des spécifications données, aussi bien du point de vue de l'application que des autres nœuds du réseau ; plusieurs personnalités peuvent collaborer au sein d'une même instance de l'intergiciel.

Les principes de l'architecture schizophrène ont été appliqués dans l'intergiciel PolyORB [Quinot, 2003]. PolyORB permet de prendre en charge une grande variété de spécifications [Hugues, 2006], notamment différentes sous-spécifications CORBA (Real-Time CORBA, Fault-Tolerant CORBA) de même que différents protocoles CORBA tels que GIOP, MIOP… La flexibilité de l'architecture schizophrène permet par ailleurs d'adapter PolyORB à la plupart des paradigmes de répartition. De récents travaux [Hugues, 2005] ont mis en évidence les capacités d'un tel intergiciel pour les applications temps-réel, et notamment les possibilités de vérifications formelles permises par l'architecture schizophrène.

La configuration de PolyORB consiste en une sélection des personnalités et des implantations de services. Cette sélection s'effectue par des fichiers de configuration parcourus au lancement de l'intergiciel, ce qui facilite le changement de la configuration.

TAO et PolyORB fournissent des mécanismes pour ajuster l'implantation de l'intergiciel en fonction des besoins de l'application. Cependant, la description de ces besoins n'entre pas dans le cadre des spécifications CORBA.

II-2.2.3 RMI

L'utilisation d'un langage de configuration tiers tel qu'IDL permet de décrire l'adaptation de l'intergiciel indépendamment du langage de programmation utilisé. En contrepartie, l'utilisateur doit manipuler plusieurs langages et outils. D'autres intergiciels ont donc pris le parti d'extraire les interfaces à partir du langage de programmation. C'est le cas de Java RMI [Grosso, 2001], qui permet de décrire les entités à répartir directement au sein du programme Java.

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3
4 public interface Calcul extends Remote {
5       int addition (int a, int b) throws RemoteException;
6 }
```

Listing II.3 – interface d'une classe Java invocable à distance

Le listing II.3 décrit la déclaration de l'interface d'une classe destinée à être invoquée à distance. Une telle interface étend simplement l'interface prédéfinie `Remote`.

```
1 import java.rmi.Naming;
2 import java.rmi.RemooteException;
3 import java.rmi.RMISecurityManager;
4 import java.rmi.server.UnicastRemoteObject;
5
6 public class CalculImpl extends UnicastRemoteObject implements
      Calcul {
7       public CalculImpl() throws RemoteException {
8             super();
9       }
10
11      public int addition(int a, int b) {
12            return a + b;
13      }
14
15      public static void main(String args[]) {
16            if (System.getSecturiyManager() == null) {
17                  System.setSecurityManager(new
                        RMISecurityManager ());
18            }
19
20            try {
21                  CalculImpl obj = new CalculImpl();
22                  Naming.rebind("//hote/ServeurCalcul", obj);
23            } catch (Exception e) {
24                  System.out.println("Erreur: " + e.getMessage())
                        ;
25            }
26      }
27 }
```

Listing II.4 – interface d'une classe Java invocable à distance

Le listingII.4 illustre une implantation de l'interface que nous avons déclarée au listing II.3. Nous voyons que la classe correspond à une entité qui peut être distante : la méthode `main` enregistre l'objet auprès du service de nommage de RMI ; de même, le constructeur peut lever une

exception relative à un appel distant. Java RMI rassemble donc au sein d'un même code source toutes les opérations nécessaires à la définition et à l'installation d'une entité répartie.

Tout comme CORBA, RMI se focalise principalement sur la mise en place de systèmes d'informations, où la gestion des ressources ne constitue pas une contrainte extrêmement forte. Néanmoins, l'adaptation de l'intergiciel RMI pour des situations particulières a fait l'objet a fait d'études [Nester et al., 1999 ; Wall et Cahill, 2001].

Jonathan

De la même façon que pour CORBA, certains projets ont conduit à produire des intergiciels offrant une grande souplesse d'implantation. C'est notamment le cas Jonathan [Krakowiak], qui constitue une solution pour la prise en charge de différents paradigmes de communication. Un tel intergiciel est dit générique : il peut être *instancié* afin de se comporter selon une spécification donnée. Un intergiciel générique constitue en fait une armature définissant un ensemble de composants abstraits ; ces abstractions doivent être instanciées afin de former une personnalité particulière de l'intergiciel.

Jonathan peut ainsi être instancié pour implanter un intergiciel RMI ou JMS (*Java Message Service*), qui correspondent à des paradigmes différents. De la même façon que pour un intergiciel configurable, il s'avère nécessaire d'assister l'utilisateur dans la mise en place d'une instance de l'intergiciel. Dans le cas de Jonathan, ce processus de configuration repose sur un outil appelé Kilim [Obj, 2003], qui permet de configurer les éléments constituant l'intergiciel.

II-2.3 Discussion

Les différentes approches que nous avons décrites ici placent l'intergiciel au centre du processus de conception de l'application. Les bibliothèques de communication telles que MPI constituent une situation extrême, dans laquelle l'application doit effectuer toutes les manipulations de données.

Des approches plus évoluées, telles que RPC, CORBA ou RMI, assistent le concepteur de l'application en automatisant la prise en charge des données. D'autres services peuvent également être fournis par l'intergiciel ; par exemple un service de nommage dans les cas de CORBA et RMI. Dans tous les cas, ces solutions sont basées sur une approche commune, illustrée sur la figure II.1.

Les spécifications d'un intergiciel peuvent être déclinées en plusieurs version spécialisées, correspondant à des cas d'utilisation particuliers ; de même, les spécifications d'un intergiciel donné donnent souvent lieu à différentes implantations, répondant à des besoins spécifiques. De nombreuses recherches sont menées pour concevoir des implantations d'intergiciel pouvant être adaptées à différentes situations d'exécution, afin de réduire les efforts de développement.

Les différentes approches que nous avons étudiées dans cette section permettent de décrire l'interface des entités réparties. Cependant, il n'est pas possible d'indiquer l'agencement des entités ni leur répartition. Ces éléments de configuration sont du ressort de l'implantation de l'intergiciel. Les langages de description utilisés ne sont pas conçus pour rendre compte d'éventuelles propriétés concernant l'application ; les possibilités de vérification se limitent donc à l'intergiciel lui-même et ne peuvent donc pas prendre en charge des considérations telles que le temps d'exécution, qui nécessitent de considérer l'application dans sa globalité.

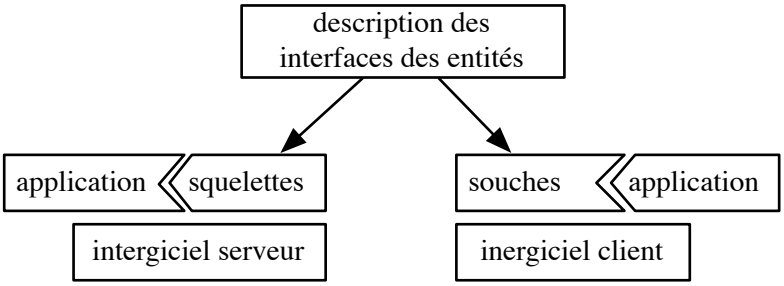


FIG. II.1 – Configuration d'une application centrée sur l'intergiciel

II-3 Approches centrées sur l'application

Les solutions d'intergiciel que nous avons présentées à la section précédente reposent sur une imbrication étroite de l'application avec son intergiciel. D'autres approches ont été développées pour permettre de mieux organiser l'application par rapport à l'intergiciel. Il s'agit souvent – mais pas toujours – de concevoir l'application comme un ensemble de *composants* assemblés et contrôlés par un exécutif ; cet exécutif intègre notamment l'intergiciel.

II-3.1 Déclinaisons orientées composant des spécifications « classiques »

La notion de composant constitue une évolution du concept d'objet. Une définition couramment acceptée d'un composant est celle de Szyperski [Szyperski, 1998] : « *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component [...] is subject to composition by third parties.* »

Un composant représente une entité de calcul. Il se caractérise par une interface, qui matérialise l'unique moyen d'accès à ses fonctionnalités. L'interface d'un composant est constituée d'un ensemble de *points d'interaction*, ou *ports*, permettant une interaction avec l'environnement extérieur. L'interface d'un composant définit explicitement non seulement les services qu'il fournit mais aussi, contrairement à des approches antérieures comme les objets, les services qu'il requiert. Une interface définit donc à la fois les contraintes d'utilisation du composant mais aussi un ensemble d'hypothèses faites sur son environnement. Les interfaces sont définies à l'aide de langages de définition d'interfaces – tel qu'IDL, par exemple.

Les *services* peuvent correspondre à des opérations (procédures, fonctions), des messages (dans une approche de composants objets) ou encore à des variables partagées. La description d'un service consiste en général en son nom, son type (type des arguments et d'un éventuel retour) et le cas échéant une liste d'exceptions.

Un composant se caractérise également par une encapsulation dans un conteneur pris en charge par l'environnement d'exécution. Ce conteneur est fourni par l'environnement d'exécution et prend en charge les interactions entre le composant et le reste du système ; il fournit un certain nombre de services au composant. Les composants ont ainsi une certaine indépendance vis-à-vis les uns des autres, ce qui facilite leur réutilisation et leur recomposition ; ils peuvent être *déployés* dans différents environnement d'utilisation.

Les spécifications d'intergiciel telles que CORBA, RMI ou DCOM sont essentiellement conçues pour implanter des objets répartis. Elles ont toutes évolué pour intégrer la notion de composant : CCM, J2EE et .NET.

II-3.1.1 J2EE

Les plates-formes comme J2EE ou .NET sont destinées à la mise en place de systèmes d'informations ; ils ne sont pas adaptés aux systèmes embarqués temps-réel. Ils fournissent néanmoins une approche architecturale intéressante. J2EE se repose à l'origine sur les mécanismes de communication de RMI ; il en constitue une évolution majeure. Un composant J2EE est appelé « EJB » (*Enterprise Java Bean*) et comporte une interface définissant les fonctions du composant, une interface fournissant les services de déploiement requis par le conteneur (création, suppression, recherche), une classe implantant les fonctions du composant et les méthodes nécessaires à la manipulation de l'EJB par le conteneur. Un EJB est associé à un descripteur de déploiement définissant les besoins du composant en terme de ressources d'exécution.

II-3.1.2 CCM

Les composants dans CCM sont aussi pris en charge par l'environnement d'exécution constitué par l'ORB. De la même façon que pour J2EE, CCM définit la notion de conteneur, qui s'appuie sur les mécanismes déjà définis dans l'architecture CORBA orientée objet. CCM définit plusieurs mécanismes de communication associés aux composants ; ils peuvent être classés selon quatre catégories :

- les facettes décrivent les fonctionnalités du composant offertes à ses interlocuteurs ;
- les réceptacles constituent des points de raccordement pour les interfaces des autres composants ;
- les attributs correspondent à des valeurs types associées au composant ;
- les événements permettent de matérialiser les communications entre composants sous la forme d'échanges de messages.

Les composants CORBA peuvent utiliser leurs interfaces, réceptacles et ports d'événement pour communiquer les uns avec les autres.

La description des composants et de leurs interface est assurée par des constructions syntaxiques supplémentaires dans IDL.

II-3.1.3 Implantations

Deux implantations principales ont été réalisées pour J2EE. JOnAS est historiquement basé sur la personnalité RMI de Jonathan (Jeremie) et peu donc bénéficier des mécanismes de configuration fournis par Kilim. JBoss est une implantation spécialisée.

De la même façon que pour la version de CORBA pour les objets répartis, différentes implantations d'ORB ont été réalisées, correspondant à différents domaines d'application. Ainsi, OpenCCM [Vadet et Merle, 2001]. Le projet TAO prend lui aussi en charge CCM [Krishna et al., 2005].

La notion de composant permet de mieux structurer la description de l'application. Les intergiciels y gagnent en possibilité de réutilisation des entités. Cependant, les informations concernant la topologie du déploiement (emplacement des nœuds, contraintes de l'environnement d'exécution) ne sont toujours pas prises en compte par les langages de description. Les différentes implantations de ces intergiciels orientés composants peuvent être configurés à des degrés divers ; il s'agit en fait de l'extension aux composants des mécanismes déjà mis en place pour les intergiciels orientés objet.

II-3.2 Conception de l'application par agencement de composants

D'autres approches sont basées sur la notion de composants, sans se rattacher à un intergiciel particulier. Ainsi, Fractal [Bruneton et al., 2002] est une approche d'assemblage de composants mise en place par le consortium ObjectWeb. Elle est basée sur la définition de conteneurs encapsulant le code des différents composants. Fractal repose sur une description de l'assemblage des composants de l'application, permettant la reconfiguration (c'est-à-dire l'évolution dynamique de l'assemblage des composants). Il est ainsi possible de décrire l'assemblage des composants à l'aide d'un langage spécialisé.

Fractal consiste en un exécutif et une interface de programmation pour différents langages. À ces éléments s'ajoute un langage de configuration. Un langage de référence – FractalADL [Bruneton, 2004], basé sur XML – est spécifié. Cependant, par soucis de généralité, les auteurs de Fractal n'imposent pas de langage de configuration particulier. De la même façon, un implantation d'exécutif de référence est définie, Julia [Bruneton, 2003] ; mais d'autres implantations ont été crées, telles que AOKell [Seinturier et al., 2006], prenant en charge une approche par aspect.

Fractal peut donc être vu comme une méthodologie architecturale pouvant être appliquée pour assembler des composants : bien qu'un langage d'assemblage et un exécutif standard aient été spécifiés, Fractal encourage l'utilisation d'autres langages de configuration ou d'exécutifs alternatifs. D'autres projets, tels que SOFA [Plášil et al., 1998] suivent également la même approche en définissant un exécutif et un langage d'assemblage associé.

II-3.3 Isolation des informations de déploiement

Les solutions que nous avons présentées jusqu'ici se basent sur une approche orientée composants. De cette façon, l'application est structurée en composants connectés entre eux et pilotés par un exécutif éventuellement réparti. Une autre méthode consiste à intégrer la mise en place de l'intergiciel au compilateur du langage. RMI applique ce principe, mais oblige l'utilisateur à prendre en compte l'aspect réparti de l'application dès le début de sa conception.

L'annexe des systèmes répartis d'Ada (DSA, *Distributed Systems Annex*) autorise l'utilisateur à concevoir dans un premier temps son application de façon centralisée, puis à en répartir les différents éléments [Pautet et Tardieu, 2005]. Cette démarche permet d'introduire une description centralisée du déploiement des nœuds, au contraire par exemple de CORBA qui effectue cette opération à l'exécution.

DSA reprend donc les mêmes principes que RMI en intégrant la description des interfaces dans le langage de programmation, et ajoute la prise en charge des informations de déploiement ; ces informations comprennent à la fois la localisation des nœuds et des information sur l'environnement d'exécution – notamment des directives concernant la compression des communications, par exemple.

Comme CORBA ou RMI, plusieurs implantations d'exécutifs existent : il s'agit principalement de GLADE [Pautet et Tardieu, 2005], l'implantation de référence, de RT-GLADE [Campos et al., 2004] et de PolyORB. L'utilisation de PolyORB pour DSA permet de bénéficier des capacités pour le temps-réel de cet intergiciel.

II-3.4 Discussion

Diverses solutions permettent de s'affranchir du caractère central de l'intergiciel. Une conception de l'application par assemblage de composants – mise en place par exemple dans J2EE, CCM et Fractal – consiste à étendre le rôle de l'intergiciel : il contrôle alors complètement l'exécution

des éléments applicatifs, qu'il encapsule. Cela facilite le redéploiement et la recomposition des composants, qui se trouvent ramenés à des éléments purement réactifs. La figure II.2 illustre cette organisation. L'intergiciel masque complètement l'aspect réparti – ou non – de l'application.

Par rapport aux solutions que nous avons décrites en section II-2, ces approches permettent une meilleure reconfiguration des application en facilitant le redéploiement des composants. Le fait de connaître l'agencement des composants permet éventuellement d'optimiser la construction des intergiciels ; le déploiement des composants demeure cependant une notion locale à chaque nœud.

Par ailleurs, des spécifications telles que CCM héritent de l'approche CORBA, qui ne se focalise pas sur la description des propriétés des applications. Le processus de vérification demeure donc une opération propre à chaque implantation d'intergiciel. J2EE ne vise pas des systèmes subissant des contraintes fortes en terme de place ou de temps d'exécution ; les considérations d'exécution correcte sont moins prégnantes. Fractal définit un cadre général pour assembler les composants et éventuellement reconfigurer l'application en cours d'exécution ; il ne s'intéresse pas spécialement au respect de contraintes extérieures ; le processus de vérification de l'application sort du cadre de Fractal.

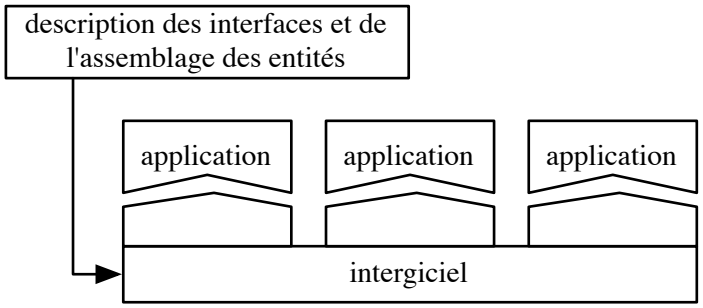


FIG. II.2 – Construction d'un exécutif pour une approche par composants

L'approche proposée par DSA est très intéressante : elle offre à l'utilisateur la possibilité de programmer sans prendre en compte les aspects répartis ; ceux-ci sont intégrés par la suite sous forme de directives additionnelles dans le code source, et d'informations de déploiement dans un fichier de configuration externe. La séparation est donc nette entre l'application elle-même et la façon de la répartir. Par rapport aux solutions que nous avons étudiées précédemment, DSA permet la prise en charge de paramètres supplémentaires pour le déploiement des nœuds de l'application. En revanche, les techniques pour la construction de l'application développée dans les approches orientées composant ne sont pas abordées. Des implantations comme GLADE permettent la prise en compte de certaines contraintes s'appliquant aux communications, comme la sélection de politiques de compression des données. Cependant, du fait que DSA n'est pas axé sur la description de l'assemblage des éléments applicatifs, il est difficile de mener une analyse sur l'application complète.

II-4 Conception des applications fondée sur les modèles

Les différentes approches que nous avons présentées dans les sections précédentes se caractérisent par une certaine interdépendance entre la conception de l'application et le choix de l'intergiciel à utiliser. En effet, l'intergiciel à utiliser influencera la conception de l'application en plusieurs

aspects, notamment le choix du langage à utiliser (pour DSA et RMI, par exemple) ou la façon d'organiser l'application (RPC, CORBA). Les approches de plus haut niveau telles que Fractal permettent dans une certaine mesure de faire abstraction de l'exécutif utilisé, mais elles guident néanmoins la conception de l'application [Hnětykna, 2005] – qui sera une application conçue en appliquant l'approche Fractal.

Contrairement à des solutions comme RMI ou DSA, les approches orientées composant reposent sur l'utilisation d'un langage pour décrire les applications comme un assemblage d'éléments éventuellement recomposables. Bien que ce ne soit pas le cas dans les méthodes que nous avons décrites, ces composants pourraient transporter, en plus de la description des interactions qu'ils prennent en charge, des caractéristiques non fonctionnelles telles que des temps d'exécution. Ces approches mettent en lumière l'intérêt de décrire les applications comme un assemblage d'éléments ; une telle démarche de modélisation permet de confronter la construction des applications à des spécifications plus ou moins formelles, ce qui peut faciliter l'analyse de leur structure et de leur exécution.

Les activités de l'OMG, outre CORBA, ont conduit à définir le langage UML (*Unified Modeling Language*) afin de faciliter la description des applications. UML était initialement conçu pour décrire l'organisation de programmes orientés objet ; il a été généralisé, de façon à permettre la description des différents aspects d'une application – tout en conservant l'approche objet dans une large mesure. UML sert de langage support à un processus de conception d'application appelé MDA (*Model Driven Architecture*). Nous donnons ici un aperçu des principes de MDA.

II-4.1 Présentation d'UML

Dans sa version 2.0, UML définit un ensemble de treize syntaxes, chacune correspondant à un type de *diagramme* : diagramme de classes, de paquetages, de structures composites, de composants, de déploiement, de cas d'utilisation, d'états, d'activités et d'interactions. Ces diagrammes définissent des notations standard permettant de décrire tous les aspects de l'application à modéliser – tant architecturaux que comportementaux. Dans le cadre de ce chapitre, nous nous intéressons essentiellement aux diagrammes architecturaux.

Les diagrammes de classe sont les plus connus. Ils permettent de décrire l'organisation des éléments de l'application, et sont largement inspirés des structures que l'on retrouve dans les langages de programmation orientée objet. Les diagrammes de paquetages permettent d'organiser les classes en sous-ensembles logiques, et ainsi de structurer les éléments applicatifs. Les diagrammes d'objets décrivent les instances des classes. Ils permettent donc de figurer l'état de l'application à un instant donné.

Les diagrammes de composants se situent à un niveau supérieur : ils permettent de représenter l'architecture comme un ensemble d'éléments proposant des interfaces qu'il est possible de connecter. Un composant peut ainsi requérir une connexion avec un autre élément fournissant une interface adéquate ; cette notion est absente des diagrammes de classes.

Les diagrammes de structures composites permettent de décrire les relations entre les différentes entités qui ont pu par exemple être définies dans les diagrammes de composants. Les diagrammes de structures composites permettent donc typiquement de représenter les connecteurs et les interfaces (c'est-à-dire les ports) ; ils permettent de décrire comment les différentes entités architecturales interagissent.

Les diagrammes de déploiement permettent de décrire la façon dont les éléments de l'application doivent être installés. Ils permettent notamment de représenter les éléments matériels sur lesquels les éléments applicatifs vont s'exécuter.

Les autres diagrammes permettent de décrire le comportement des entités. UML couvre ainsi à la fois la description de l'assemblage des composants applicatifs et les spécifications de leur comportement ; il permet de décrire tous les aspects des applications.

II-4.2 Particularisation de la syntaxe UML

Du fait qu'UML définit une syntaxe qui laisse une grande liberté pour la description des systèmes, il est souvent nécessaire d'enrichir les notations, afin de pouvoir préciser la sémantique des éléments. Ces enrichissements sont appelés *profils*. Ainsi, le profil SPT (*Scheduling, Performance and Time*) introduit des notions de dimensionnement associées à la description des systèmes temps-réel (temps de réponse, tailles de files d'attente, etc.). Son successeur MARTE (*Modeling and Analysis of Real-Time and Embedded*) [Espinoza et al., 2005] permet aussi de prendre en compte les contraintes liées aux systèmes embarqués. D'autres profils, comme Omega [Graf et Hooman, 2004], intègrent l'usage de sémantiques formelles au sein d'une modélisation UML.

II-4.3 La démarche MDA

Dans le prolongation d'UML, l'OMG a défini une méthodologie de développement appelée MDA [Blanc, 2005]. Cette approche consiste à manipuler différents *modèles* de l'application à produire, depuis une description très abstraite jusqu'à une représentation correspondant à l'implantation effective du système. Le processus MDA se décompose en trois étapes principales, représentées sur la figure II.3 :

1. la définition d'un modèle de très haut niveau, indépendant des contraintes d'implantation : le PIM (*platform independent model*) ; le PIM peut être raffiné afin d'ajouter différentes informations non fonctionnelles telles que la gestion de la sécurité ; ces informations demeurent indépendante de la plateforme d'exécution qui sera effectivement utilisée ;
2. le PIM est ensuite transformé pour prendre en compte les spécifications propres à la plateforme d'exécution ; le nouveau modèle est appelé PSM (*plateform specific model*) ; le PSM peut être également raffiner afin de prendre en compte différents paramètres liés à l'environnement d'exécution.
3. Le PSM est ensuite utilisé pour générer un système exécutable basé sur les spécifications à partir desquelles le PIM a été construit.

La démarche MDA permet de structurer les étapes de conception des application, et délimitant de façon claire la séparation entre les spécifications fonctionnelles et la prise en compte des paramètres de déploiement et d'implantation. UML est utilisé comme langage de modélisation ; ses différentes syntaxes permettent de décrire tous les aspects de l'application à produire.

La description d'un PIM consiste en l'expression des besoins fonctionnels de l'application. Les différentes constructions syntaxiques du langage représentent des concepts relativement abstraits – UML n'a pas la notion de *thread* système – qui sont interprétés à travers un profil. Bien que des travaux visent à produire une structure de communication adaptée [Basso et al., 2006], les PSM sont en général définis par rapport à des profils correspondant à des technologies d'implantation existantes [Mellor et Balcer, 2003] telles que J2EE ou CCM, .

En couvrant toutes les étapes du cycle de production des applications, l'approche MDA permet la prise en compte de tous les paramètres de déploiement et de configuration. Il est alors possible d'extraire ces informations du modèle pour configurer l'implantation d'intergiciel utilisée [Gokhale et al., 2003].

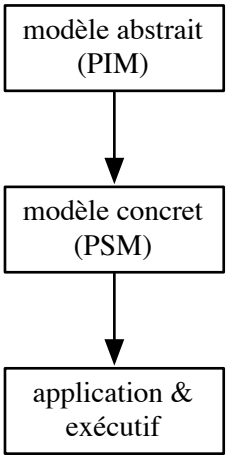


FIG. II.3 – L'approche MDA

II-4.4 Discussion

MDA propose une démarche intégrée permettant de rassembler tous les éléments pour la description d'une application. De cette façon, MDA vise la mise en place de différentes étapes de raffinement depuis la conception conceptuelle de l'application jusqu'à la production de code exécutable. Cependant, si UML permet une grande expressivité, il ne transporte par en lui-même une sémantique très précise pour décrire les architectures [Garlan et al., 2002]. L'utilisation de profils pour spécifier les PIM permet de préciser la sémantique associée aux constructions UML. Cependant, ces spécifications de PSM se basent typiquement sur CCM ou J2EE, qui imposent une vision objet de l'architecture.

Sans remettre en cause la logique de la démarche MDA, il semble intéressant de permettre une modélisation de l'application à un niveau plus bas qu'un PIM ; il serait ainsi possible de définir plus finement les paradigmes de communication et l'organisation des composants applicatifs tout en permettant la caractérisation vis-à-vis des contraintes appliquées au système.

II-5 Vers une description formelle des applications

De nombreux travaux ont conduit à la définition de langages permettant de décrire précisément les applications. Ces langages sont regroupés sous la dénomination de « langages de description d'architecture » (ADL). Plusieurs ADL ont été proposés pour la modélisation d'architectures logicielles, soit dans un but général, soit dédiées à un domaine particulier d'application.

II-5.1 Aperçu des langages de description d'architecture

Le développement des langages de description d'architectures [Perry et Wolf, 1992 ; Garlan et Shaw, 1993 ; Medvidovic et Taylor, 2000] correspond à une volonté de décrire les applications dans leur globalité. Les ADL sont en général conçus pour servir de support à l'architecte logiciel en répondant à un certain nombre de fonctionnalités parmi lesquelles :

– définir l'*architecture* d'un système comme la composition d'éléments de base, de façon abstraite et ce indépendamment de la réalisation concrète (implantation) de ces derniers ;

définir de façon explicite les interactions parle plus généralement de relations) entre ces éléments ;
- supporter une phase d'analyse architecturale qui se place entre l'analyse des besoins (l'architecture sert de support à l'analyse du respect des exigences) et la conception (l'architecture sert de support à la conception en proposant un découpage du système) ; supporter à la fois une approche descendante (servir de support à la décomposition lors de la conception du système) et une approche ascendante (servir de support à la construction d'un système par assemblage d'éléments préalablement définis et réutilisables) ;
- fournir une sémantique bien définie au plan architectural (et ne pas se limiter à une description à base « de boîtes et de lignes ») ;
- permettre l'analyse de l'architecture, soit par l'utilisation de critères liés aux respects de contraintes de style (exemple : couplage minimal entre composants), soit par l'utilisation de techniques formelles (par exemple la vérification de l'absence d'inter-blocages dans une architecture) auquel cas l'architecture permet alors l'analyse compositionnelle ;
- aider à l'implantation du système, par exemple en permettant la construction automatique du code des interactions entre composants du système.

La définition des langages de description d'architecture est assez floue, et il n'existe pas vraiment de consensus sur le niveau d'abstraction et de description auquel devrait se situer un ADL. Certaines études ont été menées pour classer les ADL [Medvidovic et Taylor, 2000]. Elles ont débouché sur un ensemble de critères permettant de classer les ADL.

II-5.2 Éléments constitutifs de la description d'une architecture

Les langages de description d'architecture sont généralement structurés autour de trois éléments de base : les *composants*, les *connecteurs* et les *configurations*. À parti de ces concepts, il est possible de décrire la construction d'un système, qui est constitué d'un ensemble de composants liés par des connecteurs au sein de configurations.

II-5.2.1 Composants

La notion de composant dans les ADL correspond à celle développée pour CCM ou J2EE, en précisant éventuellement les interfaces. Les interfaces permettent en effet d'associer une sémantique aux relations entre composants. Cette sémantique est souvent précisée par la définition de propriétés permettant de caractériser les éléments de l'interface – données, informations temporelles ou stochastiques…

Des solutions comme J2EE ou CCM reposent sur une description des services qui se limite à la description des types de données. C'est en général insuffisant pour vérifier la bonne interopérabilité des composants entre eux [Vallecillo et al., 2000]. En effet, deux composants compatibles du point de vue de leurs noms de services peuvent quand même se bloquer si leurs protocoles comportementaux (l'ordre dans lequel les services sont sensés être utilisés) sont incompatibles. De nombreux travaux universitaires ont conduit à prendre en compte des interfaces de plus haut niveau comme les interfaces comportementales décrites à l'aide de langages de description d'interfaces comportementales (Behavioral Interface Description Languages, BIDL). Ces travaux définissent ensuite des mécanismes de vérification [Bernardo et Inverardi, 2003] et de correction [Canal et al., 2006] d'architectures logicielles en se basant sur ces descriptions comportementales.

II-5.2.2 Connecteurs

Un connecteur constitue l'abstraction utilisée pour modéliser les interactions entre composants. Un connecteur décrit des correspondances entre interfaces de composants ainsi que les règles qui prévalent lors des interactions. Un connecteur représente classiquement les communications entre les composants, et peut donc être interprété dans une certaine mesure comme la représentation d'un intergiciel ; il matérialise alors des communications par *tube* à la manière Unix, par appel de sous-programme distant (RPC), etc.

Un connecteur peut se voir attribuer des *rôles*, qui représentent les rôles que peuvent tenir différents composants dans l'interaction décrite par le connecteur. Ainsi par exemple dans une architecture dont les communications sont basées sur un mécanisme de type *tube et filtre*, un connecteur de type *tube*, aura deux rôles : source, qui correspondra au composant – fichier ou processus – qui fournit les données, et puits, qui correspondra au composant – le filtre suivant ou un fichier – destinataire des données.

La notion de connecteur est plus ou moins développée d'un ADL à un autre [Oussalah, 2005 ; Medvidovic et Taylor, 2000]. Dans des langages tels que Darwin ou Rapide, un connecteur peut être une connexion directe des ports des composants, associée à une sémantique très simple voire implicite ; d'autres langages, comme Unicon, définissent un connecteur comme une combinaison de connecteurs prédéfinis ; un connecteur peut également être spécifiés à l'aide d'un langage de définition (Wright).

II-5.2.3 Configurations

Une *configuration* (architecturale), aussi appelée topologie, décrit un assemblage de composants et de connecteurs. Les composants et connecteurs correspondent à des instances nommées de types de composants et de connecteurs permettant leur référencement. Les relations entre interfaces des composants et rôles des connecteurs sont établies à l'aide de *liaisons* (*bindings*). Une configuration fournit une information structurelle, qui peut éventuellement être amenée à évoluer au cours de l'exécution du système.

Une configuration permet le déploiement automatique de l'architecture définie. Les configurations permettent aussi dans certains ADL de construire des composants *composites*, c'est-à-dire constitués d'une configuration de composants et de connecteurs. Pour cela, il doit être possible de relier les ports des sous-composants à ceux du composite. L'intérêt des composants composites est de permettre la réutilisation de configurations complètes mais aussi de prendre en charge le développement de techniques compositionnelles de vérification formelle (la vérification est effectuée au niveau du particulier, un constituant de l'architecture, puis réutilisée au niveau de la vérification globale).

D'autres caractéristiques peuvent intervenir, telles que la prise en compte de contraintes ou encore l'hétérogénéité et l'évolutivité des architectures [Medvidovic et Taylor, 2000].

II-5.3 Langages pour assister la production de systèmes exécutables

La syntaxe d'UML permet en grande partie de représenter les notions classiques d'un langage de description d'architecture ; il est utilisé comme support syntaxique pour de nombreux ADL [Garlan et al., 2002 ; Hofmeister et al., 1999 ; Elloy et Simonot-Lion, 2002 ; Rumpe et al., 1999]. De la même façon, FractalADL peut être considéré comme un ADL dédié à l'assemblage de composants applicatifs.

La plupart des ADL se focalisent sur un aspect particulier de la description architecturale : analyse et vérification (Darwin, Rapide, Wright [Allen, 1997 ; Allen et Garlan, 1997]), définition d'architectures liées à des styles architecturaux (Aesop), raffinement d'architectures (C2/SADL), lien avec l'implantation (Darwin, OLAN), définition de code permettant le recollement, les interactions entre composants existants (C2/SADL, UniCon...). La grande variété des ADL a conduit à la définition de langages destinés à jouer un rôle de pivot, comme Acme [ABLE Group] ou bien encore xADL [Dashofy et al., 2001].

Nous présentons ici certains langages plus particulièrement destinés à la spécification d'architectures en vue de générer des systèmes exécutables.

II-5.3.1 Descriptions architecturales de haut niveau

Darwin [Magee et Kramer, 1996] est un ADL qui permet avant tout de décrire des configurations ; il possède à la fois une notation textuelle et graphique. L'un des intérêts de Darwin est sa prise en compte de la dynamique de la topologie des architectures (création/suppression de composants, modification des liaisons).

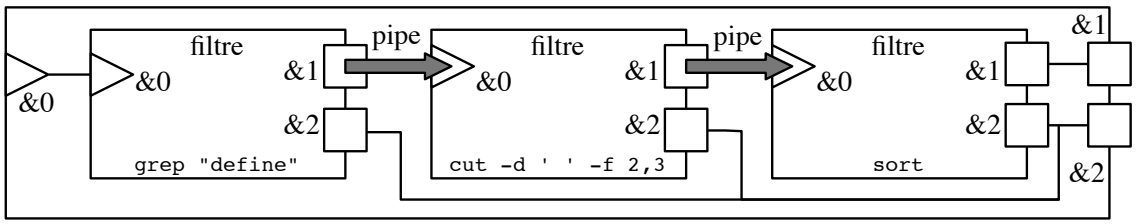


FIG. II.4 – Schéma d'une architecture constitué d'une série de filtres

Supposons n filtres à combiner en série pour obtenir un composant composite, comme représenté sur la figure II.4. Une description d'une telle description en Darwin est illustrée par le listing II.5. Darwin permet l'instanciation de composant avec la primitive `inst` (`@` permettant de spécifier sur quelle machine) et la liaison dynamique entre ports (composant/composant ou composite/sous-composant) avec `bind`.

```
component filter {
  provide output<stream char>;
  require input<stream char>;
};

component pipeline(int n) {
  provide output;
  require input;
  array F[n]:filter;
  forall k:0..n-1 {
    inst F[k]@k+1;
    when k < n-1
      bind F[k+1].input -- F[k].output;
  }
  bind
    F[0].input -- input;
    output -- F[n-1].output;
```

```
18 }
```

Listing II.5 – Connecteur en Darwin, correspondant a la figure II.4

À partir de l'exemple du listing II.5, nous pouvons remarquer que Darwin est un langage d'assez bas niveau, ce qui s'illustre notamment par l'utilisation de type de données inspirés du C. Ainsi, le type `<stream char>` correspond à un flux de caractères, destiné à être traités en lot. Ces types de donnée peuvent directement être exploités au sein d'un exécutif (Regis).

L'approche Darwin a été étendue dans le projet TRACTA [Giannakopoulou et al., 1999 ; Magee et al., 1999] pour prendre en compte le comportement des composants à l'aide de LTS puis de l'algèbre de processus FSP [Magee et Kramer, 1999]. Une architecture donnée Darwin étendue avec FSP peut alors être vérifiée à l'aide de l'outil LTSA :

- par animation (l'utilisateur peut interagir avec les ports des configurations (vues comme des composants composites) ;
- par vérification d'absence de blocages (une trace témoin est proposée en cas de blocage) ;
- par la définition d'un automate particulier, appelé *property automata*, qui définit les traces légales pour un alphabet d'événements donné et contient éventuellement des états `ERROR`. Cet automate est ensuite combiné en parallèle avec le processus correspondant à la configuration dans le but de procéder à une analyse d'atteignabilité (ou non) des états `ERROR` et à la vérification de la correspondance entre traces de la configuration et traces désirées. Cette technique permet de vérifier des propriétés de sécurité et de vivacité.

II-5.3.2 Intégration de spécifications comportementales formelles

Les langages de description d'architecture permettent en théorie de rassembler tous les éléments nécessaires à la description d'une application répartie et d'en déduire la configuration de l'intergiciel nécessaire. Les ADL tels que Darwin se situent cependant à un niveau d'abstraction trop élevé ; ils font abstraction des paramètres influant sur la configuration de l'intergiciel.

De nombreux travaux ont porté sur d'autres langages de description d'architecture, plus orientés vers le détail de l'implantation des systèmes, et notamment la prise en compte de l'environnement d'exécution. Nous étudions ici certains de ces langages.

LfP [Regep et al., 2002] (Language for Prototyping) est un langage formellement défini permettant de décrire le contrôle d'une application répartie. Il est conçu pour reprendre une description faite en UML et la raffiner pour établir les caractéristiques précises du système.

Dans sa conception, LfP s'inspire en grande partie des concepts développés dans l'approche MDA et vise notamment à permettre une vérification de l'architecture modélisée. Il s'appuie pour cela sur une définition formelle basée sur les réseaux de Petri [Girault et Valk, 2003]. Il permet ainsi la vérification formelle d'un système tout au long de sa réalisation ; il s'appuie pour cela sur trois vues : la vue fonctionnelle, la vue propriétés et la vue implantation.

La vue fonctionnelle décrit l'architecture et le comportement du système. Elle contient :

- une partie déclarative contenant des informations de traçabilité (par exemple, les références vers des composants du modèle UML d'origine) et la déclaration des types et constantes utilisés dans le modèle.
- un graphe hiérarchique décrivant l'architecture du système, ce graphe est composé de classes, média et *binders*.

Une classe LfP correspond à une classe instanciable UML. Les média permettent de décrire les schémas de communication (protocoles) entre classes. Les relations d'association, d'agrégation, et de composition d'UML peuvent être exprimées au moyen de média. Les *binders* représentent des

points d'entrée (ports de communication) entre les classes et les média. Ils décrivent les caractéristiques de la sémantique d'interaction entre les média et les classes : direction des communications, synchronisme, ordonnancement, etc. Plusieurs *binders* peuvent être associés à une classe, permettant ainsi de modéliser des interactions complexes.

La vue propriété explicite les propriétés à vérifier sur le système et se positionne comme un complément de la vue fonctionnelle. Les propriétés peuvent être considérées comme des obligations de preuve (au sens des assertions). Elles prennent la forme d'invariants (pour exprimer une exclusion mutuelle ou une condition sur un ensemble de variables), de formules de logique temporelle (pour exprimer une causalité entre des événements), ou de post-conditions sur les actions. Les informations de cette vue sont exploitées pour la vérification formelle ainsi que pour la vérification de l'exécution des programmes générés.

La vue d'implantation définit les contraintes du système telles que l'environnement d'exécution ciblé, le langage utilisé par le générateur de code, ou encore des informations de déploiement du système. Ces informations sont associées aux entités LfP et sont exploitées pour la génération automatique de programmes [Gilliers, 2005].

II-5.3.3 Modélisation des systèmes embarqués temps-réel

Différents languages de descriptions d'architecture, tels que CLARA [Faucou et al., 2002], MetaH [Vestal, 1998] ou AADL [Feiler et al., 2003], ont été mis au point pour la description des systèmes embarqués temps-réel.

MetaH est un ensemble d'outils pour assister le développement d'applications (CASE : *Computer Aided Software Engineering*) axé sur les systèmes embarqués pour l'aéronautique. MetaH définit un langage de description d'architecture [Vestal, 1998] ; les outils s'appuient sur le langage pour produire des descriptions architecturales, les analyser et générer des systèmes exécutables.

Le langage MetaH en lui-même se focalise sur une description assez bas niveau des composants du système à produire : il permet de représenter les différents composants matériels (processeurs, réseaux, mémoire…) et les principaux éléments logiciels (processus).

Dans son approche, MetaH ne vise pas la construction de systèmes répartis selon la même optique que pour les systèmes d'information : il s'agit de systèmes statiques dont tous les paramètres sont connus et peuvent faire l'objet d'une analyse statique. Les besoins en terme de communication sont donc bien définis et limités ; MetaH induit un modèle de répartition fondé sur le passage de message entre entités.

AADL (Architecture Analysis & Description Language) a été conçu dans le prolongement de MetaH [Feiler et al., 2003] ; il en reprend les principes. Par rapport à MetaH, AADL permet une plus grande précision dans la description des applications en introduisant notamment la notion de sous-programme. Une description AADL se focalise sur une identification concrète des différents éléments de l'architecture.

Tout comme MetaH, AADL a été conçu pour modéliser les systèmes embarqués temps-réel. Il a pour objectif de constituer un langage fédérateur pour décrire tous les aspects d'une application et ainsi permettre l'interopérabilité des différents outils de conception, analyse et génération. Contrairement à MetaH qui est associé à un ensemble d'outils définis, le standard AADL ne définit pas de méthodologie et laisse donc chacun libre de construire ses outils d'exploitation.

AADL poursuit en partie les mêmes objectifs qu'UML et la démarche MDA, mais à un niveau de description plus proche de l'implantation – AADL permet notamment de décrire de façon préciser l'environnement d'exécution tels que les processeurs, réseaux, etc. Même si par de nombreux aspects AADL peut être vu comme une alternative plus concrète à MDA/UML, il ne vise

cependant pas à concurrencer l'approche de l'OMG ; des profils UML pour AADL sont d'ailleurs en développement [Boisieau et Gianiel, 2006 ; SAE, 2006d] afin de rassembler les deux langages.

II-5.4 Discussion

Le concept d'ADL regroupe un vaste éventail de langages qui permettent de décrire précisément toutes les caractéristiques d'une application dans le cadre d'une exploitation déterminée.

Certains de ces langages, en se plaçant à un haut niveau d'abstraction, permettent la spécification formelle et la vérification des descriptions architecturales. Ainsi Darwin propose un cadre suffisamment expressif pour la description comportementale au niveau des composants et des connecteurs mais aussi pour la description d'architectures (topologies) dynamiques. Il en est de même pour des langages comme LfP, qui prévoit également d'être associé à UML pour constituer les dernières étapes de la production de l'application.

Le haut niveau de description de LfP et de Darwin les rend cependant difficiles à utiliser pour décrire précisément l'application et ses contraintes. D'autres langages comme MetaH ou AADL se placent à un niveau de modélisation beaucoup plus bas ; il prévoient notamment la description des composants matériels pour intégrer les informations de déploiement.

MetaH sert de langage pivot pour coordonner un ensemble d'outils de développement. AADL ne définit pas un tel ensemble d'outils standard ; son orientation est plus ouverte comme langage fédérateur. Par ailleurs, il permet une relation étroite avec la syntaxe UML.

II-6 Conclusion

Nous avons dressé un panorama des différentes façons d'aborder la description de la description d'un intergiciel. Cela nous a permis de mettre en lumière certaines insuffisances vis-à-vis des buts que nous poursuivons.

II-6.1 Insuffisances des approches centrées sur l'intergiciel

Les approches « classiques » pour la description des intergiciels, telles que RPC ou CORBA, font de l'application une considération secondaire. Le problème de la localisation des nœuds est souvent laissé à l'application.

D'autres approches, plus focalisées sur l'application, ont été développées ; c'est le cas par exemple des évolutions orientées composants des intergiciels classiques. L'application est alors décrite comme un assemblage de composants qui interagissent entre eux par l'intermédiaire d'un intergiciel ; celui-ci est construit à partir des informations associés aux composants.

Certaines solutions offrent des possibilités de description plus étendues. C'est notamment le cas de DSA ; à travers des implantations telles que GLADE, il est possible de décrire la localisation des différents nœuds de l'application, et de préciser certaines caractéristiques des liens de communication.

Toutes ces solutions se fondent sur la description des interfaces des entités ; elles ne permettent pas la description des différents aspects de l'application – contraintes temporelles ou spatiales, spécifications des comportements, etc. La configuration de l'intergiciel est possible avec certaines implantations telles que TAO ou PolyORB. Il s'agit néanmoins d'une opération complémentaire, qui nécessite l'intervention de l'utilisateur : les caractéristiques de l'application n'étant pas formalisées, il n'est pas possible de définir des politiques de configuration automatique.

II-6.2 Spécifications concrètes de l'application

Afin de pouvoir mettre en place un intergiciel complètement configuré, il est nécessaire de pouvoir décrire l'organisation de l'application. Les différents paramètres de l'intergiciels sont alors déduits des spécifications de l'application.

Dans cette optique, l'OMG a défini le processus MDA, basé sur une modélisation de l'application à construire. MDA utilise le langage de modélisation UML et défini un processus de raffinements successifs permettant de particulariser une modélisation abstraite pour une plateforme d'exécution déterminée. À travers UML, MDA permet de prendre en compte les différents aspects nécessaires à la caractérisation complète de l'application.

MDA constitue néanmoins un processus assez complexe, et UML peut apparaître comme trop abstrait pour être véritablement adapté à la modélisation concrète des applications.

Les langages de description d'architecture présentent une solution formelle intéressante pour prendre en charge cette étape de pré-implantation. Cette notion couvre un large champ de langages différents ; certains ont pour objectif la description de systèmes pour leur vérification et leur génération.

II-6.3 AADL comme langage pour la configuration d'intergiciel

AADL est un langage très récent, destiné à la modélisation des systèmes embarqués temps-réel. Il hérite de nombreux concepts de MetaH, tout en étant plus ouvert ; il vise notamment a servir de langage central pour la coordination des différents outils nécessaires à la conception, l'analyse et la génération d'applications. Plusieurs travaux portent sur les relations possibles entre AADL et UML, permettant ainsi d'envisager l'utilisation d'AADL comme langage pour décrire un PSM du MDA.

L'objectif de nos travaux est d'utiliser AADL pour la description d'applications temps-réel répartis embarqués afin de prendre en charge le processus de conception, de vérification et de génération de systèmes exécutables.

CHAPITRE III

AADL, un langage pour décrire les architectures

AADL EST UN LANGAGE DE DESCRIPTION D'ARCHITECTURE normalisé par le SAE (Society of Automotive Engineers), principalement dédié à la modélisation des systèmes embarqués temps-réel. Il joue un rôle central au sein de plusieurs projets tels que COTRE [Farines et al., 2003a], ASSERT [ASSERT, 2006] ou Topcased [Farail et Gaufillet, 2005]. La première version du standard [SAE, 2004] date de 2004, et le langage évolue régulièrement depuis.

Seuls quelques ouvrages [Haddad et al., 2006 ; Feiler et al., 2006] décrivant les différents aspects d'AADL ont été récemment publiés ou sont sur le point de l'être. Il nous paraît donc important de consacrer un chapitre à l'étude du langage, afin d'en souligner les aspects pertinents pour nos travaux. Ce chapitre porte sur la version 1.0 du standard, qui est la dernière publiée à ce jour. Nous consacrons également une section à certaines améliorations syntaxiques nécessaires à nos travaux, qui seront intégrées dans la version 1.2. Nous illustrons notre propos par une série d'exemples décrivant la description progressive d'un système réparti.

III-1 Principes du langage

AADL peut être exprimé selon différentes syntaxes. Le standard en définit trois : en texte brut, en XML, ainsi qu'une représentation graphique. Par cette multiplicité des syntaxes possibles, AADL peut être utilisé par de nombreux outils différents, graphiques ou non. Le développement de profils pour UML permet également d'envisager l'intégration d'AADL au sein d'outils de modélisation UML.

AADL a été créé avec le soucis de faciliter l'interopérabilité des différents outils ; c'est pourquoi la syntaxe de référence est textuelle. La représentation XML permet de faciliter la création de parseurs pour des applications existantes. La notation graphique se pose en complément de la notation textuelle, pour faciliter la description des architectures ; elle permet une représentation beaucoup plus claire que le texte ou XML, mais elle est moins expressive.

En tant que langage de description d'architecture, AADL permet de décrire des composants connectés entre eux pour former une architecture. Une description AADL consiste en un ensemble de déclaration de composants. Ces déclarations peuvent être instanciées pour former la modélisation d'une architecture.

III-2 Définition des composants

Les composants AADL sont définis en deux parties : l'interface et les implantations. Un composant AADL possède une interface (*component type*) à laquelle correspondent zéro, une ou plusieurs implantations (*component implementation*). Toutes les implantations d'un composant par-

tagent la même interface ; il est possible de les interchanger sans affecter les connexions avec les autres composants.

III-2.1 Catégories de composants

AADL définit plusieurs *catégories* de composants, réparties en trois grandes familles :
- les composants logiciels définissent les éléments applicatifs de l'architecture ;
- les composants de la plate-forme d'exécution modélisent les éléments matériels ;
- les systèmes permettent de regrouper différents composants en entités logiques pour structurer l'architecture.

III-2.1.1 Composants logiciels

Il existe cinq catégories de composants logiciels :
- les fils d'exécution (*threads*) ;
- les groupes de *threads* ;
- les processus ;
- les sous-programmes ;
- les données.

Les *threads* sont les éléments logiciels actifs. Ils peuvent être comparés aux processus légers tels que définis dans les systèmes d'exploitation. Les groupes de *threads* permettent de regrouper des *threads* afin de former des hiérarchies. Les processus définissent des espaces mémoire dans lesquels s'exécutent les *threads*. Les sous-programmes modélisent les procédures telles que définies dans les langages de programmation impératifs. Enfin, les données représentent les structures données qui peuvent être stockées ou échangées entre les composants.

Le standard AADL définit une sémantique assez précise pour ces composants. Ainsi, les processus AADL ne matérialisent que l'espace mémoire d'exécution des *threads* ; un processus doit donc contenir au moins un *thread* AADL ; parallèlement, un *thread* doit être contenu dans un processus. Les sous-programmes AADL ne peuvent pas avoir d'état.

(a) donnée (b) sous-programme (c) *thread* (d) groupe de *threads* (e) processus

FIG. III.1 – Syntaxe graphique des composants logiciels

La figure III.1 illustre la représentation des différentes catégories de composants logiciels selon la syntaxe graphique d'AADL.

III-2.1.2 Composants de plate-forme

Il existe quatre catégories de composants de plate-forme :
- les processeurs ;
- les mémoires ;
- les bus ;
- les dispositifs (*devices*).

Les processeurs représentent des ensembles constitués d'un microprocesseur combiné à un ordonnanceur ; ils modélisent donc d'un processeur associé à un système d'exploitation minimal. Les mémoires représentent tous les dispositifs de stockage : disque dur, mémoire vive, etc. Les *devices* permettent de représenter des éléments dont la structure interne est ignorée ; il peut s'agir par exemple de capteurs dont on ne connaît que l'interface et les caractéristiques externes. Les bus modélisent toutes les sortes de réseaux ou de bus, depuis le simple fil jusqu'à un réseau de type Internet. Les bus doivent être branchés aux processeurs, mémoires et *devices* afin de transporter les communications entre ces composants.

(a) processeur (b) dispositif (c) mémoire (d) bus

FIG. III.2 – Syntaxe graphique des composants de plate-forme

La figure III.2 illustre la représentation des différentes catégories de composants de plate-forme selon la syntaxe graphique d'AADL.

III-2.1.3 Systèmes

Les systèmes AADL permettent de rassembler différents composants pour former des blocs logiques d'entités ; ils facilitent la structuration de l'architecture. Contrairement aux autres catégories de composants, la sémantique des systèmes n'est donc pas associée à des éléments précis.

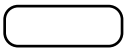


FIG. III.3 – Syntaxe graphiques des composants système

La figure III.3 illustre la représentation des systèmes selon la syntaxe graphique d'AADL.

III-2.2 Types et implantations

Le type d'un composant définit son interface tandis que l'implantation décrit les éléments (*sous-clauses*) de sa structure interne. Une implantation fait toujours référence à un type défini par ailleurs (cf. listing III.1). AADL étant un langage déclaratif, l'ordre des déclarations n'importe pas : il est possible de déclarer une implantation avant le type correspondant.

```
1  processor i486
2  end i486;
3
4  processor i486dx extends i486
5  end i486dx;
6
7  processor implementation i486.Intel
8  end i486.Intel;
9
10 processor implementation i486.AMD
```

```
11 end i486.AMD;
12
13 processor implementation i486dx.Intel extends i486.Intel
14 end i486dx.Intel;
```

Listing III.1 – Types et implantations de composants AADL

Il est possible d'étendre une déclaration de composant (type ou implantation) afin d'y ajouter ou d'en préciser des éléments (cf. listing III.1). La figure III.4 résume les mécanismes d'extensions possibles. Nous pouvons remarquer que l'implantation d'un type qui en étend un autre peut étendre une implantation du type initial.

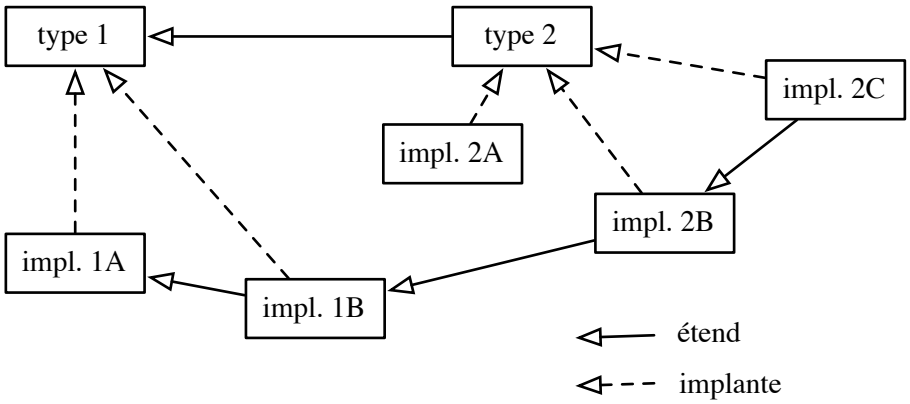


FIG. III.4 – Possibilités d'extension des composants

L'extension de composant AADL ne correspond pas exactement à la notion d'héritage des langages objet : un composant et son extension constituent des composants distincts ; il n'est pas possible d'utiliser l'un pour l'autre au sein d'une description architecturale. En revanche, les implantations d'un type de composant peuvent être substituées les unes aux autres dans la mesure où elles partagent la même interface aux autres composants.

III-3 Structure interne des composants

La structure interne des composants peut être précisée dans la déclaration des implantations des composants.

III-3.1 Sous-composants

Un sous-composant est une instance de la déclaration d'un composant. De cette façon, une architecture modélisée en AADL est une arborescence de d'instances de composants. La déclaration d'un sous-composant doit être associée à une catégorie, un type ou une implantation de composant ; il est ainsi possible d'apporter plus ou moins de précision dans la description architecturale. En ne spécifiant que la catégorie du sous-composant, la description architecturale demeure vague ; elle ne peut pas être exploitée pour générer un système exécutable, mais peut être employée à des fins de documentation.

Une description architecturale complète doit préciser l'implantation de composant utilisée pour chaque sous-composant.

AADL définit les règles de composition des composants ; seules les combinaisons ayant une sémantique cohérente sont autorisées. Le tableau III.1 fait la synthèse des compositions possibles.

composant	sous-composants
donnée	données
sous-programme	—
thread	—
groupe de *threads*	*threads*
processus	*threads*, groupes de *threads*
processeur	mémoires
mémoire	mémoires
bus	—
dispositif	—
système	données, processus, processeurs, mémoires, dispositifs, bus, systèmes

TAB. III.1 – Compositions légales des composants AADL

Nous pouvons remarquer que les sous-programmes ne sont jamais instanciés. En effet, la sémantique d'AADL traduit le fait qu'en terme de langage de programmation, un sous-programme n'est en fait qu'une séquence de code stockée dans l'espace mémoire d'un programme : ce n'est pas une entité en soi. Nous reviendrons sur ce point en section III-9.

De la même façon, un processus ne peut pas être un sous-composant d'un processeur : un programme est en effet exécuté par un processeur, mais n'est pas intégré dedans.

Dans la mesure où un dispositif modélise une boîte noire, il ne peut pas avoir de sous-composant.

Un composant mémoire peut être contenu dans un processeur ou une autre mémoire pour modéliser un cache mémoire ou une partition de disque dur.

Les systèmes peuvent contenir toutes les catégories de composants sauf les sous-programmes (qui ne peuvent pas être instanciés) et les *threads* et groupes de *threads*, qui doivent appartenir à un processus – un fil d'exécution ne peut en effet pas exister en dehors de l'espace mémoire qui délimite son exécution.

III-3.2 Appels de sous-programmes

Les appels de sous-programmes modélisent les appels de procédures dans les langages impératifs. Ils sont regroupés en séquences d'appels dans les sous-programmes et les *threads*. Bien que la syntaxe utilisée soit semblable à la déclaration d'un sous-composant, un appel de sous-programme ne représente pas une instance de sous-programme. L'approche d'AADL correspond donc aux principes des langages de programmation : chaque sous-programme appelé est implicitement instancié une fois dans la mémoire du processus, et peut-être appelé plusieurs fois.

```
1 process processus_a
2 end processus_a;
3
4 thread thread_a
5 end thread_a;
6
7 process implementation processus_a.impl
8 subcomponents
```

```
9   thread1 : thread thread_a.impl;
10 end processus_a.impl;
11
12 thread implementation thread_a.impl
13 calls
14   sequence : {appel1 : subprogram sp_a;
15               appel2 : subprogram sp_a;};
16 end thread_a.impl;
17
18 subprogram sp_a
19 end sp_a;
```

Listing III.2 – Structure interne des composants AADL

III-4 Les éléments d'interface

Les éléments d'interface (*features*) d'un composant sont déclarée dans son type. De cette façon, toutes les implantations d'un type de composant offrent la même interface aux autres composants. Il existe plusieurs sortes de *features* : les ports de communication, les sous-programmes d'interface et les accès à sous-composants. Les déclarations de *features* peuvent n'être associées à aucune déclaration de composant, permettant ainsi une modélisation très abstraite. Cependant, il est nécessaire de préciser les composants associés afin de décrire une architecture exploitable pour générer un système exécutable.

III-4.1 Les ports

Les ports correspondent à la principale façon de décrire les transmissions d'information entre les composants. Ils sont déclarés en entrée, en sortie ou en entrée/sortie.

III-4.1.1 Ports simples

On distingue trois types de ports :
- les ports d'événement ;
- les ports de donnée ;
- les ports d'événement/donnée.

Les ports d'événement (*event ports*) correspondent à la transmission d'un signal. Ils peuvent être assimilés aux signaux des systèmes d'exploitation. Ils peuvent également déclencher l'exécution des threads.

Les ports de donnée (*data ports*) correspondent à la transmission de données. Contrairement aux ports d'événements, ils ne déclenchent rien à la réception ; ils peuvent donc modéliser un registre mémoire mis à jour de façon asynchrone.

Les ports d'événement/donnée (*event data ports*) sont la synthèse des deux premiers types de ports : ils permettent de transporter des données tout en générant un événement à la réception ; ils permettent donc de modéliser un message.

Les ports peuvent constituer les interfaces des composants concernés par les flux d'exécution : *threads*, groupes de *threads*, processus, dispositifs, systèmes et processeurs. Dans le cas des sous-programmes, on parle de paramètres. Les paramètres ont une sémantique équivalente aux ports de données ou d'événement/données.

III-4.1.2 Groupes de ports

La manipulation des ports de communication peut conduire à la déclaration d'ensembles constants. Par exemple, la description de composants communicant par l'intermédiaire d'un port série de type RS232 entraînera la déclaration répétée des ports correspondant aux signaux de cette interface. Pour faciliter la manipulation de ports souvent associés, le standard AADL définit la notion de groupe de ports (*port group*).

La déclaration d'un groupe de port est similaire à celle d'un type de composant. Il est possible de définir un groupe de port comme étant l'*inverse* d'un autre. L'inverse d'un groupe de ports est constitué des mêmes ports, dont les directions sont inversées.

III-4.2 Les sous-programmes d'interface

Les sous-programmes d'interface permettent de décrire des appels de méthode. Un sous-programme d'interface peut-être fourni par un composant de donnée ; dans ce cas il définit une méthode associée à une classe, reprenant ainsi les concepts définis par la programmation objet. Un sous-programme d'interface peut également être fourni par un *thread* ; il modélise un sous-programme invocable à distance (RPC, *Remote Procedure Call*).

Les sous-programmes d'interface sont un moyen alternatif de décrire les échanges de données entre les composants AADL. Les constructions syntaxiques qui leur sont associées dans la version 1.0 ne sont pas très développées. Notamment, il n'existe pas de construction syntaxique permettant d'exprimer le fait qu'un sous-programme appelé doit correspondre au sous-programme d'interface fourni par un *thread*. La modélisation complète d'une architecture basée sur un mécanisme de RPC s'en trouve compliquée. La version 1.2 du standard fournit les constructions syntaxiques nécessaires, comme nous le verrons en section III-9.

III-4.3 Les accès à sous-composant

Les accès à sous-composants permettent d'exprimer le fait qu'un sous-composant défini au sein d'un composant peut être partagé avec d'autres composants. Il existe deux types d'accès à sous-composants : pour les bus et les données. Dans les deux cas, ils sont déclarés comme étant des accès fournis (*provides*) ou requis (*requires*).

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

III-4.4 Synthèse sur les éléments d'interface

Des exemples de syntaxe pour la déclaration des éléments d'interface sont présentés au listing III.3.

```
1 system systeme_a
2 features
3   a : in data port;
4 end systeme_a;
5
6 system systeme_b extends systeme_a
7 features
8   a : refined to in data port donnee;
```

```
9   b : requires data access donnee;
10  c : port group groupe_de_ports;
11 end systeme_b;
12
13 data donnee
14 end donnee;
15
16 port group groupe_de_ports
17 features
18  p1 : in data port donnee;
19  p2 : in out event data port donnee;
20 end groupe_de_ports;
21
22 process process_a
23 features
24  e : in event port;
25  s : out event data port;
26 end process_a;
27
28 subprogram sous_programme_a
29 end sous_programme_a;
30
31 thread thread_a
32 features
33  sp : server subprogram sous_programme_a;
34 end thread_a;
```

Listing III.3 – Exemples d'éléments d'interface

De la même façon que pour les sous-composants, le standard AADL définit les éléments d'interface possibles pour chaque catégorie de composant. Le tableau III.2 fait les synthèses de ces possibilités.

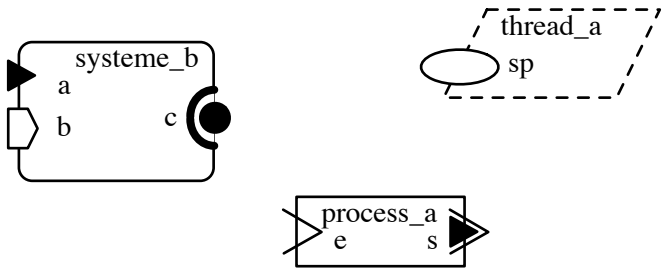


FIG. III.5 – Syntaxe graphique des éléments d'interface, correspondant au listing III.3

La syntaxe graphique des éléments d'interface est représentée sur la figure III.5.

III-5 Connexions des composants

Une description AADL est basée sur un assemblage de sous-composants connectés entre eux au moyen de *connexions*.

composant	éléments d'interface possibles
donnée	sous-programmes, accès fourni à une donnée
sous-programme	port de sortie d'événement ou d'événement/donnée, groupe de ports, accès requis à une donnée, paramètre
thread	sous-programme serveur, port, group de ports, accès requis ou fourni à une donnée
groupe de *thread*	sous-programme serveur, port, groupe de ports, accès requis ou fourni à une donnée
processus	sous-programme serveur, port, groupe de port, accès requis ou fourni à une donnée
processeur	sous-programme serveur, port, groupe de ports, accès requis à un bus
mémoire	accès requis à un bus
bus	accès requis à un bus
dispositif	port, groupe de port, sous-programme serveur, accès requis à un bus
système	sous-programme serveur, port, groupe de ports, accès requis ou fourni à une donnée ou un bus

TAB. III.2 – Éléments d'interface possibles pour chaque catégorie de composants

III-5.1 Les connexions

Les connexions permettent de relier les interfaces des différents sous-composants à celles d'autres sous-composants ou aux interfaces du composant parent. Une connexion est orientée et peut éventuellement être nommée.

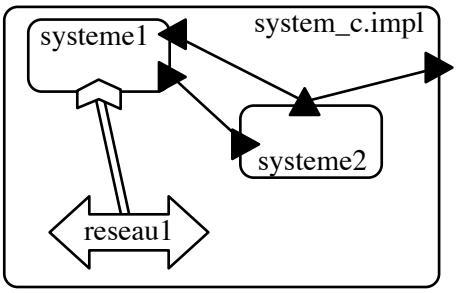


FIG. III.6 – Syntaxe graphique des connexions, correspondant au listing III.4

La figure III.6 représente la syntaxe graphique correspondant aux connexions décrites dans le listing III.4.

III-5.2 Les flux

La circulation des données est portée par les connexions, qui sont point-à-point. Afin de faciliter l'analyse de l'architecture, AADL permet de décrire des flux (*flows*) portés par les connexions. De cette façon, il est possible de décrire le cheminement logique des communications à travers

toute l'architecture.

AADL définit plusieurs types de flux :

– les flux de bout en bout (*end to end flows*) décrivent un flux complet ;
– les sources (*source flows*), les puits (*flow sink*) et les tronçons de flux (*flow path*) représentent respectivement le début, la fin et un tronçon d'un flot.

Un flux de bout en bout peut se décomposer en une source, des tronçons et un puits, qui sont assemblés en fonctions de la composition des composants.

Les flux ne traduisent pas de construction réelle ; ils sont un moyen de matérialiser la circulation des données à travers l'architecture afin de faciliter le processus d'analyse.

```
1  system systeme_a
2  features
3    a : in data port;
4    b : out data port;
5    reseau : requires bus access un_reseau;
6  flows
7    flux1 : flow path a -> b;
8  end systeme_a;
9
10 system systeme_b
11 features
12   a : in data port;
13   b : out data port;
14 flows
15   flux1 : flow source b;
16   flux2 : flow sink a;
17 end systeme_b;
18
19 bus un_reseau
20 end un_reseau;
21
22 system systeme_c
23 features
24   s : out data port;
25 end systeme_c;
26
27 system implementation systeme_c.impl
28 subcomponents
29   systeme1 : system systeme_a;
30   systeme2 : system systeme_b;
31   reseau1 : bus un_reseau;
32 connections
33   cnx1 : data port systeme2.b -> systeme1.a;
34   cnx2 : data port systeme1.b -> systeme2.a;
35   cnx3 : data port systeme2.b -> s;
36   bus access reseau1 -> systeme1.reseau;
37 flows
38   flux : end to end flow systeme2.flux1 -> cnx1 -> systeme1.flux1
39        -> cnx2 -> systeme2.flux2;
40 end systeme_c.impl;
```

Listing III.4 – Connexions et flux

Le listing III.4 illustre la description de connexions et de flux.

III-5.3 Matérialisation des connecteurs

AADL est conçu pour refléter l'architecture de systèmes réels. Les composants constituent donc les principales entités du langages, représentant les entités architecturales qui formeront les éléments principaux du système à générer.

La notion de connecteur dans AADL est donc reléguée au second plan ; elle n'est pas identifiée par une construction syntaxique unique. Les connecteurs sont principalement modélisés par les connexions AADL, qui ont une sémantique très restreinte. La description d'un connecteur en tant qu'entité de premier plan ne peut se faire qu'à travers un assemblage de composants connectés aux entités que le connecteur doit relier.

Dans ces conditions, l'utilisation des flux peut servir de support pour l'expression de propriétés s'appliquant de bout en bout du connecteur ainsi modélisé.

III-6 Configurations d'architecture

AADL est essentiellement axé sur la description de configurations d'architecture statiques ; la syntaxe permet néanmoins d'exprimer un certain degré de dynamisme. Le listing III.5 résume les constructions syntaxiques pour exprimer une configuration en AADL.

III-6.1 Développement et instanciation des déclarations

La syntaxe d'AADL permet de décrire une suite de déclarations ; les sous-composants font références à des instances des déclarations de composants. Afin d'obtenir la description architecturale en elle-même, il est nécessaire d'instancier les différentes déclarations, et ainsi obtenir une arborescence de composants instanciés. Pour cela, il faut définir un composant initial qui servira de racine à l'arborescence. Ce composant doit être un système sans interface ; il représente l'ensemble de l'architecture.

L'instanciation des déclarations AADL produit une arborescence de composants dont le système initial est la racine. Cette arborescence correspond à une certaine composition d'instances de composants AADL, représentant une configuration d'architecture.

III-6.2 Les modes

AADL décrit des architectures aux dimensions clairement définies : il n'est par exemple pas possible de d'exprimer le fait qu'un composant puisse avoir un nombre quelconque de sous-composants. Le langage ne se limite pourtant pas à la description d'architectures statiques : il est en effet possible d'introduire un certain dynamisme dans les modélisations AADL, par l'intermédiaire des modes.

Les modes AADL sont définis dans les implantations des composants. Ils correspondent à des configurations de fonctionnement, auxquels peuvent être associés des sous-composants, des connexions, etc. Ils définissent ainsi des configurations de fonctionnement pour les composants.

AADL permet de décrire les conditions de changement de mode, afin de définir une machine à états pour l'implantation du composant. Un changement de mode est déclenché par la réception d'un événements (transmis par un port d'événement).

Un composant ne peut être que dans un mode de configuration à la fois ; il n'y a pas de notion de sous-mode. Le standard précise que l'ensemble des modes de configurations d'une architecture complète correspond au produit cartésien des différents modes des composants instanciés dans l'architecture. L'utilisation des modes peut donc mener rapidement à une explosion combinatoire.

```
system global
end global;

process processus_a
features
  s : out event data port;
end processus_a;

process processus_b
features
  e : in event data port;
end processus_b;

process controleur
features
  c : out event port;
end controleur;

system systeme_a
features
  c : in event port;
end systeme_a;

system implementation systeme_a.impl
subcomponents
  processus1 : process processus_a;
  processus2 : process processus_b;
  processus3 : process processus_b in modes (double);
connections
  event data port processus1.s -> processus2.e;
  event data port processus1.s -> processus3.e in modes (double);
modes
  simple : initial mode;
  double : mode;
  simple -[ c ]-> double;
  double -[ c ]-> simple;
end systeme_a.impl;

system implementation global.config1
subcomponents
  systeme1 : system systeme_a.impl;
  controleur1 : process controleur;
connections
  event port controleur1.c -> systeme1.c;
end global.config1;
```

Listing III.5 – Système global et modes

III-7 Espaces de noms

Une description AADL est une succession de déclarations de composants et de groupes de ports. Afin d'organiser ces déclarations et de les regrouper par ensembles logiques, le langage

fournit la notion d'espace de nom. Il existe deux types d'espaces de noms : l'espace de nom anonyme (*anonymous namespace*), qui est l'espace de nom par défaut, et les paquetages (*packages*).

Les paquetages sont déclarés dans l'espace de nom anonyme. Ils possèdent une partie publique et éventuellement une partie privée (cf. listing III.6). Les déclarations contenues dans la partie publique peuvent être référencées depuis l'extérieur du paquetage, tandis que les déclarations faites dans la partie privée ne sont visibles que depuis le paquetage. La notion de sous-paquetage n'existe pas ; cependant, le nommage des paquetages AADL permet d'exprimer une hiérarchie.

L'utilisation des paquetages permet de regrouper plusieurs déclarations AADL pour former des ensembles logiques, de la même façon que les paquetages de Java ou les espaces de noms du C++. La syntaxe actuelle d'AADL ne permet pas de référencer un composant déclaré dans l'espace de nom anonyme à partir d'un paquetage. Par conséquent, les composants déclarés dans les paquetages forment des groupes autonomes, avec éventuellement des références à d'autres paquetages. Ainsi, un paquetage permet de fournir des déclarations qui pourront être instanciées par les composants de l'espace de nom anonyme.

Les paquetages structurent les déclarations tandis que systèmes structurent l'architecture.

```
1  package Capteurs
2  public
3    system systeme_a
4    features
5      res : requires bus access Reseaux::Reseau;
6    end systeme_a;
7
8    system implementation systeme_a.impl
9    subcomponents
10     capteur1 : device un_capteur;
11   connections
12     bus access res -> capteur1.res;
13   end systeme_a.impl;
14
15 private
16   device un_capteur
17   features
18     res : requires bus access Reseaux::Reseau;
19   end un_capteur;
20 end Capteurs;
21
22 package Reseaux
23 public
24   bus Reseau
25   end Reseau;
26 end Reseaux;
```

Listing III.6 – Paquetages AADL

III-8 Propriétés et annexes

AADL permet de décrire l'organisation des composants de l'architecture et leurs connexions. Il est possible de caractériser cette description structurelle l'aide des propriétés et des annexes afin de préciser un certain nombre de caractéristiques s'appliquant aux entités architecturales – temps

d'exécution, description comportementale d'un composant, protocole à utiliser pour un réseau, etc.

III-8.1 Propriétés

Les propriétés constituent un aspect fondamental d'AADL. Elles permettent d'exprimer les différentes caractéristiques des entités AADL telles que les composants, les sous-composants, les éléments d'interface, les connexions, etc. Il est ainsi possible de décrire les contraintes s'appliquant à l'architecture. Par exemple, les propriétés sont utilisées pour spécifier le temps d'exécution théorique d'un sous-programme, la période d'un thread, le protocole de file d'attente utilisé pour un port d'événement/donnée, etc.

III-8.1.1 Déclarations de propriétés

Les déclaration de propriétés sont regroupées dans des ensembles de propriétés (*property sets*), semblables aux paquetages. Il existe trois types de déclarations : les types de propriétés, les constantes et les noms de propriétés.

Généralités

Une propriété se définit par un nom, un type, la liste des éléments auxquels elle peut s'appliquer, et éventuellement une valeur par défaut. Les types de propriétés peuvent être définis à partir de types de base (chaîne de caractère, booléen, entier, réel, énumération, référence à une instance de composant ou référence à une déclaration de composant, plage de valeurs). Un type peut être une valeur simple ou une liste. Il est également possible de définir des unités. Des exemples de déclaration de propriété sont présentés sur le listing III.7.

```
property set Utilisateur is
  Compiler : aadlstring => "gcc" applies to (subprogram,
      thread);
  Pressure : type units (Pa, hPa => Pa * 100);
  Pressure_Range : range of Pressure applies to (device);
  Version : aadlinteger applies to (all);
end Utilisateur;
```

Listing III.7 – Déclarations de propriétés

Le standard AADL définit deux ensembles de propriétés : `AADL_Properties` contient les déclarations des propriétés standard et `AADL_Project` celles des propriétés de projet. Les propriétés standard ne peuvent pas être modifiées ; en revanche les propriétés de projet peuvent être adaptées en fonction des dimension de l'architecture qu'on l'on décrit ou des technologies que l'on a sa disposition.

```
property set AADL_Properties
  Source_Language : Supported_Source_Languages
    applies to (subprogram, data, thread, process, bus,
        device, processor);

  Thread_Limit : aadlinteger 0 .. value (Max_Thread_Limit)
    => value (Max_Thread_Limit)
    applies to (processor);
end AADL_Properties;
```

```
9
10 property set AADL_Project is
11   Max_Thread_Limit : constant aadlinteger
12     => <project-specified-integer-literal>;
13
14   Supported_Source_Language : type enumeration (<project-specified
       >);
15 -- The following are examples software source languages:
16 -- (Ada95, C, Simulink_6_5)
17 end AADL_Project;
```

Listing III.8 – Extrait des déclarations de propriétés standard

Le listing III.8 donne un extrait des déclarations de propriétés standard qui illustre le paramétrage des propriétés. La définition de certaines propriétés standard font référence à des définitions relatives au projet. L'ensemble `AADL_Project` dépend donc des outils AADL, qui doivent le définir en accord avec leurs propres possibilités ou limitations.

L'ensemble des propriétés standard permet d'exprimer un certain nombre de caractéristiques sur les entités AADL. Nous en décrivons ici quelques unes, qui sont pertinentes pour nos travaux.

Périodes et politiques de déclenchement des threads

Le standard définit quatre politiques de déclenchement pour les *threads*, à l'aide de la propriété `Dispatch_Protocol` :

- périodique ;
- apériodique ;
- sporadique ;
- tâche de fond (*background*).

Les *threads* périodiques se déclenchent d'eux-mêmes selon la période indiquée par la propriété `Period`. Les *threads* apériodiques se déclenchent sur l'arrivée d'un événement ou d'une donnée/événement sur l'un de leurs ports. Les *threads* sporadiques se déclenchent également à l'arrivée d'un événement, mais avec une période de garde indiquée par la propriété `Period`. Les *threads* en tâche de fond s'exécutent en continu.

Descriptions des implantations

Le standard défini trois propriétés permettant d'associer un code source aux composants :

- `Source_Language` permet de spécifier le langage utilisé dans la description ;
- `Source_Text` permet d'indiquer la liste des fichiers contenant la description ;
- `Source_Name` permet de préciser l'entité référencée dans le fichier (par exemple le nom de la procédure).

La sélection du langage peut s'appliquer aux sous-programmes, données, *threads*, processus, bus, dispositifs et processeurs, permettant ainsi de décrire les composants logiciels (sous-programmes, par exemple en C) ou matériels (processeurs, par exemple en VHDL).

Contraintes spatiales et temporelles

Diverses propriétés permettent d'exprimer les temps d'exécution ou de propagation à travers les composants. Ainsi, la propriété `Propagation_Delay` permet de spécifier une plage de temps correspondant au temps de propagation d'un signal à travers un bus, `Read_Time` permet d'indiquer

le temps d'accès en lecture à une mémoire, `Subprogram_Execution_Time` définit la plage de temps d'exécutions pour un sous-programme, etc.

De même, `Source_Code_Size`, `Source_Heap_Size` permettent d'exprimer les contraintes en taille mémoire.

Association des composants logiciels à la plate-forme d'exécution

Le déploiement des composants applicatifs sur la topologie matérielle est spécifiée par des propriétés AADL.

`Actual_Connection_Binding`, `Actual_Memory_Binding` et `Actual_Processor_Binding` peuvent être utilsées pour indiquer respectivement par quel bus, mémoire ou processeur est porté une connexion, une donnée ou processus. Par ailleurs, des propriétés telles que `Thread_Limit` permettent de spécifier les limitations du système d'exploitation des processeurs en terme de capacité d'ordonnancement.

III-8.1.2 Associations de propriétés

Les associations de propriétés permettent d'attribuer une valeur à une propriété (c'est-à-dire associer une valeur à un nom de propriété).

Déclarations

Les associations de propriétés peuvent intervenir dans trois situations différentes :
- déclarées dans un paquetage ;
- déclarées dans la section « properties » d'un composant ;
- directement attachées à une sous-clause.

Lorsqu'une association est réalisée au niveau d'un paquetage, elle concerne tous les composants du paquetage auxquels la propriété peut s'appliquer.

Une propriété déclarée dans la section « properties » d'un composant s'applique dans le cadre de ce composant. La déclaration s'applique au composant en question, sauf si l'association contient le mot-clé **`applies to`** ; dans ce cas la propriété s'applique à la déclaration désignée par **`applies to`**. Cette déclaration peut être une sous-clause du composant, ou une sous-clause du composant référencé par la sous-clause, et ainsi de suite, selon ce qu'indique le **`applies to`**.

Si l'association est réalisée au niveau d'une sous-clause (par exemple. un sous-composant), elle s'applique dans le cadre de la sous-clause en question. Si le mot-clé **`applies to`** est utilisé, la propriété s'applique à une sous-clause du composant référencé par la sous-clause. Il s'agit donc du même principe que pour le cas précédent, mais appliqué à une sous-clause au lieu de la déclaration d'un composant.

Valeurs associées aux propriétés

La valeur associée à une propriété peut être une valeur simple ou une liste de valeurs simples, selon le type de la propriété. Dans le cas d'une liste, il est possible d'*ajouter* des éléments à la valeur déjà définie pour la propriété, dans le cas où une valeur aurait déjà été associée à la propriété (valeur par défaut dans la déclaration ou déjà définie par ailleurs). Une valeur peut également être une référence à la valeur d'une autre association de propriété ou à une constante.

```
1 processor processeur_a
2 end processeur_a;
```

```
3
4 processor implementation processeur_a.f40MHz
5 properties
6   Clock_Period => 25 ns;
7 end processeur_a.f40MHz;
8
9 system ordinateur
10 end ordinateur;
11
12 system implentation ordinateur.overclock
13 subcomponents
14   processeur1 : processor processeur_a.f40MHz {Clock_Period => 20
       ns;};
15 end ordinateur.overclock;
```

Listing III.9 – Associations de valeurs aux propriétés

La valeur d'une propriété est déterminée en cherchant sa valeur dans l'ordre suivant (illustré sur la figure III.7) :

1. dans l'instance de l'entité ;
2. dans la déclaration de l'implantation correspondante ;
3. dans les éventuelles déclarations d'implantation dont dérive l'implantation du composant ;
4. dans la déclaration du type correspondant à l'implantation ;
5. dans les éventuelles types de composant dont dérive le type du composant ;
6. dans l'instance parente.

Une propriété à laquelle aucune valeur n'est associée est dite indéfinie.

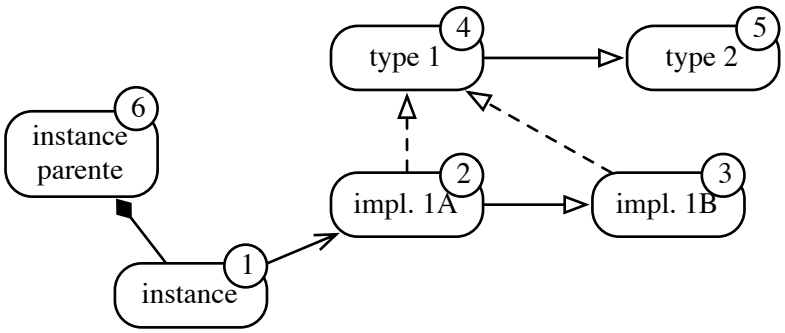


FIG. III.7 – Ordre d'évaluation des propriétés

III-8.2 Annexes

Les annexes sont un autre moyen d'associer des informations aux éléments d'une description. Contrairement aux propriétés, elles ne peuvent être associées qu'aux déclarations de composants et permettent d'insérer des informations exprimée dans une syntaxe indépendante. L'utilisation des annexes permet donc d'étendre la syntaxe standard d'AADL afin de spécifier le comportement des composants [SAE, 2006b], introduire un modélisation des erreurs [SAE, 2005 ; Rugina et al.,

2006], etc. Le listing III.10 illustre un exemple d'annexe AADL tiré de l'annexe comportementale d'AADL [SAE, 2006b].

```
subprogram start_read
features
  debug: in parameter Behavior::boolean;
end start_read;

subprogram implementation start_read.i
calls {
  p1: subprogram std::print;
};

annex behavior_specification {**
states
  s0 : initial state;
  s1 : return state;
transitions
  s0 -[ on debug ]-> s1 { std::print! };
  s0 -[ on not debug ]-> s1 {};
**};
end start_read;
```

Listing III.10 – Annexes AADL

III-8.3 Discussion

Les annexes et les propriétés permettent d'ajouter des informations à la description architecturale. Alors que les annexes ne concernent que les composants, les propriétés peuvent être associées à tous les éléments d'une description : composants, connexions, *features*, etc. Les annexes permettent d'incorporer des éléments rédigés dans une syntaxe différente de celle d'AADL. On peut donc considérer qu'elles permettent donc d'étendre la syntaxe AADL tout en permettant l'utilisation d'outils existants. Les propriétés font partie intégrante de la syntaxe AADL, et donc plus adaptées pour décrire les caractéristiques des architectures.

III-9 Évolutions de la syntaxe d'AADL

La première version du standard AADL, que nous venons de décrire, s'inspire très largement de son ancêtre MetaH ; l'une des principales conséquences est que la description des éléments applicatifs – principalement les sous-programmes – est assez primitive. Afin de pouvoir pleinement exprimer les éléments d'architecture logicielle, il est nécessaire d'enrichir la syntaxe.

La version 1.0 du standard définit les composants sous-programmes selon la même approche que dans les langages impératif. Cela facilite le passage entre AADL et des langages comme Ada ou C mais restreint l'expressivité des descriptions AADL. Par ailleurs, le standard AADL actuel ne permet pas de modéliser facilement un appel de sous-programme distant (RPC). Dans cette section nous décrivons des évolutions du langage AADL que nous avons proposées au comité de standardisation afin d'améliorer ces aspects.

III-9.1 Modélisation des séquences d'appel

III-9.1.1 Limitation du standard 1.0

Les sous-programmes AADL ne peuvent pas être explicitement instanciés dans une architecture. Il n'est donc pas possible d'y faire référence, contrairement aux autres composants, tels que les bus. Ce traitement particulier entraîne une certaine rigidité dans les modélisations.

Ainsi, les différents sous-programmes référencés par une séquence d'appel le sont de manière statique, comme l'illustre le listing III.11. L'implantation `thread_a.impl1` du *thread* `thread_a` appelle `sp_a.impl1`. Si nous souhaitons décrire une autre configuration où `thread_a` appelle `sp_a.impl2`, il est nécessaire d'écrire une autre implantation du *thread*.

```
1  thread thread_a end thread_a;
2  subprogram sp_a end sp2;
3
4  subprogram sp_a.impl1 end sp_a.impl1;
5  subprogram sp_a.impl2 end sp_a.impl2;
6
7  subprogram implementation thread_a.impl1
8  calls {
9    appel1 : subprogram sp_a.impl1;
10 };
11 end thread_a.impl1;
12
13 subprogram implementation thread_a.impl2
14 calls {
15   appel1 : subprogram sp_a.impl2;
16 };
17 end thread_a.impl2;
```

Listing III.11 – Modélisation de séquences d'appel en AADL 1.0

Parallèlement, du fait de l'instanciation implicite des sous-programmes appelé, la syntaxe d'AADL 1.0 ne permet pas de spécifier qu'un sous-programme appelé est fourni par un autre *thread* que le *thread* courant. De la même façon, aucune construction syntaxique ne permet d'indiquer un appel à un sous-programme fourni par un composant de donnée. Dans ces deux situations, le standard propose de recourir à un mécanisme d'annexe afin de préciser à quel composant les sous-programmes appelé sont attachés ; ne pas intégrer ce genre d'information directement dans les constructions architecturales complique l'exploitation des descriptions.

III-9.1.2 Extension de la syntaxe

La solution que nous proposons est de promouvoir les sous-programmes AADL au même rang que les autres catégories de composants, et ainsi de pouvoir les instancier.

Principes

Dans ces hypothèses, les processus, les *threads* et les données peuvent intégrer des sous-programmes comme sous-composants. Les processus étant des espaces de mémoire, la signification d'une telle composition est relativement simple. Dans le cas des *threads* et des données, le sous-programme sous-composant est implicitement déclaré dans les processus parent, mais uniquement accessible depuis le thread ou la donnée.

Deux nouveaux types d'interfaces doivent être ajoutés :

– **requires subprogram access**

– **provides subprogram access**

Ces deux nouveaux éléments d'interface ont la même syntaxe que les accès aux bus ou aux données. Ils s'appliquent aux sous-programmes, *thread*, processus, données et systèmes. Un sous-programme ne peut néanmoins pas fournir d'accès.

La connexion de tels interfaces est autorisée entre un type de composant et une implantation. Par exemple, une implantation de composant peut être connectée à une interface associée au type de composant correspondant. Il est ainsi possible de connecter différentes implantations d'un sous-programme à une interface donnée.

De la même façon, nous introduisons une construction syntaxique supplémentaire pour les séquences d'appel. Deux types d'appels sont alors possibles :

– **subprogram**

– **subprogram access**

De cette façon, nous différencions le paradigme d'appel « traditionnel » de celui que nous ajoutons.

Application

La nouvelle syntaxe facilite la sélection du sous-programme qui doit effectivement être appelé, comme l'illustre le listing III.12. La sélection du sous-programme à appeler ne se fait plus au niveau de l'implantation du *thread* mais au moment de son instanciation au sein du processus. La sélection du sous-programme devient donc un paramètre de configuration, et plus une caractéristique intrinsèque du *thread*.

```
1  thread thread_a
2  features
3    p : requires subprogram access sp2;
4  end thrad_a;
5
6  subprogram sp_a end sp_a;
7
8  subprogram sp_a.impl1 end sp_a.impl1;
9  subprogram sp_a.impl2 end sp_a.impl2;
10
11 thread implementation thead_a.impl
12 calls {
13   appel1 : subprogram access p;
14 };
15 end thread_a.impl;
16
17 process processus_a
18 end processus_a;
19
20 process implementation processus_a.config1
21 subcomponents
22   sp1 : subprogram sp_a;
23   thread1 : thread thread_a.impl;
24 connections
25   subprogram access sp1 -> thread1.p;
```

```
end processus_a.config1;
```

Listing III.12 – Sélection du sous-programme appelé

De la même façon, l'expression d'un appel distant se fait par les connexion des éléments d'interface, comme le montre le listing III.13. Pour des raisons de concision, nous avons fait figurer le *thread* fournissant le sous-programme dans le même processus ; la mise en place d'un « véritable » appel distant correspondrait à placer le *thread* `thread2` dans un autre processus.

```
thread thread_b
features
  p : provides subprogram access sp_a;
end thread_b;

process implementation processus_a.config2
subcomponents
  thread1 : thread thread_a.impl;
  thread2 : thread thread_b;
connections
  subprogram access thread2.p -> thread1.p;
end processus_a.config2;
```

Listing III.13 – Modélisation d'un appel distant

La syntaxe pour un sous-programme associé à une donnée est très semblable.

III-9.2 Connexion des paramètres

III-9.2.1 Limitations du standard 1.0

La syntaxe AADL actuelle permet de connecter les paramètres (ou les ports s'il s'agit de threads). Il est ainsi possible de décrire la transmission de l'information entre un *thread* ou un sous-programme, et les sous-programmes qu'il appelle.

En revanche, il est impossible de modéliser le fait que certaines données puissent être crées au niveau de l'implantation en code source du sous-programme puis transmises aux sous-programmes appelés. Le listing III.14 illustre cette limitation : aucune construction syntaxique ne permet d'exprimer le fait que la donnée transmise au paramètre `e2` du sous-programme `sp1` lors de l'appel `appel1` est générée par du code applicatif associé à `sp2.impl`

```
data donnee
end donnee;

subprogram sp1
features
  e : in parameter donnee;
  s : out parameter donnee;
end sp1;

subprogram sp2
features
  e1 : in parameter donnee;
  e2 : in parameter donnee;
  s : out parameter donnee;
end sp2;
```

```
16
17 subprogram implementation sp1.impl
18 subcomponents
19   variable : data donnee;
20 calls {
21   appel1 : subprogram sp2;
22 };
23 connections
24   parameter e -> appel1.e1;
25   parameter variable -> appel1.e2;
26   parameter appel1.s -> s;
27 end sp1.impl;
```

Listing III.14 – Connexion interne des paramètres

Une façon de contourner cette limitation serait de créer artificiellement un sous-programme `sp2` possédant un paramètre de sortie ; `sp2` symboliserait le calcul de la donnée manquante. Dans ce cas, `sp1.impl` appellerait d'abord `sp2` puis `sp1` ; Le modèle AADL ferait apparaître une connexion entre la sortie de `sp2` et l'entrée correspondante de `appel1.e2`. Il s'agirait cependant d'une autre architecture ; cette approche ne convient donc pas.

III-9.2.2 Extension de la syntaxe

Afin de modéliser les connexions entre des données calculées par les implantations en code sous des sous-programmes et les sous-programmes qu'ils appellent, il est nécessaire de faire en sorte que les sous-programmes et les threads puissent avoir des composants de données comme sous-composants. Ces composants de données modélisent alors des variables. Le listing III.15 illustre la nouvelle construction syntaxique.

```
1  data donnee
2  end donnee;
3
4  subprogram sp1
5  features
6    e : in parameter donnee;
7    s : out parameter donnee;
8  end sp1;
9
10 subprogram sp2
11 features
12   e1 : in parameter donnee;
13   e2 : in parameter donnee;
14   s : out parameter donnee;
15 end sp2;
16
17 subprogram implementation sp1.impl
18 calls {
19   appel1 : subprogram sp2;
20 };
21 connections
22   parameter e -> appel1.e1;
23   parameter appel1.s -> s;
```

```
24 end sp1.impl;
```

Listing III.15 – Connexion interne des paramètres

III-10 Conclusion

AADL permet de décrire les architectures avec une approche très concrète ; le langage offre un ensemble de catégories de composant à la sémantique bien définie. Des notions telles que les connecteurs sont très peu développées. AADL vise donc un niveau de modélisation centré sur la description des nœuds de l'architecture. En ce sens il peut être vu comme un langage de pré-implantation.

AADL se focalise sur les aspects architecturaux : il permet la description des dimensions des composants et leur connexions, mais ne traite pas directement de leur implantation comportementale, ni de la sémantique des données manipulées. Cet aspect de la description peut être ajouté au moyen d'annexes, ou en associant des descriptions externes à l'aide des propriétés. De la même façon, les différentes contraintes s'appliquant au système ou le déploiement des applications sur les topologies matérielles peuvent être exprimées au moyen de propriétés.

Le langage permet ainsi de décrire la structure d'une application par une collection de composants logiciels s'exécutant sur des composants matériels ; tous ces composants peuvent être regroupés de façon logique au sein de systèmes.

Dans sa version 1.0, AADL a une approche très primitive de la description des éléments applicatifs. Notamment, la sélection des appels de sous-programme sont relèvent de la structure interne des sous-programmes et des *threads*, ce qui complique la description de l'assemblage d'applications ou la modélisation d'appels de procédures distantes. Par ailleurs, la syntaxe actuelle ne permet pas de décrire des variables internes aux sous-programmes, ce qui restreint la description des interactions entre les constructions architecturales et les descriptions comportementales.

Dans ce chapitre nous avons introduit des extensions à la syntaxe du standard AADL 1.0 afin de permettre l'instanciation des sous-programmes et la modélisation de variables locales. De cette façon, nous pouvons modéliser l'assemblage des composants applicatifs AADL de la même manière que pour les autres composants, et décrire les points d'interaction entre l'architecture et les descriptions comportementales. Ceci fournit une grande flexibilité pour la réutilisation et le re-composition des composants, et fournit tous les éléments nécessaires pour décrire des composants logiciels.

Ces modifications de syntaxe seront intégrées dans la future version 1.2 du standard AADL.

Dans les chapitres suivants, nous montrerons comment exploiter l'assemblage des composants, les connexions et les propriétés pour décrire une application répartie et les contraintes d'exécutions associées, et nous présenterons comment exploiter ces informations pour s'assurer de la validité des architectures et produire des systèmes exécutables.

CHAPITRE IV

Utilisation d'AADL pour décrire une application répartie

LA SYNTAXE D'AADL permet une grande flexibilité dans la façon de décrire une architecture. Il est possible d'adopter différents niveaux de précision selon la finalité de la description : documentation, génération automatique de code, évaluation des performances du système, etc. Cette flexibilité implique de définir un certain nombre d'orientations dans la façon de décrire les architectures.

Dans ce chapitre nous montrons comment utiliser AADL pour supporter une démarche de modélisation. Nous abordons principalement deux approches différentes, l'une centrée sur la description du déploiement, et l'autre prenant en charge la description complète de l'application vis-à-vis des critères que nous avons énoncés au chapitre II, à savoir l'interfaçage avec l'application, la configuration de chaque instance de l'intergiciel et le déploiement sur le réseau.

IV-1 Utilisation d'AADL pour décrire le déploiement d'une application

Nous exposons dans cette section une façon d'utiliser AADL pour décrire le déploiement d'une application répartie. Nous utilisons alors AADL comme langage de description, de la même façon que le langage utilisé par GLADE pour décrire le déploiement d'une application basée sur DSA. Ce langage permet de décrire tous les éléments de configuration au sein d'un même fichier en adoptant une organisation plate. Il permet de déclarer les différentes partitions ainsi que des attributs qui s'y rapportent.

IV-1.1 Identification des composants pour la description de la topologie

Pour décrire la répartition des différents nœuds de l'application, nous nous plaçons à un niveau d'abstraction assez élevé, dans lequel la structure des nœuds n'est pas explicitée. Nous nous focalisons sur la description du déploiement et des relations entre les nœuds, sans préciser la nature des communications. Les composants à considérer sont donc principalement les processus et les processeurs. Ces deux catégories de composants permettent en effet de décrire complètement la répartition d'une application. Les systèmes AADL permettent de structurer l'architecture.

Dans la mesure où ce niveau de description architecturale ne rend pas compte du détail des communications, nous pouvons décrire celles-ci de la façon la plus abstraite possible. Pour cela, nous pouvons associer un port de donnée/événement en entrée/sortie à chaque processus de la modélisation. Ces ports ne sont associé à aucune déclaration de composant de donnée ; la description

architecturale reste donc très abstraite. Les relations entre les différents processus se traduisent par des connexions entre les ports des processus.

La description AADL de la répartition d'une application se base donc sur la déclaration de composants processus et processeurs banalisés, comme illustré sur le listing IV.1. Afin de simplifier la démarche de modélisation, nous pouvons rassembler dans un paquetage AADL ces déclarations de composants génériques ; elles seront instanciées et particularisées dans l'implantation de système décrivant l'architecture en elle-même.

```
1 package GLADE
2   process Partition
3   features
4     communications : in out event data port;
5   end Partition;
6
7   processor Machine
8   end Machine;
9 end GLADE;
```

Listing IV.1 – Composants AADL génériques

Ces déclarations n'ont rien de contraignant ; elles permettent simplement d'éviter à l'utilisateur d'avoir à déclarer systématiquement ces composants de base. Elles doivent être instanciées au niveau de l'implantation d'un système global représentant l'architecture complète, comme représenté sur le listing IV.2 : deux partitions communiquent, chacune s'exécutant sur un processeur ; les ports des processus sont connectés pour modéliser les communications entre les deux partitions.

```
11 system Global
12 end Global;
13
14 system implementation Global.implem
15 subcomponents
16   partition_1 : process GLADE::Partition;
17   partition_2 : process GLADE::Partition;
18   processeur_1 : processor GLADE::Machine;
19   processeur_2 : processor GLADE::Machine;
20 connections
21   event data port partition_1.communications -> partition_2.
       communications;
22 properties
23   actual_processor_binding => reference (processeur_1) applies to
        partition_1;
24   actual_processor_binding => reference (processeur_2) applies to
        partition_2;
25 end Global.implem;
```

Listing IV.2 – Mise en place des composants

Nous pouvons noter que le réseau de communication n'est pas modélisé puisque nous nous limitons à une approche de haut niveau, selon un point de vue essentiellement logiciel. Les informations telles que la bande passante ou la latence de transmission des réseaux de communication ne sont pas pris en compte dans cette approche ; les éventuelles informations concernant la transmission des données entre les partitions devraient être indiquées au moyen de propriétés associées

à la connexion. Les processeurs AADL ne sont utilisés que comme un moyen propre de décrire la localisation des partitions.

Cette approche de modélisation permet de restreindre AADL à la description du déploiement des différents nœuds de l'application.

IV-1.2 Intégration des informations de déploiement

Les composants AADL permettent de décrire la topologie de l'architecture répartie. Les informations telles que l'emplacement des nœuds correspondent à la spécifications de caractéristiques associées aux instances des composants.

Pour déterminer les informations pertinentes à associer aux constructions AADL, nous nous sommes inspirés des travaux menés autour de l'implantation de DSA, GLADE [Pautet et Tardieu, 2005]. GLADE reconnaît un langage de configuration permettant de décrire les différentes partitions de l'application, ainsi qu'un certain nombre de paramètres permettant de caractériser les communications ou le contenu des nœuds.

Ces éléments de configuration correspondent à des caractérisations des entités architecturales de base (partitions ou liaisons inter-partition) ; elle se traduisent en AADL sous la forme de propriétés associées aux processus, processeurs ou connexions AADL.

La déclaration d'une partition dans GLADE se fait de la façon suivante :

```
1 Partition_1 : Partition;
```

Cette déclaration se traduit par une construction AADL telle que celle décrite au listing IV.2, en faisant référence aux déclarations de composants définies au listing IV.1.

Le langage de GLADE définit un certain nombre d'attributs se rapportant à une partition déjà déclarée. La syntaxe pour la spécification de la méthode de terminaison pour une partition est définie ainsi :

```
1 TERMINATION_LITERAL ::= Global_Termination |
2                         Local_Termination |
3                         Defered_Termination
4
5 REPRESENTATION_CLAUSE ::=
6    for PARTITION_IDENTIFIER'Termination use TERMINATION_LITERAL
```

Les politiques de terminaison possibles correspondent à des mot-clé définis par la BNF du langage. Leur traduction en AADL correspond naturellement à un type énuméré, représenté au listing IV.3.

```
19 Termination_Type : type enumeration (Global_Termination,
20                                      Local_Termination,
21                                      Defered_Termination);
```

Listing IV.3 – Définition des valeurs pour la terminaison des partitions

L'application des attributs se traduit par l'association des propriétés AADL correspondantes aux composants, comme l'illustre le listing suivant :

```
141 Termination : Glade::Termination_Type
142             => Global_Termination applies to (process);
```

Listing IV.4 – Terminaison des partitions

Tous les attributs du langage de GLADE sont transformés en propriétés AADL.

De la même façon, la localisation des nœuds sur le réseau est indiquée par un attribut associé aux partitions :

```
REPRESENTATION_CLAUSE ::=
   for PARTITION_IDENTIFIER'Host use STRING_LITERAL;
 | for PARTITION_IDENTIFIER'Host use FUNCTION_IDENTIFIER;
```

L'utilisation d'AADL fait apparaître la distinction entre partition logicielle et composant matériel supportant l'exécution des partitions. La localisation des partitions est donc exprimée par une propriété associée aux processeurs :

```
Host_URL : aadlstring applies to (processor);
```

Listing IV.5 – Emplacement des nœuds

GLADE est dans ce cas plus concis puisqu'il fait abstraction des éléments matériels qui sous-tendent le déploiement de l'application. Les informations de localisation des partitions leur sont directement associées. La description des communications possibles entre les partitions fait en revanche l'objet de déclarations séparées :

```
for Partition_1'Host use "127.0.0.1";
for Partition_2'Host use "127.0.0.1";
for Partition_3'Host use "127.0.0.1";
Canal1_2 : Channel := (Partition1, Partition2);
Canal2_3 : Channel := (Partition2, Partition3);
Canal3_1 : Channel := (Partition3, Partition1);
```

Listing IV.6 – Description du déploiement des nœuds avec Gnatdist

La construction AADL équivalente est plus complexe, mais plus structurée :

```
system implementation Global.implem
subcomponents
  partition_1 : process Partition;
  partition_2 : process Partition;
  partition_3 : process Partition;
  processeur_1 : processor Machine {GLADE::Localisation =>
      "127.0.0.1"};
connections
  event data port partition_1.communications -> partition_2.
      communications;
  event data port partition_2.communications -> partition_3.
      communications;
  event data port partition_3.communications -> partition_1.
      communications;
properties
  actual_processor_binding => reference (processeur_1) applies to
       partition_1;
  actual_processor_binding => reference (processeur_1) applies to
       partition_2;
  actual_processor_binding => reference (processeur_1) applies to
       partition_3;
end Global.implem;
```

Listing IV.7 – Description du déploiement des nœuds en AADL

Elle permet de regrouper les informations de localisation. Les connexions AADL représentent les canaux de communication entre les partitions.

La syntaxe AADL permet de séparer clairement d'une part l'implantation des nœuds applicatifs et d'autre part la topologie de l'application, ce qui facilite le redéploiement d'une application répartie ainsi décrite. Le listing suivant illustre un déploiement sur deux machines :

```
1 system implementation Global.implem
2 subcomponents
3   partition_1 : process Partition;
4   partition_2 : process Partition;
5   partition_3 : process Partition;
6   processeur_1 : processor Machine {GLADE::Localisation =>
        "137.194.160.84"};
7   processeur_2 : processor Machine {GLADE::Localisation =>
        "137.194.192.52"};
8 connections
9   event data port partition_1.communications -> partition_2.
        communications;
10  event data port partition_2.communications -> partition_3.
        communications {GLADE::Filter => "ZIP"};
11  event data port partition_3.communications -> partition_1.
        communications {GLADE::Filter => "ZIP"};
12 properties
13  actual_processor_binding => reference (processeur_1) applies to
        partition_1;
14  actual_processor_binding => reference (processeur_2) applies to
        partition_2;
15  actual_processor_binding => reference (processeur_1) applies to
        partition_3;
16 end Global.implem;
```

Listing IV.8 – Autre déploiement des nœuds en AADL

Dans cet exemple nous indiquons que certains canaux de communications doivent être compressés. Ces spécifications s'intègrent de façon naturelle à la description du déploiement ; la syntaxe classique de GLADE les exprime par une déclaration complémentaire :

```
1 for Partition_1'Host use "137.194.160.84";
2 for Partition_2'Host use "137.194.192.52";
3 for Partition_3'Host use "137.194.160.84";
4 Canal1_2 : Channel := (Partition1, Partition2);
5 Canal2_3 : Channel := (Partition2, Partition3);
6 Canal3_1 : Channel := (Partition3, Partition1);
7 for Canal2_3'Filter use "ZIP";
8 for Canal3_1'Filter use "ZIP";
```

Listing IV.9 – Autre déploiement des nœuds avec Gnatdist

IV-1.3 Discussion

L'utilisation d'AADL pour modéliser la répartition d'application fait intervenir à la fois des constructions architecturales (c'est-à-dire des instances de composant) et un ensemble de propriétés AADL permettant de décrire les conditions de déploiement. L'implantation des différents

nœuds de l'application demeure en dehors du cadre d'AADL, et doit par exemple être réalisée à l'aide d'un langage de programmation.

Nous avons retranscrit en AADL les paramètres de configuration reconnus par GLADE sous la forme d'un ensemble de composants et de propriétés AADL ; nous nous appuyons sur un système de configuration concret. En pouvant ainsi exprimer un fichier de configuration de GLADE en AADL, nous montrons que notre approche est viable, puisqu'il est possible d'utiliser AADL comme langage alternatif pour une application industrielle.

Dans la mesure où elles sont relativement concises, les descriptions architecturales que nous pouvons faire peuvent également être exploitées à des fins de documentation pour décrire l'allure générale des applications.

Néanmoins, nous n'exploitons en fait qu'une partie des possibilités offertes par le langage, et nous ne décrivons que la répartition des applications. Cette approche permet une approche de description très souple, mais sous-utilise donc AADL. Par ailleurs, nous ne spécifions pas la partie applicative : nous n'indiquons que les communications *possibles*, sans pouvoir déterminer leur fréquence ni la structure exacte des données échangées.

Grâce à la richesse de sa syntaxe et de sa sémantique, AADL peut être utilisé pour modéliser les applications selon une approche beaucoup plus fine ; nous pouvons alors envisager de décrire complètement les applications – et pas seulement leur déploiement sur les différents nœuds. L'utilisation d'AADL pour décrire complètement les applications est décrite dans les sections suivantes.

IV-2 Vers un développement conjoint de l'application et de l'intergiciel

Dans la section précédente nous avons montré comment utiliser AADL pour décrire le déploiement d'une application. Cela suppose l'utilisation d'un intergiciel configuré par ailleurs. Le développement séparé de l'intergiciel implique que celui-ci est relativement indépendant de l'application qu'il sous-tend. Dans ces hypothèses, l'adéquation de l'intergiciel avec l'application peut être évaluée *a posteriori*, typiquement à l'aide de suites de tests. Cette approche pose deux problèmes majeurs :

- le test d'un intergiciel est en général très difficile à réaliser, étant donné que ce genre de logiciel présente beaucoup de cas d'utilisation ;
- le développement séparé de l'intergiciel implique une certaines décorrélation par rapport à l'application, qui peut rendre difficile l'évaluation des caractéristiques de l'ensemble formé par l'application et son intergiciel ; cela concerne par exemple l'évaluation des temps d'exécution ou la taille totale en mémoire.

Une solution appropriée à ce problème serait de concevoir un intergiciel spécifiquement adapté à l'application considérée. Cet intergiciel serait conçu en même temps que l'application, et constituerait donc une sous-couche applicative à part entière. Une application extrême de ce principe revient en fait à ne pas avoir d'intergiciel, en laissant au concepteur de l'application le soin de gérer tous les mécanismes de communications ; cela engendrerait un coût de développement prohibitif.

Notre objectif est donc de maintenir l'existence de l'intergiciel en tant qu'assemblage d'éléments logiciels réutilisables, tout en associant étroitement son élaboration au processus de création de l'application. Une approche efficace pour la réalisation de l'intergiciel consiste donc à intégrer la configuration de l'intergiciel dans le processus de conception de l'application.

AADL nous fournit cela un formalisme permettant de décrire précisément les caractéristiques de l'application pour en déduire les éléments de configuration de l'intergiciel. L'utilisation d'AADL comme support central pour tous les aspects de la modélisation permet de rationaliser le processus de développement et de vérification.

IV-3 Cycle de développement

De par sa complexité, le développement d'une application répartie pour un système embarqué temps-réel nécessite une vérification régulière du respect des spécifications. Une approche de développement itérative (ou cycle en « spirale ») permet une rétroaction entre l'application et ses spécifications initiales. Une telle approche est dite « par prototypage » ; elle consiste en une démarche de conception pas à pas permettant la validation régulière de l'architecture. Il est ainsi possible de détecter les problèmes relativement tôt, ce qui permet d'éviter de coûteuse modifications sur l'architecture finale.

IV-3.1 Principes du prototypage

Le terme « prototypage » recouvre deux notions différentes [Kordon et Luqi, 2002]. Le premier type de prototypage est dit « jetable » (*throw-away*). Il s'agit de la création de prototypes dans le but de valider les concepts avant d'implanter le système lui-même. On parle alors de *maquettage*. Le maquettage est rarement utilisé lors du développement : du fait que les maquettes ne sont pas réutilisées, elles ne sont exploitées qu'à des fins de raffinement des besoins du système. L'utilisation d'AADL que nous avons présentée dans la section IV-1.

La seconde approche est dite « par évolution ». À la différence du maquettage, les prototypes font partie intégrante du développement du système final. Chaque prototype est une pré-version plus ou moins aboutie du système final, développée pour vérifier des propriétés que l'on attend du système final. Les différents prototypes sont des raffinements les uns des autres afin de tendre vers un système de plus en plus proche de ce que l'on souhaite. Le dernier prototype est en fait le système final lui-même.

Nous basons notre méthodologie de développement sur une démarche de prototypage. Les différents prototypes sont issus de la description AADL par transformation de modèle.

IV-3.2 Phases de conception

Nous définissons un cycle de conception en deux phases principales, correspondant à deux granularités d'intergiciel différentes. La succession de ces deux phases est illustrée sur la figure IV.1. Lors de chaque phase, le modèle AADL est exploité de différentes manières, comme illustré sur la figure IV.2 :

- différentes opérations peuvent être effectuées pour vérifier la cohérence des contraintes exprimées sur l'architecture ;
- un système exécutable peut être généré afin de tester la conformité vis-à-vis des contraintes temporelles et spatiales ;
- différentes représentations formelles peuvent être extraites du modèle AADL afin de valider le comportement des entités ;

Dans une première phase nous nous basons sur une description AADL simple de l'application que nous voulons construire. Cette description correspond à l'utilisation d'un intergiciel très général supposé capable de fournir tous les services de communication nécessaires.

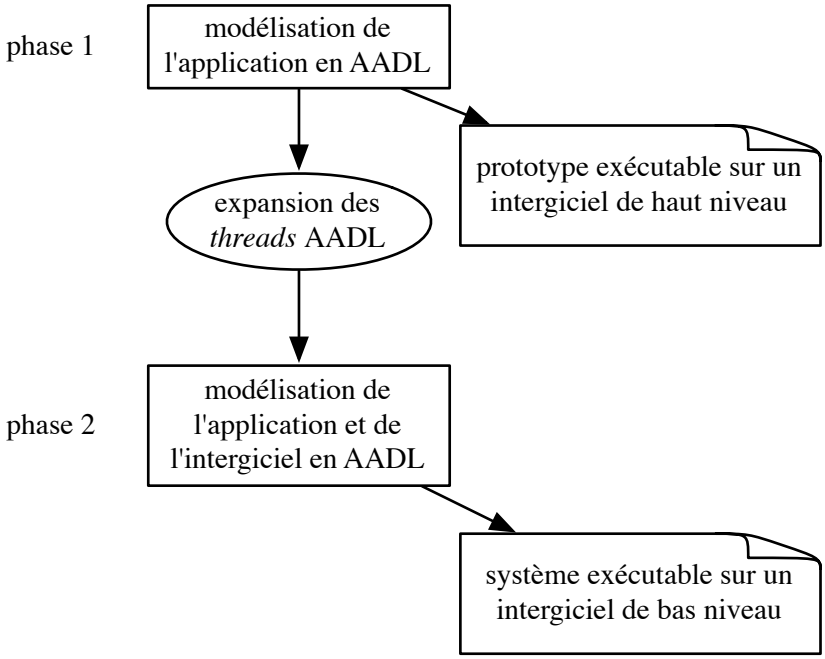


FIG. IV.1 – Cycle de conception en deux phases

Cette première phase permet de valider la structure en nœuds de l'application. La modélisation produite par les premières itérations du raffinement peuvent éventuellement être trop vague pour permettre la génération d'un système exécutable ; elle peut cependant être exploitée soit à des fins de documentation (dans la mesure où la description reste encore assez abstraite) soit pour certaines vérifications formelles.

Tous les éléments de configuration doivent être contenus dans le modèle, afin de caractériser complètement l'application. À la fin de cette phase, nous devons obtenir un prototype conforme aux spécifications fonctionnelles (notamment comportementales) de l'application. Dans la mesure où nous n'avons qu'un contrôle limité sur la structure de l'intergiciel, les implantations exécutables issues de cette modélisation peuvent faire une certaine abstraction des éléments de performance et de dimensionnement comme la taille mémoire ou les temps d'exécution. Nous pouvons néanmoins faire une première évaluation du respect des contraintes temporelles ou spatiales.

La seconde phase du développement consiste à déduire une modélisation en AADL de l'intergiciel à partir de la description de l'application. La description complète (application et intergiciel) peut alors être analysée et simulée afin d'en évaluer les dimensions exactes, s'assurer de sa fiabilité et de son adéquation avec les besoins. Dans cette dernière phase, le concepteur a une maîtrise complète de tous les aspects de l'application répartie, et peut donc en ajuster finement les différents paramètres, par exemple en sélectionnant tel ou tel composant.

L'approche que nous avons décrite à la section IV-1 ne s'intègre pas dans notre cycle de conception. Les modélisations décrites en IV-1 peuvent être exploitées comme une phase de maquettage. Compte-tenu de la capacité d'AADL à assembler des composants prédéfinis, des éléments définis dans cette phase de maquettage peuvent être directement intégrés dans notre processus et servir de point de départ à la première phase.

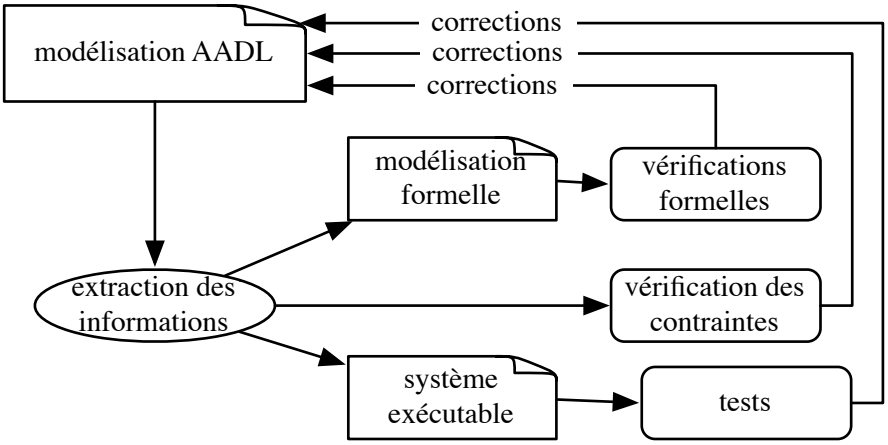


FIG. IV.2 – Processus de raffinement

IV-4 Utilisation d'AADL pour le cycle de développement

Dans le chapitre III, nous avons vu qu'AADL se focalise essentiellement sur les descriptions architecturales, et permet de modéliser les architectures avec une unique représentation centrale. Certains aspects, comme la modélisation du comportement des composants, n'entrent pas directement dans le champ d'application d'AADL. L'usage des propriétés et des annexes permet d'associer les descriptions comportementales aux composants, que ce soit en spécifiant les codes sources correspondants ou en utilisant des descriptions plus formelles.

La syntaxe d'AADL permet une grande souplesse : il n'est pas nécessaire de fournir tous les détails d'une architecture pour pouvoir en exploiter la description. Il est ainsi possible de ne spécifier que les informations qui sont pertinentes pour une exploitation donnée de la modélisation. Il est par exemple inutile de spécifier la taille des codes sources si nous souhaitons juste vérifier l'ordonnançabilité : il suffira d'indiquer en propriété les temps d'exécution, les périodes des *threads*, etc. De la même façon, il n'est pas indispensable de préciser le type des sous-composants : nous pouvons nous contenter d'indiquer leur catégorie. Il est alors facile de décrire très simplement le déploiement d'une architecture répartie sur différents nœuds.

IV-4.1 Conception d'un exécutif pour AADL

Notre objectif final est de décrire tous les éléments architecturaux d'une application avec AADL pour la générer automatiquement. La génération d'un système exécutable à partir de sa description architecturale nécessite l'utilisation d'un exécutif qui fournira les fonctionnalités nécessaires à la bonne exécution du code généré à partir des composants AADL.

IV-4.1.1 Analyse du standard 1.0

L'annexe actuelle du standard AADL [SAE, 2006c] sur les langages de programmation définit un exécutif pour AADL en s'inspirant des travaux déjà menés sur MetaH. Cet exécutif est contrôlé par les descriptions comportementales associées aux composants AADL. La description AADL est exploitée pour générer un ensemble de squelettes dans lesquelles l'utilisateur doit ajouter le code source des descriptions comportementales.

L'utilisateur a donc une grande liberté vis-à-vis du comportement de l'application, et la description AADL n'est utilisée que pour indiquer les communications *possibles* – le détails des communications, et notamment leur ordre et leur date d'émission, dépendent des descriptions comportementales.

Cette approche offre une grande souplesse dans la conception des applications, puisque la description AADL n'impose que peu de contraintes ; l'utilisateur a la possibilité de décrire des comportements complexes au niveau des communications. En contrepartie, cette liberté entraîne une dispersion des éléments de description qui limite les moyens de vérification et d'analyse de l'architecture : les communications sont spécifiées à la fois dans la description AADL et dans les descriptions comportementales.

IV-4.1.2 Séparation entre comportement et architecture

Afin de faciliter la maintenance de l'application, il semble important d'adopter une séparation la plus stricte possible entre les descriptions comportementales (qui ne sont pas du domaine d'AADL) et les structures d'assemblage AADL qui les coordonnent. Notamment, nous devons nous efforcer de garantir que les propriétés architecturales exprimées dans la description AADL seront respectées dans l'application générée. Il est par conséquent souhaitable de retranscrire autant que possible les constructions AADL dans l'exécutif.

Afin de respecter le plus possible ce principe, nous proposons donc une approche alternative pour la définition d'un exécutif AADL. De la même façon que les composants AADL contiennent chacun la description comportementale qui décrit leur implantation, l'exécutif doit contrôler les descriptions comportementales fournies, comme illustré sur la figure IV.4. Pour cela une structure d'exécution doit être générée à partir de la description AADL ; cette structure sert d'interface entre l'exécutif AADL et les descriptions comportementales.

Les descriptions comportementales sont donc contrôlées par l'exécutif AADL et n'ont aucun contrôle sur lui ; les communications entre les différents composants sont pris en charge par l'exécutif. Nous utilisons donc AADL comme un langage d'assemblage pour coordonner les différentes implantations des composants.

De cette façon, la reconfiguration de l'application s'effectue exclusivement au niveau de la description AADL, ce qui facilite la maintenance. Par ailleurs, étant donné que les implantations comportementales ne font aucune hypothèse sur leur environnement extérieur, il est relativement aisé de réutiliser les composants AADL. Nous pouvons ainsi constituer une bibliothèque de composants recomposables.

IV-4.2 Structuration de la description AADL

L'utilisation d'AADL nous permet de rassembler au sein d'une même modélisation les éléments nécessaires à la description de l'application et de l'intergiciel. Le rôle de l'intergiciel est de prendre en charge les communications et de supporter le fonctionnement de l'application. Il constitue donc la base de l'exécutif AADL.

Pour être exploitée de façon rationnelle, la modélisation AADL doit être structurée de façon à faire apparaître la séparation entre les éléments de l'application et l'intergiciel. Deux catégories principales de composants interviennent dans la description de la partie logicielle d'une architecture : les sous-programmes et les *threads* ; ils décrivent les flux d'exécutions logiciels. Les *threads* modélisent des composants actifs tandis que les sous-programmes représentent des éléments réactifs, pilotés par les *threads*. Cette différence correspond à la distinction entre l'intergiciel et

l'application elle-même : l'intergiciel pilote les éléments applicatifs.

Nous pouvons donc interpréter une modélisation AADL de façon structurée et en isoler facilement les éléments relevant purement du domaine applicatif (les sous-programmes) des composants sous-tendant leur exécution (les *threads*). La gestion des communications est modélisée par les éléments d'interface des *threads*, qui décrivent les fonctionnalités que l'intergiciel doit assurer. La figure IV.3 illustre cette séparation.

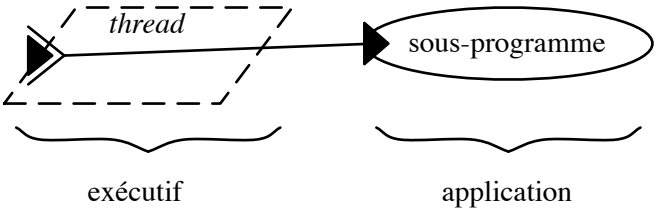


FIG. IV.3 – Structuration de la description AADL

Cette séparation dans la description AADL permet de construire un exécutif AADL destiné à piloter les descriptions comportementales, comme illustré sur la figure IV.4. Cette figure fait apparaître plusieurs parties que nous définissons ici.

Définition IV.1 (Description comportementale)
Une description comportementale implante les algorithme régissant le fonctionnement interne d'un composant AADL. Une telle description est typiquement fournie par l'utilisateur, dans un langage de programmation ; elle peut également être fournie dans un autre formalisme, selon la façon dont on souhaite l'exploiter au sein de la modélisation AADL. Une description comportementale ne fait pas d'hypothèse sur l'environnement dans lequel est situé le composant dont elle décrit le comportement.

Définition IV.2 (Intergiciel d'exécution)
Un intergiciel d'exécution est constitué d'un intergiciel de communication s'exécutant sur un système d'exploitation. Cet ensemble supporte l'exécution de l'application décrite en AADL. Un intergiciel d'exécution peut également être appelé *machine virtuelle*.

Définition IV.3 (Exécutif AADL)
L'exécutif AADL prend en charge les communications et l'exécution des éléments applicatifs. Il se compose de trois parties :
- une enveloppe pour les descriptions comportementales des éléments applicatif ;
- un intergiciel d'exécution ;
- une interface entre l'enveloppe applicative et le noyau de l'exécutif.

L'enveloppe applicative joue le rôle de conteneur (au sens où il est défini pour les intergiciels orientés composant) pour le code source fourni par l'utilisateur. Elle est déduite de la description des sous-programmes AADL. L'intergiciel d'exécution est configuré en fonction des propriétés associées aux composants AADL. Par analogie avec une approche basée sur CORBA, l'interface correspond à la couche d'adaptation qui est déduite d'une description IDL.

L'intergiciel de communication utilisé doit être suffisamment générique pour prendre en charge les différents modèles de répartitions exprimés dans la modélisation AADL ; il doit également être configurable, de façon à être adapté en fonction des caractéristiques exprimées dans la description

AADL (politique d'ordonnancement, nombre de *threads* à prendre en charge, etc.). Un intergiciel adaptable tel que PolyORB [Vergnaud et al., 2004] correspond typiquement à ces besoins et permet obtenir un prototype de l'application complète offrant de bonne performances vis-à-vis des besoins exprimés pour l'application.

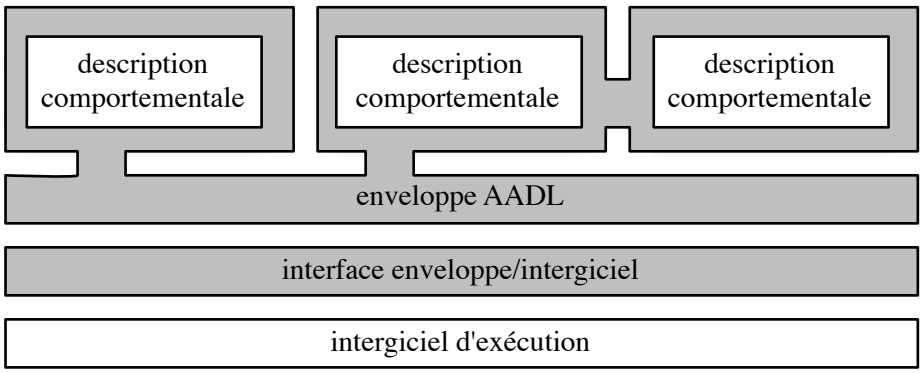


FIG. IV.4 – Place de l'exécutif AADL dans l'application

IV-4.3 Matérialisation de la modélisation de l'intergiciel

La distinction que nous pouvons effectuer entre l'application et l'intergiciel d'exécution au niveau de la description architecturale permet de faciliter la description de l'intergiciel lors de la seconde phase du cycle de modélisation. Cette phase de modélisation fine de l'intergiciel correspond à l'expansion des *threads* AADL pour faire apparaître les structures de l'intergiciel sous forme de sous-programmes AADL.

Dans cette seconde phase la plus grande part de l'intergiciel de communication est elle-même décrit en AADL. Ceci implique de décrire en AADL les composants assurant la gestion des communications. La description architecturale est plus précise, et offre donc un meilleur support pour la vérification complète des nœuds de l'application.

IV-5 Spécifications de l'exécutif AADL

Dans la section précédente nous avons proposé une façon de structurer une description AADL de façon à séparer clairement les descriptions comportementales, l'enveloppe applicative et la configuration de l'intergiciel. Nous avons établi que la description AADL doit contenir l'ensemble des caractéristiques de l'application.

L'objectif de nos travaux étant de produire automatiquement un système réparti, nous devons nous assurer que l'exécutif AADL peut prendre en charge les constructions architecturales décrites. Il est nécessaire de préciser quelles sont les constructions AADL valides et comment les interpréter vis-à-vis de spécifications d'exécutif. Pour cela, nous devons définir des patrons architecturaux permettant de guider la description des systèmes.

Ces patrons dépendent des spécifications de l'exécutif que nous souhaitons utiliser pour prendre en charge l'exécution du code produit. Ils ne correspondent pas à la notion classique de patrons de conception (*design patterns*, [Gamma et al., 2001]). Ils présentent des consignes générales pour la description des architectures, mais ne décrivent pas une solution technique à un problème précis.

L'interprétation des composants AADL proposée dans le standard AADL [SAE, 2006c] se focalise sur des systèmes centralisés ; elle ne prend pas en compte la présence d'un intergiciel pour assurer les communications. De la même façon que nous concevons une architecture d'exécutif différente, notre interprétation des composants diverge également quelque peu des orientations proposées actuellement par le standard.

Dans cette section nous décrivons les spécifications de l'exécutif AADL que nous définissons. Ces spécifications correspondent à un raffinement de la sémantique décrite dans le standard. Nous présentons également les patrons architecturaux AADL correspondant à ces spécifications.

IV-5.1 Relation avec les descriptions comportementales

Nous avons établi en section IV-4.2 que les sous-programmes AADL constituent la description de l'application tandis que les *threads* AADL représentent l'intergiciel d'exécution. Seuls les sous-programmes sont donc censés encapsuler des descriptions comportementales.

L'exploitation des *threads* AADL est totalement prise en charge par le générateur d'exécutif ; seules les descriptions comportementales associées aux sous-programmes sont pris en compte.

IV-5.2 Interprétation des séquences d'appel

Un sous-programme AADL peut contenir une séquence d'appels qui décrit son flux d'exécution. Dans le cas de plusieurs séquences, le standard AADL stipule que celles-ci doivent être associées à des modes de configuration distincts. Ainsi un sous-programme ne peut exécuter qu'une seule séquence, correspondant à son mode de configuration. Cette approche limite la description d'éventuels branchement conditionnels dans les composants ; de tels branchements correspondraient en effet à un ensemble de modes de configuration.

Une telle logique de modélisation empêche de représenter de façon efficace des séquences d'exécutions alternatives ; il faudrait alors les intégrer dans une modélisation complexe basée sur des changements de mode systématiques. L'interprétation des séquences d'appel suggérée par le standard suppose donc que les applications modélisées effectuent des actions systématiques et inconditionnelles.

Nos travaux s'intéressent à la modélisation d'applications éventuellement complexes, qui sont donc susceptibles de définir plusieurs flux d'exécution différents. Dans ces conditions, recourir aux modes AADL conduirait à une explosion combinatoire des modes de configuration.

Nous adoptons donc une interprétation alternative.

Spécification IV.1 (Interprétation des séquences d'appels multiples)
L'implantation d'un sous-programme AADL peut contenir plusieurs séquences d'appels. Ces séquences modélisent différents scénarios d'exécution possibles ; elles doivent être coordonnées par une description comportementale associée au sous-programme.

Plusieurs séquences d'appel peuvent coexister au sein d'un même mode de configuration ; elles modélisent alors différents flux d'exécution possibles. La description comportementale permettant de contrôler ces flux doit être fournie par l'utilisateur, comme nous le décrirons dans les chapitres V et VII.

Nous étendons ainsi la sémantique d'AADL sans la remettre en cause : il est toujours possible de décrire une séquence d'appel par mode de configuration.

IV-5.3 Description des modèles de répartition

Les modèles de distribution sont décrits par les éléments d'interface fournis par les *threads* AADL. La modélisation fait apparaître une distinction entre les *threads* qui reçoivent et ceux qui envoient les données. La politique de déclenchement d'un *thread* récepteur est typiquement apériodique ; ils sont alors déclenchés à la réception de données. La politique de déclenchement des *threads* émetteurs est en général périodique ou sporadique.

IV-5.3.1 Passage de message

Les architectures basées sur une communication par passage de messages sont traduites par la présence de ports associés aux *threads* AADL. Nous pouvons considérer les trois catégories de ports (événement, donnée, événement/donnée) ; chacune d'elles correspond à un certain type de message.

Typiquement, un message apporte deux informations : la donnée effectivement transportée et le fait que le *thread* doit être activé. Ce genre d'information correspond aux ports *d'événement/donnée*. Ces ports permettent de modéliser la gestion de messages synchrones qui peuvent déclencher le *thread* récepteur. La plupart des applications basées sur le passage de messages sont donc modélisées à l'aide de ces ports.

Les ports de *donnée* permettent de modéliser les communications asynchrones. Dans cette situation, le message ne déclenche le *thread* récepteur ; la politique de déclenchement associée est *a priori* périodique ou sporadique. Cependant, le *thread* récepteur peut être déclaré apériodique si l'application utilise par ailleurs des ports d'événement/donnée. Dans ce cas, le *thread* sera déclenché à l'arrivée du message synchrone et lira les données sur tous les ports.

Les ports d'*événement* permettent de modéliser la transmission d'un signal simple entre deux nœuds, sans aucune donnée. Ce genre de port permet par exemple de modéliser la transmission d'un signal d'interruption (par exemple. un top d'horloge). Les *threads* offrant des ports d'événement en entrée sont typiquement apériodiques, étant donné que le seul usage d'un port d'événement est le déclenchement de *threads*.

Spécification IV.2 (Communications par passage de message)
Les ports d'événement/donnée permettent de modéliser une architecture basée sur le passage de messages. Les ports de donnée permettent de modéliser des transmissions de données asynchrones.

Le listing IV.10 donne un exemple d'architecture AADL modélisant un client pouvant envoyer des messages à un serveur.

```
1  thread a_client
2  features
3    out_msg : out event data port;
4  end a_client;
5
6  thread implementation a_client.impl
7  properties
8    dispatch_protocol => periodic;
9    period => 10 s;
10 end a_client.impl;
11
12 thread a_server
13 features
```

```
14   in_msg : in event data port;
15 properties
16   dispatch_protocol => aperiodic;
17 end a_server;
18
19 thread implementation a_server.impl
20 end a_server.impl;
21
22 system global end global;
23
24 system implementation global.impl
25 subcomponents
26   client1 : thread a_client.impl;
27   server1 : thread a_server.impl;
28 connections
29   event data port client1.out_msg -> server1.in_msg;
30 end global.impl;
```

Listing IV.10 – Modélisation d'une architecture basée sur le passage de message

Le protocole de déclenchement du serveur est ici apériodique, tandis que le client est périodique, avec une période de 10 secondes. Un client peut être constitué d'un *thread* en tâche de fond (*background*).

IV-5.3.2 Appel de procédure distante

Spécification IV.3 (Appels de procédure distante)
Les appels de procédures distantes sont matérialisés par des sous-programmes fournis comme éléments d'interface d'un *thread*. Un appel à de tels sous-programmes à partir d'un autre *thread* modélise un appel distant.

Un tel mécanisme de communication suppose que le *thread* serveur se déclenche à l'appel des sous-programmes qu'il fournit. Ceci implique que le *thread* a une politique de déclenchement apériodique.

Du côté du client, le sous-programme ou le *thread* appelle le sous-programme distant par l'intermédiaire d'un accès à sous-programme. Le listing IV.11 illustre une architecture basée sur un mécanisme d'appel distant.

```
1 subprogram a_rpc
2 features
3   a : in parameter;
4   b : out parameter;
5 end a_rpc;
6
7 thread a_server
8 features
9   rpc1 : server subprogram a_rpc;
10 properties
11   dispatch_protocol => aperiodic;
12 end a_server;
```

Listing IV.11 – Modélisation d'une architecture basée sur l'appel de sous-programme distant

Un appel de sous-programme distant se manifeste par une connexion d'accès à sous-programme entre deux *threads*. Si les deux *threads* appartiennent au même processus, le générateur de code est susceptible de procéder à des optimisations, de façon à ne pas faire appel aux couches du réseau. Du point de vue de la modélisation, ce cas ne constitue cependant pas une situation particulière.

Notons qu'un sous-programme instancié au sein d'un processus et connecté à un accès requis à sous-programme d'un *thread* de ce processus ne constitue pas un appel distant ; le *thread* appelle simplement le sous-programme local.

IV-5.3.3 Objets distants

AADL ne permet pas de modéliser des objets à proprement parler, dans la mesure où des notions telles que la surcharge et l'héritage n'existent pas. Notamment, l'extension de composant permet seulement de réutiliser des déclarations de façon statique ; l'extension d'un composant ne peut pas instancié en lieu et place du composant étendu. Il n'est pas non plus possible de créer dynamiquement des composants, comme on peut le faire avec des systèmes comme CORBA.

Il est néanmoins possible de décrire des objets « statiques » comme étant un ensemble de sous-programmes fournis par un composant de donnée. Le composant de donnée peut lui-même être accessible à partir de sous-programmes exécutés par d'autres *threads* ; il fait partie des éléments d'interface fournis par le *thread* serveur.

Spécification IV.4 (Objets répartis)
Les objets répartis sont modélisés par des composants de donnée dont l'accès est fourni en interface d'un *thread*. Ces composants de donnée fournissent eux-même un accès à des sous-programmes, qui constituent les méthodes de l'objet.

Les objets distants en AADL correspondent en fait à des sous-programmes distants associés à un composant de donnée. Le listing IV.12 illustre un exemple de manipulation d'objet réparti en AADL.

```
1  subprogram a_method
2  features
3    a : in parameter;
4    b : out parameter;
5  end a_method;
6
7  data a_class
8  features
9    method1 : subprogram a_method;
10 end a_class;
11
12 thread a_server
13 features
14   object_stub : provides data access a_class;
15 properties
16   dispatch_protocol => aperiodic;
17 end a_server;
18
19 thread implementation a_server.impl
20 subcomponents
21   object : data a_class;
22 connections
```

```
23   data access object -> object_stub;
24 end a_server.impl;
```

Listing IV.12 – Modélisation d'une architecture basée sur des objets distants

IV-5.4 Cycle de fonctionnement des *threads*

Les représentations en AADL des modèles de répartition que nous avons présentées entraînent certaines conséquences sur la sémantique de fonctionnement des *threads* AADL qui les prennent en charge.

Le standard AADL spécifie que les *threads* récepteurs apériodiques ne peuvent être déclenchés que par les ports de donnée/événement. Au moment de son déclenchement, un *thread* lit tous ses ports de donnée. Par conséquent, un *thread* ne doit avoir *a priori* qu'un seul port d'événement/-donnée déclencheur, qui provoque l'exécution de son unique activité.

Spécification IV.5 (Déclenchement des *threads* apériodiques)
Un *thread* apériodique peut avoir au plus un port d'événement/donnée déclenchant son exécution.
Si un seul port d'événement/donnée est spécifié dans son interface, le *thread* considéré se déclenche à la réception d'un message sur ce port. Si plusieurs ports sont spécifiés, le port déclencheur doit être indiqué à l'aide d'une propriété AADL.

Le standard AADL actuel ne fournit pas de propriété standard permettant de spécifier le caractère déclencheur d'un port. Nous définissons une nouvelle propriété, nommée `Dispatch_Port` (listing IV.13), permettant de remplir ce rôle.

```
1 Dispatch_Port : aadlboolean applies to (event data port);
```

Listing IV.13 – Définition de la propriété AADL pour le déclenchement des *threads*

De la même façon, un *thread* périodique lit les données présentes sur tous ses ports à chaque déclenchement.

Spécification IV.6 (Cycle de traitement des données)
Une fois déclenché, un *thread* exécute la séquence d'appels de sous-programmes qui est spécifiée dans son implantation. À la fin de l'exécution complète de la séquence, les données issues des sous-programmes appelés sont récupérées et envoyées vers les autres nœuds, selon les ports et connexions définis dans la description architecturale.

Dans ces conditions, un *thread* en tâche de fond n'émettra ses données qu'une fois, à la fin de son exécution.

Il est de cette façon possible d'évaluer précisément la date d'émission des données d'un *thread* en fonction des temps d'exécution associés aux sous-programmes appelés.

Les sous-programmes fournis en interface correspondant à un appel distant correspondent à un autre mécanisme : ils constituent en effet des activités indépendantes du passage de messages et traduisent des flux d'exécution clairement délimitées. Du point de vue du traitement des données entrantes, les sous-programmes fournis en interface peuvent être assimilés à des ports, dans la mesure où ils constituent un groupe autonome de données. Ils peuvent donc être comparés à des ports de données/événement puisqu'ils provoquent le déclenchement du *thread*.

Les sous-programmes d'interface n'interfèrent pas avec les ports d'événement/donnée ou de donnée ; ils constituent un mécanisme parallèle. Un *thread* périodique peut donc avoir à la fois

un port d'événement/donnée et des sous-programmes d'interface. Par rapport à la sémantique que nous avons définie en IV-5.2, les sous-programmes fournis en interface d'un *thread* peuvent être interprétés comme des séquences d'appel constituée d'un seul appel de sous-programme.

IV-6 Conclusion

Nous avons présenté deux façons d'exploiter le langage AADL. Tout d'abord une utilisation restreinte, limitée à la description du déploiement des applications. Nous avons ensuite défini un processus de conception dans lequel AADL est utilisé comme langage central pour rassembler tous les aspects de la description de l'application.

Ce processus de conception repose sur la spécification d'un exécutif assurant les fonctions de communication. La construction et la configuration de cet exécutif nécessite la prise en compte des paramètres fonctionnels, tels que les communications à assurer, ainsi que des paramètres non fonctionnels, comme les dimensions de l'application – tant temporelles que spatiales. Ces paramètres doivent être exprimés dans la modélisation AADL.

Ces contraintes doivent être respectée par l'ensemble constitué de l'application et de son intergiciel ; il doivent être développés conjointement. Nous avons présenté un processus de conception permettant d'intégrer les différents aspects de l'application au sein d'une unique modélisation AADL. Notre processus se base sur une approche par prototypage afin de produire l'application finale par raffinement successifs. Par sa capacité à transporter toutes les informations nécessaires à la description des architectures, AADL nous sert de langage fédérateur : il est ainsi possible d'utiliser plusieurs outils pour tester l'ordonnançabilité, s'assurer du respect des contraintes de place mémoire, simuler l'architecture à partir des descriptions comportementales puis finalement générer un exécutable. À chaque raffinement, il est possible d'exploiter la description AADL afin de vérifier les propriétés théoriques de l'application, que ce soit par analyse des dimensions ou modélisation formelle.

L'approche que nous suivons dans la modélisation consiste à considérer que l'exécutif est matérialisé par la description AADL. L'exécutif coordonne les descriptions comportementales encapsulées par les différents composants AADL. De cette façon, la reconfiguration et le paramétrage de l'application se fait exclusivement au niveau de la modélisation AADL. Sur différents aspects, notre démarche constitue une alternative à l'approche actuellement proposée dans le standard AADL.

Notre processus se partage en deux phases, correspondant au degré de précision dans la description de l'intergiciel supportant l'exécution de l'application. La première phase consiste à assimiler l'exécutif AADL à l'intergiciel, alors représenté par les *threads* AADL. La seconde phase consiste à préciser la structure de l'intergiciel en AADL afin de l'intégrer dans les processus d'analyse et de vérification. Cette étape se matérialise par une expansion des *threads* AADL pour faire apparaître les composants de l'intergiciel.

Pour pouvoir donner lieu à la génération d'un système exécutable, il est nécessaire de guider l'utilisateur pour qu'il décrive une architecture cohérente vis-à-vis des spécifications de l'exécutif que nous définissons. L'application de patrons architecturaux pour décrire l'application permet d'avoir une description normalisée, qui sera reconnue par le générateur d'application que nous décrivons dans les chapitres V, VI et VII. Ces chapitres exposent la mise en place des opérations permettant la réalisation de notre processus de conception.

CHAPITRE V

Génération du code pour l'enveloppe applicative

UNE MODÉLISATION en AADL permet de décrire les éléments architecturaux d'une application, éventuellement répartie. Dans ce chapitre nous étudions la démarche de transformation d'une description AADL en application répartie exécutable.

Les problématiques que nous abordons ici sont l'interprétation des éléments AADL en terme de construction logicielle, ainsi que l'intégration de descriptions comportementales au sein de la description. Nous avons montré en section IV-4.1.1 que nous adoptions une démarche de construction alternative par rapport à celle suggérée par le standard AADL actuel. Cette différence d'approche a une incidence sur la définition des règles de production de code source. Nous étudions les différents cas à considérer dans le cadre de la description des composants logiciels en AADL.

V-1 Identification des éléments applicatifs d'AADL

Dans le chapitre III, nous avons établi qu'AADL définit un certain nombre de composants pour formaliser la partie logiciel d'un système : il s'agit des sous-programmes, des fils d'exécution (*threads*) et des composants de donnée. À ces trois catégories de composants s'ajoutent les processus, qui représentent les exécutables en eux-mêmes.

Les *threads* et les sous-programmes définissent les flux d'exécution et représentent donc des instructions en code source ; les composants de données représentent les structures de données, c'est-à-dire les variables et les paramètres des procédures. Les processus définissent des espaces de mémoire dans lesquels les *threads* s'exécutent.

Au chapitre IV, nous avons expliqué que les éléments d'une architecture AADL modélisaient la couche d'adaptation entre les composant de l'application et un intergiciel d'exécution. Cette couche d'adaptation est constituée de deux parties :

- une enveloppe permettant de lier les composants applicatifs fournis par l'utilisateur à l'intergiciel ;
- une couche d'interface assurant la coordination entre l'enveloppe et l'intergiciel lui-même.

Dans ce chapitre, nous nous focalisons sur la génération de l'enveloppe à partir de la description AADL. Cette enveloppe correspond aux composants purement applicatifs de la modélisation, c'est-à-dire les sous-programmes et les composants de donnée.

Nous nous intéressons à la traduction des éléments applicatifs – les sous-programmes et les composants de donnée – en langage de programmation, comme illustré sur la figure V.1. Les éléments architecturaux doivent être retranscrits en appliquant des règles de correspondance entre AADL et le langage cible. Cette partie de la génération dépend donc uniquement du langage de programmation à utiliser. Étant donné qu'elle ne dépend pas de l'implantation de l'exécutif,

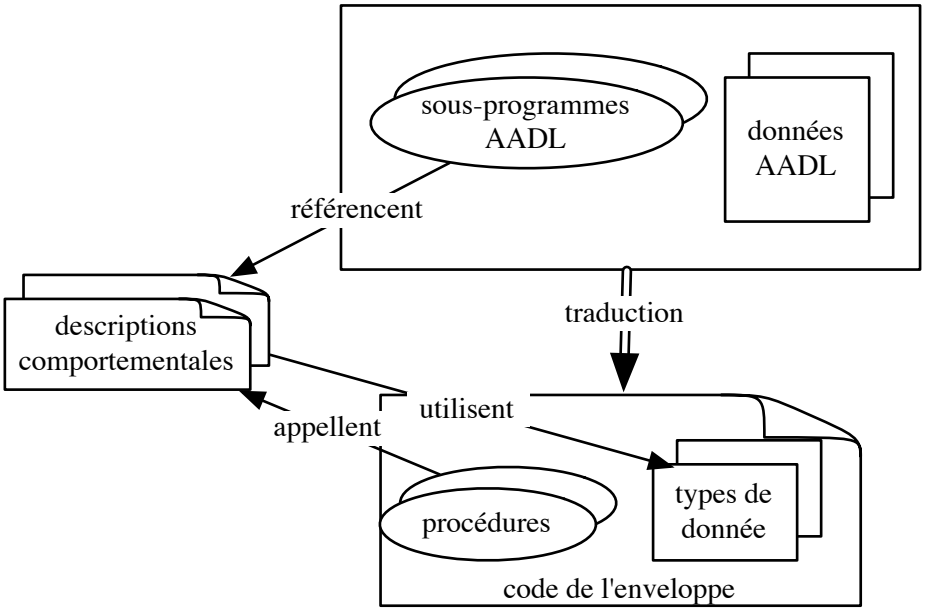


FIG. V.1 – Génération de l'enveloppe applicative

elle demeure valable pour les deux étapes du processus de conception que nous avons décrit en section IV-3.

V-2 Traduction des composants AADL applicatifs en langage de programmation

Nous nous focalisons sur la génération de code pour les langages impératifs tels que C, Ada ou Java. Cette section expose d'abord les règles de traduction depuis AADL vers un langage de programmation impérative quelconque, puis indique les traductions précises pour des langages en particulier. Nous choisissons Ada et C car ces langages sont très utilisés dans la communauté des systèmes temps-réel embarqués. Par ailleurs, le fait que la norme Ada définit l'interopérabilité entre les deux langages les rend intéressant pour illustrer l'utilisation d'AADL pour construire des applications dont les composants sont implantés dans des langages différents. Nous détaillons également une traduction vers le langage Java. De nombreux travaux sont en effet menés pour utiliser Java dans des systèmes embarqués.

V-2.1 Organisation du processus de traduction

La génération d'une application à partir d'une description AADL consiste à traduire les constructions AADL que nous avons identifiées à la section précédente en constructions syntaxiques compilables. Nous nous intéressons à la traduction vers des langages de programmation impératifs – par exemple C.

Quel que soit le langage de programmation impérative ciblé, notre objectif est de générer un ensemble de fichiers nécessaires à la compilation de programmes exécutables. Nous devons générer un programme par nœud de l'application répartie. Pour cela, nous considérons chaque

processus AADL séparément.

Définition V.1 (nœud applicatif)
Un nœud applicatif est représenté par une instance de processus AADL.

Pour chaque processus AADL, le traducteur produit donc les constructions syntaxiques correspondant aux différents sous-programmes et composants de donnée.

V-2.2 Traduction des composants de données

Les déclarations de composant de donnée AADL correspondent à des déclarations de type dans les langages de programmation. Ces déclarations se traduisent donc par une définition de type. Une instance de composant de donnée correspond alors naturellement à la définition d'une variable.

Règle V.1 (Interprétation des composants de donnée)
La déclaration d'un composant de donnée AADL correspond à la définition d'un type de donnée dans un langage de programmation impérative. Une instance de composant de donnée (comme sous-composant d'un *thread* ou d'un sous-programme correspond à la définition d'une variable.

La déclaration d'un composant AADL traduit l'existence d'un type de donnée ; la sémantique du type de donnée n'est cependant pas couverte par la seule déclaration de composant. Ainsi, le code AADL V.1 ne donne aucune indication sur la nature de la donnée déclarée.

```
1 data une_donnee
2 end une_donnee;
```

Listing V.1 – Déclaration d'une donnée AADL

Afin de produire un type de donnée exploitable, il est nécessaire de spécifier la sémantique de la donnée. Le langage AADL ne transportant pas en lui-même la description sémantique des données, il est nécessaire de préciser cette sémantique. Au cours de nos travaux nous avons défini un ensemble de propriétés AADL permettant de spécifier le type de donnée modélisé par les composants de donnée AADL. Le comité de standardisation AADL a repris cette approche et défini un ensemble plus complet de propriétés, appelé `Language_Support`. Les règles que nous présentons ici peuvent être vues comme des compléments aux directives architecturales que nous avons présentées en section IV-5, page 65.

```
40 Data_Format: enumeration (Integer, int, Float, flt, Boolean,
41 String, str, Character, char, Wide_String, Natural, Positive,
42 Enum, long, long_long)
43 applies to (data);
```

Listing V.2 – Définition de la sémantique des données

La principale propriété de cet ensemble est `Data_Format`, dont la définition est reproduite au listing V.2 ; elle permet de spécifier la sémantique d'un composant de donnée. Les différentes valeurs possibles correspondent aux types classiques définis dans les langages de programmation (entiers, flottants, caractères, chaînes de caractères, type énuméré). Certains types sont mentionnés plusieurs fois sous différentes dénominations, afin d'être plus proche des appellations utilisées dans les langages cibles. Par exemple, `Integer` et `int` désignent une sémantique d'entier en reprenant les mots-clés utilisés en Ada et en C.

Règle V.2 (Spécification de la sémantique des données)
La sémantique d'un composant de donnée est indiqué par les propriétés AADL `Language_Support::Data_Format`, `Language_Support::Data_Type`, `Language_Support::Data_Structure`, `Language_Support::Dimension` et `Language_Support::Constant_String`.

La combinaison des ces différentes propriétés permet de spécifier la sémantique des composants de donnée.

À partir de la sémantique indiquée dans les propriétés, il est possible de traduire une déclaration de composant de donnée AADL en un type simple (entier, à virgule flottante, booléen), comme illustré sur le listing V.3.

```
4 data implementation une_donnee.entier
5 properties
6   Language_Support::Data_Format => Integer;
7 end une_donnee.entier;
```

Listing V.3 – Déclaration d'un entier en AADL

La déclaration des énumérations implique la spécification des valeurs énumérées possibles. Nous utilisons pour cela la propriété `Constant_String`, qui est une liste de chaînes de caractères. Le listing V.4 donne un exemple d'une déclaration de donnée correspondant à un type énuméré.

```
9  data implementation une_donnee.enumeration
10 properties
11   Language_Support::Data_Format => Enum;
12   Language_Support::Constant_String => ("bleu", "jaune");
13 end une_donnee.enumeration;
```

Listing V.4 – Déclaration d'un type énuméré en AADL

Nous assimilons les tableaux à des types simples dans la mesure où leur définition repose uniquement sur des propriétés AADL, comme illustré sur le listing V.5. Nous utilisons trois propriétés pour spécifier la structure en tableau, le type de donnée (ici des entiers) et la dimension du tableau (ici un tableau à une dimension, de taille 6). Nous pouvons remarquer que l'usage de la propriété `Language_Support::Data_Format` ne permet pas de spécifier l'usage d'un composant de donnée AADL ; ainsi cette syntaxe restreint les possibilités de définition à des tableaux contenant des types scalaires simples, spécifiés par la propriétés `Language_Support::Data_Format` . Pour pallier cette limitation, nous proposons d'introduire une propriété AADL alternative, appelée `Language_Support::Data_Type`. Cette nouvelle propriété ainsi définie permet de spécifier l'usage d'un composant de donnée AADL défini par ailleurs.

```
15 data implementation une_donnee.tableau_entier
16 properties
17   Language_Support::Data_Format => Integer;
18   Language_Support::Data_Structure => Array;
19   Language_Support::Dimension => (6);
20 end une_donnee.tableau_entier;
21
22 data implementation une_donnee.tableau_entier2
23 properties
24   Language_Support::Data_Type => classifier (une_donnee.entier);
25   Language_Support::Data_Structure => Array;
26   Language_Support::Dimension => (6);
```

```
27 end une_donnee.tableau_entier2;
```

Listing V.5 – Déclaration de tableaux en AADL

Les chaînes de caractères constituent un mécanisme à mi-chemin entre les types scalaires et les tableaux. Pour des raisons pratiques, nous ne considérons que les chaînes de taille fixée ; nous évitons ainsi de traiter le redimensionnement des chaînes. Ceci permet d'une part de simplifier la manipulation des chaînes, et également de pouvoir déterminer de façon statique la taille des communications. Nous utilisons donc une propriété supplémentaire pour spécifier la taille de la chaîne, comme illustré sur le listing V.6. Les chaînes de caractères sont vues implicitement comme des tableaux de caractères unidimensionnels.

```
29 data implementation une_donnee.chaine
30 properties
31   Language_Support::Data_Format => String;
32   Language_Support::Dimension => (7);
33 end une_donnee.chaine;
```

Listing V.6 – Déclaration d'une chaîne de caractères en AADL

Les structures de données constituées de plusieurs champs (aussi appelées *enregistrements* dans certains langages) constituent des données complexes. Leur organisation peut être représenté par une construction AADL à l'aide de sous-composants, comme illustré sur le listing V.7. Une telle déclaration traduit exactement le fait que le type de donnée `une_donnee.structure` est constitué de deux champs `i` et `c`, correspondant tous deux à un type défini par ailleurs. Aucune propriété AADL n'a besoin d'être spécifiée, car la syntaxe AADL suffit à fournir toutes les informations nécessaires.

```
35 data implementation une_donnee.structure
36 subcomponents
37   i : data une_donnee.entier;
38   c : data une_donnee.chaine;
39 end une_donnee.structure;
```

Listing V.7 – Déclaration d'une structure de données complexe en AADL

Les listes constituent un cas limite pour lequel AADL ne permet pas une description efficace. Les listes ne peuvent pas être représentées comme des structures de données. En effet, une liste est une succession de données de même type chaînées les unes aux autres, le dernier éléments de la liste n'ayant pas de successeur. Une telle construction ne peut pas être représentée comme une structure de donnée telle que décrite dans le listing V.7, dans la mesure où AADL ne possède pas la notion de pointeur ou de référence : un composant s'instanciant lui-même comme sous-composant produit systématiquement une architecture infiniment récursive, qui ne peut pas être exploitée. Une solution consiste à utiliser des propriétés AADL permet de décrire un type de donnée opaque, qui masque la structure exacte de la donnée, comme représenté sur le listing V.8. Nous masquons alors la structure interne du type de donnée, en suivant une approche similaire à celle appliquée pour les conteneurs de la bibliothèque standard du C++ [Stroustrup, 2003]. Pour cela, nous introduisons une nouvelle valeur pour la propriété `Data_Structure` : `chained_list`. De la même façon que pour les tableaux, nous pouvons utiliser `Language_Support::Data_Format` ou `Language_Support::Data_Type`.

```
41 data implementation une_donnee.liste
42 properties
43   Language_Support::Data_Structure => Chained_List;
```

```
44   Language_Support::Data_Container => classifier (une_donnee.entier
         );
45 end une_donnee.liste;
```

Listing V.8 – Déclaration d'une liste en AADL

Contrairement aux autres types de données, qui traduisent des constructions classiques dans les langages de programmation, le type liste tel que nous le décrivons en AADL suppose la génération d'une structure assez évoluée, fournissant des mécanismes permettant d'ajouter, d'extraire, etc. des éléments d'une liste.

Les types de donnée AADL sont traduit dans le langage de programmation cible. Afin de faciliter la correspondance entre AADL et le langage cible, le nom AADL est repris pour la définition du type dans le langage cible.

V-2.3 Traduction des sous-programmes

Les sous-programmes AADL constituent les unités structurelles abritant les descriptions comportementales. Ils correspondent naturellement aux procédures ou aux fonctions dans les langages de programmation. Un sous-programme AADL possède des paramètres en entrée, sortie, ou entrée/sortie, mais pas de type valeur de retour.

Règle V.3 (sémantique des sous-programmes)
Un sous-programme AADL a la même sémantique qu'une procédure. Il se traduit par une procédure ou une fonction, selon les constructions autorisées par le langage cible.

La description d'un sous-programme en AADL correspond à la définition d'une enveloppe qui contient une description comportementale et la description des connexions avec d'autres sous-programmes.

Au sein d'un sous-programme, AADL permet de décrire des séquences d'appels à d'autres sous-programmes, ainsi que les connexions entre les différents paramètres de ces sous-programmes. Ces constructions reflètent l'organisation architecturale des branchements possibles entre les différents sous-programmes d'une application. Les structures algorithmiques classiques (boucles, conditions, etc.) sont du ressort de la description comportementale.

En tant que langage de description d'architecture, AADL traite donc des flots d'appels de sous-programmes, mais ne décrit pas la façon dont ces flux sont organisés ou séquencés les uns par rapport aux autres. Il est donc nécessaire de définir des règles de correspondance pour établir une relation entre les descriptions comportementales et les séquences d'appel. La traduction d'un sous-programme AADL en langage de programmation nécessite donc la prise en compte de deux paramètres principaux :

- présence de séquences d'appel dans l'implantation du sous-programme ;
- référence à un fichier contenant la description comportementale du sous-programme.

Nous avons étudié les différentes combinaisons possibles de ces deux paramètres afin de définir des règles générales concernant d'une part le code à générer à partir des descriptions AADL et d'autre part la signature des descriptions comportementales fournies par l'utilisateur. Nous distinguons quatre cas de sous-programmes AADL différents, selon leur implantation :

- pas d'implantation ;
- implantation opaque ;
- séquence d'appel pure ;
- implantation hybride.

V-2.3.1 Interfaces des sous-programmes

Un sous-programme AADL n'a pas de type de retour : les paramètres sont en entrée, en sortie ou en entrée/sortie. Cette signature doit être reproduite par le code source généré, en reprenant la même bijection que celle utilisée pour générer les noms de types de donnée.

Les paramètres d'un sous-programme AADL s'interprètent naturellement comme des paramètres de procédure. Les accès requis à des composants de données correspondent à des références sur variable. Les accès requis à des sous-programmes correspondent à des références sur les procédures correspondantes.

V-2.3.2 Sous-programme sans implantation

Un sous-programme sans implantation ne fournit aucune information quand à sa structure interne. Il s'agit d'un type sous-programme ne possédant aucune implantation, ou possédant une implantation vide.

Règle V.4 (Traduction des sous-programmes sans implantation)
La génération de code consiste à produire une procédure ou une fonction vide.

À l'exécution, ce code ne fera rien et engendrera des valeurs indéterminées pour les paramètres de sortie, ce qui correspond effectivement à la description AADL.

V-2.3.3 Implantations opaques

Dans cette situation, l'implantation du sous-programme est décrite en langage de programmation dans un fichier externe. Elle est associée au sous-programme AADL à l'aide de propriétés standards : `Source_Name`, `Source_Text` et `Source_Language` tels qu'elles sont définies dans le standard AADL :

- `Source_Language` indique le langage utilisé dans la description comportementale ;
- `Source_Text` indique le nom du paquetage ou du fichier dans lequel se situe la description comportementale ;
- `Source_Name` indique le nom de la procédure contenant la description comportementale.

Le listing V.9 donne un tel exemple de sous-programme AADL.

```
1  data entier
2  properties
3    Language_Support::Data_Format => integer;
4  end entier;
5
6  subprogram sp
7  features
8    e : in parameter entier;
9    s : out parameter entier;
10 end sp;
11
12 subprogram implementation sp.impl
13 properties
14   source_language => Ada95;
15   source_text => ''''Paquetage''
16   source_name => ''Sp_Implantation'';
```

```
17 end sp.impl;
```

Listing V.9 – Exemple de sous-programme AADL avec une implantation opaque

Règle V.5 (Traduction des sous-programmes ayant une implantation opaque)
Un sous-programme AADL n'ayant pas de séquences d'appel et dont les propriétés `Source_Language`, `Source_Text` et `Source_Name` sont définies est traduit par une enveloppe appelant le code spécifié par les propriétés.

L'enveloppe générée doit faire le lien avec le code source que l'utilisateur doit fournir, en faisant correspondre les paramètres. L'utilisateur doit donc écrire le code source d'implantation sous forme d'une procédure ou d'une fonction dont la signature correspond à celle décrite en AADL. Les types de données utilisés correspondent aux types générés à partir des déclarations de composants de donnée AADL. Dans notre approche, la cohérence entre la déclaration AADL et l'implantation comportementale est vérifiée par le compilateur.

Nous matérialisons ainsi une séparation nette entre la partie architecturale et la description algorithmique, comme nous l'avions illustré sur la figure V.1. Le fait de fournir le code complet d'une procédure – au lieu d'insérer du code au sein d'un squelette généré – permet d'avoir une séparation nette entre l'enveloppe AADL et l'implantation comportementale. Cela facilite la maintenance du code source d'implantation : l'implantation fournie par l'utilisateur n'est pas affectée par les changements dans la description architecturale, pourvu que l'interface du sous-programme AADL ne soit pas modifiée. Cela autorise également un découplage entre le langage cible produit par le traducteur AADL et le langage utilisé pour l'implantation en code source : l'enveloppe générée à partir de l'AADL dépend à la fois du langage cible et du langage d'implantation, et contient toutes les instructions nécessaires pour faire la liaison entre les deux code sources.

V-2.3.4 Séquences d'appel pures

AADL permet de modéliser un sous-programme comme étant une séquence d'appel à d'autres sous-programmes AADL, comme illustré sur le listing V.10. En l'absence de description comportementale, nous considérons que le comportement du sous-programme consiste à exécuter la séquence.

```
1  data entier
2  properties
3    Language_Support::Data_Type => integer;
4  end entier;
5
6  subprogram spA
7  features
8    s : out parameter entier;
9  end spA;
10
11 subprogram spB
12 features
13   s : out parameter entier;
14 end spB;
15
16 subprogram spC
17 features
18   e : in parameter entier;
```

```
19   s : out parameter entier;
20 end spC;
21
22 subprogram implementation spA.impl
23 calls
24  {appel1 : subprogram spB;
25   appel2 : subprogram spC;};
26 connections
27   cnx1 : parameter appel1.s -> appel2.e;
28   cnx2 : parameter appel2.s -> s;
29 end spA.impl;
```

Listing V.10 – Exemple de séquence d'appel pure

Le sous-programme apparaît alors comme un moyen de connecter entre eux plusieurs autres sous-programmes. La séquence est accompagnée de la description des connexions entre les paramètres. Une telle modélisation permet de générer directement un code complet : les appels aux autres sous-programmes correspondent à des appels de procédures, et les connexions correspondent à des variables intermédiaires. Le code généré reflète alors strictement la description AADL.

Règle V.6 (Traduction des séquences d'appels pures)
Un sous-programme constitué seulement d'une séquence d'appel est traduit par une procédure ou une fonction effectuant les différents appels indiqués par la séquence.

Là encore, le code est généré de façon systématique ; les erreurs de structure, telles que l'absence ou la mise en concurrence de certaines connexions, doivent être détectées par d'autres moyens.

V-2.3.5 Implantation hybride

Les sections précédentes ont présenté des cas simples, dans lesquels un sous-programme AADL était soit un composant à l'implantation opaque soit une séquence d'appels inconditionnelle. Ces situations extrêmes n'offrent pas beaucoup de souplesse : dans un cas général, un sous-programme est susceptible d'appeler différents sous-programmes, en fonction des valeurs des paramètres qu'il reçoit.

Une modélisation efficace de ce genre de situation doit faire apparaître une séparation claire entre les conditions d'appel (c'est-à-dire la manipulation algorithmique des données) et les sous-programmes à appeler (c'est-à-dire l'enchaînement des appels et la description des connexions) ; il demeure ainsi facile de changer l'assemblage des composants tout en conservant l'algorithme inchangé.

Coordination de plusieurs séquences d'appel

Cette approche nécessite de pouvoir décrire plusieurs séquences d'appel possibles au sein du sous-programme, en adjoignant une description comportementale capable de contrôler l'exécution des séquences en fonction des paramètres du sous-programme.

Le standard actuel d'AADL permet de décrire plusieurs séquences d'appel au sein d'un même sous-programme. Il précise que chaque séquence doit être associée à un mode de configuration particulier. Il s'agit donc de se ramener au cas d'une séquence d'appel pure que nous avons abordé

en section V-2.3.4. L'enchaînement des séquences se fait donc par une succession de changements de mode.

Cette approche n'est pas satisfaisante pour l'implantation d'architectures logicielles complexes car elle conduit à une explosion combinatoire des modes d'exécution ; elle implique également la mise en place de mécanismes de reconfiguration complexes au sein de l'exécutif pour résoudre un problème qui est traité simplement dans la majorité des langages de programmation.

Nous choisissons donc une autre interprétation sémantique à la présence de plusieurs séquences d'appel au sein d'un sous-programme : nous considérons que les séquences d'appel présentes dans l'implantation d'un sous-programme représentent les différentes phases d'exécution possibles. Les différentes séquences doivent être coordonnées par une description comportementale fournie par l'utilisateur. Ainsi, le listing V.11 met en jeu trois phases d'exécution possibles, modélisées par les séquences `seq1`, `seq2` et `seq3`.

```
1  data entier
2  properties
3    Language_Support::Data_Type => integer;
4  end entier;
5
6  subprogram spA
7  features
8    a : in parameter entier;
9    d : out parameter entier;
10 end spA;
11
12 subprogram spB
13 features
14   e : in parameter entier;
15   s : out parameter entier;
16 end spB;
17
18 subprogram spC
19 features
20   s : out parameter entier;
21 end spC;
22
23 subrogram spD
24 features
25   e : in parameter entier;
26   s : out parameter entier;
27 end spD;
28
29 subprogram implementation spA.impl
30 subcomponents
31   default_value : data entier;
32 calls
33   seq1 : {spB1 : subprogram spB;};
34   seq2 : {spC2 : subprogram spC;
35           spB2 : subprogram spB;};
36   seq3 : {spD3 : subprogram spD;};
37 connections
38   parameter a -> spB1.e;
39   parameter spB1.s -> spD3.e;
```

```
40
41   parameter spC2.s -> spB2.e;
42   parameter spB2.s -> spD3.e;
43
44   parameter spD3.s -> d;
45 properties
46   Source_Name => ''Algo'';
47   Source_Text => ''Implantation'';
48   Source_Language => Ada95;
49 end sp1.impl;
```

Listing V.11 – Exemple de sous-programme AADL hybride

Afin d'illustrer notre propos, considérons un exemple de scénario d'exécution pour le sous-programme `spA.impl`. Celui-ci reçoit une donnée a, dont la sémantique correspond à la définition d'un entier. `spA.impl` peut avoir différents comportements possibles en fonction de la valeur de la donnée ; ces comportements correspondent à des séquences d'appel différentes et sont décrits par l'algorithme suivant :

Requiert a est un entier
  si $a < 4$ **alors**
    $b \leftarrow$ `spB` (a)
  sinon
    $c \leftarrow$ `spC` ()
    $b \leftarrow$ `spB` (c)
  fin si
  $d \leftarrow$ `spD` (b)
  renvoyer d

Selon la valeur de a, `spA.impl` appelle directement `spB`, ou `spC` puis `spB`. Dans tous les cas, la valeur retournée par `spB` est ensuite transmise à un dernier sous-programmes `spD` ; la valeur de retour de `spD` est ensuite retournée par `spA`.

Afin d'avoir une séparation claire entre la partie architecturale (décrite en AADL) et la partie comportementale (décrite en code source), cette dernière doit pouvoir manipuler les différentes séquences d'appel comme des blocs opaques ; il est alors possible de modifier le contenu des séquences (afin de changer les sous-programmes effectivement appelés) sans affecter l'algorithme. Celui-ci ne manipule donc seulement les séquences :

Requiert a est un entier
  si $a < 4$ **alors**
    `seq1`
  sinon
    `seq2`
  fin si
  `seq3`

Nous voyons que les appels aux trois séquences `seq1`, `seq2` et `seq3` s'apparentent en fait à des appels de procédure (ou de fonction) qui doivent être générée d'après la description AADL.

Connexion avec les données de la description comportementale

Nous avons ainsi décrit comment coordonner des séquences d'appel en fonction des données reçues par le sous-programme AADL. Nous n'avons cependant pas pu exprimer le fait que le

sous-programme puisse introduire de nouvelles données dans le flux d'exécution. En effet, les connexions ne permettent que de connecter les paramètres du sous-programme AADL aux paramètres des sous-programmes appelés.

Afin d'avoir la plus grande souplesse possible dans l'expression des algorithmes, il est nécessaire de pouvoir exprimer le fait que des données sont générées par le code source d'implantation et transmises aux sous-programmes appelés. Il est donc nécessaire de pouvoir introduire la notion de variables locales au sous-programme.

Une variable locale à un sous-programme s'apparente à un composant de données, déclaré comme sous-composant d'un sous-programme. La version 1.0 de la syntaxe AADL ne permet de modéliser de tels sous-composants dans les sous-programmes ; il est donc nécessaire de recourir aux extensions que nous introduisons en III-9, page 46. Dans ces conditions, nous pouvons reprendre l'exemple précédent, en remplaçant le sous-programme `spC` par une donnée locale, comme illustré sur le listing V.12.

```
1  data entier
2  properties
3    Language_Support::Data_Type => integer;
4  end entier;
5
6  subprogram spA
7  features
8    a : in parameter entier;
9    d : out parameter entier;
10 end spA;
11
12 subprogram spB
13 features
14   e : in parameter entier;
15   s : out parameter entier;
16 properties
17   Source_Text => "Algo";
18   Source_Name => "spB";
19   Source_Language => Ada95;
20 end spB;
21
22 subprogram spD
23 features
24   e : in parameter entier;
25   s : out parameter entier;
26 properties
27   Source_Text => "Algo";
28   Source_Name => "spD";
29   Source_Language => Ada95;
30 end spD;
31
32 subprogram implementation spA.impl
33 subcomponents
34   default_value : data entier; -- proposition pour AADL 1.2
35 calls
36   seq1 : {spB1 : subprogram spB;};
37   seq2 : {spB2 : subprogram spB;};
38   seq3 : {spD3 : subprogram spD;};
```

```
39 connections
40   cnx1 : parameter a -> spB1.e;
41   cnx2 : parameter spB1.s -> spD3.e;
42
43   cnx3 : parameter default_value -> spB2.e;
44   cnx4 : parameter spB2.s -> spD3.e;
45
46   cnx5 : parameter spD3.s -> d;
47 properties
48   Source_Text => "Algo";
49   Source_Name => "spA_impl";
50   Source_Language => Ada95;
51 end spA.impl;
```

Listing V.12 – Exemple de sous-programme avec une donnée locale

En terme de code source, la donnée locale se traduit par une variable, qui doit être manipulable par toutes les séquences et par la partie algorithmique.

Nommage des éléments architecturaux

La syntaxe AADL impose de nommer les sous-composants de donnée représentant les variables locales. Les séquences d'appels doivent également toutes être nommées afin de pouvoir y faire référence. Des séquences d'appel anonymes sont en effet syntaxiquement correctes, mais inexploitables dans notre situation car elles ne pourraient alors pas être désignées, donc pas appelées, par le code source de la description comportementale.

Le cas des connexions de paramètres est légèrement différent. En effet, les connexions se traduisent en effet par des variables intermédiaires, dont la valeur doit pouvoir être lue par l'algorithme. Le code source généré doit donc permettre à la partie algorithmique de manipuler les connexions AADL. Cependant, une connexion anonyme est tout à fait admissible : sa valeur ne pourra simplement pas être lue par l'implantation comportementale.

Traduction des sous-programmes hybrides

Une séquence d'appel est tout naturellement traduite en une procédure (ou fonction sans type de retour) dans le langage de programmation cible, de la même façon que le sous-programme AADL englobant. Les signatures de toutes les procédures correspondant aux séquences d'appels doivent être identiques, afin de pouvoir les manipuler de façon uniforme, quel que soit leur contenu.

> **Règle V.7 (Traduction des séquences d'appel d'un sous-programme hybride)**
> Chaque séquence d'un sous-programme hybride est traduit par une procédure ou une fonction dont la signature accepte un paramètre en entrée/sortie. Ce paramètre est une structure de donnée permettant de stocker les différentes variables correspondant aux connexions AADL déclarées dans le sous-programme.

Le code sources correspondant à l'algorithme doit pouvoir manipuler les informations décrivant l'état du sous-programme AADL, c'est-à-dire les différentes variables correspondant aux paramètres ou aux connexions dans le sous-programme AADL, ainsi que les références aux procédures correspondant aux séquences.

Dans la mesure où la signature du code de l'algorithme n'est pas censée dépendre des informations d'état du sous-programme AADL, ces informations doivent regroupées dans une structure de données propre au sous-programme AADL. À chaque sous-programme hybride est donc associé une structure de donnée permettant d'accéder à toutes les déclarations issues de la description AADL. De cette façon, l'ajout de nouvelles déclarations n'a pas d'influence sur l'algorithme ; de même le retrait de déclarations qui ne sont pas manipulées par l'algorithme n'a aucun impact. Les procédures des séquences prennent également cette donnée en paramètre, afin de pouvoir mettre à jour les différentes variables intermédiaires correspondant aux connexions.

Pour les sous-programmes hybrides, le traducteur AADL doit donc générer une structure de contrôle que l'implantation comportementale peut manipuler. L'association du code source est spécifiée de la même façon que pour une implantation opaque.

V-2.3.6 Sous-programme fourni par un composant de donnée

Un sous-programme fourni par un composant de donnée correspond à la modélisation d'un objet, bien qu'AADL ne possède pas la notion d'objet en tant que telle – l'extension de composant ne correspond pas exactement à la notion d'héritage, comme nous l'avons expliqué au chapitre III.

La traduction de tels sous-programmes est effectuée en suivant les même mécanismes que ceux utilisés classiquement dans la compilation des langages orientés objets tels que C++ ou Ada 95 : la donnée fournissant le sous-programme est passées par référence en premier paramètre de la procédure correspondante. Elle est donc intégrée à la structure de donnée que nous avons décrite dans la section précédente. Elle peut donc être manipulée par l'éventuelle description comportementale fourni par l'utilisateur.

Par convention, le champ correspondant au composant de donnée est appelé « data » ; ce nom est en effet un mot-clé d'AADL, et nous avons donc la garantie qu'il ne sera pas déjà employé pour un nom de connexion ou de séquence.

V-2.3.7 Synthèse sur la traduction des sous-programmes

Les trois premiers types de sous-programmes que nous avons étudiées sont en fait des cas particuliers de sous-programmes hybrides. Dans le cas général hybride, un sous-programme AADL contient des séquences d'appel et des sous-composants de données ; tous ces éléments sont coordonnés par le code source de la description comportementale associée au sous-programme.

Les sous-programmes sans implantation correspondent à une absence de séquences, de sous-composants de donnée et de code source associé ; les sous-programmes opaques correspondent à une absence de séquences et de sous-composants de donnée ; les séquences d'appel pures correspondent à une absence de code source associé et de sous-composant de donnée, avec pour restriction la présence d'une unique séquence d'appel. Les sous-programmes sans implantation correspondent à une absence de séquences, de composants de donnée et de code source associé ; les sous-programmes opaques correspondent à une absence de séquences et de sous-composants de donnée ; les séquences d'appel pures correspondent à une absence de code source associé et de sous-composant de donnée, avec pour restriction la présence d'une unique séquence d'appel.

Lorsque les propriétés `Source_Language`, `Source_Name` et `Source_Text` sont renseignées, le code source issu de la description AADL doit appeler la description comportementale fournie par l'utilisateur. Le sous-programme AADL est alors opaque ou hybride. Nous définissons une règle systématique pour la signature de l'implantation comportementale.

Règle V.8 (Signature des implantations comportementales)
La procédure fournie par l'utilisateur doit toujours avoir la signature suivante :
– un premier paramètre en entrée/sortie appelé **`data`**, de type `<nom_du_sous_programme>_data`, qui reçoit une structure de donnée contenant les éventuelles références des procédures des séquences d'appel, ainsi que les variables locales et les variables issues des connexions AADL ;
– les différents paramètres du sous-programme AADL.

Le type de données `<nom_du_sous_programme>_data` est généré par le traducteur pour chaque sous-programme AADL encapsulant une description comportementale. Du fait que « **`data`** » est un mot-clé AADL, nous pouvons l'utiliser en ayant l'assurance qu'il ne correspond pas à un nom de paramètre utilisé dans la modélisation.

V-2.4 Gestion des entités distantes

Les évolutions de syntaxe que nous avons proposées au chapitre IV permettent la modélisation de sous-programmes et d'objets répartis. Dans cette section nous traitons ces cas. Nous montrons notamment que les aspects répartis de l'application relève exclusivement de la description AADL ; ils n'ont aucun impact sur la définition des description comportementales.

V-2.4.1 Traduction des appels de sous-programmes distants

Dans la section précédente, nous avons traité le cas des sous-programmes locaux. Un appel vers un accès à un sous-programme instancié dans le processus local ne pose aucune difficulté de traduction. Un appel à un sous-programme distant est représenté en AADL par un appel à un accès à sous-programme qui est instancié dans un autre processus.

Le passage de message, matérialisé par les ports AADL, permet une stricte séparation entre les deux parties de l'application AADL, l'interface avec l'intergiciel contrôlant l'exécution de l'enveloppe sans que celle-ci aie une quelconque visibilité sur l'exécutif. L'appel de sous-programme implique en revanche un appel à l'interface avec l'intergiciel de la part de l'enveloppe applicative.

Un appel à un accès à un sous-programme instancié dans un autre processus doit donc donner lieu à la génération d'un couple de souche et de squelette, de la même façon que pour les approches RPC ou CORBA.

Le traducteur doit donc substituer au sous-programme instancié à distance un sous-programme local, qui sera fourni par l'interface avec l'intergiciel. Ce sous-programme joue le rôle de souche ; le squelette correspondant, généré au niveau du nœud distant, appellera le sous-programme effectivement instancié.

Ce processus de génération de souche et de squelette est seulement paramétrée par des éléments de la description AADL – les processus dans lesquels sont instanciés les sous-programmes. Les descriptions comportementales n'ont aucune visibilité sur la localisation effective des sous-programmes, et ne sont donc pas affectées.

V-2.4.2 Traduction des objets répartis

Dans le cadre de ces travaux, nous ne considérons les objets répartis que sous la forme d'appel à des sous-programmes fournis par des composants de donnée instanciés dans des processus distants.

Dans la section V-2.3.6, nous avons décrit la traitement des sous-programmes fournis par un composant de donnée, et montré qu'il se ramenait au traitement des sous-programmes normaux. Il en est donc de même pour la traduction des objets répartis : nous pouvons nous ramener au cas des sous-programmes distants.

V-2.5 Organisation des fichiers générées

Afin de structurer la génération de code, nous nous efforçons de conserver l'organisation des entités exprimée dans la description AADL ; il s'agit essentiellement de retranscrire les paquetages AADL dans le langage cible.

Une description AADL met en jeu des composants instanciés et des composants qui ne le sont pas. Au niveau d'un processus, les composants à considérer sont les *threads*, les données et les sous-programmes.

- les *threads* AADL sont instanciés au sein du processus ;
- la déclaration d'un composant de donnée correspond à la déclaration d'un type de donnée, tandis qu'une instance de donnée correspond à une variable ;
- dans le cas général, les sous-programmes sont seulement appelés, sans être instanciés ; cependant, un sous-programme peut-être instancié, comme nous l'avons montré au chapitre IV.

Les déclarations d'entités – données et sous-programmes – correspondent à la notion de déclaration de type ou de procédure dans les langages de programmation impératifs. Ils est donc possible de les regrouper en reproduisant l'organisation des paquetages AADL. Les entités instanciées ne sont en revanche pas associées à un paquetage donné ; elles constituent un ensemble séparé.

Règle V.9 (Regroupement du code source)
Les paquetages AADL représentent les unités d'organisation des déclarations AADL.
Ils sont traduits par leur équivalent dans le langage cible.

Nous structurons donc le code généré en plusieurs ensemble de code correspondant à des paquetages, espaces de nom, etc. selon le langage cible considéré. Le code généré est structuré ainsi :

- le code correspondant aux entités instanciées est généré dans un ensemble de code que nous appelons « partition » ;
- le code correspondant aux entités qui ne sont pas instanciées est généré dans un ensemble dont le nom correspond au paquetage AADL dans lequel les entités AADL sont déclarées. Cet ensemble de code est un sous-ensemble de « partition » ;
- le code correspondant aux entités non instanciées, déclarées dans l'espace de nom anonyme AADL, est généré dans l'ensemble de code « partition ».

La traduction en langage de programmation conserve ainsi au maximum l'organisation des déclarations AADL.

V-3 Traduction des composants applicatifs en langage Ada

Le langage Ada est très répandu dans le domaine des systèmes embarqués temps-réel [Burns et Wellings, 2001]. Nous nous sommes donc particulièrement focalisés sur la génération de code vers ce langage. Dans cette section nous décrivons la traduction en Ada des éléments AADL que nous avons étudiés en section V-2.

V-3.1 Traduction des espaces de nom

Ada possède la notion de paquetage, bien qu'elle ne corresponde pas exactement à celle d'AADL. En effet, contrairement à Ada, il n'y a aucune relation particulière en AADL entre un paquetage et ses sous-paquetages. Nous pouvons donc appliquer directement le mécanisme de traduction que nous avons expliqué en section V-2.5.

Règle V.10 (Traduction des paquetages AADL en Ada)
L'espace de nom anonyme est transformé en un paquetage Ada appelé `Partition`. Les différents espaces de nom (qui sont des fils de l'espace anonyme) sont transformés en sous-paquetages de `Partition`. Une hiérarchie de paquetages est générée pour chaque nœud de l'application. Elle contient les déclarations de tous les types de données et sous-programmes utilisés dans le nœud.

V-3.2 Traduction des composants de donnée

En l'absence d'informations sémantiques associées à la déclaration d'un composant de donnée, nous générons un type Ada vide. Ainsi, le type `une_donnee` décrit au listing V.1 donne lieu à la génération d'un type vide, comme illustrée au listing V.13.

```
3 type une_donnee is new null record;
```

Listing V.13 – Type Ada correspondant à une donnée AADL sans sémantique

Les autres types de donnée se traduisent relativement simplement, comme illustré sur le listing V.14. Les propriétés AADL ne permettent pas d'indiquer les indices de début et de fin pour les tableaux ; nous choisissons de fixer le premier indice à 0 afin de faciliter les échanges de données avec des langages comme le C. La syntaxe AADL utilise le point (« . ») pour marquer le nom d'une implantation de composant. Comme la syntaxe d'Ada n'autorise pas ce caractère dans les identificateurs, nous le remplaçons par la chaîne « _i_ ».

```
5 type une_donnee_i_entier is new Integer;
6 type une_donnee_i_enumeration is ("bleu", "jaune");
7 type une_donnee_i_tableau_entier is array (0 .. 5) of Integer;
8 type une_donnee_i_tableau_entier2 is array (0 .. 5) of
      une_donnee_i_entier;
9 type une_donnee_i_chaine is String (0 .. 6);
10
11 type une_donnee_i_structure is record
12   i : une_donnee_i_entier;
13   c : une_donnee_i_chaine;
14 end record;
```

Listing V.14 – Types Ada issus des déclarations AADL

La traduction d'une structure de liste est plus complexe car il ne s'agit pas d'une construction prise en charge directement par le langage Ada. Nous recourrons donc à un type paramétré, matérialisé par l'instanciation d'un paquetage générique (listing V.15).

```
16 package une_donnee_i_liste is new
17   Arao.Data_Structure (une_donnee_i_entier);
```

Listing V.15 – Type Ada correspondant à un composant de liste en AADL

La définition de ce paquetage générique doit faire partie de la bibliothèque d'exécution associée au générateur de code.

V-3.3 Traduction des sous-programmes

Règle V.11 (Traduction des sous-programmes AADL en Ada)
Les sous-programmes AADL sont traduits en Ada par des procédures.

Les accès aux composants de donnée sont traduits par un accès au type de donnée considéré ; il en va de même pour les accès aux sous-programmes. Les paramètres AADL sont traduits par des paramètres de procédure AADL selon les règles suivantes :

- le nom des paramètres correspond à celui des éléments d'interface ;
- le type des paramètres correspond à la traduction des types AADL correspondants (cf. V-2.2) ;
- la direction des paramètres est la même que celle déclarée dans la description AADL (« in », « out » ou « in out »).

Dans le cas général d'un sous-programme hybride, les différentes séquences d'appel sont traduites en procédures ; le sous-programme AADL est également traduit en une procédure contenant le code source d'implantation. Cette procédure reprend la signature du sous-programme AADL, en ajoutant le premier paramètre `Data` que nous avons décrit en section V-2.3.7. Le type de donnée associé à ce paramètre est un enregistrement contenant les noms des connexions AADL et les accès aux procédures correspondant aux séquences AADL. Dans le cas d'un sous-programme fourni en interface d'un composant de donnée AADL, l'enregistrement contient aussi un champ `Data`, qui est un accès au type de donnée issu du composant de donnée AADL. Les séquences d'appel sont traduites par des procédures qui acceptent comme argument le paramètre `Data`.

Les listings V.16 et V.17 décrivent tout le code généré à partir de la description AADL du listing V.12

```
1  package Partition is
2    type entier is new Integer;
3
4    type spA_impl_control is record
5      a : entier;
6      d : entier;
7      default_value : entier;
8      spD3_e : entier;
9    end record;
10
11   type spA_impl_sequence is
12     access procedure (Status : in out spA_impl_control);
13
14   procedure spA_impl (a : in entier; d : out entier);
15   procedure spB (e : in entier; s : out entier);
16   procedure spD (e : in entier; s : out entier);
17 end Partition;
```

Listing V.16 – Spécifications Ada générées à partir de la description AADL du listing V.12

```
1  package body Partition is
2    with Algo;
3
4    procedure spA_impl_seq1 (Status : in out spA_impl_control);
5    procedure spA_impl_seq2 (Status : in out spA_impl_control);
6    procedure spA_impl_seq3 (Status : in out spA_impl_control);
7
```

```
8   procedure spB (e : in entier; s : out entier) is
9   begin
10    Algo.spB (e, s);
11  end spB;
12
13  procedure spD (e : in entier; s : out entier) is
14  begin
15    Algo.spD (e, s);
16  end spD;
17
18  procedure spA_impl_seq1 (Status : in out spA_impl_control) is
19  begin
20    spB (Status.a, Status.spD3_e);
21  end spA_impl_seq1;
22
23  procedure spA_impl_seq2 (Status : in out spA_impl_control) is
24  begin
25    spB (Status.default_value, Status.spD3_e);
26  end spA_impl_seq1;
27
28  procedure spA_impl_seq3 (Status : in out spA_impl_control) is
29  begin
30    spD (Status.spD3_e, Status.d);
31  end spA_impl_seq1;
32
33  procedure spA_impl (a : in entier; d : out entier) is
34    Status : spA_impl_control;
35  begin
36    Status.a := a;
37    Algo.spA_impl (Status,
38                   spA_impl_seq1'Access,
39                   spA_impl_seq2'Access,
40                   spA_impl_seq3'Access);
41    d := Status.d
42  end sp1_impl;
43 end Partition;
```

Listing V.17 – Code Ada généré à partir de la description AADL du listing V.12

Le code source Ada correspondant à l'implantation de l'algorithme est fourni par l'utilisateur et doit suivre l'interface prévue :

```
1  with Partition;
2
3  package body Algo is
4
5    procedure spA_impl
6           (Status : in out Partition.spA_impl_control;
7            Seq1 : Partition.spA_impl_sequence;
8            Seq2 : Partition.spA_impl_sequence;
9            Seq3 : Partition.spA_impl_sequence)
10   is
11   begin
12     Status.default_value := 4;
```

```
13
14    if Status.a < 4 then
15      Seq1 (Status);
16    else
17      Seq2 (Status);
18    end if;
19
20    Seq3 (Status);
21  end spA_impl;
22
23  procedure spB (e : in partition.entier; s : out partition.entier
        ) is
24  begin
25        s := e + 1;
26  end spB;
27
28  procedure spD (e : in partition.entier; s : out partition.entier
        ) is
29  begin
30        s := e * 6;
31  end spD;
32
33 end Algo;
```

Listing V.18 – Code Ada implantant l'algorithme

La cohérence entre la procédure-enveloppe générée à partir de l'AADL et la procédure fournie par l'utilisateur sera vérifiée par le compilateur Ada 95.

V-4 Traduction des composants AADL applicatifs dans d'autres langages

Dans cette section nous nous intéressons à la traduction des constructions AADL vers d'autres langages qu'Ada. Nous considérons les langages C et Java. C est un langage purement procédural, largement répandu dans les systèmes embarqués temps-réel. Java est un langage orienté objet ; il n'est pas encore très utilisé, mais il tend à se développer, notamment grâce à des spécifications de machines virtuelles spécialisées telles que JavaCard [Sun, 2006] ou Real-Time Java [Bollella et al., 2002]. La prise en compte de ce deux langages nous permet de couvrir un large panorama de langages de programmation impérative.

V-4.1 Traduction vers C

La traduction des constructions AADL en Ada ne fait pas intervenir les constructions orientées objet d'Ada. La traduction vers C lui est donc similaire. Nous n'abordons donc que les principales différences qui se posent entre les deux langages.

V-4.1.1 Traduction des espaces de nom

Le langage C n'a pas la notion de paquetages. Les espaces de noms sont donc matérialisés par différents fichiers. Le fichier principal, correspondant à l'espace de nom anonyme, est

nommé `partition.c`, et les fichiers correspondant aux espaces de nom sont nommés `partition -<espace_de_nom>.c`. Étant donné que toutes les déclarations C se trouvent dans l'unique espace de nom géré par le langage, nous appliquons une règle différente d'Ada pour le nommage des entités issues des composants AADL : leur nom est le résultat de la concaténation du nom de leur paquetage de définition et de leur nom à l'intérieur de ce paquetage.

V-4.1.2 Traduction des composants de donnée

Le langage C permet de manipuler les mêmes types de données qu'Ada. La traduction des composants de donnée suit donc les mêmes principes. Les chaînes de caractères sont implantées par des tableaux de caractères, de la dimension spécifiée en AADL, plus un octet pour stocker le caractère de fin de chaîne `0`.

Dans la mesure où C n'offre pas de mécanisme permettant de définir des sous-programmes génériques, il est nécessaire de générer plus de code, s'appuyant sur une structure de liste manipulant des pointeurs vers les structures de données à stocker.

V-4.1.3 Traduction des sous-programmes

Règle V.12 (Traduction des sous-programmes AADL en C)
Les sous-programmes AADL sont traduits en C par des fonctions dont le type de retour est **`void`**.

Les accès aux composants de donnée sont traduits par un pointeur sur le type de donnée considéré ; de même, les accès aux sous-programmes sont traduits par des pointeurs sur procédure. Les autres paramètres sont traduits selon les mêmes principes que pour Ada. Les paramètres AADL déclarés « out » ou « in out » sont traduits par des pointeurs sur les types de données correspondants. Les paramètres « in » sont traduits par un passage de paramètre classique.

V-4.2 Traduction vers Java

La traduction en Java reprend en partie les mêmes principes que pour Ada et C. Cependant, du fait qu'il s'agisse d'un langage orienté objet implique quelques différences notables. AADL ne développe en effet pas les concepts que l'on retrouve habituellement dans les langages objets (telles que les classes, notamment). La traduction d'AADL vers Java conduit donc à générer du code qui, dans sa logique, n'est pas orienté objet ; le code produit, bien que correct, ne correspond donc pas nécessairement à ce qu'écrirait naturellement un programmeur.

V-4.2.1 Traduction des espaces de nom

Java possède la notion de paquetage ; la notion de paquetages d'AADL s'en inspire d'ailleurs largement. Cependant, du fait que Java se fonde sur l'usage des classes alors que cette notion n'existe pas dans AADL, il n'est pas possible de traduire simplement les paquetages AADL en paquetages Java ; bien que de sémantique très proche, les paquetages Java et AADL ne contiennent en effet pas les mêmes genres de déclarations. Notamment, il n'est pas possible de déclarer une méthode Java directement dans un paquetage. Il est donc impératif de tenir compte à la fois des notions de classe et de paquetage dans la traduction d'AADL vers Java, tout particulièrement pour les sous-programmes.

Règle V.13 (Traduction des paquetages AADL en Java)
Un paquetage AADL se traduit par un paquetage Java. Les conventions de nommage sont similaire à Ada.

Les déclarations placées dans ce paquetage ne se traduisent en revanche pas de façon aussi directe, comme nous l'étudions dans les sections suivantes. De la même façon que pour Ada et C, le paquetage principal, correspondant à l'espace de nom anonyme, est nommé « partition » ; il contient les différentes déclarations de l'espace de nom anonyme. Ce paquetage est placé dans un fichier `Partition.java` ; les différents fichiers correspondant aux espaces de nom sont nommés `<espace_de_nom>.java` et placés dans les sous-répertoires correspondants.

V-4.2.2 Traduction des composants de donnée

La définition de nouveaux types de données en Java s'effectue par la déclaration de classes.

Règle V.14 (Traduction des déclarations de donnée AADL en Java)
Chaque déclaration de composant de donnée AADL se traduit par une classe Java de même nom, déclarée dans le paquetage correspondant.

De la même façon que pour Ada, la traduction d'un composant de donnée AADL sans sémantique – tel que `une_donnee` au listing V.1 – se traduit par une classe vide, comme illustré au listing V.19.

```
5 class une_donnee { }
```

Listing V.19 – Classe Java correspondant à une donnée AADL sans sémantique

Les classes Java permettant de décrire les types de base, telle que `Integer` pour les entiers, ne permettent que de décrire des objets « constants » : ces objets représentent une valeur constante, qu'il n'est pas possible de modifier. De façon parallèle, la classe Java `String` ne permet pas de représenter des chaînes de caractères de taille fixée. Dans le cadre de la traduction des composants de données AADL en Java, nous nous reposons donc sur les types « classiques » inspirés du C, tels que le type **`int`** ou les tableaux de caractères.

Les différents types de donnée se traduisent donc par des classes contenant un champ `val` dont le type correspond à la sémantique de la donnée AADLi, comme représenté au listing V.20.

```
7  class une_donnee_i_entier {
8          public int val;
9  }
10
11 enum une_donnee_i_enumeration {
12         bleu, jaune;
13 }
14
15 class une_donnee_i_tableau_entier {
16         public int[] val = new int[5];
17 }
18
19 class une_donnee_i_tableau_entier2 {
20         public une_donnee_i_entier[] val = new une_donnee_i_entier
               [5];
21 }
22
```

```
23 class une_donnee_i_chaine {
24       public char[] val = new char[6];
25 }
26
27 class une_donnee_i_structure {
28       public une_donnee_i_entier i = new une_donnee_i_entier();
29       public une_donnee_i_chaine c = new une_donnee_i_chaine();
30 }
```

Listing V.20 – Classes Java issues des déclarations AADL

De la même façon, les tableaux et les chaînes de caractères sont déclarés par un champ `val` contenant un tableau de la dimension spécifiée dans le composant AADL. Les composants de donnée correspondant à une énumération sont traduits par une structure de donnée Java `enum`, introduite dans Java 5.0.

Contrairement à Ada, la traduction d'une structure de liste est relativement simple, dans la mesure où Java définit la classe `LinkedList`, correspondant à une liste doublement chaînée. La traduction consiste donc à avoir une classe dont le champ `val` est objet de cette classe (listing V.21).

```
32 class une_donnee_i_liste {
33       public LinkedList val = new LinkedList();
34 }
```

Listing V.21 – Classe Java correspondant à un composant de liste en AADL

V-4.2.3 Traduction des sous-programmes

La syntaxe de Java ne permet pas de déclarer une méthode en dehors d'une classe. Les différents sous-programmes AADL doivent donc être traduits par des méthodes d'une classe Java particulière, définie dans chaque paquetage Java. Nous choisissons d'appeler cette classe `subprogram`, qui est un mot-clé AADL ; de cette façon, nous nous assurons que ce nom n'est pas utilisé par ailleurs dans la description AADL. Dans la mesure où la classe `subprogram` n'est pas censée être instanciée, les méthodes issues des sous-programmes AADL sont nécessairement des méthodes de classe.

Règle V.15 (Traduction des sous-programmes AADL en Java)
Les sous-programmes d'un paquetage AADL sont traduits en Java par des méthodes d'une classe `subprogram` déclarée dans le paquetage Java correspondant.

Le passage des objets en paramètre se fait systématiquement par référence en Java. Par rapport aux règles de traductions que nous avons présentées en section V-4.2.2, tous les types de donnée Java issus des déclarations AADL sont encapsulés dans des classes. Ces constructions syntaxiques ne permettent pas de contrôler la modification des données qu'elles contiennent, ce qui revient à considérer tous les paramètres comme étant **`in out`**.

Afin de conserver la sémantique du passage des paramètres, nous encapsulons l'ensemble des paramètres dans un conteneur qui permettra de contrôler les accès aux données. Un conteneur peut être comparé dans une certaine mesure à une classe *Holder* telle que définie dans les règles de conversion entre CORBA IDL et Java [OMG, 2002a]. Il fournit les méthodes d'accès correspondant aux directions des différents paramètres du sous-programme considéré. Un conteneur fournit alors les méthodes suivantes :

– une méthode `read_p` si le paramètre AADL `p` est déclaré **`in`** ;

– une méthode `write_p` si le paramètre AADL `p` est déclaré **`out`**.

Un paramètre `p` déclaré **`in out`** engendre la création de deux méthodes `read_p` et `write_p`. Les méthodes d'accès permettent de lire ou d'écrire des attributs du conteneur correspondant aux paramètres du sous-programme AADL considéré. Ces attributs sont publics, de sorte que l'exécutif peut les modifier ; les méthodes d'accès ne sont donc utilisées que par l'utilisateur.

De cette façon, nous évitons que l'implantation en code source des sous-programmes AADL n'interfère avec la sémantique du passage des paramètres. La traduction conserve donc complètement la signature des sous-programmes.

Les différentes séquences d'appel sont réunies dans une classe ; l'implantation en code source reçoit un objet de cette classe en paramètre au lieu d'une référence sur chaque procédure. Le langage Java impose cependant de définir chaque classe dans un fichier séparé. Les listings V.22 à V.28 illustrent le code source à générer à partir du listing AADL V.12 ; tous ces codes sources sont associés au paquetage Java `Partition`.

Le listing V.22 représente le type de donnée généré à partir du type de donnée AADL `entier`.

```
1 package Partition;
2
3 public class entier {
4       public int val;
5 }
```

Listing V.22 – Type de donnée généré à partir de la description AADL du listing V.12

Les listings V.23, V.24 et V.25 représentent le code Java correspondant aux paramètres des sous-programmes `spA.impl`, `spB` et `spD`.

```
1 package Partition;
2
3 public class spA_i_impl_parameters {
4       public entier d;
5       public entier a;
6
7       public entier read_a () {
8             return a;
9       };
10
11      public void write_d (entier param) {
12            d = param;
13      };
14 }
```

Listing V.23 – Classe correspondant aux paramètres du sous-programme `spA.imp`

```
1 package Partition;
2
3 public class spB_parameters {
4       public entier s;
5       public entier e;
6
7       public entier read_e () {
8             return e;
9       };
10
11      public void write_s (entier param) {
```

```
12                s = param;
13        };
14 }
```

Listing V.24 – Classe correspondant aux paramètres du sous-programme `spB`

```
1  package Partition;
2
3  public class spD_parameters {
4         public entier s;
5         public entier e;
6
7         public void write_s (entier param) {
8                 s = param;
9         };
10
11        public entier read_e () {
12                return e;
13        };
14 }
```

Listing V.25 – Classe correspondant aux paramètres du sous-programme `spD`

Les listings V.26 et V.27 représentent respectivement le code Java correspondant à la structure de contrôle associée au sous-programme `spA.impl` et le code correspondant aux différentes séquences d'appel de `spA.impl`.

```
1  package Partition;
2
3  public class spA_i_impl_control {
4         public entier a = new entier();
5         public entier d = new entier();
6         public entier default_value = new entier();
7         public entier spD3_e = new entier();
8  }
```

Listing V.26 – Classe de contrôle pour le sous-programme `spA.impl`

Le code généré pour les différentes séquences prend en charge les connexions AADL : celles-ci sont traduites par des affectations des différentes attributs des conteneurs de paramètres à partir des attributs de l'objet de contrôle (défini au listing V.26).

```
1  package Partition;
2
3  public class spA_i_impl_subprogram {
4         public void seq1 (Partition.spA_i_impl_control Status) {
5                 param = new spB_parameters ();
6                 param.e = Status.a;
7                 param.s = Status.spD3_e;
8                 Partition.subprogram.spB (param);
9         }
10
11        public void seq2 (Partition.spA_i_impl_control Status) {
12                param = new spB_parameters ();
13                param.e = Status.default_value;
14                param.s = Status.spD3_e;
```

```
15                Partition.subprogram.spB (param);
16        }
17
18        public void seq3 (Partition.spA_i_impl_control Status) {
19                param = new spD_parameters ();
20                param.e = Status.spD3_e;
21                param.s = Status.d;
22                Partition.subprogram.spD (param);
23        }
24 }
```

Listing V.27 – Classe contenant les séquences d'appel du sous-programme `spA.impl`

Enfin, le listing V.28 représente le code généré pour l'ensemble des sous-programmes AADL définis dans le listings V.12.

```
1  package Partition;
2  import Algo.*;
3
4  public class subprogram {
5         public static void spA_i_impl (spA_i_impl_parameters param)
               {
6                 spA_i_impl_control Status = new spA_i_impl_control();
7                 Status.a = param.read_a ();
8                 Algo.spA_impl (Status, new spA_i_impl_subprogram());
9                 param.write_d (Status.d);
10        }
11
12        public static void spB (spB_parameters param) {
13                Algo.spB (param);
14        }
15
16        public static void spD (spD_parameters param) {
17                Algo.spD (param);
18        }
19 }
```

Listing V.28 – Code Java correspondant aux déclarations de sous-programmes AADL du listing V.12

De la même façon que pour Ada, les méthodes correspondant aux sous-programmes `spB` et `spD` se contentent d'appeler le code fourni par l'utilisateur. En revanche, l'enveloppe générée pour le sous-programme `spA.impl` initialise la structure de contrôle et les différentes séquences avant d'appeler l'implantation fournie par l'utilisateur.

L'implantation de l'algorithme fournie par l'utilisateur en Java est représentée au listing V.29.

```
1  package Algo;
2  import Partition.*;
3
4  public class Algo {
5     public static void spA_impl (spA_i_impl_control Status,
          spA_i_impl_subprogram sequences) {
6         Status.default_value.val = 4;
7
8         if (Status.a.val < 4)
```

```
9             sequences.seq1 (Status);
10        else
11            sequences.seq2 (Status);
12
13        sequences.seq3 (Status);
14    }
15
16    public static void spB (Partition.spB_parameters param) {
17        entier s = new entier();
18        entier e = param.read_e ();
19        s.val = e.val + 1;
20        param.write_s (s);
21    }
22
23    public static Partition.spD_out spD (Partition.spD_parameters
          param) {
24        entier s = new entier();
25        entier e = param.read_e ();
26        s.val = e.val * 6;
27        param.write_s (s);
28    }
29 }
```

Listing V.29 – Code Java implantant l'algorithme

V-5 Conclusion

Nous avons défini un processus de traduction des constructions AADL purement applicatives – les données et les sous-programmes – en langage de programmation. Notre approche se base sur une séparation nette des constructions architecturales et du code des algorithmes : les types de données et les appels de sous-programmes sont définis par la description AADL, tandis que le comportement des composants AADL est fournie par l'utilisateur sous forme de code source.

L'implantation en code source des composants doit correspondre à la signature décrite en AADL. De la même façon que pour IDL, nous avons défini des règles de traduction vers différents langages de programmation. Les constructions AADL peuvent être traduites dans une grande diversité de langages, aussi bien purement procéduraux (C) qu'orientés objet (Java). Nous avons particulièrement étudié Ada, utilisé pour les systèmes embarqués temps-réel ; la richesse de sa syntaxe permet de retranscrire facilement les descriptions AADL. À l'inverse, Java est à l'origine conçu pour les systèmes d'information ; il n'offre pas toutes les constructions syntaxiques d'Ada, ce qui rend la traduction un peu plus complexe.

Dans tous les cas, les règles de traductions permettent une intégration relativement simple des descriptions comportementales. L'utilisation d'AADL pour construire une application constitue donc une solution viable.

Nous exploitons la description pour construire l'assemblage et la coordination des différents sous-programmes. Nous nous démarquons du standard AADL au niveau de la sémantique associées aux séquences d'appel : nous nous autorisons en effet à décrire plusieurs séquences au sein d'un sous-programme. Ces différentes séquences correspondent aux différentes phases d'exécution possible du sous-programme.

Nous définissons des règles de traduction pour permettre la manipulation des séquences par les

descriptions comportementales. L'enchaînement des appels au niveau des séquences relève cependant uniquement du domaine de la description AADL. Il est par conséquent possible de modifier l'assemblage des composants ou leur déploiement sans nécessairement affecter l'implantation des algorithmes.

Notre approche permet donc une synthèse des possibilités offertes par les langages de description d'interface tels qu'IDL et les langages d'assemblage de composant tels que FractalADL ou les approches orientées composants telles que J2EE. En autorisant de façon transparente l'appel de sous-programmes distants ou locaux, nous permettons un redéploiement facile de l'architecture applicative, de la même façon que des approches telles que GLADE.

CHAPITRE VI

Construction et configuration de l'interface avec l'intergiciel d'exécution

DANS LE CHAPITRE IV nous avons établi une séparation entre la sémantique des *threads* et des sous-programmes AADL. Le code source produit à partir des sous-programmes doit reposer sur un exécutif chargé de coordonner l'ensemble de l'application. Les *threads* AADL servent de point d'entrée pour les communications ; ils constituent également le point de départ de l'exécution des éléments logiciels AADL (c'est-à-dire les sous-programmes AADL). Ils matérialisent donc l'intergiciel d'exécution.

Dans le chapitre V nous avons expliqué comment traduire la partie applicative d'une description AADL en code source. Nous nous focalisons ici sur l'interprétation des *threads* afin de paramétrer l'intergiciel d'exécution. Les éléments architecturaux doivent être retranscrits en appliquant d'une part des règles de correspondance entre AADL et le langage cible, et d'autre part en tenant compte de la bibliothèque d'exécution et de l'intergiciel sous-jacents. Nous montrons comment AADL peut être exploité comme langage de configuration pour l'intergiciel – correspondant à la première phase du processus de conception – ou pour sa modélisation – correspondant à la seconde phase du processus.

VI-1 Spécifications de l'intergiciel d'exécution

La description d'une architecture doit tenir compte des capacités de l'exécutif, comme nous l'avons souligné dans le chapitre IV-5, page 65.

L'approche pour la génération de systèmes exécutables que nous avons présentée en section IV-4, page 61, se fonde sur le fait que l'exécutif doit contrôler l'application, afin de préserver au maximum les caractéristiques de l'architecture telles que les temps d'exécution. Les règles de production proposées par le standard 1.0 AADL s'appuient sur un principe opposé ; elles permettent une grande souplesse dans l'implantation des systèmes, mais ne prennent pas en compte les architectures basées sur un intergiciel, rendant ainsi difficile la prise en charge des communications entre les nœuds.

Nous décrivons ici les spécifications de l'intergiciel d'exécution que nous avons définies dans le cadre de nos travaux. Ces spécifications permettent de définir une interface entre l'enveloppe applicative dont nous avons étudié la génération au chapitre V et l'intergiciel d'exécution lui-même.

VI-1.1 Sémantique des *threads* AADL

L'intergiciel d'exécution est symbolisé par les *threads* AADL, et doit donc fournir les fonctionnalités correspondant à la sémantique de ces composants. Ces fonctionnalités correspondent à deux grandes catégories de services :

– la gestion des fils d'exécution ;
– la gestion des communications.

La gestion de l'exécution des composants peut se reposer sur une bibliothèque de *threads* systèmes fournie par l'exécutif. Nous considérons que chaque *thread* AADL représente une entité autonome gérée par l'exécutif. Cette correspondance permet de transposer les constructions AADL en terme de ressources système effectivement nécessaires.

VI-1.1.1 Contrôle de l'application

Le standard AADL définit quatre politiques de déclenchement pour les *threads*, spécifiées par la propriété AADL standard `Dispatch_Protocol` :

– périodique ;
– apériodique ;
– sporadique ;
– tâche de fond.

Un *thread* périodique est déclenché régulièrement, selon la période indiquée par la propriété standard `Period`. Un *thread* apériodique est déclenché par les données arrivant sur un port de donnée/événement déclencheur ou l'appel de l'un des sous-programmes dont il fournit l'accès. Un *thread* sporadique se comporte comme un *thread* apériodique, mais ne peut se déclencher avant le délai spécifié par la propriété `Period`. Enfin, un *thread* en tâche de fond n'est déclenché qu'au lancement du système.

VI-1.1.2 Prise en charge des communications

Les communications sont typiquement assurées par un intergiciel de communication configurable. En section IV-5, page 65, nous avons défini des patrons architecturaux pour représenter différents modèles de répartition, représentés par les interfaces des *threads* AADL : le passage de message, l'appel de sous-programme distant et de méthodes d'objets répartis.

La bibliothèque de communication fournie par l'exécutif doit être capable de prendre en charge ces trois paradigmes. Pour cela, nous interprétons les éléments d'interface comme des requêtes.

Spécification VI.1 (Sémantique des éléments d'interface)
Chaque élément d'interface d'un *thread* AADL est traduit par une requête au niveau de l'intergiciel. Les données associées à l'élément d'interface constituent les paramètres de la requête.

La structure des requêtes est la suivante :

– les ports sont traduits par une requête à sens unique (correspondant à un message, ou une interface *oneway* CORBA) dont l'unique paramètre correspond à la donnée associée au port ;
– les sous-programmes fournis en interface correspondent à une requête/réponse dont les paramètres correspondent aux paramètres du sous-programme (noms et types).

Spécification VI.2 (Spécification d'un port déclencheur)
Si l'interface d'un *thread* apériodique ou sporadique comporte plusieurs ports d'événement/donnée, le port déclenchant l'exécution doit être indiqué à l'aide d'une propriété appelée `Dispatch_Port`, de type booléen.

L'exécution d'un *thread* apériodique ou sporadique est déclenchée par l'arrivée d'une requête correspondant à un port d'événement/donnée ou un appel de sous-programme distant. Dans la mesure où toutes les données des ports sont traitées en même temps par le *thread*, l'exécution de celui-ci ne peut être déclenchée que par un seul port. Nous avons défini la propriété `Dispatch_Port` en section IV-5.4, page 69.

VI-1.2 Niveau des services fournis par l'intergiciel d'exécution

Nous pouvons distinguer deux grandes catégories d'intergiciel, selon le niveau des services qu'ils fournissent.

Définition VI.1 (intergiciel de haut niveau)
Un intergiciel est dit de *haut niveau* s'il peut prendre directement en charge des modèles de distribution tels que le passage de message, les sous-programmes distants et les objets partagés.

Définition VI.2 (intergiciel de bas niveau)
Un intergiciel est dit de *bas niveau* s'il ne fournit qu'un service de communication rudimentaire, limité à des envois et réceptions de messages. C'est typiquement la cas d'une bibliothèque de *sockets*.

Ces deux catégories d'exécutif correspondent respectivement à la première et à la deuxième phase du cycle de conception que nous avons proposé en section IV-3, page 59.

Un intergiciel d'exécution de haut niveau permet de prendre directement en charge les modèles de distribution décrits au niveau des *threads* AADL. Il peut donc être utilisé pour la première phase du processus de conception.

Le cas d'un intergiciel d'exécution bas niveau correspond en fait à l'absence d'intergiciel de communication. Dans ce dernier cas, les patrons architecturaux pour la répartition que nous avons décrits à la section IV-5 ne peuvent donc pas être pris directement en charge ; il est nécessaire de transformer la modélisation AADL afin de faire apparaître les services de communication au niveau de l'application, c'est-à-dire au niveau de l'exécutif AADL. Il s'agit alors de la seconde phase du processus de conception.

Dans les deux situations, les éléments AADL qui décrivent l'enveloppe applicative demeurent les mêmes, et conservent un rôle purement passif dans l'architecture : ils sont appelés par l'exécutif, qui leur transmet les données.

VI-2 Première phase : utilisation d'un intergiciel de haut niveau

Dans cette section nous décrivons la conception d'un exécutif basé sur un intergiciel de haut niveau. Cette situation correspond à la première phase du cycle de conception que nous avons décrit en section IV-3, page 59.

L'intergiciel sur lequel se base un exécutif de haut niveau doit être capable de traiter les types de données manipulés par l'application. Il doit également pouvoir prendre en charge les différents paradigmes de répartition exprimés dans la description AADL. Par ailleurs, l'intergiciel utilisé doit

pouvoir être configuré afin de fournir les fonctions nécessaires en terme de nombre de *threads* système tout en garantissant les performances requises par l'architecture, énoncées comme propriétés des *threads* AADL.

VI-2.1 Organisation de l'interface avec l'intergiciel

L'interface avec l'intergiciel doit pouvoir contrôler l'enveloppe applicative générée à partir des sous-programmes AADL (cf. chapitre V) selon les deux fonctionnalités que nous avons identifiées en section VI-1.1. Elle doit donc remplir deux fonctions :

- prendre en charge les requêtes entrantes à destination des entités matérialisant les *threads* AADL ;
- exécuter ces entités en fonction des politiques de déclenchement spécifiées dans la description AADL.

La réception des requêtes correspondant aux ports d'événement/donnée et aux sous-programmes invoqués à distance doit déclencher l'exécution de l'enveloppe applicative. Les données associées aux requêtes correspondant aux ports de donnée doivent être mémorisées en attendant les conditions de déclenchement du *thread* AADL – arrivée d'une requête associée à un port d'événement/donnée ou échéance de la période de déclenchement du *thread*. Il est donc nécessaire de séparer le traitement des données associées aux requêtes et le mécanisme de déclenchement de l'application. La figure VI.1 illustre l'architecture que nous avons définie pour la couche d'interface.

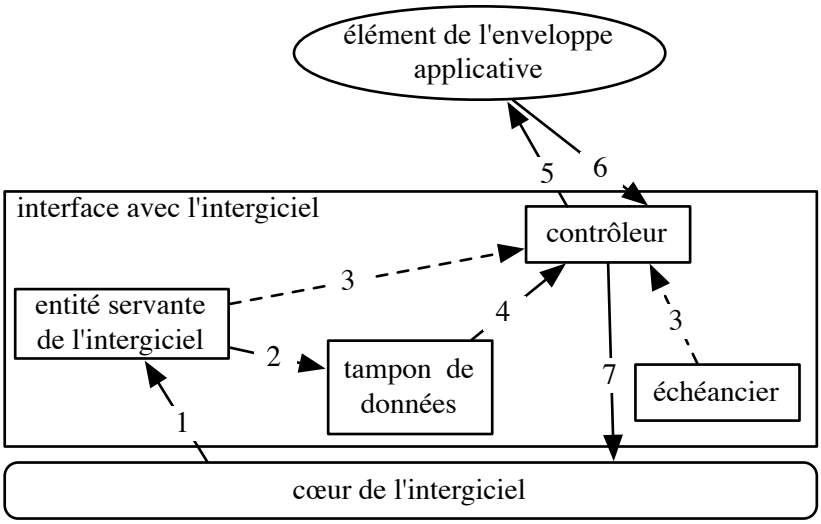


FIG. VI.1 – Organisation de l'interface avec l'intergiciel

Définition VI.3 (Prise en charge des *threads* AADL)
Chaque *thread* AADL correspond à une entité servante dans l'intergiciel.

Chaque *thread* AADL correspond à une instance de l'architecture représentée en figure VI.1. L'entité servante correspond à une entité de base gérée par l'intergiciel, par exemple un *servant* si l'intergiciel est un ORB CORBA ; elle est enregistrée auprès de l'intergiciel, et reçoit toutes les requêtes correspondant aux éléments d'interface du *thread* AADL qu'elle représente. Chaque

thread AADL donne donc lieu à la création d'une référence dans le cœur de l'intergiciel (qui correspond par exemple à une IOR dans le cas d'un ORB).

Lorsque l'intergiciel reçoit une requête destinée à cette référence, il la transmet à l'entité servante correspondante (1). Les données de la requête sont extraites et transmises à un tampon mémoire (2). Le tampon mémoire gère les données associées à chaque élément d'interface et prend en charge les politiques associées aux éventuelles files d'attente – pour les ports d'événement/donnée ou les sous-programmes d'interface.

Si la requête correspond à un port d'événement/donnée ou à un sous-programme fourni en interface, le servant appelle le contrôleur de l'application (3) ; sinon le traitement s'arrête. Dans le cas d'un *thread* AADL périodique, un échéancier appelle le contrôleur (3).

Lorsque le contrôleur est appelé, il récupère toutes les données des ports, ou les données associées à l'appel du sous-programme fourni en interface (4). L'enveloppe applicative AADL qui correspond à la séquence d'appel du *thread* AADL est ensuite appelée (5) par le contrôleur avec les données correspondantes.

Les éventuelles données retournée (6) par la séquence d'appel – c'est-à-dire les paramètres de sortie du sous-programme fourni en interface ou les ports de sortie du *thread* AADL, selon le cas – sont ensuite récupérées (7). Dans le cas d'un sous-programme fourni en interface, les données sont intégrées dans la réponse à la requête et envoyées vers le client initial. Dans le cas de ports de données ou événement/donnée, elles sont intégrées dans de nouvelles requêtes et envoyées aux entités AADL destinataires.

VI-2.2 Application à l'intergiciel PolyORB

PolyORB permet de produire un intergiciel adapté prenant en compte différents modèles de répartition et offrant une grande latitude de configuration [Vergnaud et al., 2004]. De récents travaux ont consisté à rationaliser son architecture afin de faciliter son intégration dans des systèmes répartis temps-réel [Hugues, 2005]. Il constitue par conséquent une bonne solution pour la mise en place d'un intergiciel d'exécution pour une application AADL.

VI-2.2.1 Organisation de PolyORB

PolyORB est une implantation de l'architecture d'intergiciel dite « schizophrène » [Pautet, 2001]. Cette architecture est structurée en trois couches : neutre, applicative et protocolaire.

La couche neutre constitue un cœur d'intergiciel mettant en place des mécanismes génériques pour le traitement des données. Elle fournit un ensemble de services de communication canoniques, qui ne sont pas rattachés à un modèle de répartition particulier. La couche applicative effectue la traduction des requêtes entre la sémantique de l'application et les représentations neutres correspondantes. La couche protocolaire implante les fonctions relatives aux communications avec les autres nœuds. L'organisation des trois couches est représentée sur la figure VI.2.

Plusieurs personnalités applicatives ou protocolaires peuvent être utilisées simultanément au sein d'une instance donnée d'un intergiciel schizophrène. Ces différentes personnalités interagissent alors par l'intermédiaire de la couche neutre. Il est ainsi possible d'utiliser plusieurs personnalités protocolaires afin d'insérer le nœud applicatif au sein d'un système réparti constitué de nœuds hétérogènes, ou de déployer sur un même nœud plusieurs entités applicatives conçues pour des spécifications d'intergiciel différentes.

La structure de la couche neutre forme un intergiciel neutre configurable, basé sur un ensemble de services fondamentaux [Quinot, 2003] dont la combinaison permet de traiter les communica-

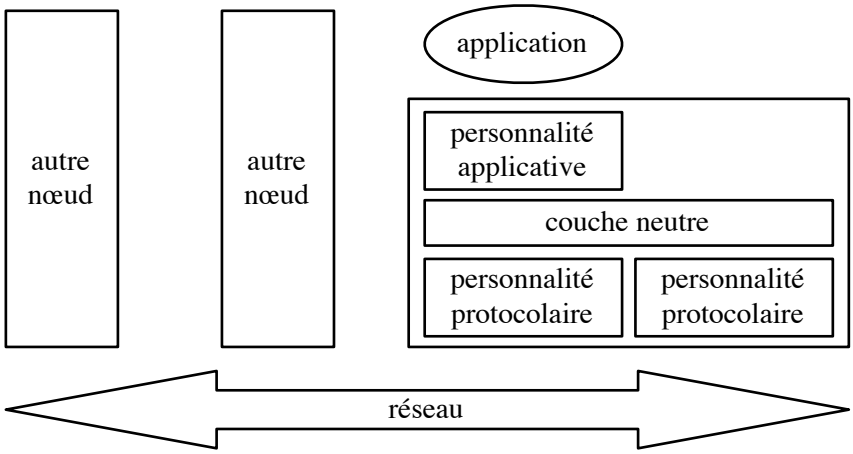


FIG. VI.2 – Organisation d'une instance de PolyORB

tions sur chaque nœud local de l'application. Ces services sont les suivants :

le service d'adressage enregistre les entités servants et crée les références vers ces entités pour qu'elles puissent être appelées par la suite.

le service de transport crée les points d'accès pour les requêtes de connexion entrantes.

le service de liaison associe un subrogé (c-à-d une entité de liaison) à une entité répartie. Il gère la qualité de service associés aux connexions.

le service de représentation prend en charge la conversion des structures de données entre la représentation neutre et celle des applications ou des protocoles.

le service de protocole prend en charge l'émission et la réception des requêtes. Il contrôle notamment le service de transport.

le service d'activation reçoit une référence à une entité servante and lui associe l'entité correspondante.

le service d'exécution alloue les ressources nécessaires à l'exécution de l'entité servante.

Ces différents services sont coordonnés par un composant central, le μBroker. Le μBroker prend en charge la gestion des *threads* systèmes et l'ordonnancement des requêtes.

Par la sélection et l'assemblage des implantations de ces différents services, et par la sélection des politiques de gestion du μBroker, il est possible d'adapter les mécanismes de traitement des requêtes, indépendamment des personnalités utilisées.

VI-2.2.2 Construction d'une personnalité applicative pour AADL

La couche applicative constitue l'interface entre l'application AADL en elle-même et la partie de l'exécutif décrite en AADL. Elle doit piloter les composants de l'application : elle reçoit les requêtes, en extrait les données, exécute les composants applicatifs correspondants et récupère les résultats.

Dans le cadre d'une utilisation de PolyORB comme intergiciel de haut niveau pour notre exécutif AADL, la couche d'interface entre l'enveloppe applicative et l'intergiciel lui-même est matérialisée par une personnalité applicative. À chaque *thread* AADL nous associons un servant

PolyORB. Ce servant reçoit toutes les requêtes correspondant aux éléments d'interface du *thread* AADL.

VI-2.2.3 Éléments de configuration de l'intergiciel

L'architecture de PolyORB permet une grande flexibilité dans la configuration, que ce soit par le choix de personnalités adaptés ou la configuration des éléments de la couche neutre [Hugues, 2006]. Certains paramètres de configuration peuvent être déduits de l'assemblage des composants, tandis que d'autres peuvent correspondre à des propriétés AADL. Nous exposons ici comment transposer les descriptions AADL en éléments de configuration pour l'exécutif basé sur PolyORB.

Politique de gestion des threads *systèmes*

Une application stricte de la sémantique des *thread* AADL correspondrait à associer un *thread* système – c'est-à-dire une tâche de PolyORB – à chaque servant. Une telle gestion des tâches n'est pas implantée dans des intergiciels comme PolyORB, qui sont orientés sur le traitement des requêtes. Cela conduit à une interprétation différente des *threads* AADL.

Dans une implantation basée sur un intergiciel de haut niveau tel que PolyORB, les différents *threads* AADL représentent un ensemble de *threads* du système, qui sont affectés au traitement des différentes requêtes. Parmi les différentes politiques de gestion des tâches offertes par PolyORB, la plus adaptée est la définition d'un ensemble de *threads* (*thread pool*) dont le cardinal est défini par le nombre de *threads* AADL instanciés dans chaque processus.

Ces *threads* sont indifférenciés, dans la mesure où il doivent pouvoir être affectés au traitement des requêtes correspondant aux éléments d'interface de tous les *threads* AADL du processus considéré ; ils peuvent prendre en charge l'écoute sur le service de transport et le traitement des requêtes. Cela correspond à l'application d'une politique de contrôle des tâches équitable. PolyORB propose pour cela deux politiques, *basic* et *leader/follower*.

Gestion des priorités

AADL permet de spécifier une priorité dans le traitement des différents *threads*, au moyen de la propriété `Priority` définie dans l'ensemble `Language_Support`. Cette notion peut se traduire par un mécanisme de priorités associées aux servants de l'intergiciel, telle que définie dans l'implantation RT-CORBA de PolyORB.

Sélection des personnalités protocolaires

Il est nécessaire de sélectionner les protocoles à utiliser pour les communications inter-processus AADL. Les protocoles que nous considérons ici s'appliquent aux flux de données entre les processus (relevant des couches session, présentation et application du modèle OSI [Tanenbaum, 2003]), qui sont représentés par les connexions AADL. Les couches de protocole plus basses (couches liaison, réseau et transport dans le modèle OSI) concernent les bus AADL, qui modélisent des réseaux physiques.

Deux stratégies peuvent être suivies pour prendre en charge la sélection des protocoles du niveau applicatif :

- la mise en place de politiques de sélection de protocole en fonction des constructions architecturales.
- la définition de propriétés AADL pour la spécification explicite des protocoles à utiliser ;

Ces deux approches peuvent être combinées. La description architecturale permet de sélectionner les personnalités par défaut, et les propriétés AADL permettent de « forcer » l'utilisation d'une personnalité particulière. Les principales personnalités de PolyORB sont les implantations du protocole GIOP de CORBA :

- IIOP [1] correspond au protocole de communication point à point fiable ;
- DIOP [2] correspond à une communication point à point non fiable, utilisable pour les requêtes sans réponse ;
- MIOP [3] correspond à une diffusion non fiable.

L'utilisation de protocoles de communication non fiables ne peut se faire qu'avec la garantie que le support de communication est lui-même fiable. C'est typiquement le cas des communication entre deux processus s'exécutant sur le même processeur. Le protocole IIOP devrait être utilisé pour les autres cas.

La spécification explicite du protocole à utiliser doit se faire par l'utilisation d'une propriété d'énumération s'appliquant aux connexions AADL. Le standard définit pour cela la propriété `Connection_Protocol`, dont les valeurs possibles doivent être définies par l'utilisateur dans l'ensemble `AADL_Project`.

Dimension des files d'attente

La modélisation des files d'attente de données dans une description AADL apparaît en deux endroits :

- sur les éléments d'interface des processus ;
- sur les éléments d'interface des *threads*.

La propriété AADL standard `Queue_Size` permet de spécifier une taille de file d'attente pour ces entités.

La taille des files d'attente associée aux éléments d'interface des processus – ports d'événement/donnée et sous-programmes – se traduit par la dimension des tampons de mémoire mis en place dans le service de transport de PolyORB. Ces tampons de mémoire permettent de stocker les données entrantes avant leur traitement par les servants. Les files d'attentes associées aux *threads* AADL correspondent aux dimensions des tampons de la personnalité applicative.

Optimisation des temps d'exécution

Dans la section V-2.2 nous avons défini les types de données AADL manipulables dans le cadre de nos travaux. La plupart de ces types de données ont une taille déterminée. Par ailleurs, AADL ne permet pas la création dynamique d'entités. Tous les caractéristiques des requêtes sont donc connues lors de la construction de l'intergiciel.

Ces informations peuvent être exploitées pour optimiser l'implantation de certains services de l'intergiciel. Il est ainsi possible de définir précisément la taille des tampons mémoire utilisés dans les personnalités protocolaires, au niveau du service de représentation. L'indexation des entités référencées au niveau du service d'activation peut prendre la forme d'un tableau statique, dont les temps d'accès sont fixes.

De tels travaux d'optimisation sont déjà réalisés dans le cadre du compilateur IDL IAC [Zalila et al., 2006] associé à PolyORB. Ils peuvent être transposés dans le cadre d'AADL.

[1] IIOP : Internet InterORB Protocol
[2] IIOP : Datagram InterORB Protocol
[3] IIOP : Multicast InterORB Protocol

VI-3 Seconde phase : utilisation d'un intergiciel de bas niveau

Dans la section précédente nous avons décrit la mise en place d'un exécutif de haut niveau pour le première phase de notre processus de conception. Cet exécutif doit reposer sur un intergiciel de communication configurable et capable de prendre directement en charge les différents modèles de répartition décrits en AADL. Cette approche permet de générer facilement une application répartie tout en permettant la configuration de l'exécutif.

La structure d'un tel intergiciel de communication est en général complexe ; l'analyse de ses dimensions spatiales et temporelles peut donc se révéler délicate. Nous avons donc intérêt à réduire sa taille au strict minimum. Cela constitue la seconde étape de notre processus de conception, que nous avons décrite en section IV-3, page 59.

Dans cette étape, nous considérons que la plupart des éléments de l'exécutif peuvent être intégrés à l'application, dont ils constituent alors la partie basse. Les fonctionnalités de communication fournies par l'exécutif AADL sont alors être elles-mêmes modélisées en AADL. Ainsi, l'exécutif se retrouve être réduit à un ensemble minimal de composants qui gèrent l'ordonnancement des *threads* système et prennent en charge les opérations de communication de base (telle que la gestion des *sockets* si nous utilisons un réseau de type Internet).

VI-3.1 Expansion des *threads* AADL

Nous utilisons cette fois une bibliothèque de communication très primitive ; elle ne peut notamment pas gérer les appels de sous-programme distant, ni même les messages avec file d'attente. La mise en place de tous ces mécanismes seront donc décrits par des composants AADL.

Il en résulte que la description des *threads* AADL telle qu'elle a été faite dans la première phase de modélisation ne peut pas être conservée ; les différents éléments d'interface correspondent en effet à des concepts de trop haut niveau, qui ne sont pas pris en charge par un intergiciel de bas niveau.

L'utilisation d'un exécutif bas niveau implique donc la transformation des *threads* AADL de l'application ; cette transformation vise à supprimer les éléments d'interface des *threads* et à faire apparaître les composants AADL correspondant à la modélisation de l'intergiciel de communication. Les *threads* AADL ainsi transformés symbolisent l'exécutif de bas niveau, c'est-à-dire l'ordonnanceur. Ce processus de transformation est illustré sur la figure VI.3.

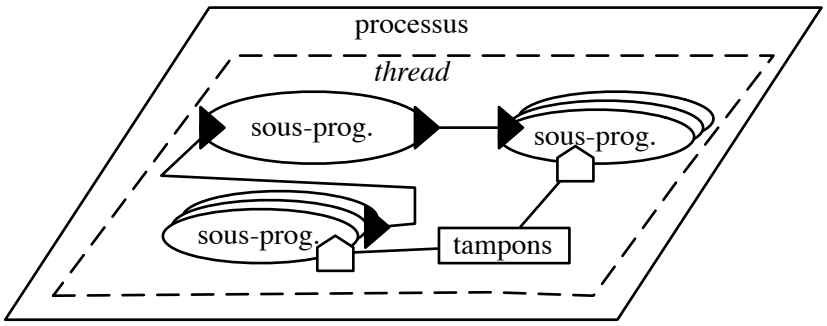


FIG. VI.3 – Expansion du *threads* AADL pour décrire l'intergiciel de communication en AADL

Le fait de supprimer les éléments d'interface des *threads* AADL implique que les communications entre les nœuds de l'application ne sont plus modélisées. Par conséquent, nous n'avons plus

de visibilité globale à cette étape du développement ; nous manipulons désormais chaque nœud séparément.

VI-3.2 Transposition de l'architecture schizophrène en AADL

Dans la section VI-2, nous avons montré que la mise en place d'un exécutif de haut niveau nécessitait l'utilisation d'un intergiciel capable de prendre en charge les différents modèles de répartition descriptibles en AADL.

Un exécutif modélisé en AADL doit pouvoir prendre en charge les différents modèles de distribution de l'application, de la même façon que dans le cas d'un intergiciel de haut niveau. Le fait de modéliser toutes les fonctions de communication en AADL doit en outre permettre de concevoir un intergiciel exactement dimensionné en fonction des besoins exprimés dans la description AADL de l'application, tant du point de vue temps-réel et embarqué qu'au niveau de l'interopérabilité entre les nœuds de l'application. L'exécutif doit donc être constitué d'un ensemble de composant combinés afin de construire un environnement d'exécution approprié.

L'architecture schizophrène offre de telles propriétés, qui ont été appliquée pour la conception de PolyORB : elle décrit une armature canonique pour construire facilement un intergiciel pouvant prendre en charge les différents modèles de répartition tout en permettant la configuration en fonction des besoins spécifiques de l'application.

La phase d'expansion des *threads* AADL de la description de haut niveau consiste donc à faire apparaître une modélisation en AADL de l'architecture schizophrène.

L'approche suivie dans PolyORB repose sur des mécanismes dynamiques permettant l'inscription des entités applicatives (les *servants*) pendant l'exécution de l'intergiciel et l'assemblage des différents éléments de l'architecture schizophrène. AADL ne permet pas de décrire des constructions dynamiques : notamment, la notion de pointeur – ou de référence – n'existe pas, ce qui empêche par exemple de modéliser l'inscription d'un composant auprès d'un dictionnaire. Nous ne pouvons donc pas appliquer strictement la même démarche pour la description en AADL des services de communication que celle suivie pour la conception de PolyORB. La modélisation en AADL des couches basses de l'application n'est donc pas une modélisation de PolyORB, mais l'implantation d'un intergiciel de communication spécifique, dont la structure est basée sur l'architecture schizophrène.

La définition exacte de cette structure dépend des paramètres de l'application, qui vont être retranscrits dans la définition des composants de l'intergiciel. Il est néanmoins possible de fournir une définition de haut niveau d'un ensemble de composants. Ces composants « abstraits » devront être étendus pour former une description architecturale concrète.

Dans les sections suivantes nous décrivons ces composants et l'armature qu'ils forment. Afin de refléter l'organisation de l'architecture schizophrène, nous regroupons les composants dans différents paquetages AADL. Le paquetage principal regroupe les déclarations de composants utilisées par la plupart des services, notamment la donnée modélisant les requêtes (cf. listing VI.1). Dans le cadre de nos travaux, nous avons désigné l'ensemble des composants AADL devant être générés par le nom « Ipao [4] »

```
1 package Ipao
2 public
3   data Request
4   end Request;
```

[4] Ipao est l'acronyme de « instanciation de PolyORB en AADL pour Ocarina ». Ocarina est l'outil AADL que nous avons conçu dans le cadre de nos travaux ; nous le présentons au chapitre VIII.

```
5 end Ipao;
```

Listing VI.1 – Modélisation des requêtes

Nous ne détaillons pas la structure exacte du composant de donnée modélisant la structure des requêtes. Selon la façon dont sont implantés les différents composants de l'intergiciel, la modélisation des requêtes peut être détaillée en AADL ou au contraire demeurer opaque.

VI-3.3 Modélisation en AADL de l'interface avec l'intergiciel

La modélisation de la couche applicative en AADL reprend les mêmes principes que pour la génération de code pour PolyORB, que nous avons décrite en VI-2.1. La modélisation en AADL de la couche applicative consiste en groupes de trois éléments, attachés à chaque *thread* AADL :

- un servant ;
- un tampon de mémoire ;
- un contrôleur.

Le servant est un sous-programme qui sera appelé par les composants AADL de la couche neutre. Le tampon mémoire est modélisé par un composant de donnée AADL fournissant en interface des sous-programmes d'accès. Ces sous-programmes contrôlent les tailles de file d'attente (p.ex. pour les ports d'événement/donnée). Le contrôleur est un sous-programme qui peut être appelé soit par le sous-programme servant dans le cas d'une requête déclenchante ou par un échéancier dans le cas d'un *thread* périodique. Le listing VI.2 donne une description de haut niveau des composants de l'interface avec le cœur de l'intergiciel.

```
1  package Ipao::Interface
2  public
3    data Buffer
4    end Buffer;
5
6    subprogram Push_Data
7    features
8      Parameters : requires data acces Buffer;
9    end Push_Data;
10
11   subprogram Get_Data
12   features
13     Parameters : requires data acces Buffer;
14   end Get_Data;
15
16   subprogram Servant
17   features
18     Request : in parameter Ipao::Request;
19   end Servant;
20
21   subprogram Controler
22   end Controler;
23 end Ipao::Interface;
```

Listing VI.2 – Modélisation de l'interface avec le cœur de l'intergiciel

Un sous-programme `Servant` doit être défini pour chaque *thread* AADL. Il en est de même pour le sous-programme `Controler`. Les sous-programmes `Push_Data` et `Get_Data` doivent être

étendus pour chaque élément d'interface de chaque *thread* AADL ; la donnée `Parameter_Buffer` contient toutes les files d'attente des différents paramètres.

VI-3.4 Modélisation du cœur de l'intergiciel en AADL

Une personnalité applicative constitue une implantation de certains des services en fonction des paradigmes caractéristiques du modèle d'application considéré ; il en va de même pour une personnalité protocolaire.

Les travaux autour de PolyORB [Hugues, 2005] ont permis de rationaliser les services de l'architecture schizophrène sous forme d'entités simples [Hugues et al., 2004]. Nous détaillons ici la structure des différents services.

La modélisation en AADL des différents services est spécialisée vis-à-vis de l'application à prendre en charge ; il est impossible d'en fournir une modélisation complète. Nous pouvons néanmoins définir les principes de cette modélisation. Nous présentons ici l'organisation de la description AADL et une définition de haut niveau des composants modélisant les services.

VI-3.4.1 Adressage

La fonction d'adressage attribue à chaque entité de l'application une référence unique la désignant sans ambiguïté. Elle combine plusieurs informations pour construire une référence : l'identifiant local d'un objet, l'ensemble des identifiants représentant les différents canaux de communication permettant de le joindre (un profil de liaison), des paramètres de qualité de service, etc.

Dans le cadre de la conception d'un intergiciel de type CORBA, la référence construite par le service d'adressage peut être transformée sous forme de chaîne de caractères (IOR CORBA, corbaloc URI, etc.), puis échangée avec les autres nœuds – typiquement, en notant les références dans un fichier, etc.

L'implantation de ce service au sein d'une architecture dynamique comme celle de PolyORB peut se révéler relativement sophistiquée, en ayant recours à des mécanismes de *call-backs* et d'exclusion mutuelle pour enregistrer les informations issues des différents modules de l'intergiciel.

Dans le cadre d'une modélisation en AADL, le service d'adressage est en fait déjà exprimé par la modélisation AADL, par les *threads* AADL et leurs connexions. Par conséquent ce service n'est traduit par aucun composant actif dans la modélisation, puisque toutes les références peuvent être pré-calculées lors de la construction de l'intergiciel. Les références aux entités doivent en revanche être stockées en mémoire. La modélisation AADL du service d'adressage se traduit donc par un composant de donnée. Le listing VI.3 définit une structure de donnée abstraite matérialisant le service d'adressage.

```
1 package Ipao::Addressing
2 public
3   data References
4   end References;
5 end Ipao::Addressing;
```

Listing VI.3 – Modélisation du service d'adressage

VI-3.4.2 Liaison

Le rôle du service de liaison est de créer une structure locale, l'objet de liaison, qui représente l'entité désignée par une référence, et de mettre en place les mécanismes permettant d'interagir

avec elle. Ainsi, un objet de liaison est un subrogé de l'entité réelle, qui prend en charge le transfert des invocations.

Dans le cadre de PolyORB, lorsque la référence renvoie à une entité locale au nœud, l'objet de liaison est l'entité encapsulant le code de l'utilisateur. Lorsque la référence pointe vers un objet distant, l'objet de liaison regroupe l'ensemble des mécanismes permettant le dialogue avec l'objet distant : pile protocolaire, fonction de représentation, connexion, etc. La fonction de liaison nécessite alors plusieurs fabriques d'objets, et une fonction de choix sélectionner la fabrique appropriée. Cette fonction de choix utilise des informations de configuration et de contexte pour choisir un mécanisme.

Dans le cadre de la modélisation en AADL de l'intergiciel, le service de liaison permet d'orienter les requêtes vers la couche applicative ou protocolaire adéquate, selon la référence transportée par la requête. Afin de permettre la mise en place de politiques d'ordonnancement des requêtes, il est nécessaire de découpler la réception des requêtes par le service de liaison de leur traitement effectif. Pour cela, nous modélisons le service de liaison par trois composants principaux :

- un composant de donnée représentant une file d'attente ;
- un sous-programme prenant en charge l'insertion des requêtes dans la file d'attente ;
- un sous-programme extrayant les requêtes de la file pour les traiter.

Les dimensions de la file d'attente sont fixées à la construction de l'intergiciel, en fonction du nombre d'entités à prendre en charge et de la politique d'ordonnancement. Le composant de donnée AADL correspondant peut donc avoir la sémantique d'un tableau. Ce composant est partagé entre les différents *threads* AADL du nœud local ; le service de liaison consiste en effet en un aiguillage commun entre les différentes entités, c'est-à-dire entre les différents *threads* AADL.

Le sous-programme d'insertion prend une requête en paramètre et nécessite un accès à la file d'attente afin de l'y insérer. Son implantation ne change pas d'une configuration d'intergiciel à une autre.

L'implantation du sous-programme de traitement dépend en revanche de la configuration de l'intergiciel. Le sous-programme possède autant de séquences d'appel que de personnalités définies dans l'intergiciel. Il doit avoir un accès aux composants de donnée du service d'adressage, afin de déterminer si la requête s'adresse à une entité locale ou non. Le code source de son implantation extrait l'identité de l'entité destinataire de la requête, décide si l'entité est locale ou distante et exécute la séquence d'appel correspondante.

Pour les requêtes qui sont orientées vers une personnalité protocolaire, le service de liaison doit également fournir un modèle AADL du subrogé correspondant. Les subrogés sont des sous-programmes qui correspondent à chaque élément d'interface de l'entité AADL distante (c'est-à-dire du *thread* distant). Le subrogé coordonne les services de transport et de protocole.

Le listing VI.4 représente la définition des composants constituant le service de liaison. Nous n'en présentons que les interfaces ; la définition exacte de ces composants doit être construite par extension de ces composants de base, afin de préserver l'implantation du tampon et de composant de donnée modélisant les requêtes.

```
1 package Ipao::Bind
2 public
3   data Buffer
4   properties
5     Language_Support::Data_Format => Array;
6     Language_Support::Data_Type => classifier (Ipao::Request);
7   end Buffer;
8
9   subprogram Add_Request
```

```
10   features
11     Request : in parameter Ipao::Request;
12     Buffer : requires data access Buffer;
13   end Add_Request;
14
15   subprogram Bind_Request
16   features
17     Requests : requires data access Buffer;
18   end Bind_Request;
19 end Ipao::Bind;
```

Listing VI.4 – Modélisation du service de liaison

VI-3.4.3 Représentation

La fonction de représentation convertit une donnée d'un type du modèle de données de la couche neutre en un message conforme à la représentation imposée par la personnalité utilisée (protocolaire ou applicative). Il s'agit donc d'un ensemble de fonctions (au sens mathématique) assimilables à des filtres : une donnée est transformée en une autre suivant un processus déterministe.

Dans le cadre d'une modélisation en AADL, le service de représentation définit les composants de donnée qui peuvent être associés aux paramètres des sous-programmes de l'intergiciel, et définit également les sous-programmes de conversion.

Ces sous-programmes de conversion possèdent un paramètre d'entrée et un autre de sortie. Afin de rationaliser la description AADL, ils ont une implantation opaque.

Le listing VI.5 illustre la signature générale des sous-programmes associés au service de représentation. Nous définissons deux sous-programmes génériques qui devront être étendus pour préciser le type de donnée à traduire.

```
1 package Ipao::Representation
2 public
3   subprogram Generic_To_Neutral
4   features
5     Generic : in parameter;
6     Neutral : out parameter;
7   end Generic_To_Neutral;
8
9   subprogram Neutral_To_Generic
10  features
11    Neutral : in parameter;
12    Generic : out parameter;
13  end Neutral_To_Generic;
14 end Ipao::Representation;
```

Listing VI.5 – Modélisation du service de représentation

VI-3.4.4 Protocole

La fonction de protocole coordonne le déroulement d'une invocation distante : à partir d'une demande d'invocation, un message est préparé puis émis. L'intergiciel client est mis en attente

d'une réponse si nécessaire ; lorsque celle-ci est reçue, elle est retraduite sous forme neutre et signalée à la couche neutre, qui la transmet à la personnalité applicative.

À la réception d'une demande d'invocation, l'identité de l'entité cible et de la requête concernées sont extraites, ainsi que les paramètres associés. Une requête sous forme neutre est construite et confiée à la couche neutre, pour que celle-ci l'envoie vers la personnalité applicative locale ou la retransmette vers un autre nœud lorsqu'une passerelle entre deux protocoles est établie. Une fois la requête traitée, la couche neutre signale à la couche protocolaire que la réponse éventuelle peut être retournée à l'intergiciel client.

Le service de protocole s'apparente à un automate. La modélisation AADL correspondante est donc un sous-programme appelant les sous-programmes d'autres services, tels que le service de représentation et le service de transport. Il est constitué de séquences d'appels vers les différents composants AADL pilotés par le protocole ; l'implantation en code source remplit les fonctions de coordination des différents appels. Son interface est vide (cf. listing VI.6).

```
1 package Ipao::Protocol
2 public
3   subprogram Protocol
4   end Protocol;
5 end Ipao::Protocol;
```

Listing VI.6 – Modélisation du service de protocole

VI-3.4.5 Transport

La fonction de transport s'occupe de transférer une information d'un nœud à un autre. Les points d'accès et de terminaison du service de transport représentent des entités permettant à l'intergiciel d'interagir avec les autres nœuds de l'application. Les instants où des événements se produisent sur ces entités et requièrent l'attention de l'intergiciel ne sont pas connus a priori. Ces objets constituent donc des sources d'événements asynchrones qui doivent être scrutées par l'intergiciel pour que ces événements externes soient pris en compte.

Ce service doit prendre appui sur une bibliothèque existante, telles que les primitives d'entrées/sorties du système d'exploitation. Il s'agit typiquement de deux opérations `send` et `receive`, auxquelles s'ajoutent des mécanismes de configuration.

Dans le cadre de la modélisation en AADL, ce service est donc principalement constitué des deux sous-programmes AADL `send` et `receive`, comme illustré sur le listing VI.7. Ils matérialisent la bibliothèque de communication qui constitue l'intergiciel de communication de bas niveau.

```
1  package Ipao::Transport
2  public
3    subprogram Send
4    features
5      Communication : in parameter;
6    end Send;
7
8    subprogram Receive
9    features
10     Communication : out parameter;
11   end Receive;
12 end Ipao:Transport;
```

Listing VI.7 – Modélisation du service de transport

VI-3.4.6 Activation

Le service d'activation prend en charge l'association entre une entité et la requête demandant à interagir avec celle-ci. Le service a ainsi pour but de retrouver l'entité correspondant parmi celles enregistrées dans l'intergiciel.

Dans le cadre de PolyORB, cette association peut être réalisée suivant plusieurs modes (statiques, dynamiques) et avec plusieurs niveaux de configuration (par exemple grâce aux multiples politiques du POA de CORBA). Ce service peut donc être ramené à un *dictionnaire* qui retourne une entité correspondant à certains critères.

Dans la modélisation en AADL, le service d'activation établit une relation statique entre les références transportées par les requêtes et les entités servantes. Afin de permettre la mise en place de politiques de gestion de priorité entre les requêtes, il est nécessaire de découpler la réception des requêtes par le service d'activation et l'appel effectif aux entités associées à ces requêtes. Le service d'activation est donc formé de trois composants AADL, de la même façon que pour le service de liaison :

- un composant de donnée qui représente la file d'attente des requêtes ;
- un sous-programme permettant d'insérer les requêtes dans la file d'attente ;
- un sous-programme permettant d'extraire les requêtes de la file d'attente et d'appeler l'entité correspondante dans la couche applicative.

Les deux sous-programmes doivent avoir un accès au composant de donnée. Les dimensions du composant de données sont fixées ; elles font partie des paramètres de dimensionnement de l'intergiciel. Cette file d'attente peut donc être représentée par un tableau. L'implantation du sous-programme d'insertion ne change pas d'une configuration à une autre de l'intergiciel ; elle consiste à récupérer la requête fournie en paramètre et à la stocker dans la file d'attente.

En revanche, le sous-programme d'activation dépend de la configuration de l'intergiciel. Il s'agit d'un sous-programme AADL contenant différentes séquences d'appels, chacune correspondant à une requête – c'est-à-dire un élément d'interface du *thread* AADL. La mise en place de ce sous-programme est donc fonction de deux aspects :

- les interfaces des *threads* AADL ;
- la politique de gestion de requêtes.

Les interfaces des *threads* conditionnent la définition des différentes séquences d'appels du sous-programme ; chaque séquence est nommé d'après l'élément d'interface qu'elle représente, et porte donc le nom de la requête à laquelle elle correspond.

L'implantation en code source du sous-programme reprend les fonctionnalités de l'adaptateur d'objet tel qu'il est défini dans l'architecture schizophrène. Elle extrait une requête de la file d'attente en fonction de la politique d'activation définie et exécute la séquence d'appel correspondante.

Le listing VI.8 représente la déclaration des composants de base du service d'activation ; ils doivent être étendus en fonction de la configuration à modéliser.

```
1  package Ipao::Activation
2  public
3    data Buffer
4    properties
5      Language_Support::Data_Format => Array;
6      Language_Support::Data_Type => classifier (Ipao::Request);
7    end Buffer;
8
9    subprogram Add_Request
10   features
```

```
11     Request : in parameter Ipao::Request;
12     Buffer : requires data access Buffer;
13   end Add_Request;
14
15   subprogram Exec_Request
16   features
17     Buffer : requires data access Buffer;
18   end Exec_Request;
19 end Ipao::Activation;
```

Listing VI.8 – Modélisation du service d'activation

VI-3.4.7 Exécution

Dans l'architecture schizophrène, le service d'exécution s'occupe d'attribuer une tâche (*thread* système) au sous-programme applicatif approprié. La tâche utilisée peut être soit une tâche prêtée temporairement à l'intergiciel par l'application, soit une tâche propre à l'intergiciel.

Dans le cas d'une modélisation en AADL, toutes les ressources d'exécutions sont connues au moment de générer l'intergiciel ; le service d'exécution n'a donc pas de traduction à proprement parler en terme de composant AADL, et nous pouvons considérer qu'il est confondu avec le service d'activation.

VI-3.4.8 Le μBroker

Les services sont coordonnés par un composant central [Hugues, 2005] appelé μBroker, qui gère les *threads* supportant l'exécution des services de communication. L'utilisation d'un intergiciel de bas niveau revient donc à modéliser les services en AADL tandis que l'exécutif en lui-même correspond à l'implantation du μBroker.

Le μBroker coordonne l'exécution des différents services. Il s'agit donc essentiellement d'une machine à état. En conséquence, il est modélisé par un sous-programme pouvant appeler les différents sous-programmes des services schizophrènes. Ce sous-programme a accès à un composant de donnée AADL qui stocke l'état du μBroker.

Les services à coordonner dépendent de la nature des requêtes à prendre en charge. Deux cas doivent être considérés : une requête venant d'une entité locale ou du réseau. La réception d'une requête à partir du réseau s'effectue en deux phases : d'abord la réception de la requête et ensuite son traitement par le service de liaison. Pour une requête émise par une entité locale, il n'y a qu'une seule phase, qui est l'invocation du service de liaison. Les requêtes issues d'une entité locale sont modélisées par des séquences d'appel dont l'origine est située dans les entités applicatives.

L'implantation du sous-programme du μBroker décrit les différentes séquences d'appels possibles. Le listing VI.9 illustre une implantation générique du μBroker pour un nœud applicatif de serveur, n'ayant qu'une seule personnalité protocolaire.

```
1 package Ipao:Mubroker
2 public
3   data Mubroker_State
4   end Mubroker_State;
5
6   subprogram Mubroker
7   features
```

```
8     Protocol : requires subprogram access Ipao::Protocol::
          Protocol;
9     Queue_Request : requires subprogram access Ipao::Bind::
          Add_Request;
10    Bind_Request : requires subprogram access Ipao::Bind::
          Bind_Request;
11    Add_Request : requires subprogram access Ipao::Activation::
          Add_Request;
12    Activate_Request : requires subprogram access Ipao::
          Activation::Add_Request;
13    State_Machine : requires data access Mubroker_State;
14  end Mubroker;
15
16  subprogram implementation Mubroker.Generic
17  calls
18    Listen : {Listen1 : subprogram access Protocol;
19              Listen2 : subprogram access Queue_Request;};
20    Bind_Local : {Bind_Local1 : subprogram access Bind_Request;
21                  Bind_Local2 : subprogram access Add_Request;};
22    Activate : {Activate1 : subprogram access Activate_Request;};
23  end Mubroker.Generic;
24 end Ipao::Mubroker;
```

Listing VI.9 – Modélisation du μBroker

Contrairement à des services comme l'activation ou la liaison, dont l'implantation dépend de l'organisation des entités applicatives et protocolaires, le μBroker coordonne toujours les mêmes services. Nous utilisons donc des accès à sous-programmes afin de pouvoir mettre en place une description comportementale indépendante des services.

VI-3.5 Assemblage des composants

Chaque nœud applicatif est modélisé par un processus AADL contenant des *threads* AADL. Ces *threads* représentent l'exécutif de bas niveau ; ils sont issus de l'expansion des *threads* AADL de la première phase du processus de conception. Ils sont exécutés par des *threads* du système d'exploitation.

Le processus et ses *threads* contiennent les composants de donnée et appellent les sous-programmes modélisant l'instance de l'architecture schizophrène pour chaque nœud. Le listing VI.10 donne un exemple d'instanciation des composants de l'intergiciel.

```
1 process application_node
2 end application_node;
3
4 thread Minimal_Thread
5 features
6     Protocol : requires subprogram access Ipao::Protocol::
          Protocol;
7     Queue_Request : requires subprogram access Ipao::Bind::
          Add_Request;
8     Bind_Request : requires subprogram access Ipao::Bind::
          Bind_Request;
9     State_Machine : requires data access Mubroker_State;
10 properties
```

```
11   Dispatch_Protocol => background;
12 end Minimal_Thread;
13
14 thread implementation Minimal_Thread.Generic
15 subcomponents
16   activation_buffer : data Ipao::Activation::Buffer;
17   add_request : subprogram Ipao::Activation::Add_Request;
18   activate_request : subprogram Ipao::Activation::Activate_Request;
19 calls
20   {broker : subprogram Ipao::Mubroker::Mubroker.Generic;};
21 connections
22   subprogram access Protocol -> broker.Protocol;
23   subprogram access Queue_Request -> boker.Queue_Request;
24   subprogram access Bind_Request -> broker.Bind_Request;
25   subprogram access Add_Request -> broker.Add_Request;
26   subprogram access Activate_Request -> broker.Activate_Request;
27   subprogram access State_Machine -> broker.State_Machine;
28   data access activation_buffer -> add_request.Buffer;
29   data access activation_buffer -> activate_request.Buffer;
30 end Minimal_Thread.Generic;
31
32 process implementation Application_Node.Generic
33 subcomponents
34   thread1 : thread Minimal_Thread.Generic;
35   thread2 : thread Minimal_Thread.Generic;
36   middleware_state : data Ipao::Mubroker::Mubroker_State;
37   binding_buffer : data Ipao::Bind::Buffer;
38   protocol1 : subprogram Ipao::Protocol::Protocol;
39   queue_request : subprogram Ipao::Bind::Queue_Request;
40   bind_request : subprogram Ipao::Bind::Bind_Request;
41 connections
42   subprogram access protocol1 -> thread1.Protocol;
43   subprogram access protocol1 -> thread2.Protocol;
44   subprogram access queue_request -> thread1.Queue_Request;
45   subprogram access queue_request -> thread2.Queue_Request;
46   subprogram access bind_request -> thread1.Bind_Request;
47   subprogram access bind_request -> thread2.Bind_Request;
48   data access middleware_state -> thread1.State_Machine;
49   data access middleware_state -> thread2.State_Machine;
50   data access binding_buffer -> queue_request.buffer;
51   data access binding_buffer -> bind_request.buffer;
52 end Application_Node.Generic;
```

Listing VI.10 – Assemblage des composants de l'intergiciel

Nous considérons un système composé de deux *threads*, instanciés aux lignes 34 et 35. Les composants `Minimal_Thread.Generic` représentent les *threads* de l'exécutif de bas niveau correspondants. Ils s'exécutent en tâche de fond et appellent le μBroker, dont la description comportementale doit former une boucle d'exécution. Les *threads* minimaux instancient les sous-programmes et les données du service d'activation, qui leur sont propre (lignes 16 à 18). En revanche, les services de protocole, de liaison, ainsi que l'état du μBroker sont partagés entre les différents *threads* ; il doivent être instanciés dans le processus (lignes 36 à 40) puis connectés aux *threads* (lignes 42 à 51).

VI-3.6 Évaluation des caractéristiques des services de communications

La modélisation des couches de communication de l'application – constituant l'intergiciel de communication – en AADL fait intervenir un ensemble de composants dont la description exacte dépend de la configuration de l'application.

La description de ces composants permet de retranscrire les différents paramètres de configuration que nous avons mentionnés en section VI-2.2.3. Elle permet également de spécifier les dimensions spatiales et temporelles des différents éléments de l'intergiciel.

La prise en compte de ces caractéristiques est difficilement conciliable avec une génération complète des implantations des composants AADL. Les temps d'exécution et la place en mémoire dépendent en effet non seulement du code des implantations, mais aussi du compilateur utilisé, du système d'exploitation sous-jacent, etc.

Il est donc nécessaire de prédéfinir la plus grande partie des composants utilisables. Nous avons montré que certains composants ne changent pas d'une configuration à une autre ; par exemple, une personnalité protocolaire constitue un ensemble fixe de composants. Certaines composants, comme les sous-programmes de conversion du service de représentation, peuvent être construit par assemblage de composants élémentaires – en l'occurrence des sous-programmes AADL permettant de convertir les types AADL de base que nous avons définis en section V-2.2.

Il est ainsi possible de définir un ensemble de composants soit complètement prédéfinis soit paramétrables. La construction de l'intergiciel consiste alors en l'assemblage de ces composants connus pour former la configuration souhaitée, plutôt qu'en une génération complète.

VI-4 Conclusion

Dans ce chapitre nous avons présenté la conception de l'exécutif que nous associons aux descriptions AADL. Cet exécutif fournit les fonctionnalités nécessaires à la mise en place des mécanismes de communication et de traitement des flux de données que nous avons définis au chapitre IV-4.2. Cette conception d'exécutif constitue une alternative à l'approche proposée dans la version 1.0 du standard AADL. Nous avons présenté deux approches possibles pour sa mise en place.

Nous avons tout d'abord montré comment utiliser un exécutif de haut niveau. Un tel exécutif correspond à la première phase de notre processus de conception. L'utilisation de PolyORB permet de prendre en charge tous les modèles de communication que nous avons définis en AADL ; il permet également une grande flexibilité dans la configuration de l'exécutif.

L'utilisation d'un exécutif de bas niveau pour la seconde phase de notre processus de conception suppose la modélisation en AADL des différentes fonctionnalités de l'intergiciel de communication. Nous nous sommes reposés sur l'architecture schizophrène pour concevoir cette modélisation AADL. Bien que PolyORB soit lui-même une mise en pratique de l'architecture schizophrène, il repose sur des mécanismes dynamiques, qui ne correspondent pas à la démarche de modélisation d'AADL. Notre modélisation ne correspond donc pas exactement à l'implantation actuelle de PolyORB, même si elle en reprend les principes architecturaux.

La définition des composants en AADL permet d'avoir une vision précise des mécanismes mis en jeu ; il est ainsi possible de dimensionner les couches de communication de l'application tout en reproduisant les politiques de configuration utilisées dans PolyORB.

Nous avons présenté un ensemble de composants fournissant l'armature de la description architecturale de l'intergiciel. La description d'une instance appropriée de l'intergiciel de commu-

nication en AADL consiste à préciser ces composants en fonction de la configuration de l'application.

Bien que la définition exacte des composants soit spécifique à chaque instance de l'intergiciel en AADL, la grande majorité d'entre eux correspond à la particularisation d'un ensemble relativement restreint de composants AADL génériques. La description automatique des couches de communication en AADL consiste donc en l'assemblage de composants parmi un ensemble prédéfini et connu.

CHAPITRE VII

Vérification formelle de la structure des applications

DANS LES CHAPITRES V ET VI nous avons étudié l'exploitation d'AADL pour produire automatiquement une application répartie, ainsi que la configuration de l'intergiciel sous-jacent. Notre démarche permet de produire une application de façon automatisée ; cependant elle ne garantit pas le bon fonctionnement de l'architecture.

L'analyse des architectures logicielles a fait l'objet de nombreux travaux [Bernardo et Inverardi, 2003 ; Poizat et al., 2004]. Dans ce chapitre, nous montrons une autre interprétation des spécifications de construction AADL que nous avons présentées au chapitre IV ; nous produisons ici une représentation formelle de l'architecture pour permettre l'étude de certaines propriétés structurelles. Nous nous intéressons aux réseaux de Petri, qui permettent de modéliser le comportement des systèmes.

Nous présentons la correspondance que nous avons définie entre AADL et les réseaux de Petri, puis nous étudions différentes propriétés que ce formalisme permet de mettre en évidence sur l'architecture.

VII-1 Objectifs de la vérification

La vérification d'une architecture peut porter sur la structure de la description, comme la validité des connexions ; elle peut aussi concerner le respect des contraintes exprimées dans les propriétés AADL, telles que les temps d'exécution et les politiques d'ordonnancement [Singhoff et al., 2005].

De la même façon que la génération de code source à partir d'une description AADL a pour objectif la production d'un système exécutable, la production d'une représentation formelle doit correspondre à un objectif de vérification défini ; un seul formalisme ne peut pas couvrir tous les aspects nécessaires à l'analyse complète d'une architecture.

VII-1.1 Éléments de vérification

Dans ce chapitre nous nous focalisons sur la validation de la structure de l'architecture. Nous exploitons la description de l'assemblage des composants AADL pour nous assurer de l'intégrité des flux d'exécution et de données au sein de l'architecture. Nous avons pour objectif de vérifier des propriétés telles que :

- les données utilisées dans les sous-programmes sont définies de façon déterministe ;
- l'assemblage des composants n'engendre pas d'interblocage ;

– la structure de l'architecture n'engendre pas systématiquement de débordement de file d'attente.

Ces propriétés concernent la structure même de la description architecturale et ne dépendant pas de facteurs d'ajustement tels que les temps d'exécution. Elles révèlent donc des erreurs dans la conception de l'architecture et ne se basent que sur les spécifications que nous avons énumérées au chapitre IV.

La vérification de telles propriétés nécessite l'utilisation d'une représentation formelle permettant de représenter l'organisation des flux d'exécution et de donnée. Les réseaux de Petri correspondent à ces besoins.

VII-1.2 Définition des réseaux de Petri

Les réseaux de Petri sont issus des travaux de Carl Adam Petri, en 1962. Ils définissent une notation formelle pour la modélisation et la vérification de systèmes concurrents. La manipulation de tels modèles par des outils d'analyse permet une vérification automatique de propriétés structurelles, ainsi que la vérification des systèmes considérés suivant des formules de logique temporelle. Il est ainsi possible de vérifier l'absence d'un état interdit du système, la causalité entre états ou de caractériser les flux d'exécution dans l'architecture.

Les réseaux de Petri se représentent traditionnellement graphiquement, sous la forme de graphes « places/transitions ». Les places sont représentées par des cercles et les transitions par des rectangles ; ces différents éléments sont reliées alternativement par des connexions. Des jetons circulent dans le réseau, passant des places aux transitions par l'intermédiaire des connexions. Afin de pouvoir manipuler formellement ces représentation, les réseaux de Petri sont définis mathématiquement ainsi [Girault et Valk, 2003] :

Définition VII.1 (réseau de Petri)
Un réseau de Petri est un réseau de type places et transitions défini par un tuple $N = (P, T, Pre, Post)$ où :
– P est un ensemble fini (l'ensemble des *places* de N) ;
– T est un ensemble fini (l'ensemble des *transitions* de N), disjoint de P ;
– $Pre, Post \in \mathbb{N}^{|P|\times|T|}$ sont des matrices (les matrices d'incidence *amont* and *aval* de N).
$C = Post - Pre$ est appelée *matrice d'incidence* de N.

Un réseau de Petri définit donc un graphe dans lequel des jetons circulent, passant de places en transitions. Les transitions « consomment » les jetons entrants et en produisent d'autres en sortie. Elles ne peuvent être déclenchées que lorsque tous les jetons nécessaires sont disponibles ; elles permettent donc de synchroniser la circulation des jetons dans le réseau. Les places agissent comme des tampons permettant de stocker les jetons.

Dans le cadre de nos travaux, nous utilisons une famille de réseaux de Petri, dits colorés bien formés [Girault et Valk, 2003], qui sont définis mathématiquement ainsi :

Définition VII.2 (réseau de Petri coloré bien formé)
Un réseau de Petri coloré bien formé (*well-formed colored Petri net* en anglais) est un 5-uplet $N = (P, T, Pre, Post, Types, M_0)$ où :
– P est l'ensemble fini des places de N ;
– T est l'ensemble fini des transitions de N ;
– $Pre(t)$ et $Post(t)$ sont respectivement les fonctions de pré et post-condition associées à la transition $t \in T$;
– $Types$ est l'ensemble fini des types de base ;
– M_0 est le marquage initial.

Un réseau de Petri coloré est donc un réseau de Petri « classique » dans lequel les différents jetons peuvent appartenir à des ensembles de définition différents. À chaque place $p \in P$, nous associons un domaine $Dom(p)$, qui est le produit cartésien de certains types de bases. $Dom(p)$ correspond à l'ensemble des jeton que la place p peut contenir. Par analogie avec le formalisme graphique, $Dom(p)$ est appelé domaine de couleur ; les jetons ont donc des couleurs différentes, et peuvent être contenus dans les places acceptant les domaines de couleur correspondants.

Un marquage $M(p)$ est également associé à chaque place p. Ce marquage indique le nombre de jetons de chaque domaines présents dans la place p.

Un marquage M du réseau N est défini comme étant la fonction associant un marquage à chaque place p de P. Graphiquement, un marquage correspond à l'ensemble des jetons présents dans les places à un instant donné.

Les fonctions *Pre* et *Post* décrivent comment un marquage est modifié lorsqu'une action est réalisée. Les actions sont associées aux transitions du modèle, de sorte qu'on dit souvent qu'une transition est *tirée* (franchie) pour signifier qu'une action est réalisée.

VII-1.2.1 Exemple de réseau de Petri

À titre d'illustration, considérons le réseau de Petri représenté sur la figure VII.1. Ce réseau définit deux domaines de couleurs, $type1$ et $type2$, auxquels les jetons peuvent appartenir. Le marquage initial M_0 du réseau contient un jeton de chaque domaine : l'un situé sur la place `p1` avec pour valeur 1 ; l'autre sur la place `p2` avec pour valeur b.

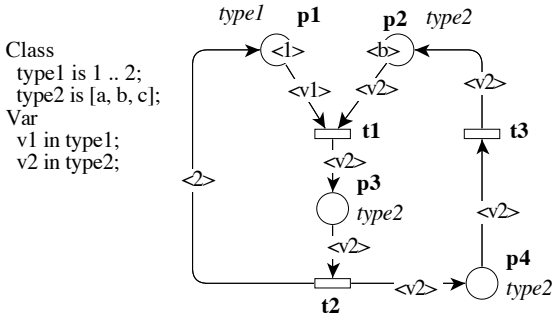


FIG. VII.1 – Exemple de réseau de Petri coloré

À partir du marquage initial, seule la transition `t1` peut être déclenchée ; en effet, contrairement à toutes les autres transitions, tous les jetons nécessaires à son déclenchement sont présents – en l'occurrence dans `p1` et `p2`. La transition `t1` consomme les deux jetons, dont les valeurs sont désignées par $v1$ et $v2$; elle produit en retour un jeton de valeur $v2$, qui est stocké dans la place `p3`. La valeur $v1$ correspondant au jeton issu de la place `p1` est donc perdue.

La transition `t2` consomme ensuite le jeton stocké dans `p3` et produit deux jetons. L'un des jetons prend la valeur $v2$ et est stocké dans la place `p4` ; l'autre jeton, dont la valeur est fixée à 2, est stocké dans `p1`.

Le jeton stocké dans `p4` est ensuite consommé par la transition `t3` et réintroduit dans la place `p3`. Dès que les places `p1` et `p2` contienne chacune un jeton, la transition `t1` peut être à nouveau déclenchée.

Un tel réseau de Petri illustre les possibilités de synchronisation (transition `t1`), ainsi que la transmission des valeurs de jeton, leur destruction ou leur création (transition `t2`).

VII-1.2.2 Analyse des réseaux de Petri

Lorsqu'une transition est activée, les jetons correspondants sont consommés des places en entrée, d'autres jetons sont générés dans les places de sortie. En se basant sur cette évolution, l'espace d'états associé au modèle peut être construit. L'étude de l'espace d'états du système permet d'analyser son comportement et vérifier la présence d'états particuliers (par exemple pour savoir si une situation donnée peut se produire) ou vérifier la relation causale entre deux états (par exemple, si un état e_1 du système est atteint, est-ce que un autre état e_2 sera atteint par la suite ?).

Les réseaux de Petri permettent aussi une analyse structurelle. Des propriétés telles que les invariants – c'est-à-dire les sous-ensembles de place pour lesquels le nombre de jetons présents reste constant – ou les symétries sont calculées sur la structure du modèle et ne nécessitent pas la construction de l'espace d'états. Ceci permet de vérifier des systèmes dont l'espace d'états est infiniment grand [Girault et Valk, 2003].

VII-2 Principes de la traduction des constructions AADL en réseaux de Petri

Les réseaux de Petri constituent un formalisme abstrait pour décrire les architectures concurrentes ; ils fournissent un support mathématique à leur analyse. La construction d'un réseau de Petri pour modéliser une architecture donnée traduit un choix sémantique pour l'interprétation des places, des transitions et des jetons. La construction des réseaux de Petri dépend en partie des propriétés que l'on souhaite vérifier. Dans cette section nous présentons la traduction des éléments architecturaux AADL de base.

La modélisation du comportement d'un système (réparti ou non) fait essentiellement intervenir les éléments logiciels de la modélisation. De la même façon que pour la génération de code, la traduction d'une architecture AADL en réseau de Petri doit s'effectuer sur l'architecture AADL instanciée ; nous ne considérons que les instances des composants, et pas leurs déclarations. Par ailleurs, contrairement à la génération de code, la traduction s'effectue cette fois sur l'architecture globale : nous considérons tous les nœuds applicatifs en même temps, afin de rendre compte des interactions entre eux. La traduction en réseaux de Petri n'exploite donc pas la description AADL de la même façon que la production d'applications exécutables, étudiée aux chapitres V et VI.

Dans le cadre d'une modélisation en réseaux de Petri, nous nous intéressons principalement aux composants applicatifs actifs, c'est-à-dire les *threads* et les sous-programmes, ainsi que les instances de composants de donnée. Les composants décrivant la plate-forme d'exécution sont ignorés. Les autres composants logiciels (groupes de threads et processus) et les systèmes sont considérés comme de simple conteneurs.

Une description AADL est constituée de composants qui acceptent des données en entrée et en produisent d'autres en sortie. Il en découle la sémantique suivante pour la construction des réseaux de Petri :

Règle VII.1 (interprétation des places et des transitions)
Nous interprétons les éléments d'un réseau de Petri ainsi :
- les jetons correspondent aux flux de données et d'exécution circulant à travers l'architecture ;
- les transitions modélisent les composants actifs, qui consomment des données et en produisent d'autres, ainsi que les connexions AADL ;
- les places symbolisent les entités permettant de stocker des données.

La traduction d'une architecture en réseau de Petri modélise donc les flux d'exécutions à travers l'architecture. Dans la mesure où nous considérons les instances de composants, les différentes construction des réseaux de Petri sont nommées à partir des instances qu'elle traduisent.

Règle VII.2 (Nommage des éléments d'un réseau de Petri)
Les noms des entités du réseau de Petri sont construits par concaténation des noms des parents des entités AADL correspondantes.

VII-3 Définition des domaines de couleur du réseau

La traduction en réseaux de Petri vise à étudier les flux d'exécution dans l'architecture. Les domaines de couleurs associés à un réseau donné doivent permettre de représenter les données et les structures de contrôle des *threads*.

Règle VII.3 (Domaines de couleur)
Un réseau de Petri généré à partir d'une description AADL comprend deux classes de couleurs :
- les jetons représentant les flux de donnée appartiennent à une classe nommée *Value* qui contient deux couleurs : u et d ;
- les jetons de contrôle des *threads* appartiennent à une classe nommée *Control* qui contient autant de couleurs que d'instances de *threads* dans la description AADL.

Un domaine *Comm* est défini comme le produit des classes *Value* et *Control* : $Comm = Value \times Control$. Les jetons de ce domaine correspondent aux messages envoyés entre les *threads*.

La classe *Value* contient deux couleurs u et d. La couleur u correspond à une valeur indéfinie (*undefined*) ; elle permet de modéliser les situations dans lesquelles aucune donnée n'est spécifiée. La couleur d correspond à tout valeur renseignée (*defined*) ; c'est la couleur de tous les jetons de donnée circulant dans l'architecture.

La classe *Control* permet d'exprimer les flux de contrôle dans les *threads*. Le fait de définir une couleur par instance de *thread* permet de les différencier ; nous développons les raisons de cette distinction dans la section VII-4.2.3.

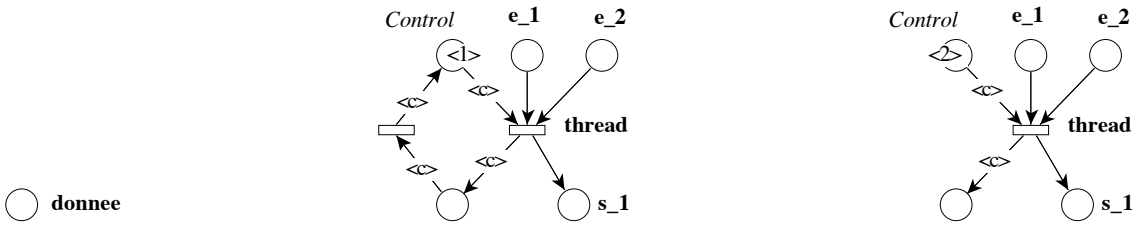
Le domaine *Comm* correspond au multiplexage d'un jeton de contrôle et d'un jeton de valeur. Son rôle sera également étudié à la section VII-4.2.3.

VII-4 Modélisation des éléments de haut niveau

Dans cette section nous détaillons la traduction des composants AADL en réseau de Petri. Par modélisation de haut niveau, nous entendons modélisation dans laquelle le détail de l'implantation des *threads* n'apparaît pas. La traduction de l'implantation des *threads* sera traitée dans la section VII-5.

VII-4.1 Traduction des composants

La figure VII.2 résume les différentes modélisations en réseaux de Petri, correspondant aux déclarations du listing VII.1.



(a) donnée

(b) *thread* périodique, apériodique ou sporadique

(c) *thread* en tâche de fond

FIG. VII.2 – Modélisation en réseaux de Petri de composants AADLi, décrit au listing VII.1

```
1  data donnee
2  end donnee;
3
4  thread thread1
5  features
6    e_1 : in data port;
7    e_2 : in event data port;
8    s_1 : out data port;
9  properties
10   Dispatch_Protocol => perdiodic;
11 end thread1;
12
13 thread thread2
14 features
15   e_1 : in data port;
16   e_2 : in event data port;
17   s_1 : out data port;
18 properties
19   Dispatch_Protocol => background;
20 end thread2;
```

Listing VII.1 – Exemples de déclarations de composants AADL, traduites en réseaux de Petri sur la figure VII.2

VII-4.1.1 Traduction des composants de donnée

Nous distinguons les composants de données des autres composants logiciels.

Règle VII.4 (Traduction des composants de donnée)
Une instance de composant de donnée est traduite par une place.

Nous sommes ainsi cohérents avec la règle VII.1 : un composant de donnée ne donne pas lieu à une opération de traitement et ne peut donc pas contenir de transition.

Les déclarations de composant de donnée AADL ne sont pas traduites dans le réseau, puisque celui-ci ne rend compte que des entités instanciées. Par ailleurs, contrairement à la traduction en langage de programmation, nous ne nous intéressons pas à la sémantique des données : toutes les déclarations de données sont donc considérées équivalentes vis-à-vis des flux d'exécution. La vérification de la cohérence des types doit être réalisée par ailleurs.

VII-4.1.2 Traduction des composants « actifs »

Tous les composants autres que les données recueillent des données en entrée, les traitent et produisent des données en sortie. Les figures VII.2(b) et VII.2(c) illustrent les modélisations correspondant aux différents composants AADL considérés.

Règle VII.5 (Traduction des types de composants)
Les composants « actifs » de haut niveau (systèmes, processus, *threads*) d'une architecture sont traduits par une transition (représentant le composant lui-même) entourée de places modélisant ses éléments d'interface.

Cette approche assure la cohérence du formalisme vis-à-vis de la syntaxe AADL : en effet, les ports des composants – que nous modélisons en réseaux de Petri par des places – sont typiquement associés à une déclaration de composant de donnée. Un composant de donnée est donc toujours représenté par une place, qu'il s'agisse d'une instance – c'est-à-dire un sous-composant – ou d'un élément d'interface.

Notons que cette modélisation de haut niveau implique qu'un composant a besoin de recevoir des données sur tous ces ports pour s'exécuter. Pour les *threads*, cette hypothèse correspond aux spécifications que nous avons définies au chapitre IV ; ce n'est en revanche pas le cas pour les autres composants, tels que les systèmes. En l'absence d'information sur la structure interne du composant considéré, cette hypothèse, bien que forte, est cependant assez naturelle et permet une modélisation systématique.

Afin de systématiser la traduction, nous créons une place par élément d'interface. De cette façon, il est notamment possible d'appliquer éventuellement des traitement différents à chaque élément d'interface lors de la génération du réseau de Petri – par exemple des fusions de places.

Règle VII.6 (Traduction des éléments d'interface)
Chaque élément d'interface est traduit par une place reliée à la transition représentant le composant considéré.
Un port d'entrée d'événement/donnée ou un accès à un sous-composant de donnée est traduit par une place connectée à la transition du composant.
Un port d'entrée de donnée est traduit par une place dont le marquage initial est u. La transition du composant consomme le jeton de cette place et le restitue. La transition du composant comporte une garde l'empêchant de se déclencher avec un jeton de couleur u.
Tous les ports de sortie sont traduits par une place de sortie reliée à la transition du composant.
Les ports d'entrée/sortie sont assimilés à un couple de ports, l'un en entrée, l'autre en sortie.
Toutes les places représentant un port sont du domaine *Comm*.
Les accès aux bus ou aux sous-programmes ne sont pas traduits.

Un port de donnée ne comporte pas de file d'attente ; une donnée stockée dans un tel port n'est donc pas consommée : elle est seulement lue. Le marquage initial des places correspondantes correspond à la valeur indéfinie u afin d'empêcher le composant de se déclencher avec ce qui correspond en fait à une absence de valeur.

Nous ne traduisons pas les accès aux bus car ces composants ne sont pas pris en considération dans notre traduction. La prise en charge des accès à sous-programmes sera étudiée dans la section VII-5.3.

Les *threads* AADL sont les éléments actifs de l'architecture. Pour rendre compte de cet aspect, leur traduction en réseau de Petri fait apparaître un jeton représentant le contrôle d'exécution du thread. Ce jeton de contrôle constitue le marquage initial d'une place associée au *thread*. La modélisation du circuit du jeton de contrôle dépend de la politique de déclenchement du *thread*. Les *threads* périodiques, apériodiques et sporadiques s'exécutent en boucle. Ils consomment donc leur jeton de contrôle et le restituent dans la place initiale (figure VII.2(b)). Ces trois types de *threads* se modélisent de la même façon alors qu'ils correspondent à des politiques de déclenchement différentes, comme nous l'avons étudié au chapitre VI. Ceci est dû au fait que notre modélisation en réseau de Petri ne tient pas compte des propriétés temporelles de l'architecture : nous ignorons donc les situations dans lesquelles un *thread* devrait se déclencher alors que les données entrantes ne sont pas disponibles.

Les *threads* s'exécutant en tâche de fond ne bouclent pas. Leur traduction en réseau de Petri ne doit donc pas restituer le jeton de contrôle, qui est consommé définitivement (figure VII.2(c)). Pour des raisons de régularité dans la représentation en réseau de Petri, tous les *threads* ont une structure semblable ; la transition de retour des *threads* périodiques trouvera par ailleurs sa justification dans la section VII-5.3.1.

La traduction d'une description AADL en réseau de Petri a pour objet l'analyse des flux de données et d'exécution. Par conséquent, seuls les composants ayant un rôle dans ces flux doivent apparaître.

Règle VII.7 (Traduction des processus et des systèmes)
Un processus ou un système contenant des sous-composants n'est pas traduit en réseau de Petri. Seuls ses sous-composants sont transcrits dans le réseau de Petri. À l'inverse, un processus ou un système dont l'implantation n'est pas spécifiée apparaît dans le réseau de Petri, assimilé à un *thread*.

Le réseau de Petri correspondant à une architecture AADL de haut niveau est la mise à plat de toutes les instances des *threads*, qui représentent les entités réellement impliquée dans les flux d'exécution et de données.

Les processus et les systèmes sans sous-composants sont traduits dans le réseau de Petri. Dans la mesure où aucune information n'est disponible quant à leur structure interne, nous pouvons considérer qu'ils contiennent un *thread* ; nous les assimilons donc à ce *thread* supposé afin de les intégrer dans la modélisation du flux d'exécution. De tels systèmes ou processus sont donc traduit par un sous-réseau de Petri tel que représenté sur la figure VII.2(b).

VII-4.2 Traduction des connexions

Les connexions AADL se traduisent de différentes façon, selon leur nature. Nous traduisons les connexions AADL représentant une transmission de données par des transitions. Contrairement aux transitions des composants, ces transitions ne correspondent pas à une transformation des données ; elles permettent simplement d'exprimer la coordination de la transmission des données.

Par rapport à ce que nous avons indiqué en section VII-4.1.2, nous considérons des connexions directes entre les différentes entités de haut niveau de plus fine granularité – c'est-à-dire les *threads* AADL ou les *thread* implicites correspondant aux processus ou aux systèmes. Comme les différents composants englobants ne sont pas traduits par les construction en réseau de Petri, les différents ports des composants sont considérés comme étant directement reliés par une seule connexion, au lieu d'une suite de connexions point-à-point comme c'est le cas dans les descriptions AADL.

VII-4.2.1 Connexion des accès à composant de donnée

La connexion de sous-composants correspond à une situation simple : la place correspondant à l'instance du composant de donnée considéré doit être assimilé à l'interface offrant ou requérant le sous-composant.

Règle VII.8 (Connexion des accès à sous-composants)
Les connexions aux composants de donnée reviennent à fusionner la place modélisant le composant de donnée considéré avec la place correspondant à l'élément d'interface correspondant.

Nous ne considérons ici que les composants de données. En effet, les bus sont des composants matériels que nous ignorons dans le cadre de notre traduction. Les accès aux sous-programmes seront décrits dans la section VII-5.3.

VII-4.2.2 Connexions des ports

Le cas des connexions de ports est plus délicat. En effet, elles font partie intégrante des flux de données que nous souhaitons analyser.

Dans la mesure où nous ne considérons que des connexions directes, il s'agit d'interposer une transition entre les différents ports connectés. Dans le cas de multiples connexions issues d'un seul port de sortie, nous devons exprimer le fait que les données sont dupliquées et envoyées simultanément.

Règle VII.9 (Modélisation des connexions de ports)
La connexion de deux ports AADL se traduit par une transition. Les multiples connexions d'un port de sortie à plusieurs ports d'entrée sont traduites par une seule transition qui modélise la duplication des données transférées.
Une connexion de port d'événement/donnée se traduit par une simple transition.
Une connexion de port de donnée se traduit par une transition consommant à la fois le jeton transmis et le jeton placé dans la place de destination. Le jeton précédemment stocké dans la place de destination est ainsi remplacé par le nouveau jeton.

La traduction exacte des connexions AADL diffère selon que nous considérons une connexion de ports d'événement/donnée ou de donnée. En effet, les ports de donnée ne peuvent pas être associés à des files d'attente ; la place correspondante doit donc ne comporter qu'un seul jeton, qui est remplacé lorsqu'une nouvelle donnée arrive. Nous appliquons la même démarche que pour la connexion des éléments d'interface aux composants eux-mêmes (règle VII.6).

La figure VII.3 illustre les différentes traductions possibles pour les connexions de ports. Les modélisations des *threads* sont repérées par des zones grisées.

Le *thread* `thread1` possède deux ports de sortie : `s1` (port d'événement/donnée) et `s2` (port de donnée). Ces deux ports sont reliés aux ports d'entrée `s1` et `s2` du *thread* `thread2` par l'intermédiaire des connexions `cnx1` et `cnx2`. La connexion `cnx1` est une connexion de ports de données, caractérisée par les doubles connexions entre les transitions et les places. La connexion `cnx2` est une connexion de port d'événement/donnée, caractérisée par des connexions simples entre les transitions et les places.

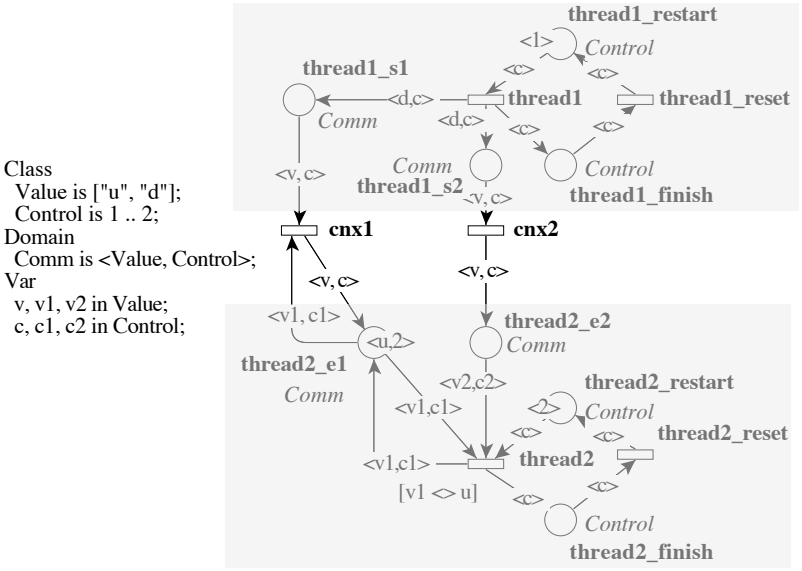


FIG. VII.3 – Description des connexions en réseau de Petri, correspondant à la figure VII.4

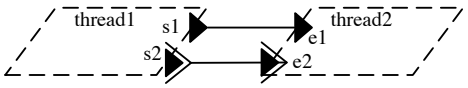


FIG. VII.4 – Connexions AADL, traduites par le réseau de la figure VII.3

VII-4.2.3 Optimisation des réseaux générés

Le fait de ne pas considérer les propriétés temporelles des architectures implique que la modélisation en réseau de Petri d'un *thread* émetteur n'a aucune contrainte dans son débit d'émission de jetons. L'architecture décrite au listing VII.2 fait apparaître un *thread* émetteur et deux *threads* récepteurs. Si le *thread* émetteur s'exécute à une fréquence plus élevée que les récepteur, les files d'attente des *threads* récepteur déborderont. Ce scénario d'exécution se traduirait par un réseau de Petri non borné.

Pour pallier ce problème, nous exploitons le fait que nous ignorons délibérément les propriétés temporelles de l'architecture. Nous pouvons alors considérer que les *threads* sont bien synchronisés ; le *thread* émetteur ne produit une nouvelle donnée que lorsque les précédentes ont été consommées. Cette hypothèse permet de borner le réseau. Sa traduction en terme de construction syntaxique consiste à mettre en place une boucle de rétroaction entre les *threads* récepteur et les *threads* émetteurs.

Règle VII.10 (Jetons modélisant les communications inter-*threads*)
Les jetons de donnée émis par la transition d'un *thread* est un couple $< d, c >$ du domaine *Comm*, où c appartient à la classe *Control* et a la valeur correspondant au *thread* émetteur.

Chaque message émis par un *thread* porte ainsi la marque de son émetteur. Cela permet de mettre en place un mécanisme d'acquittement à partir du *thread* récepteur.

```
data donnee
end donnee;

thread thread_a
features
  s : out event data port donnee;
properties
  Dispatch_Protocol => periodic;
end thread_a;

thread thread_b
features
  e : in event data port donnee;
properties
  Dispatch_Protocol => aperiodic;
end thread_b;

process processus_a
features
  s : out event data port donnee;
end processus_a;

process processus_b
features
  e : in event data port donnee;
end processus_b;

process implementation processus_a.impl
subcomponents
  thread1 : thread thread_a;
connections
  cnx1 : event data port thread1.s -> s;
end processus_a.impl;

process implementation processus_b.impl
subcomponents
  thread1 : thread thread_b;
  thread2 : thread thread_b;
connections
  cnx1 : event data port e -> thread1.e;
  cnx2 : event data port e -> thread2.e;
end processus_b.impl;

system global
end global;

system implementation global.impl
subcomponents
  process1 : process processus_a.impl;
  process2 : process processus_b.impl;
connections
  cnx1 : event data port process1.s -> process2.e;
```

```
53 end global.impl;
```

Listing VII.2 – Exemple de connexion AADL, traduit par le réseau VII.5

Règle VII.11 (Limitation du nombre d'états du réseau)
À chaque transition modélisant un *thread* recevant des données doit correspondre une place de sortie. Cette place de sortie reçoit les différents jetons de contrôle dont les couleur correspondent à celles des jetons reçu dans les places d'entrée.
Les jetons accumulés dans cette place de sortie sont envoyés vers les places `reset` des *threads* émetteurs afin de réguler leur production de jetons.

La place `reset` du *thread* émetteur doit donc comporter une garde s'assurant que seul les jetons de contrôle correspondant à sa couleur sont acceptés. Il est par conséquent nécessaire de définir une couleur de contrôle par *thread* AADL, comme nous l'avons énoncé dans la règle VII.3.

Notons que dans le cas d'un *thread* en tâche de fond, cette rétroaction est inutile, puisque le *thread* considéré ne boucle pas ; ces *thread* ne possèdent d'ailleurs pas de transition `reset`.

La figure VII.5 donne le réseau de Petri issu de la modélisation du listing VII.2. Les représentations des *threads* sont repérées par les zones grisées. Les deux connexions `cnx1` et `cnx2` du processus `processus_b` sont fusionnées en une seule transition lors de la traduction en réseau de Petri. Nous nommons la transition résultante d'après la concaténation des différents *threads* impliqués : `process1_thread1_process2_threads1_process2_thread2`.

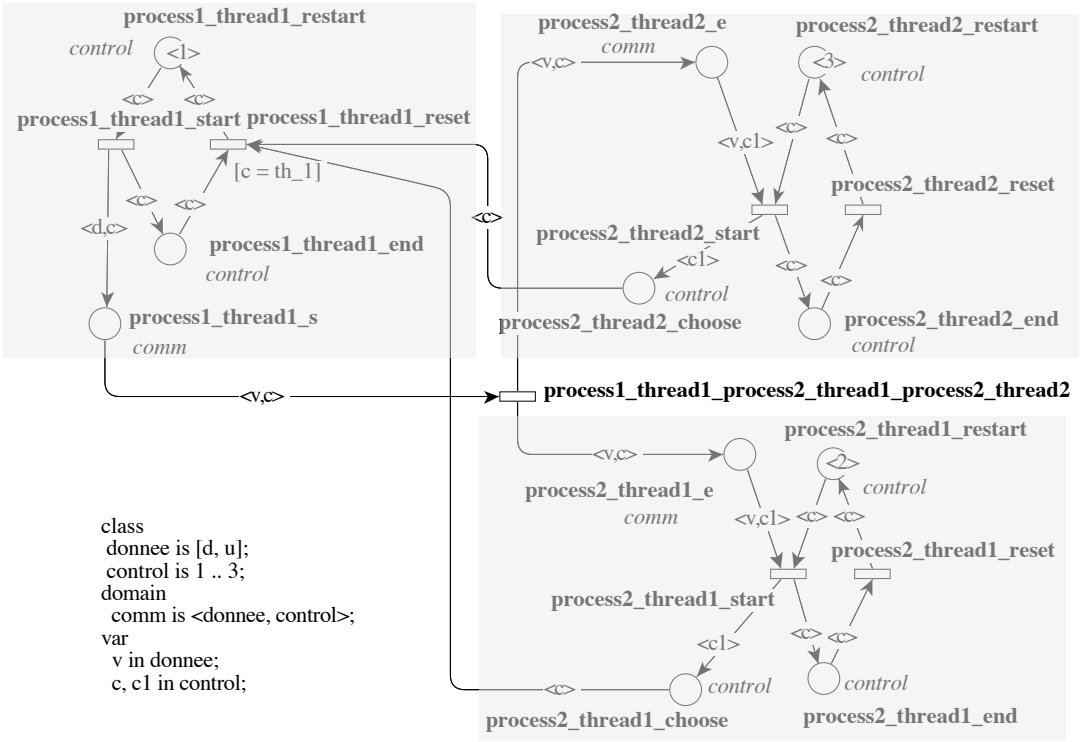


FIG. VII.5 – Réseau de Petri borné modélisant une architecture de haut niveau correspondant au listing VII.2, page 131

VII-5 Modélisation de l'implantation des composants

Jusqu'ici, nous avons modélisé les opérations de traitement des composants par une transition. La représentation de l'implantation d'un composant consiste à remplacer cette transition par le sous-réseau de Petri décrivant les séquences d'appel décrits dans l'implantation du composant. Le réseau de Petri fait alors apparaître les connexions entre les différents sous-éléments du composant.

VII-5.1 Modélisation des sous-composants

En section VII-4.1.2, nous avons indiqué que la représentation en réseaux de Petri se basait sur les composants actifs de l'architecture. D'après ce que nous avons établi dans la règle VII.7, un composant possédant des sous-composants est systématiquement remplacé par ces derniers. Seuls les sous-composants sont donc représentés dans un réseau de Petri ; cette situation a donc déjà été traitée en section VII-4.1.2.

VII-5.2 Modélisation des sous-programmes

Contrairement à la génération de code que nous avons présentée au chapitre V, la traduction en réseau de Petri rend compte des flux d'exécution. Selon cette approche, chaque sous-programme est modélisé dans le réseau autant de fois qu'il est appelé. Chaque appel de sous-programme est donc assimilé à une instance. Il en est par conséquent de même pour un accès requis à un sous-programme.

Règle VII.12 (Traduction des accès à sous-programme)
L'appel d'un sous-programme dont l'accès est requis par un *thread* ou un sous-programme est assimilé à un appel normal au sous-programme correspondant.

De la même façon que les composants de haut niveau, les sous-programmes sont modélisés par une transition entourée de places modélisant les paramètres d'entrée et de sortie.

Règle VII.13 (Traduction des sous-programmes)
Un appel de sous-programme AADL est traduit par une transition et des places correspondant à ses différents paramètres. Le flux d'exécution contrôlant l'appel du sous-programme est matérialisé par deux places transmettant un jeton de contrôle à la transition du sous-programme. Ce jeton de contrôle est celui du *thread* dans lequel l'appel est effectué.
Les places d'entrée sont connectées à la transition du sous-programme par un arc simple, de la même façon que pour les places de contrôle. Les places de sortie ont un marquage initial fixé à u. La transition du sous-programme consomme et réinjecte les jetons issus de ces places. Une garde est associée à la transition pour empêcher la consommation de la valeur indéfinie u.
Un paramètre d'entrée sortie est traduit par une place d'entrée et une place de sortie.

Les paramètres de sortie des sous-programmes modélisent des variables : leur valeur doit pouvoir être lue plusieurs fois, par différents sous-programmes. C'est pourquoi la transition du sous-programme doit remplacer un jeton qui est constamment à disposition.

Le cas d'un appel à un sous-programme distant – dont l'accès est fourni par un autre *thread* – présente quelques spécificités. En effet, un tel sous-programme met en jeu à la fois le *thread* local, qui effectue l'appel, et le *thread* distant, qui exécute effectivement le sous-programme. L'exécution du sous-programme nécessite donc les ressources d'exécution des deux *threads*.

Règle VII.14 (Traduction des appels de sous-programme distant)
Un appel de sous-programme distant est modélisé comme un appel de sous-programme local en ajoutant une paire de places de contrôle ; ces places correspondent au *thread* fournissant le sous-programme.

Un appel de sous-programme distant possède donc deux places d'entrée pour les jetons de contrôle : l'une correspondant au jeton de contrôle du *thread* local, l'autre recevant le jeton de contrôle du *thread* distant.

VII-5.2.1 Modélisation des connexions des paramètres

Les connexions de paramètres doivent laisser les jetons de donnée à la disposition d'éventuelles autres connexions.

Règle VII.15 (Traduction des connexions de paramètres)
L'ensemble des connexions dirigée vers un appel de sous-programme donné est traduit par une unique transition. Cette transition consomme et réinjecte les jetons issus des places correspondant aux paramètres de sortie des appels de sous-programme situés en amont dans la séquence d'appels. Elle consomme également le jeton de contrôle destiné à l'appel de sous-programme.
Elle distribue l'ensemble de ces jetons aux places d'entrée du sous-programme.

De la même façon que les connexions de ports de donnée, les connexions de paramètres consomment et réinjectent les jetons de donnée. Elles centralisent l'ensemble des jetons destinés au sous-programme destinataire. Cette synchronisation permet de limiter les états possibles au sein du réseau de Petri. En effet, dans la mesure où la transition d'une connexion consomme et réinjecte les jetons issus des places de sortie des sous-programmes amonts, il est nécessaire de synchroniser leur déclenchement avec le jeton de contrôle d'exécution, qui est consommé normalement. Nous évitons ainsi la création d'états artificiels liés au fait qu'une transition réinjectant le jeton consommé peut toujours être déclenchée.

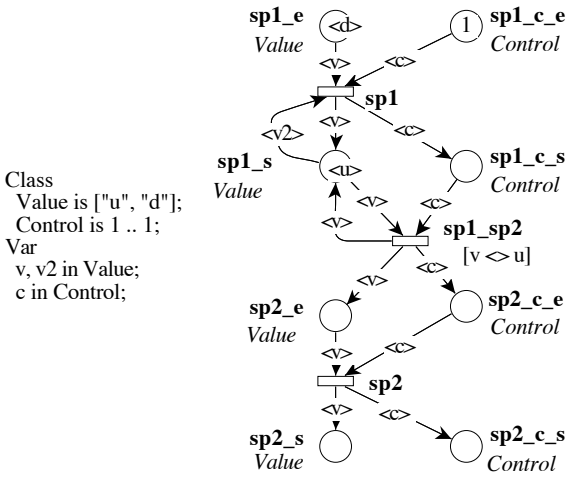


FIG. VII.6 – Réseau de Petri d'une connexion entre deux sous-programmes

Le réseau de Petri de la figure VII.6 illustre une connexion entre deux sous-programmes (`sp1` et `sp2`) d'une même sequence d'appels.

L'ensemble des connexions entre les paramètres des sous-programmes est rassemblé dans une seule transition, que nous appelons `sp1_sp2` d'après les deux appels de sous-programme qu'elle connecte. Cette transition prend en charge à la fois la connexion AADL entre les paramètres des sous-programmes et la circulation du jeton de contrôle. La place de sortie `sp1_s` est initialisée à une valeur invalide (u), qui ne peut pas franchir la connexion `sp1_sp2`. Ce jeton de valeur invalide modélise le fait qu'à l'état initial, la valeur de sortie de `sp1` est indéterminée. Lorsque le sous-programme `sp1` produit un résultat, le jeton du résultat remplace le jeton invalide et peut donc être transmis au sous-programme `sp2`. La valeur de sortie stockée dans `sp1_s` correspond correspond à une variable, et doit donc pouvoir être lue plusieurs fois. C'est pourquoi le jeton est réinjecté dans la place `sp1_s` lorsqu'il traverse la transition `sp1_sp2`.

VII-5.3 Modélisation des séquences d'appel de sous-programmes

Les séquences d'appel d'un sous-programme regroupent les appels de sous-programmes pour constituer des ensembles pouvant être manipulés par une description comportementale éventuellement associée au sous-programme considéré. Leur traduction en réseau de Petri doit donc faire apparaître une structure d'encapsulation.

Règle VII.16 (Traduction des séquences d'appel)
Une séquence d'appel `seq` est traduite par deux transitions `seq_begin` et `seq_end` qui jouent le rôle de connexions vers le premier appel et depuis le dernier appel de sous-programme de la séquence.

De la même façon que pour les connexions, les transitions des séquences synchronisent les différents jetons issus des paramètres du sous-programme englobant, ainsi que le jeton de contrôle du *thread*. Cette synchronisation correspond aux spécifications d'exécutif que nous avons établies au chapitre IV : un *thread* a besoin de toutes ses données pour s'exécuter ; cette exigence est retranscrite dans les séquences d'appel.

L'appel d'un sous-programme distant mobilise à la fois le jeton de contrôle du *thread* local effectuant l'appel et le jeton de contrôle du *thread* distant exécutant effectivement le sous-programme. Un appel distant correspond donc à une séquence d'appel particulière au niveau du *thread* distant. Cette séquence n'utilise aucun paramètre ; les transitions de début et de fin ne consomment et ne produisent que le jeton de contrôle du *thread* distant.

VII-5.3.1 Agencement des séquences d'appels

D'après les spécifications d'exécutif que nous avons établies, un sous-programme AADL peut contenir plusieurs séquences d'appels. Un *thread* peut également avoir plusieurs séquences : une séquence « normale » et différentes séquences correspondant aux sous-programmes dont l'accès est fourni pour constituer des appels distants.

Séquences « normales »

Les séquences « normales » correspondant à un traitement des paramètres ou des ports constituent un mécanisme séparés des séquences correspondant aux sous-programmes appelés à distance. Ces deux types de séquences ne peuvent pas être mélangés lors d'un même cycle d'exécution.

Règle VII.17 (Agencement des séquences de traitement des paramètres)
L'agencement des différentes séquences « normales » au sein d'un sous-programme ou d'un *thread* appelé `composant` est coordonné par deux transitions `composant_begin` et `composant_end`.
Ces deux transitions sont organisées comme des connexions de paramètres. La transition `composant_begin` synchronise les jetons des données entrant dans l'entité et le jeton de contrôle du *thread* correspondant. Les jetons consommés sont stockés dans des places correspondant à chaque jeton de paramètre et au jeton de contrôle – cette dernière place est nommé `composant_c_begin`. Toutes ces places sont reliées aux transitions `seq_begin` de chaque séquence `seq`.
Chaque transition `seq_end` envoie les jetons issus de la séquence `seq` dans des places correspondantes. Elles envoient le jeton de contrôle dans une place `composant_c_end`. Elle est construite comme une transition de connexion de paramètre.
Les différentes places correspondant aux paramètres de fin de séquence sont reliées à la transition `composant_end`. La place `composant_c_end` associée au jeton de contrôle est reliée à la fois à la transition `composant_end` et à la place `composant_c_begin` par l'intermédiaire d'une transition.
Comme les autres, la transition `composant_end` est structurée comme une transition de connexion de paramètre. Les jetons consommés sont envoyés vers les places correspondant aux données de sortie de l'entité. Le jeton de contrôle est envoyé vers la place de sortie du jeton de contrôle.

Ces différentes constructions de places et de transition permettent de recueillir les différentes données entrantes de l'entité, d'exécuter les différentes séquences d'appel possibles dans un ordre quelconque avec répétition puis de renvoyer le jeton de contrôle à l'extérieur du *thread* ou du sous-programme.

L'aspect essentiel de cet assemblage est de mettre en place deux chemins possibles pour le jeton de contrôle. Celui-ci peut traverser plusieurs fois les séquences s'il est orienté vers la place `composant_c_begin` à partir de la place `composant_c_end`. L'exécution des séquences s'arrête lorsque le jeton de contrôle est consommé par la transition `composant_end` ; cette transition ne peut être déclenchée que si toutes les places des paramètres de sortie des séquences contiennent un jeton de valeur définie d.

Nous modélisons ainsi le fait qu'il existe plusieurs scénarios d'exécution correcte d'un sous-programme (ou d'un *thread*) ; tous ces scénarios correspondent à l'exécution des différentes séquences dans un ordre quelconque avec répétitions possibles, aboutissant à la définition d'une valeur valide pour chaque paramètre de sortie du sous-programme ou du *thread*. La figure VII.7 fournit un exemple complet comprenant deux séquences d'appels, repérées par des zones grisées.

Séquences des sous-programmes distants

La traduction des séquences des sous-programmes distants est plus simple.

Règle VII.18 (Agencement des séquences des appels distants)
L'agencement des différentes séquences d'un *thread* appelé `composant` correspondant aux appels distants est constitué de ces différentes séquences. La transition `seq_begin` de chaque séquence `seq` consomme le jeton du *thread*. La transition `seq_end` de la fin de la séquence envoie le jeton de contrôle sur la place de sortie du *thread*.

Les séquences d'appel distant sont donc placées, séparément, au même niveau que l'ensemble des séquences de traitement des données du sous-programme ou du *thread*. Contrairement à cet ensemble, elles ne peuvent donc pas monopoliser le jeton de contrôle.

Un *thread* comportant ces deux constructions peut donc alternativement traiter ses paramètres ou exécuter un sous-programme appelé depuis un autre *thread*.

Réinitialisation des valeurs calculées

Entre chaque exécution d'un *thread* ou d'un sous-programme, les valeurs de sortie des sous-programmes appelés doivent être réinitialisées à la valeur invalide u. De cette façon, nous modélisons le fait que les éventuelles valeurs affectées aux variables dans le code source sont périmées lors de l'exécution suivante du *thread* ou du sous-programme.

Règle VII.19 (Réinitialisation des valeurs des sous-programmes)
La transition `thread_reset` associée à un *thread* **`thread`** est reliée à toutes les places de sortie des sous-programmes appelés. Au passage du jeton de contrôle, elle consomme le jeton stocké dans chacune de ces places, que sa valeur soit u ou d et y injecte un jeton de couleur u.

Nous réinitialisons ainsi toutes les valeurs de sortie. Cette réinitialisation est effectuée lorsque le jeton de contrôle quitte le *thread* et traverse la transition de retour `reset` qui apparaît dans la modélisation de haut niveau (cf. figure VII.2(c).

VII-5.4 Exemple complet

Afin d'illustrer les différentes règles de construction que nous avons énoncées, considérons un processus composé de deux *threads*, représenté sur le listing VII.3. Seul le premier *thread* reçoit les données transmises au processus ; le second *thread* fournit un point d'accès pour un appel de sous-programme distant. La figure VII.7 illustre le réseau de Petri correspondant à cette architecture.

```
1  data donnee end donnee;
2
3  subprogram sspg_a
4  features
5    e1 : in parameter donnee;
6    s1 : out parameter donnee;
7  end sspg_a;
8
9  subprogram sspg_b
10 features
11   e1 : in parameter donnee;
12   e2 : in parameter donnee;
13   s1 : out parameter donnee;
14 end sspg_b;
15
16 thread thread_a
17 features
18   rpc : subprogram sspg_b;
19 end thread_a;
20
```

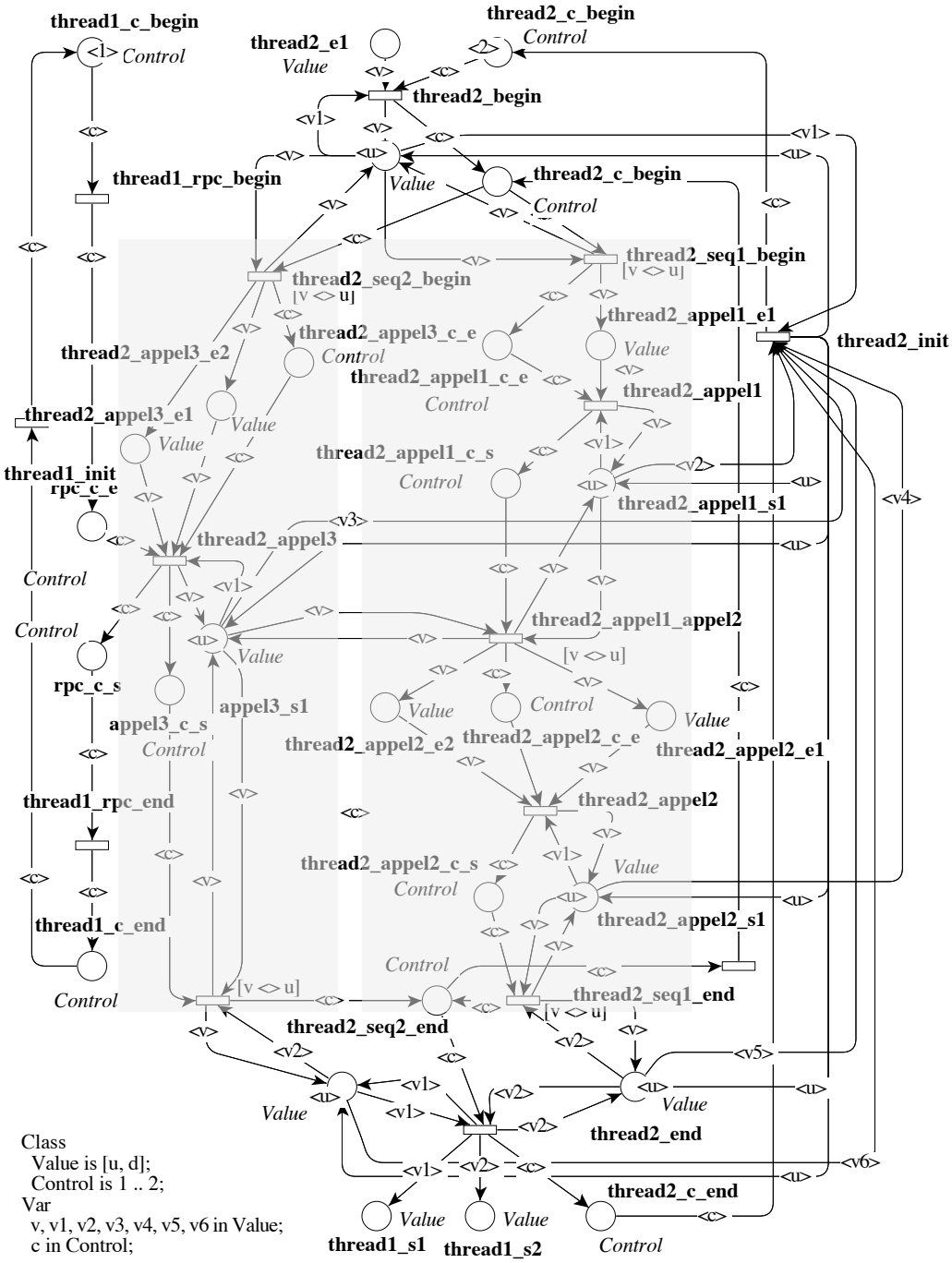


FIG. VII.7 – Modélisation de séquences d'appel, correspondant au listing VII.3

```
thread thread_b
features
  rpc : requires subprogram access sspg_b;
  e1 : in event data port donnee;
  s1 : out event data port donnee;
  s2 : out event data port donnee;
end thread_b;

thread implementation thread_b.impl
calls
  seq1 : {appel1 : subprogram sspg_a;
          appel2 : subprogram sspg_b;};
  seq2 : {appel3 : subprogram access rpc;};
connections
  cnx1 : parameter e1 -> appel1.e1;
  cnx2 : parameter e1 -> appel3.e1;
  cnx3 : parameter e1 -> appel3.e2;
  cnx4 : parameter appel1.s1 -> appel2.e1;
  cnx5 : parameter appel3.s1 -> appel2.e2;
  cnx6 : parameter appel3.s1 -> s1;
  cnx7 : parameter appel2.s1 -> s2;
end thread_b.impl;

process processus
features
  e : in event data port donnee;
  s : out event data port donnee;
end processus;

process implementation processus.impl
subcomponents
  thread1 : thread thread_a;
  thread2 : thread thread_b.impl;
connections
  subprogram access thread1.rpc -> thread2.rpc;
  event data port e -> thread2.e1;
  event data port thread2.s1 -> s;
end processus.impl;
```

Listing VII.3 – Appels de sous-programmes, traduits en réseau de Petri sur la figure VII.7

Le second *thread*, `thread2` comporte deux séquences d'appel. La première est constituée de deux appels de sous-programmes ; la seconde est en fait un appel distant au sous-programme fourni par le premier *thread*, `thread1`. La circulation des données, indiquée par les connexions, est la suivante :

- `appel1` effectue un calcul à partir de `e1` et transmet le résultat à `appel2` ;
- `appel3` (exécuté sur le premier *thread*) effectue un calcul à partir de `e1` et transmet le résultat à `appel2` ainsi qu'à la sortie `s2` du second *thread* ;
- `appel2` utilise les données fournies par `appel1` et `appel3` et renvoie le résultat vers la sortie `s1` du *thread*.

Les identificateurs de place et de transition sont construits par concaténation des identificateurs des différentes conteneurs des entités AADL. Ainsi `thread2_appel1_s1` est issu de la *feature* `s1` de l'appel `appel1` du *thread* `thread2` ; pour des raisons de lisibilité nous avons omis l'identificateur

du processus et du système global.

Nous retrouvons les éléments de la description de haut niveau des *threads* : les transitions `thread2_begin` et `thread2_end` correspondent à la transition de traitement du *thread*, les places `thread2_c_begin` et `thread2_c_end` demeurent celles de la modélisation de haut niveau, de même que la transition `thread2_init`. Il en est de même pour `thread1`.

Les deux séquences d'appels de `thread2` sont délimitées par des transitions `begin` et `end` ; il s'agit du même principe que pour les transitions encadrant le contenu des *threads*. La transition `thread2_appel1_appel2` permet de connecter les deux appels de sous-programme `thread2_appel1` et `thread2_appel2` ; elle centralise les différentes connexions AADL vers les paramètres de l'appel `thread2_appel2`.

VII-6 Intégration des descriptions comportementales

Comme nous l'avons expliqué au chapitre III, AADL décrit les aspects architecturaux des systèmes, c'est-à-dire principalement l'agencement des composants ; les aspects algorithmiques sont en dehors du cadre des modélisations. De la même façon que pour la génération de code (cf. chapitre V) une description comportementale des composants en réseau de Petri doit pouvoir être fournie ; elle doit correspondre à l'implantation en code source. Par conséquent, seuls les sous-programmes peuvent avoir une telle description comportementale. De la même façon qu'au chapitre V, nous distinguons quatre cas de sous-programmes :

VII-6.1 Composant sans modélisation comportementale

En l'absence d'éléments complémentaires, la modélisation d'un composant est celle que nous avons décrite en section VII-4.

Règle VII.20 (Traduction d'un composant sans modélisation comportementale)
Un composant sans modélisation comportementale est traduit par une simple transition.

VII-6.2 Composant possédant une description comportementale

Dans le cas d'un composant simple, l'introduction d'une modélisation comportementale consiste à substituer la transition centrale par le réseau de Petri modélisant le comportement du composant.

Règle VII.21 (Traduction d'un composant dont l'implantation est opaque)
Le sous-réseau de Petri modélisant le comportement d'un sous-programme doit comporter une place de départ `begin` et une place de fin `end`.
La transition représentant le corps du sous-programme *composant* est transformée en deux transitions `composant_begin` et `composant_end`. `composant_begin` est reliée à la place `begin` ; la place `end` est reliée à la transition `composant_end`.
Le sous-réseau comportemental doit faire apparaître les différentes places correspondant aux paramètres du sous-programme ; ces places sont reliées aux transitions `composant_begin` et `composant_end`.

Le réseau de Petri de la modélisation comportementale des composants doit faire apparaître les places initiales et finales. De cette façon, il est possible de fusionner les places contenues dans la description comportementale avec celles générées automatiquement à partir de la description AADL. La description fournie par l'utilisateur doit reprendre les noms de places utilisées dans la description AADL.

VII-6.3 Séquence d'appel pure

Dans le cas où un composant (sous-programme ou *thread*) est constitué seulement d'une séquence d'appel, nous considérons que cette séquence décrit complètement le comportement du composant – nous suivons donc les mêmes hypothèses que pour la génération de code. Le réseau de Petri issu de la séquence constitue alors la description comportementale du composant.

VII-6.4 Composant hybride

Lorsque la description AADL d'un composant comporte plusieurs séquences d'appel, toutes les combinaisons d'exécutions sont possibles. Contrairement à la génération de code, qui génère les séquences possibles mais n'en exécute aucune par défaut, nous avons vu en section VII-5.3 que la traduction en réseau de Petri permet de représenter tous les scénarios d'exécutions possibles.

Dans un cas comme dans l'autre, il est nécessaire d'adjoindre une description comportementale pour indiquer dans quel ordre et sous quelles conditions les séquences doivent s'exécuter.

Le réseau de Petri comportemental décrit alors la circulation du jeton de contrôle à travers les séquences. Afin de pouvoir en effectuer un traitement automatique, les places et les transitions du réseau comportemental doivent respecter certaines règles de nommage :

Règle VII.22 (Descriptions comportementales pour les sous-programmes hybrides)
La description de l'enchaînement des séquences d'appel à exécuter est constituée d'un réseau de Petri dont la première place doit s'appeler `begin` et la dernière place `end`. Les séquences d'appel qui doivent être exécutée sont représentées par des transition dont le nom correspond au nom AADL des séquences à coordonner.

La description complète du composant s'obtient par fusion des transitions avec les sous-réseaux des séquences.

Règle VII.23 (Coordination des séquences d'appel)
La coordination des séquences d'appel d'un sous-programme `composant` est effectuée par la fusion des transitions de la description comportementale avec les séquences correspondantes. Les places `begin` et `end` du réseau comportemental sont reliées aux transitions `composant_begin` et `composant_end`.

Les transitions `composant_begin` et `composant_end` ont été définies dans la règle VII.17.

```
1  subprogram sspg_a
2  features
3    e : in parameter;
4  end sspg_a;
5
6  subprogram sspg_b
7  features
8    e : in parameter;
9  calls
10   seq1 : {appel1 : subprogram sspg_a;};
11   seq2 : {appel2 : subprogram sspg_a;};
12 connections
13   parameter e -> appel1.e;
14   parameter e -> appel2.e;
```

```
15 end sspg_b;
```

Listing VII.4 – Description d'un sous-programme avec deux séquences d'appel, correspondant au réseau de Petri de la figure VII.8(a)

Le listing VII.4 décrit un sous-programme possédant deux séquences d'appel `seq1` et `seq2`. Chaque séquence appelle un seul sous-programme, qui prend un paramètre en entrée et ne rend rien en sortie. La traduction en réseau de Petri est illustrée sur la figure VII.8(a). Nous voyons que les deux séquences peuvent s'exécuter dans un ordre arbitraire, un nombre quelconque de fois.

La figure VII.8(b) modélise le comportement que nous souhaitons appliquer au sous-programme : nous voulons d'abord exécuter la séquence `seq1` puis la séquence `seq2`, puis terminer l'exécution du sous-programme.

La figure VII.8(c) est le résultat de la fusion du réseau issu de la description AADL et du réseau comportemental. Nous voyons que les transitions `seq1` et `seq2` sont fusionnées avec les sous-réseaux de séquences correspondantes ; les places `begin` et **`end`** sont connectée à la transition `init` pour que le réseau du sous-programme puisse contrôler le sous-réseau comportemental.

VII-7 Propriétés étudiées sur l'architecture

La traduction d'une modélisation AADL en réseau de Petri permet d'extraire les flux d'exécution et de les représenter dans un formalisme facilitant leur analyse. Il est possible d'utiliser les formules de logique temporelle afin de vérifier certains comportements sur l'architecture complète. Ces vérifications sont spécifiques à la modélisation considérée, et notamment dépendent *a priori* des modélisations comportementales fournies pour les différents composants.

Parallèlement, la vérification de certaines caractéristiques de l'architecture peut être effectuée systématiquement. Ces vérifications concernent des propriétés structurelles concernant la viabilité des flux d'exécution ; par exemple, nous voulons systématiquement nous assurer que la construction architecturale ne fait pas apparaître de blocage dans les flux d'exécution.

Le réseau de Petri généré à partir d'une description AADL modélise la circulation des données et des flux d'exécution. Nous pouvons donc l'exploiter afin d'analyser les éventuels dysfonctionnements dans l'exécution de l'architecture.

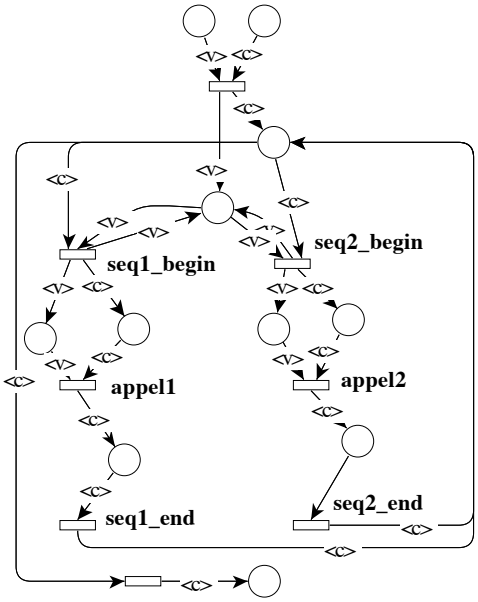
Un dysfonctionnement dans l'architecture correspond à un arrêt de la circulation des données, et peut se manifester de plusieurs façons :

- les communications entre les différents *threads* ne sont pas coordonnées ;
- l'absence d'une donnée dans l'exécution des séquences d'un *thread* ;
- la « famine » d'un *thread* ;
- le blocage de l'exécution d'un *thread* ;
- l'utilisation de données dont la valeur n'est pas déterministe.

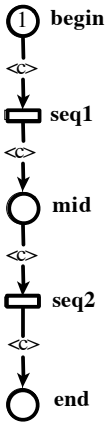
L'apparition de ces situations dans l'architecture peut être traduite par des propriétés structurelles sur le réseau ou des formules de logique temporelle. Dans la suite nous présentons les différentes formules correspondantes.

VII-7.1 Cohérence des communications

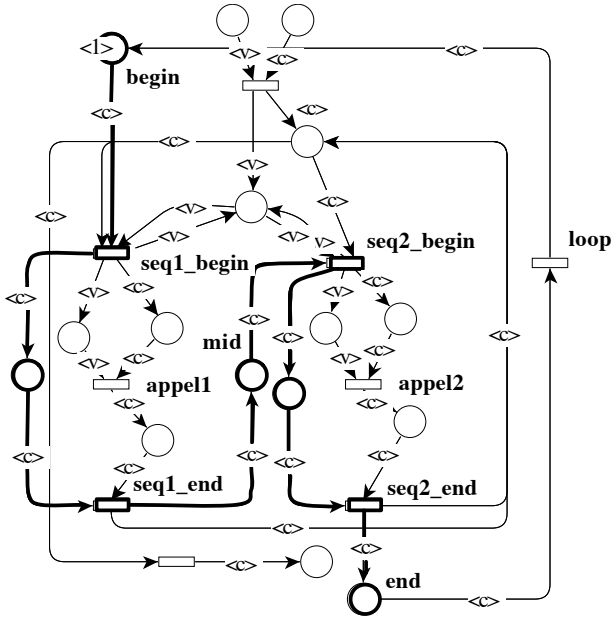
Nous avons vu en section VII-4.2.3 que les réseaux de Petri que nous construisons sont censés être bornés. Certaines configurations architecturales conduisent cependant à une surproduction de jetons qui ne peut pas être régulée par la boucle de rétroaction. Le listing VII.5 est un exemple d'une telle configuration ; la figure VII.9 représente de réseau de Petri correspondant.



(a) réseau de Petri correspondant au listing VII.4



(b) réseau de Petri de coordination



(c) réseau de Petri final

FIG. VII.8 – Réseau de Petri pour un composant hybride

```
data donnee
end donnee;

thread emetteur
features
  s1 : out event data port donnee;
properties
  Dispatch_Protocol => background;
end emetteur;

thread transmetteur
features
  e1 : in event data port donnee;
  s1 : out event data port donnee;
  s2 : out event data port donnee;
properties
  Dispatch_Protocol => aperiodic;
end emetteur;

process processus
end processus;

process implementation processus.i
subcomponents
  thread0 : thread emetteur;
  thread1 : thread transmetteur;
  thread2 : thread transmetteur;
connections
  cnx0 : event data port thread0.s1 -> thread2.e1;
  cnx1 : event data port thread1.s1 -> thread2.e1;
  cnx2 : event data port thread1.s2 -> thread2.e1;
  cnx3 : event data port thread2.s1 -> thread1.e1;
  cnx4 : event data port thread2.s2 -> thread1.e1;
end processus.i;
```

Listing VII.5 – Architecture impliquant un débordement des files d'attentes, traduit par le réseau de Petri de la figure VII.9

À chaque cycle, le *thread* `thread1` envoie systématiquement deux données au *thread* `thread2` à chaque exécution, qui se déclenchera donc deux fois, renvoyant ainsi deux jetons de contrôle à `thread1`. À chaque déclenchement, `thread2` envoie lui-même deux données à *thread1*, qui renvoie en retour deux jetons de contrôle. Il y aura donc des débordements de file d'attente au niveau des ports d'entrée de `thread1` et de `thread2`, qui pour chaque donnée consommée reçoivent deux données ; les jetons de contrôle n'empêchent pas cette situation, puisque leur nombre n'est pas borné.

Ce mauvais agencement des *threads*, dans lequel les *threads* récepteurs ne peuvent pas consommer les données aussi vite qu'elles sont produites, se traduit par un réseau non borné et une accumulation de jeton dans certaines places. Dans la mesure où nous faisons abstraction des propriétés temporelles de l'architecture, cette situation apparaît systématiquement ; elle est inhérente à l'architecture et révèle une erreur de conception.

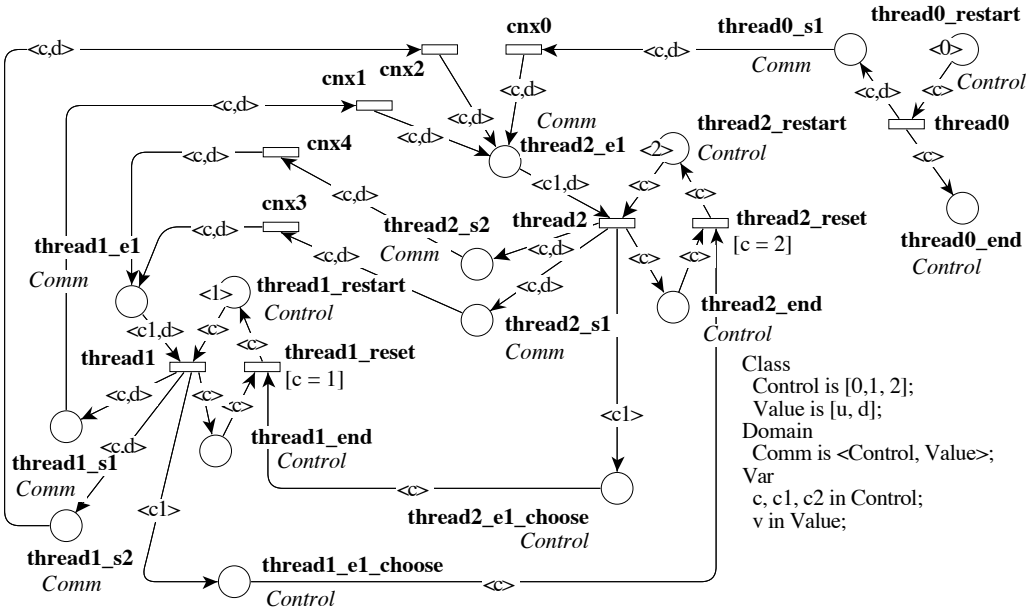


FIG. VII.9 – Réseau de Petri non borné correspondant au listing VII.5

VII-7.2 Absence de blocages

L'absence d'une donnée dans les séquences d'un *thread* peut correspondre à une dépendance mutuelle de deux sous-programmes de séquences différentes : un sous-programme prend en entrée le résultat d'un autre sous-programme, lequel prend en entrée la sortie du premier. Elle peut aussi correspondre à l'absence d'une connexion ; dans ce cas un sous-programme utilise des données dont la valeur n'est pas définie.

La « famine » d'un *thread* se produit lorsque le *thread* ne reçoit pas toutes les données nécessaires à son exécution. Il s'agit typiquement d'une connexion manquante.

Un blocage dans l'exécution du *thread* peut se produire lorsqu'un *thread* appelle un sous-programme distant fourni par un autre *thread*, lequel sous-programme fait lui-même un appel à un sous-programme distant fourni par le premier *thread* : le second appel distant restera indéfiniment en attente, puisque le premier *thread* est déjà occupé à attendre la réponse à son appel distant.

Tous ces cas correspondent à la présence d'une donnée invalide, ou l'absence d'une quelconque donnée, dans le flux d'exécution. L'absence de donnée se traduit par l'absence de jeton, empêchant ainsi la transition concernée d'être déclenchée. La présence d'une donnée invalide se traduit par la présence d'un jeton marqué *u*, empêchant également la transition concernée de se déclencher, du fait de la garde qui lui est associée. Un dysfonctionnement dans un flux de donnée se traduit donc par une transition du réseau de Petri qui ne se déclenche jamais.

Afin de vérifier l'absence de blocage dans l'exécution des *threads*, il est donc nécessaire de s'assurer que le jeton de contrôle de chaque *thread* passera de la place initiale à la place finale. Cela se traduit par la formule VII.1.

$$\forall t \in Threads, card(t\_c\_start) > 0 \rightarrow AF(card(t\_c\_end) > 0) \qquad \text{(VII.1)}$$

Pour chaque *thread* t, le fait que la place t_c_start (place de départ du jeton de contrôle t) ne soit pas vide à un instant donné implique que la place t_c_end (place de fin du jeton de contrôle

de t) finira elle-même par contenir un jeton, et ce dans tous les futurs possibles.

Avec $Threads$ l'ensemble des *threads* de l'architecture, et les places t_c_start et t_c_end correspondant respectivement aux places de début et de fin du *threads* t considéré.

De cette façon, nous pouvons nous assurer que les *threads* s'exécutent tous correctement ; si ce n'est pas le cas, il est nécessaire d'effectuer d'autres vérifications afin d'en identifier la cause.

Le blocage d'un *thread* peut être dû à l'absence de certaines données en entrée ; la formule VII.2 permet de s'assurer qu'aucune place d'entrée des *threads* n'est vide en permanence.

$$\forall p \in Places, AF(card(p) > 0) \tag{VII.2}$$

Avec $Places$ l'ensemble des places associées à chaque *thread*.

Si en revanche le blocage du *thread* est dû à un problème dans la structure de ses séquences d'appel, il est nécessaire de vérifier que celles-ci s'exécutent correctement. Dans ce cas, nous devons suivre la même démarche de vérification que pour l'extérieur des *threads* : si un jeton de contrôle atteint la première place d'une séquence, il doit atteindre la dernière place par la suite (formule VII.3).

$$\forall s \in Sequences, implies(card(s\_begin) > 0, AF(card(s\_end) > 0)) \tag{VII.3}$$

Avec $Sequences$ l'ensemble des séquences des *threads* de l'architecture.

Lorsque la séquence bloquante est identifiée, il suffit alors de vérifier que le jeton de contrôle passe dans toutes les places de contrôle des sous-programmes. Pour cela, nous devons vérifier que chaque place de contrôle de la séquence accueille exactement un jeton à un moment donné (formule VII.4).

$$\forall p \in Controles, AF(card(p) > 0) \tag{VII.4}$$

Avec $Controles$ l'ensemble des places de contrôle des sous-programmes associés aux séquences qui ne vérifient pas la formule VII.3.

Le blocage du sous-programme peut être dû à l'absence d'une donnée à l'entrée de celui-ci. Une vérification alternative consiste donc à s'assurer que toutes les places d'entrée de chaque sous-programme de la séquence considérée accueilleront un jeton à un moment donné (formule VII.5).

$$\forall p \in Entrees, AF(card(p) > 0) \tag{VII.5}$$

Avec $Entrees$ l'ensemble des places d'entrée des sous-programmes associés aux séquences qui ne vérifient pas la formule VII.3.

Les formules VII.4 et VII.5 permettent d'identifier les places d'entrée qui seront toujours vides. Ces situations sont invalides, puisqu'elles traduisent un problème dans le flux d'exécution (formule VII.4) ou dans les flux de données (formule VII.5).

VII-7.3 Déterminisme des valeurs

Le dernier cas est différent. Nous considérons qu'une valeur n'est pas déterministe lorsqu'elle est peut être définie plusieurs fois. C'est notamment le cas lorsque les sorties de plusieurs sous-programmes sont connectée à l'entrée d'un autre sous-programme ou à la sortie du *thread* correspondant. Cela peut entraîner des redéfinitions successives de la valeur d'une donnée.

Cette situation se traduit par la possibilité d'avoir plusieurs jetons dans une place appartenant à une séquence d'appel. Nous devons donc nous assurer que cette situation n'apparaît jamais (formule VII.6).

$$\forall p \in Entrees, AF(card(p) < 2) \tag{VII.6}$$

Avec *Entrees* l'ensemble des places d'entrée de chaque sous-programme de l'architecture.

Il est important de noter que les formule VII.5 et VII.6 ne peuvent pas être rassemblées en une seule formule. En effet, la formule VII.6 concerne tous les sous-programmes de l'architecture tandis que la formule VII.5 ne porte que sur les sous-programmes des séquences bloquées. Il est parfaitement acceptable qu'une entrée de sous-programme ne reçoive jamais de jeton si celui-ci appartient à une séquence qui n'est jamais exécutée.

VII-8 Conclusion

La vérification *a priori* est une phase importante de la conception des systèmes embarqués temps-réel : elle permet de s'assurer d'un certain nombre de caractéristiques de l'architecture considérée avant sa construction effective. Pour cela, il est nécessaire de pouvoir extraire une description formelle de l'architecture.

Dans ce chapitre nous avons montré qu'il était possible d'établir une relation entre une description AADL et un formalisme de vérification. Nous avons choisi les réseaux de Petri pour illustrer notre propos car ils permettent une modélisation comportementale des systèmes.

Nous avons établi des règles de traduction depuis les constructions AADL vers les réseaux de Petri. Cette traduction prend essentiellement en compte les éléments logiciels de la modélisation en AADL, et se fonde uniquement sur les spécifications d'exécutif que nous avons énoncées au chapitre IV ; ces hypothèses sont par ailleurs utilisées pour la génération de l'exécutif AADL (cf. chapitres V et VI). Nous avons notamment développé les mêmes concepts quant à l'intégration de descriptions comportementales dans les composants AADL. Il est donc possible d'établir une équivalence sémantique entre le réseau de Petri extrait d'une description AADL et le code source généré à partir de celle-ci.

L'analyse du réseau de Petri permet d'évaluer la cohérence des flux d'exécution. Pour cela, nous avons défini un certain nombre de formules et de propriétés structurelles correspondant à des propriétés qui doivent être systématiquement vérifiées, telles que l'absence de blocage dans les *threads*. Des vérifications plus spécifiques, dépendant par exemple des implantations comportementales des composants peuvent également être effectuées par l'utilisateur. En faisant abstraction des considérations temporelles de la description AADL, nous avons pu établir la vérification d'un certain nombre de propriétés inhérentes à la construction de l'architecture ; les vérifications que nous avons présentées dans ce chapitre permettent donc une vérification de la cohérence du système.

La modélisation en réseaux de Petri permet ainsi de détecter une mauvaise construction de l'architecture avant même de produire le système exécutable correspondant. Cette modélisation doit naturellement être couplée à d'autres méthodes d'analyse, telles que la simulation de l'ordonnancement des *threads*, la vérification de la taille de l'application en mémoire, de la consommation électrique, etc. Nous avons présenté ici un aspect de vérification particulier, qui est directement lié aux communications dans le système modélisé. L'utilisation d'AADL permet de centraliser toutes les informations nécessaires à l'application des différentes phases d'analyse et de génération nécessaires, ce qui permet d'avoir une représentation unique de l'architecture.

CHAPITRE VIII

Mise en pratique

DANS LES CHAPITRES PRÉCÉDENTS nous avons présenté un processus de conception basé sur AADL. Nous avons montré comment exploiter AADL pour produire du code et l'exécutif associé ; nous avons également montré comment exploiter une description AADL pour la vérification formelle des applications.

Dans ce chapitre nous appliquons notre méthodologie sur des exemples concrets afin d'en valider les performances. Nous présentons Ocarina, un compilateur pour AADL que nous avons développé pour démontrer la validité de notre approche. Nous utilisons Ocarina pour évaluer les performances d'exécution d'une application générée à partir d'AADL par rapport à une application « classique ». Nous considérons également la modélisation d'un sous-système industriel afin de d'étudier la viabilité de la transformation en réseaux de Petri.

VIII-1 Ocarina, un compilateur pour AADL

Afin de démontrer la faisabilité de notre approche, nous avons conçu et réalisé un outil permettant de manipuler les descriptions AADL pour générer du code ou des réseaux de Petri : Ocarina. Dans cette section nous décrivons son architecture et son fonctionnement.

VIII-1.1 Principes généraux

L'objectif de nos travaux est la génération d'applications vérifiées. Pour cela, nous avons dû concevoir un outil permettant la génération de différentes représentations (essentiellement du code source et des réseaux de Petri) à partir d'une description AADL.

VIII-1.1.1 Compilation et méta-modélisation

Le terme *compilation* désigne classiquement la transformation d'un programme décrit selon une certains syntaxe en une autre représentation. Un compilateur est conçu pour reconnaître une grammaire sans contexte, souvent définie en notation BNF (Backus-Naur Form). De façon simplifiée, un compilateur lit les fichiers du programme, construit un arbre de syntaxe abstraite qui subit éventuellement des transformations et des optimisations, puis génère un fichier exécutable.

La *modélisation* est une approche plus récente que la compilation. Elle est centrée sur l'arbre de syntaxe abstraite, appelé modèle, qui est déconnecté de la représentation syntaxique. La modélisation est souvent associée à la *méta-modélisation* et à la transformation de modèle. La méta-modélisation consiste à utiliser un modèle pour décrire un ensemble de modèles, en permettant de passer d'un modèle à un autre en appliquant des règles de transformation. La méta-modélisation

est le fondement de l'approche MDA, qui utilise le langage UML pour décrire les différents modèles.

VIII-1.1.2 Spécification des besoins

AADL peut être représenté selon différentes syntaxes, comme nous l'avons vu au chapitre III. La conception d'un outil pour AADL est donc plus complexe que celle d'un compilateur classique, puisque nous ne pouvons pas nous baser sur une syntaxe particulière. Par ailleurs, l'exploitation d'AADL ne se restreint pas à la génération de code : une description AADL peut être analysée (vis-à-vis des contraintes temporelles, de l'ordonançabilité, etc.), exploitée pour générer du code, utilisée à des fins de documentation, etc. Le traitement d'une description AADL ne doit donc pas dépendre de la représentation syntaxique, ni d'une exploitation particulière.

Panorama des outils existants

Parmi les différents outils existent pour la manipulation ou l'exploitation de descriptions AADL, Osate et Stood sont les plus aboutis.

Osate (Open Source AADL Tool Environment) est un logiciel développé par des membres du comité AADL [SAE, 2006a] afin de fournir un outil de référence pour la manipulation de descriptions AADL. Il s'intègre dans l'environnement de développement Eclipse dont il utilise les fonctionnalités de méta-modélisation EMF [Budinsky et al., 2003]. Osate en lui-même fournit essentiellement la gestion du méta-modèle et le parseur pour la syntaxe textuelle ; il est conçu pour accueillir des modules d'extensions qui se reposent sur le méta-modèle central, tels que l'éditeur graphique issu du projet Topcased [Farail et Gaufillet, 2005].

Stood est un logiciel commercialisé par Ellidiss [Dissaux, 2003] qui implante la méthode de conception HOOD [Dissaux, 1999] au sein d'un méta-modèle central. Stood offre des passerelles entre le méta-modèle de HOOD et les notations UML et AADL. Cette approche permet notamment de centraliser les mécanismes de production de code.

En se basant sur une approche par méta-modèle, Osate et Stood permettent de représenter l'ensemble du modèle AADL selon différentes syntaxes. Osate intègre un éditeur de texte basé sur Eclipse, ainsi qu'un éditeur graphique, fourni par le projet Topcased, tandis que Stood propose plusieurs vues graphiques (en UML, HOOD et AADL) d'un même modèle.

Champs d'application d'Ocarina

Dans le cadre de nos travaux, nous avions besoin d'un outil permettant à la fois de manipuler AADL, et également de s'intégrer dans des applications existantes, notamment GLADE afin de valider l'approche que nous développons dans la section IV-1. L'objectif était donc de concevoir un outil modulaire qui puisse être intégré dans des applications existantes ; il s'agit donc de la démarche inverse de celle qui sous-tend la conception d'Osate. Par ailleurs, Stood est un outil propriétaire, et ne peut pas être réutilisé.

L'existence de multiples syntaxes permet de structurer la description AADL en différentes vues, chacune correspondant à un aspect de la modélisation architecturale. Ainsi, la syntaxe textuelle est bien adaptée à la définition des différents composants de l'architecture : la totalité des propriétés peut être décrite avec précision, les types associés aux éléments d'interface apparaissent clairement, etc. À l'inverse, l'usage de la syntaxe graphique facilite la description du déploiement des nœuds de l'application en fournissant une approche « visuelle » et relativement intuitive de l'architecture.

Afin de concrétiser complètement notre approche, nous avons donc développé une suite d'outils pour AADL :

- Ocarina, qui constitue le programme central ;
- un mode d'édition pour l'éditeur de texte Emacs ;
- un module pour l'éditeur de diagrammes Dia.

Emacs [Free Software Foundation, 2006] est un éditeur de texte libre pouvant être programmé afin de fournir des fonctionnalités supplémentaires ; nous avons écrit une extension pour fournir la coloration syntaxique pour AADL. Dia [Clausen, 2006] est un éditeur de diagramme libre, permettant de produire différents types de schémas (électriques, logiques, SADT, UML, etc.).

Une approche basée sur un modèle est plus adaptée à l'exploitation d'AADL que l'approche de compilation traditionnelle car elle permet de s'abstraire des formalismes de représentation. Nos travaux de génération de code ou de réseaux de Petri se basent cependant largement sur des mécanismes relevant des principes de compilation ; nous ne faisons pas usage des théories relatives à la méta-modélisation. Ocarina ne prend en charge qu'un seul modèle, permettant la représentation des constructions AADL indépendamment de toute syntaxe. Il permet un certain nombre de manipulations sur ce modèle, ce qui en fait un outil plus polyvalent qu'un simple compilateur, tout en demeurant moins complexe qu'un outil de méta-modélisation.

Cela engendre une restriction au niveau de l'utilisation : Ocarina est un outil spécialisé pour AADL, qui ne peut pas facilement être adapté à d'autres formalismes. Nous conservons ainsi une approche simple ; Ocarina n'a pas besoin d'un ordinateur particulièrement performant pour s'exécuter.

VIII-1.2 Structure du modèle d'Ocarina

Ocarina manipule deux représentations d'une description AADL, introduites en section III-6.1 : le modèle, qui correspond à l'ensemble des déclarations, et l'instance de l'architecture, qui correspond à l'instanciation d'un système AADL. Au niveau de la description du modèle, nous avons rationalisé la structure syntaxique d'AADL en introduisant la notion d'*entité*. Les entités sont les éléments AADL identifiables ; elles correspondent à tous les éléments définis dans le standard AADL :

- espaces de noms :
  - l'espace de nom anonyme,
  - les paquetages,
  - les ensembles de propriétés ;
- les composants :
  - les types et les implantations de composants,
  - les groupes de ports ;
- les propriétés :
  - noms, types et constantes de propriétés,
  - associations de propriétés ;
- sous-clauses :
  - éléments d'interface (ports, spécifications des groupes de ports, sous-programmes d'interface, accès à sous-composants),
  - sous-composants,
  - séquences d'appels et appels de sous-programmes,
  - connexions,
  - modes,

  - flux ;
- annexes (bibliothèques et sous-clauses).

La structure de l'arbre d'instance reprend la plupart des constructions de l'arbre de modèle, en y introduisant quelques modifications. L'une des différences fondamentales entre le modèle et l'instance est que cette dernière décrit une hiérarchie complète de composants alors que le modèle ne décrit en définitive qu'un arbre de profondeur 2 dont les feuilles sont les déclarations de composants. Les entités présentes dans l'arbre d'instance sont plus restreintes :

- espaces de nom (paquetages) ;
- instances de composants ;
- instances de sous-clauses :
  - instances d'éléments d'interface,
  - instances de sous-composants,
  - instances de séquences d'appels et appels de sous-programmes,
  - instances de connexions,
  - instances de modes,
  - instances de flux ;
- associations de propriétés.

Les définitions de propriétés n'apparaissent pas dans l'arbre d'instance : seul les associations sont représentées. Les types et implantations de composants sont agrégés au niveau des instances. Les sous-composants sont des simples liens entre les instances de composant et leurs sous-instances. Par convention, les instances de composants sont détachées de leur éventuel paquetage de déclaration et sont insérés dans la hiérarchie ; l'espace de nom anonyme devient donc implicite.

L'utilisation de deux représentations distinctes pour les déclarations et les instances permet la création de multiples instanciations d'architectures, correspondant à différents systèmes, au sein d'une même exécution d'Ocarina. Cette séparation permet également de préserver l'ensemble des déclarations, et ainsi d'effectuer des traitements séparés sur les déclarations et les instances. Nous en illustrerons l'intérêt en section VIII-1.4.

VIII-1.3 Organisation d'Ocarina

Ocarina est constitué d'un cœur sur lequel sont branchés différents modules. Ces modules se comportent comme des applications faisant appel aux services et à l'API du cœur. Il en existe trois types, dont l'agencement est décrit sur la figure VIII.1 :

- modules d'entrée/sortie ;
- modules de transformation ;
- générateurs.

Les modules d'entrée/sortie permettent de parser des fichiers AADL ou d'afficher l'arbre du cœur selon une différentes syntaxes (textuelle, graphique, représentation XML...). Ocarina fournit actuellement un parseur/afficheur pour la syntaxe textuelle d'AADL ainsi qu'un module pour les fichiers de l'éditeur de diagramme Dia.

Les modules de transformation peuvent permettre l'expansion de l'arbre du cœur pour mettre en place des transformations d'architecture que nous avons décrites en section VI-3, page 107.

Les générateurs permettent de produire différentes représentations – en l'occurrence du code source ou des réseaux de Petri – à partir de la représentation AADL. Nous avons développé un générateur appelé Gaia [Vergnaud et Zalila, 2006] qui prend en charge les différentes générations que nous avons décrites dans les chapitres V et VII, et un autre générateur appelé PN permettant

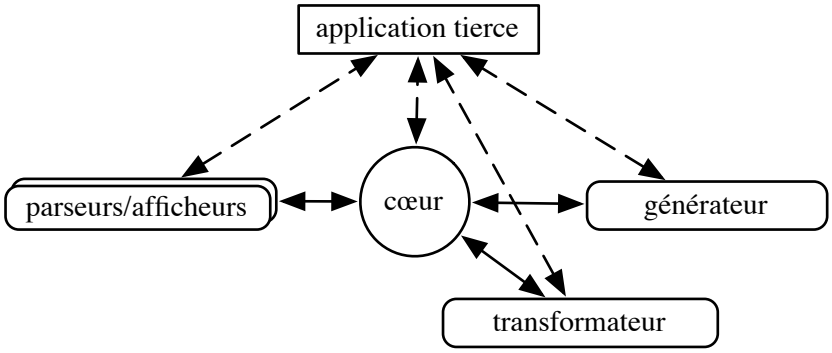


FIG. VIII.1 – Organisation des modules d'Ocarina

de produire des réseaux de Petri selon les règles que nous avons présentées au chapitre VII.

Ocarina [Vergnaud et Zalila, 2006] est une application autonome utilisable en ligne de commande permettant de produire du code exécutable, un réseau de Petri, de passer d'une représentation AADL à une autre ou simplement de vérifier la validité d'une description AADL. Tous les modules ainsi que le cœur d'Ocarina peuvent également être utilisés comme des bibliothèques logicielles, intégrées dans une application tierce afin de permettre à des outils existants de manipuler une description AADL.

VIII-1.3.1 Organisation du cœur

Le cœur permet de manipuler les deux niveaux d'une description AADL : le modèle et l'instance d'architecture. Il fournit un ensemble de fonctions pour manipuler l'arbre de modèle AADL pour le construire, le vérifier ou le consulter. D'autres fonctions permettent de manipuler l'arbre d'instance. De la même façon que pour le modèle, il est possible de construire, vérifier et consulter l'arbre d'instance. Cependant, l'arbre d'instance ne peut pas être construit directement : il est calculé à partir de l'arbre de modèle par instanciation des déclarations de composants. L'utilisation typique des fonctions du cœur est la suivante :

1. appel des fonctions de construction de l'arbre de modèle par les parseurs ou l'éventuelle application tierce ;
2. vérification de l'arbre de modèle ;
3. transformations éventuelles dans l'arbre de modèle ;
4. éventuelle instanciation de l'arbre de modèle ;
5. exploitation de l'arbre d'instance par un générateur ou une application tierce.

L'arbre de modèle permet de recueillir les informations décrites dans les différentes syntaxes AADL. La phase de vérification consiste à s'assurer que tous les composants référencés par les sous-composants et les éléments d'interface ont bien été déclarés, que les associations de propriétés sont cohérentes avec leur définitions, etc.

La racine de l'arborescence des instances de composants est le système racine décrit dans le modèle, c'est-à-dire une implantation de système dont le type n'a aucune interface.

Les composants de données associés aux éléments d'interface demeurent dans leur paquetage de déclaration dans la mesure où ils ne font pas partie de la hiérarchie d'instance. Les déclarations

de composants qui ne sont pas référencées par des sous-composants à partir du système racine ne sont pas instanciées.

VIII-1.3.2 Organisation des générateurs de code

Ocarina comprend deux modules de génération de code : l'un pour produire des systèmes exécutables, l'autre pour produire des réseaux de Petri. Ils mettent tous deux en place un arbre intermédiaire comprenant les informations pertinentes issues de l'arbre d'instance du cœur d'Ocarina.

Gaia : le générateur d'applications

Le module Gaia rassemble toutes les fonctions de génération que nous avons exposées dans les chapitres V et VI. La figure VIII.2 résume le processus de génération au sein de Gaia.

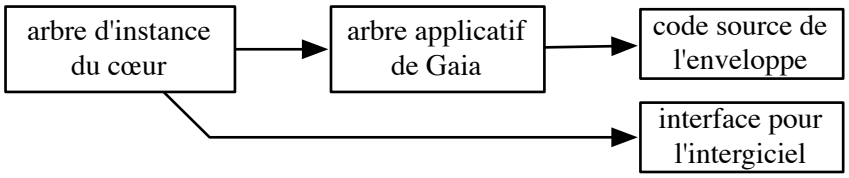


FIG. VIII.2 – Processus de génération dans Ocarina/Gaia

Gaia est un générateur d'application [1] ; l'instance d'architecture AADL est donc considérée essentiellement à travers ses composants logiciels (*threads*, sous-programmes et données) tandis que les les autres composants de l'architecture (processus, processeur, bus...) sont exploités pour la configuration de l'exécutif. Afin de faciliter l'exploitation de l'architecture pour la génération de code, Gaia est constitué de deux parties : le traducteur en code source et le générateur d'exécutif. L'organisation des deux parties de Gaia est structurée selon les processus de l'architecture, qui représentent les nœuds de l'application répartie.

La traduction en code source ne dépend que du langage cible, comme nous l'avons montré au chapitre V ; un arbre est extrait de l'arbre de Gaia pour la traduction d'AADL en code source, reproduisant les structures que nous avons décrites pour les langages impératifs dans la section V-2, page 72. Les paquetages sont transformés en espaces de noms associés à chaque processus ; ces espaces de nom contiennent les déclarations de sous-programmes et de composants de donnée correspondant à des appels ou des éléments d'interface instanciés dans chaque processus. Un générateur syntaxique est ensuite appliqué à l'arbre du traducteur afin de produire le code source dans le langage de programmation choisi. La prise en charge d'un nouveau langage consiste donc à changer la dernière étape de la génération.

La génération de l'exécutif dépend à la fois du langage cible et de l'intergiciel utilisé. Il est donc *a priori* nécessaire de construire un générateur complet pour chaque intergiciel pris en charge ; dans le cas général, la génération de l'exécutif AADL correspondant à chaque processus est par conséquent effectuée à partir de l'arbre d'instance. Dans le cadre de nos travaux, nous avons construit un générateur qui produit une personnalité applicative pour PolyORB.

[1] Gaia est l'acronyme de « Générateur d'applications depuis une instance d'architecture »

PN : le générateur de réseaux de Petri

La génération de réseaux de Petri [2] se fait également à l'aide d'un arbre intermédiaire (figure VIII.3) permettant d'avoir une représentation abstraite permettant de produire différentes syntaxes de représentation. Nous n'exploitons que les éléments logiciels, en ignorant cette fois complètement les informations de déploiement associées aux composants de plate-forme.

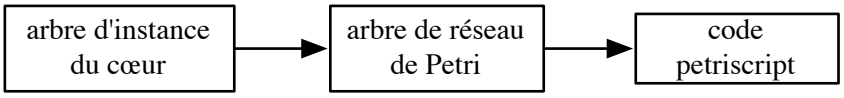


FIG. VIII.3 – Processus de génération dans Ocarina/PN

Les réseaux de Petri sont habituellement représentés sous forme graphique, peu adaptée à la génération de code. Afin de faciliter la génération des réseaux, nous nous reposons sur une syntaxe textuelle développée au laboratoire d'informatique de Paris 6, appelé PetriScript [Hamez et Renault, 2005]. PetriScript est un langage de construction de réseaux de Petri associé à la plateforme de *model checking* CPN-AMI [Kordon et Paviot-Adet, 1999] conçue au laboratoire d'informatique de Paris 6 ; il permet notamment d'automatiser les opérations de fusion entre places et transitions, ce qui facilite grandement les constructions que nous avons décrites dans le chapitre VII.

VIII-1.4 Intégration d'Ocarina dans des applications tierces

Afin de valider les solutions que nous avons exposées pour la première phase du processus de génération (décrites au chapitre IV), nous avons développé une extension au programme GLADE. Cette extension utilise Ocarina pour lire des descriptions AADL afin de fournir une syntaxe alternative à la syntaxe native de GLADE [Pautet et Tardieu, 2005].

Ocarina a également été intégré dans le projet Cheddar. Cheddar [Singhoff et al., 2005] est un analyseur d'ordonnancement développé à l'université de Bretagne occidentale ; il permet de simuler l'exécution de tâches concurrentes, selon certains protocoles d'ordonnancement. L'utilisation d'Ocarina permet à Cheddar d'extraire les informations de période et de protocole d'ordonnancement à partir d'une description AADL. L'utilisation combinée de l'application Ocarina et de Cheddar permet donc de valider à la fois le comportement d'une application répartie (par les réseaux de Petri) et certaines propriétés temporelles (par Cheddar) avant de générer le code de l'application.

L'exploitation d'AADL dans Cheddar est très éloignée de la démarche de compilation. Un outil tel que Cheddar doit pouvoir lire un modèle AADL, en déduire des instances d'architecture pour effectuer des tests d'ordonnançabilité. À partir des résultats des tests, l'utilisateur de Cheddar est susceptible de modifier le modèle pour changer certaines déclarations de composants ou ajuster des valeurs de propriétés. À l'issue des différents tests et modifications, l'ensemble des déclarations doit pouvoir être sauvegardé dans un fichier AADL. Ce processus de raffinement rend nécessaire la préservation de l'arbre du modèle dans Ocarina vis-à-vis des différents arbres d'instance.

L'intégration d'Ocarina dans ces deux applications illustre l'approche collaborative induite par AADL : un processus de conception reposant sur AADL comme langage fédérateur permet de faire intervenir différents outils spécialisés afin de couvrir plusieurs aspects de la vérification

[2] PN est l'acronyme de *Petri Nets*

d'architecture et de la génération d'application. L'utilisation d'applications telles que Cheddar permet un ajustement de la description architecturale, qui s'intègre au cycle de raffinement illustré sur la figure IV.2, page 61.

VIII-2 Évaluation des performances d'exécution

Nous souhaitons évaluer l'impact de la génération automatique sur les performances d'exécution. Pour cela, nous comparons le temps d'exécution d'une application générée à partir d'une modélisation en AADL à celui d'applications-témoins.

Afin d'évaluer uniquement l'impact de la génération automatique sur les performances d'exécution, nous devons utiliser des intergiciels semblables pour toutes les applications. L'implantation de PolyORB pour CORBA a fait l'objet de nombreuses comparaisons de performances, et constitue donc une bonne implantation de référence pour évaluer les performances de nos travaux. Nos expérimentations consistent donc à mesurer les temps d'exécution de l'application générée automatiquement et d'une application équivalente basée sur CORBA. Nous exécutons les deux types d'application sur la même configuration de la couche neutre de PolyORB et les mêmes personnalités protocolaires ; seules les personnalités applicatives diffèrent. Ces expérimentations nous permettent d'évaluer le coût en terme de performances à l'exécution de l'automatisation de la génération de l'application.

VIII-2.1 Définition de l'architecture AADL

Bien qu'il soit possible de représenter différents modèles de répartition en AADL (cf. section IV-5.3, page 66), une architecture décrite en AADL repose plus naturellement sur un mécanisme de passage de messages. Nous avons d'ailleurs montré que les appels de procédure distante et les objets partagés ne peuvent être modélisés correctement qu'en étendant la syntaxe initiale d'AADL.

Nous mettons en place une architecture basée sur deux processus se renvoyant des messages. Les deux processus s'exécutent sur le même processeur (c'est-à-dire la même machine) afin de ne pas être perturbés par des transferts sur le réseau.

Chaque processus contient un *thread* apériodique recevant et émettant un message (voir listing VIII.1). Les spécifications de l'exécutif impliquent qu'un *thread* apériodique est déclenché par l'arrivée d'un message puis émet les données en sortie à la fin de son cycle de traitement.

```
1  data Donnee
2  properties
3    Language_Support::Data_Format => Integer;
4  end Donnee;
5
6  subprogram Application
7  features
8    e : in parameter Donnee;
9    s : out parameter Donnee;
10 end Application;
11
12 subprogram implementation Application.i
13 properties
14   Source_Language => Ada95;
15   Source_Name => "Application";
```

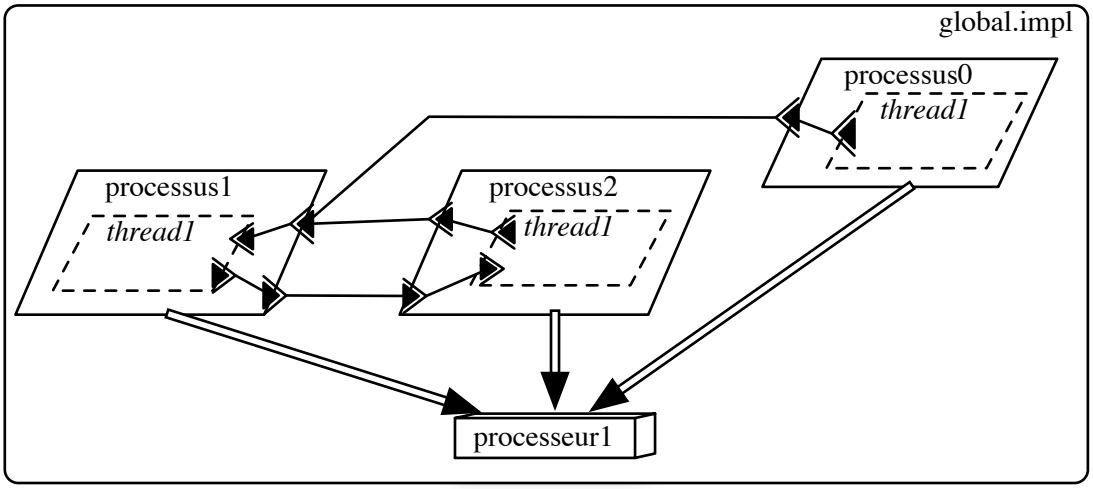


FIG. VIII.4 – Architecture de l'application de test

```
16   Source_Text => ("Repository");
17 end Application.i;
18
19 thread Noeud
20 features
21   e : in event data port Donnee;
22   s : out event data port Donnee;
23 properties
24   Dispatch_Protocol => aperiodic;
25 end Noeud;
26
27 thread implementation Noeud.i
28 calls
29   {appel : subprogram Application.i;};
30 connections
31   parameter e -> appel.e;
32   parameter appel.s -> s;
33 end Noeud.i;
34
35 process Processus_Applicatif
36 features
37   e : in event data port Donnee;
38   s : out event data port Donnee;
39 end Processus_Applicatif;
40
41 process implementation Processus_Applicatif.i
42 subcomponents
43   thread1 : thread Noeud.i;
44 connections
45   event data port e -> thread1.e;
46   event data port thread1.s -> s;
```

```
47 end Processus_Applicatif.i;
```

Listing VIII.1 – Description des processus de l'application de test

Afin d'initier ces processus, nous décrivons un processus tiers qui déclenchera les autres (listing VIII.2). Ce processus contient un *thread* en tâche de fond qui émet un message destiné à déclencher l'un des processus de l'application.

```
49 subprogram Initiateur
50 features
51   s : out parameter Donnee;
52 end Initiateur;
53
54 subprogram implementation Initiateur.i
55 properties
56   Source_Language => Ada95;
57   Source_Name => "Initiateur";
58   Source_Text => ("Repository");
59 end Initiateur.i;
60
61 thread Lanceur
62 features
63   s : out event data port Donnee;
64 properties
65   Dispatch_Protocol => background;
66 end Lanceur;
67
68 thread implementation Lanceur.i
69 calls
70   {appel : subprogram Initiateur.i;};
71 connections
72   parameter appel.s -> s;
73 end Lanceur.i;
74
75 process Processus_Lanceur
76 features
77   s : out event data port Donnee;
78 end Processus_Lanceur;
79
80 process implementation Processus_Lanceur.i
81 subcomponents
82   thread1 : thread Lanceur.i;
83 connections
84   event data port thread1.s -> s;
85 end Processus_Lanceur.i;
```

Listing VIII.2 – Description du processus d'initialisation

Nous assemblons deux processus applicatifs, ainsi qu'un processus d'initialisation, au sein du système global ; tous les processus sont attachés au même processeur (listing VIII.3). La figure VIII.4 illustre l'architecture générale de l'application.

```
87 processor Processeur
88 properties
89   Arao::Location => "127.0.0.1";
```

```
90 end Processeur;
91
92 system Client_Serveur
93 end Client_Serveur;
94
95 system implementation Client_Serveur.i
96 subcomponents
97   lanceur : process Processus_Lanceur.i {Arao::Port_Number =>
       10000;};
98   noeud1 : process Processus_Applicatif.i {Arao::Port_Number =>
       10002;};
99   noeud2 : process Processus_Applicatif.i {Arao::Port_Number =>
       10003;};
100  processeur : processor Processeur;
101 connections
102  event data port lanceur.s -> noeud1.e;
103  event data port noeud1.s -> noeud2.e;
104  event data port noeud2.s -> noeud1.e;
105 properties
106  Actual_Processor_Binding => reference processeur applies to
       lanceur;
107  Actual_Processor_Binding => reference processeur applies to
       noeud1;
108  Actual_Processor_Binding => reference processeur applies to
       noeud2;
109 end Client_Serveur.i;
```

Listing VIII.3 – Déploiement de l'application de test

L'implantation des sous-programmes des nœuds applicatifs effectue une addition. L'essentiel du temps de traitement correspond donc à l'exécution des mécanismes de la couche applicative AADL et des composants de PolyORB. Les deux nœuds principaux de l'application sont symétriques ; nous n'avons pas la notion de client et de serveur. Une seule description comportementale est donc nécessaire (listing VIII.4).

```
1  with Partition;
2  with Ada.Text_IO;
3  with Ada.Calendar;
4
5  package body Repository is
6
7    Date_Start, Date_End : Ada.Calendar.Time;
8
9    procedure Initiateur (s : out Partition.Donnee) is
10     use Ada.Text_IO;
11   begin
12     Put_Line ("initiation");
13     s := 1;
14   end Initiateur;
15
16   procedure Application (e : in Partition.Donnee; s : out
         Partition.Donnee) is
17     use Ada.Calendar;
18     use Ada.Text_IO;
```

```
19      use Partition;
20
21      Elapsed_Time : Duration;
22    begin
23      if e = 1 then
24        Date_Start := Clock;
25        s := e + 1;
26      elsif e = 20001 then
27        Date_End := Clock;
28        Elapsed_Time := Date_End - Date_Start;
29        Put_Line ("Elapsed time : " & Duration'Image (Elapsed_Time))
              ;
30        s := 0;
31      else
32        s := e + 1;
33      end if;
34    end Application;
35
36 end Repository;
```

Listing VIII.4 – criptions comportementales des composants de l'application

L'utilisateur n'a pas à fournir de code source supplémentaire pour les différents nœuds de l'application ; les procédures d'initialisation sont générées automatiquement à partir de la description AADL.

VIII-2.2 Mise en place des applications-témoins

CORBA privilégie des architectures basées sur des objets répartis mettant en jeu des appels de méthodes distantes. L'équivalent « naturel » de l'architecture AADL que nous étudions est donc formé d'un client envoyant des entiers à un serveur et récupérant une réponse à chaque requête.

Cette organisation d'application ne correspond cependant pas exactement à notre modélisation AADL. Afin d'effectuer une comparaison complète, nous établissons alors deux applications-témoin :

- une première application basée sur un appel de méthode distante ;
- une seconde application mettant en place des méthodes *oneway*, pouvant être assimilées à des envois de messages.

VIII-2.2.1 Application CORBA basée sur l'invocation de méthode

Notre première application-témoin est constituée de deux objets, un client et un serveur. Le client invoque une méthode fournie par le serveur, qui renvoie la réponse. Le listing VIII.5 détaille les interfaces des deux objets, en IDL CORBA.

```
1 interface Echo {
2   long echoInt (in long Mesg);
3 };
```

Listing VIII.5 – Interfaces de l'application-témoin basée sur un modèle client/serveur CORBA

L'utilisateur doit fournir le code source décrivant la description comportementale de la méthode `echoInt` (listing VIII.6).

```
with Ada.Text_IO;
with CORBA;

with Echo.Skel;
pragma Warnings (Off, Echo.Skel);
-- No entity from Echo.Skel is referenced.

package body Echo.Impl is

   function EchoInt
     (Self : access Object;
      Mesg : in   CORBA.Long)
     return CORBA.Long
   is
      use CORBA;
      pragma Warnings (Off);
      pragma Unreferenced (Self);
      pragma Warnings (On);
   begin
      return Mesg + 1;
   end EchoInt;

end Echo.Impl;
```

Listing VIII.6 – Code source de la méthode CORBA

L'utilisateur doit également écrire le code de lancement du serveur et du client. Le programme du serveur initialise l'ORB, crée un objet CORBA et l'inscrit auprès de l'ORB ; la référence correspondante est affichée à l'écran ou inscrite dans un fichier. Le programme du client initialise également l'ORB, récupère la référence de l'objet serveur et invoque la méthode `echoInt` sur cet objet. Le listing VIII.7 illustre le code source du client.

```
with Ada.Command_Line;
with Ada.Text_IO;
with Ada.Calendar;
with CORBA;
with CORBA.ORB;

with Echo;

with PolyORB.Setup.Client;
pragma Warnings (Off, PolyORB.Setup.Client);

procedure Client is
   use Ada.Command_Line;
   use Ada.Text_IO;
   use Ada.Calendar;
   use CORBA;

   Sent_Msg, Rcvd_Msg : CORBA.Long;
   myecho : Echo.Ref;
   Date_Start, Date_End : Time;
   Elapsed_Time : Duration;

```

```
23 begin
24    CORBA.ORB.Initialize ("ORB");
25    if Argument_Count /= 1 then
26       Put_Line ("usage : client <IOR_string_from_server>|-i");
27       return;
28    end if;
29
30    -- Getting the CORBA.Object
31
32    CORBA.ORB.String_To_Object
33      (CORBA.To_CORBA_String (Ada.Command_Line.Argument (1)), myecho);
34
35    -- Checking if it worked
36
37    if Echo.Is_Nil (myecho) then
38       Put_Line ("main : cannot invoke on a nil reference");
39       return;
40    end if;
41
42    -- Sending message
43
44    Put_Line ("Start experiment");
45
46    loop
47       Date_Start := Clock;
48       Sent_Msg := 1;
49
50       for i in 1 .. 10000 loop
51          Rcvd_Msg := Echo.echoInt (myecho, Sent_Msg + 1);
52       end loop;
53
54       Date_End := Clock;
55       Elapsed_Time := Date_End - Date_Start;
56       Put_Line ("Elapsed time is : " & Duration'Image (Elapsed_Time))
            ;
57    end loop;
58
59 exception
60    when E : CORBA.Transient =>
61       declare
62          Memb : CORBA.System_Exception_Members;
63       begin
64          CORBA.Get_Members (E, Memb);
65          Put ("received exception transient, minor");
66          Put (CORBA.Unsigned_Long'Image (Memb.Minor));
67          Put (", completion status: ");
68          Put_Line (CORBA.Completion_Status'Image (Memb.Completed));
69
70       end;
71 end Client;
```

Listing VIII.7 – Code source du client CORBA

Par rapport à l'application AADL, nous émettons moitié moins de communications ; en effet, chaque communication est un couple requête/réponse alors que les communications AADL sont unidirectionnelles.

Une grande partie du code du client est consacrée à la mise en place de l'ORB ; la partie applicative elle-même est relativement restreinte en comparaison.

Nous exécutons le client et le serveur sur la même machine, afin de limiter les perturbations liées à une éventuelle activité sur la réseau. Ce choix de déploiement se fait à l'exécution du client et du serveur, en fournissant l'IOR du serveur au client.

VIII-2.2.2 Application CORBA basée sur des méthodes *oneway*

L'utilisation de méthodes *oneway* nous permet de mettre en place un mécanisme de passage de messages. Nous pouvons alors adopter une architecture semblable à celle utilisée pour l'application AADL, basée sur trois nœuds.

La description des interfaces des nœuds en IDL est représentée sur le listing VIII.8.

```
interface Addition {
  oneway void Add (in long Mesg);
};
```

Listing VIII.8 – Interfaces de l'application-témoin basée sur des méthodes *oneway* CORBA

Le code des deux nœuds applicatifs est comparable à l'implantation que nous avons réalisée pour l'application AADL ; il est représenté sur le listing VIII.9.

```
with Ada.Text_IO;
with CORBA;
with CORBA.ORB;
with Ada.Calendar;

with Addition.Skel;
pragma Warnings (Off, Addition.Skel);
-- No entity from Echo.Skel is referenced.

package body Addition.Impl is

   myadd : Addition.Ref;
   Date_Start, Date_End : Ada.Calendar.Time;
   Elapsed_Time : Duration;

   procedure Add
     (Self : access Object;
      Mesg : in   CORBA.Long)
   is
      use CORBA;
      use Ada.Calendar;
      pragma Warnings (Off);
      pragma Unreferenced (Self);
      pragma Warnings (On);

      Elapsed_Time : Duration;
   begin
      if Mesg = 1 then
```

```
29          Date_Start := Clock;
30          Addition.Add (myadd, Mesg + 1);
31       elsif Mesg = 200001 then
32          Date_End := Clock;
33          Elapsed_Time := Date_End - Date_Start;
34          Ada.Text_IO.Put_Line
35            ("Elapsed time : " & Duration'Image (Elapsed_Time));
36          Addition.Add (myadd, 0);
37       else
38          Addition.Add (myadd, Mesg + 1);
39       end if;
40    end Add;
41
42    procedure Set_Target is
43       Ior : String (1 .. 1024);
44       L : Natural;
45    begin
46       Ada.Text_IO.Put ("target ior =");
47       Ada.Text_IO.Get_Line (Ior, L);
48       CORBA.ORB.String_To_Object
49         (CORBA.To_CORBA_String (Ior (1 .. L)), myadd);
50    end Set_Target;
51
52 end Addition.Impl;
```

Listing VIII.9 – Code source de la méthode pour le passage de messages CORBA

La procédure `Set_Target` permet de spécifier la destination des messages ; elle doit être appelée avant l'exécution du nœud initiateur afin de mettre en place les références entre les deux nœuds de l'application.

L'implantation du nœud initiateur est semblable à celle du client CORBA de la première application-témoin ; l'implantation des nœuds applicatifs est semblable à celle du serveur CORBA.

VIII-2.3 Résultats

Nous effectuons deux séries de mesures, pour 20 000 messages et 200 000 messages. De cette façon, nous pouvons détecter un éventuel délai d'exécution fixe, correspondant à la mise en place des nœuds applicatifs. Nous nous servons d'un PC mono-processeur Athlon 2500+. Les résultats des mesures sont rassemblés dans le tableau VIII.1.

communications	20 000	200 000
PolyORB/AADL	4,3 s	43,5 s
PolyORB/CORBA	3,2 s	32,5 s
rapport AADL/CORBA	+37%	+33%
PolyORB/CORBA *oneway*	2,7 s	27,1 s
rapport AADL/CORBA *oneway*	+63%	+60%

TAB. VIII.1 – Comparaison des temps d'exécution des applications AADL et CORBA pour la transmission d'entiers

VIII-2.3.1 Analyse quantitative

Nous pouvons observer que l'application AADL est légèrement plus lente – de 30 à 60 % – dans le traitement des communications que les implantations basées sur CORBA. Cette perte de performance s'explique simplement, à partir de l'organisation de la couche d'interface enveloppe/intergiciel que nous avons présentée en section VI-2.1, 102.

En effet, les données transportées dans chaque requête sont dans un premier temps extraites puis stockées dans une tampon de mémoire ; dans un deuxième temps elles sont récupérées et envoyées à l'application. Dans le cas de nos mesures, les *threads* AADL n'ont qu'un seul port d'entrée, correspondant à une seule requête ; la gestion du tampon de mémoire engendre donc un surcoût. En comparaison, les applications-témoin basées sur CORBA transfèrent directement les données ; elles sont donc plus efficaces. Dans la mesure où les applications font des calculs très rapides – des additions – les temps de traitement des données ressortent clairement.

Nous avons donc étudié une situation qui est très défavorable pour AADL ; nous pouvons considérer qu'il s'agit du pire cas. Une situation favorable consisterait à associer plusieurs ports d'entrée aux *threads* AADL ; les tampons de mémoire trouveraient alors leur justification pour synchroniser l'arrivée des différentes données. Une solution basée sur CORBA nécessiterait le développement d'une architecture semblable à celle que nous avons présentée en section VI-2.1, qui s'exécuterait alors au-dessus de la personnalité CORBA de PolyORB. L'application résultante serait alors nécessairement plus lente que l'application AADL, qui s'adresse directement à la couche neutre.

Par ailleurs, nous comparons ici une personnalité qui est à l'état de prototype (AADL) à une personnalité développée et optimisée depuis plusieurs années (CORBA). Une optimisation sur le code de la personnalité AADL permettrait naturellement de réduire l'écart de performances.

VIII-2.3.2 Analyse qualitative

Parallèlement aux mesures de performances, il est intéressant de considérer la facilité de construction des applications, de façon qualitative.

L'utilisation d'AADL permet de rassembler tous les éléments relatifs à la topologie de l'application. Le redéploiement ou l'ajout de nœuds s'en trouve grandement facilité.

La quantité de code à écrire est également réduite par rapport à une approche CORBA, puisque l'initialisation des nœuds est produite automatiquement ; l'utilisateur doit seulement fournir la description des algorithmes correspondant aux sous-programmes AADL.

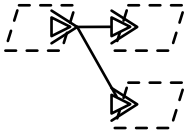
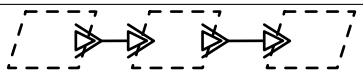
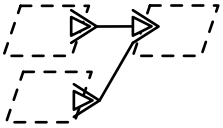
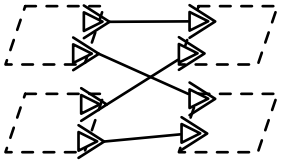
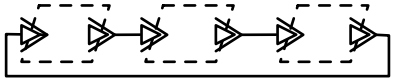
En contrepartie, la description du déploiement est plus importante puisqu'elle rassemble les interfaces des nœuds applicatifs (de la même façon que pour le listing IDL VIII.5) ainsi que leur structure et les informations de déploiement. Ce surcoût ne se manifeste que lorsque l'application à mettre en place est très simple. Une description précise du déploiement devient de toute façon nécessaire dès que l'architecture atteint une certaine complexité.

VIII-3 Évaluation de la transformation en réseaux de Petri

L'évaluation des performances de la transformation d'une description AADL en réseau de Petri consiste à considérer le temps d'analyse pour une architecture donnée. Ce temps d'analyse est reflété par le nombre d'états associés au réseau de Petri d'une description architecturale donnée.

VIII-3.1 Motifs AADL de base

Nous nous sommes dans un premier temps intéressés au calcul du nombre d'état pour des configurations de *threads* simples. Nous avons pour cela étudié différents assemblages de *threads* AADL, qui sont synthétisés sur le tableau VIII.2.

configuration d'architecture	nombre d'états du réseau
	24
	32
	32
	32
	1

TAB. VIII.2 – Nombre d'états du réseau de Petri correspondant à diverses configurations architecturales

Les deux premières architectures n'ont pas le même nombre d'états, bien qu'elles soient constituées du même nombre de *threads* et de connexions. Par ailleurs, trois architectures ont le même nombre d'états alors que leur configuration est différente.

Aucun calcul simple ne permet donc de prévoir le nombre exact d'états correspondant à une architecture donnée à partir du nombre de *threads*, dans la mesure où il dépend en grande partie de l'agencement des différents éléments.

La dernière architecture fait apparaître un cycle dans le flux de données. Le réseau de Petri correspondant ne contient qu'un seul état ; le réseau est bloqué. Cette situation peut être interprétée de deux façon, selon la nature des *threads* que nous considérons. S'il s'agit de *threads* périodiques, alors le blocage du réseau traduit le fait que le premier *thread* à se déclencher utilisera une donnée dont la valeur est indéterminée, ce qui peut entraîner des erreurs dans l'application modélisée. Si les *threads* sont apériodiques, alors aucun d'eux ne se déclenchera, puisqu'aucun d'eux ne recevra de donnée au démarrage de l'application.

Le nombre d'état du réseau de Petri correspondant à une modélisation AADL est difficile à prévoir ; il a cependant tendance à croître avec le nombre de *threads* et de connexions. Un nombre d'états relativement faible dans une architecture complexe peut traduire la présence de blocages dans la totalité ou une partie de l'architecture.

VIII-3.2 Exemple de taille réelle

La section précédente a permis d'évaluer le nombre d'état générés pour différents assemblages de composants. Nous nous intéressons maintenant à une modélisation de taille réelle afin d'évaluer la viabilité de la transformation en réseaux de Petri.

AADL étant un langage encore assez récent, peu de descriptions d'architectures sont accessibles publiquement. Par ailleurs, la plupart d'entre elles ne prennent pas en considération la problématique de la génération automatique d'application ; elles représentent des constructions qui ne correspondent pas aux spécifications d'exécutif que nous avons faites au chapitre IV. Les exemples réellement exploitables sont donc à l'heure actuelle peu nombreux.

VIII-3.2.1 Description de l'architecture

Nous prenons comme objet d'étude la modélisation d'un système d'affichage d'une cabine de pilotage d'avion [Statezni, 2004]. Cette architecture est composée de 90 *threads* s'exécutant sur 13 processeurs. La modélisation AADL qui en a été faite avait pour objectif de démontrer la viabilité d'AADL comme langage de description. Cette modélisation fait 21 000 lignes ; elle constitue donc un exemple significatif en terme de taille d'architecture.

Il s'agit d'une modélisation de haut niveau, ne comprenant que la description du déploiement des *threads* AADL sur les processus. La description des composants applicatifs n'est pas abordée ; les différentes propriétés dont nous avons présenté la vérification en section VII-7 ne peuvent donc pas être vérifiées, puisque la description architecturale n'est pas assez précise.

VIII-3.2.2 Expérimentations

Nous exploitons la description AADL pour générer le réseau de Petri correspondant aux flux d'exécution entre les différents *threads* de l'architecture. L'application de nos règles de traduction produit un réseau constitué de 1564 places, 572 transitions et 2576 arcs. Il fait intervenir 166 variables de contrôle ; ces variables découlent du fait que certains *threads* possèdent de nombreux ports d'entrée. Ce grand nombre de variables entraînent la génération d'un trop grand nombre d'états, ce qui sature les outils actuels. Il est donc impossible d'analyser une telle architecture avec les techniques actuelles.

Pour pouvoir analyser l'architecture, il est nécessaire de la réduire, tout en préservant sa sémantique vis-à-vis de la transformation en réseau de Petri. Nous nous appuyons pour cela sur la sémantique d'exécution que nous avons associée aux *threads* AADL : un *thread* lit toutes ses données d'entrée, effectue un traitement puis émet toutes ses données de sortie.

L'architecture que nous étudions se caractérise par la présence de *threads* possédant de nombreux ports d'entrée et de sortie. Nous exploitons cette caractéristique, dans le cadre de la sémantique d'exécutif que nous avons définie, pour réduire la taille du réseau généré.

Première réduction

En section VII-3, page 125, nous avons établi que tous les types de données étaient confondus lors de la transformation en réseau de Petri. Tous les ports des *threads* sont donc considérés comme étant associés à un unique composant de donnée.

Par ailleurs, selon la règle IV.6, énoncée en page 69, un *thread* AADL émet toutes les données sortantes en même temps, à la fin de son exécution.

La modélisation en réseau de Petri correspondant à un *thread* AADL émet donc systématiquement un jeton dans chacune de ses places de sortie, correspondant aux ports de sortie. Toutes les données sont censées être émises et transmises en même temps. Dans la transformation en réseau de Petri, nous pouvons donc agréger tous les ports de sortie d'un *thread* donné en un seul. Les connexions issues des différents ports de sortie sont alors rattachées à l'unique port de sortie de chaque *thread*. De cette façon nous éliminons des états artificiels qui correspondent à des transmissions différées des jetons de données.

Cette première réduction permet d'obtenir un autre réseau de Petri constitué cette fois de 799 places, 241 transitions et 1138 arcs – c'est-à-dire environ moitié moins que le premier réseau. Nous avons cependant toujours 166 variables de contrôle, qui ne dépendent que des ports d'entrée des *threads*. Ce deuxième réseau génère encore trop d'états pour pouvoir être analysé avec les outils actuels.

Seconde réduction

Afin de produire un réseau analysable, nous nous intéressons ensuite aux ports d'entrée des *threads*.

L'architecture que nous considérons possède une propriété remarquable : elle est organisée en deux grands groupes de composants organisés de façon plus ou moins symétrique, qui communiquent entre eux. Il s'ensuit qu'un grand nombre de connexions issues d'un *thread* donné pointent vers les ports d'entrée d'un seul autre *thread*.

Par rapport aux propriétés que nous étudions, ces différentes connexions parallèles sont censées transmettre les données de façon synchronisée. Le fait d'avoir plusieurs transitions de réseau de Petri implique que les jetons peuvent circuler dans le réseau indépendamment les uns des autres. Cette situation engendre là aussi des états artificiels, qui ne sont pas pertinents pour notre étude.

Toujours en vertu la règle IV.6, nous pouvons donc agréger ces connexions parallèles et les ports d'entrée correspondants. Nous réduisons alors grandement le nombre d'interfaces de chaque *thread*, qui ne possèdent alors plus que quelques ports d'entrée. Le nouveau réseau que nous générons ne possède plus que 356 places, 241 transitions et 695 arcs. Il possède 3 variables de contrôle. Ce réseau possède de l'ordre de 10^{21} états. Bien que ses dimensions soient encore importantes, un tel système peut être exploité – du moins en partie – par des outils d'analyse. Un si grand nombre d'état montre notamment que le système n'est pas bloqué, ce qui constitue une information intéressante vis-à-vis de sa validité.

VIII-3.3 Étude de l'application de test

Afin d'avoir une expérimentation complète, nous avons généré le réseau de Petri correspondant à l'application de test décrite en section VIII-2.1. Le réseau produit 11 états, ce qui permet une analyse extrêmement rapide.

À titre d'expérience, nous avons considéré une configuration d'architecture alternative (listing VIII.10) dans laquelle nous avons supprimé le processus initiateur.

```
95 system implementation Client_Serveur.i2
96 subcomponents
97   noeud1 : process Processus_Applicatif.i {Arao::Port_Number =>
         10002;};
98   noeud2 : process Processus_Applicatif.i {Arao::Port_Number =>
         10003;};
99   processeur : processor Processeur;
```

```
100 connections
101   event data port noeud1.s -> noeud2.e;
102   event data port noeud2.s -> noeud1.e;
103 properties
104   Actual_Processor_Binding => reference processeur applies to
          noeud1;
105   Actual_Processor_Binding => reference processeur applies to
          noeud2;
106 end Client_Serveur.i2;
```

Listing VIII.10 – Configuration alternative de l'application de test

L'analyse du réseau de Petri correspondant fait immédiatement apparaître que le marquage initial est bloqué, traduisant le fait qu'en l'absence du processus initiateur, les deux processus de l'architecture ne se déclenchent pas.

VIII-4 Conclusion

Ce chapitre a été consacré à l'évaluation de notre approche de génération.

Au cours de nos travaux nous avons développé une application, Ocarina, qui implante les différentes transformations que nous avons décrites aux chapitres V, VI et VII. Ocarina constitue donc un compilateur pour AADL.

Pour mesurer les performances d'exécution, nous avons comparé une application AADL simple et différentes contreparties basées sur CORBA. Pour pouvoir établir facilement une comparaison, nous avons dû nous placer dans un contexte correspondant à une utilisation classique de CORBA. Cette situation se révèle en l'occurrence défavorable à l'utilisation d'AADL ; notre application AADL est donc légèrement moins efficace. Nos mesures doivent donc être considérées comme une évaluation d'un pire cas.

Une architecture « naturelle » pour AADL, mise en place avec CORBA ou tout autre implantation d'intergiciel impliquerait le développement d'une couche logicielle pour synchroniser l'arrivée des données des ports, semblable à celle que nous avons présentée en section VI.1, page 102, et qui constitue la personnalité AADL pour PolyORB. Le développement d'une telle application pour PolyORB/CORBA serait donc nécessairement plus coûteuse qu'une solution basée sur la personnalité AADL, puisqu'elle ajouterait les traitements de la personnalité CORBA à la couche de synchronisation.

La génération de réseau de Petri permet de mettre en évidence certaines propriétés structurelles comme le blocage des architectures. Le nombre d'état des réseaux produits dépend à la fois du nombre de *threads* AADL et de leur inter-connexions ; il est assez difficile de prévoir le nombre d'états.

L'étude d'une architecture de grande taille nécessite *a priori* un travail de préparation afin de réduire la modélisation AADL, tout en conservant la sémantique d'exécution. Certaines opérations de réduction peuvent effectuées de façon systématique ; d'autres s'appuient sur des caractéristiques remarquables de l'architecture considérée.

Nous avons étudié ici un système dans lequel les *threads* étaient fortement couplés, ce qui a permis de simplifier considérablement l'architecture. Nous n'avons cependant pas eu à prendre en compte les séquences d'appels de l'implantation des *threads*, qui auraient généré beaucoup d'états. Dans une situation moins favorable, il peut être nécessaire de ne considérer que des sous-parties de l'architecture.

CHAPITRE IX

Conclusions et perspectives

Pour le savant, croire la science achevée est toujours une illusion aussi complète que le serait pour l'historien de croire l'histoire terminée.

Louis DE BROGLIE, *in* Physique et microphysique

La mise en place d'un système réparti repose en général sur la construction d'une couche applicative particulière – l'intergiciel – pour prendre en charge les communications inter-nœuds. L'intergiciel doit pouvoir fournir tous les mécanismes de communication requis par l'application.

Nos travaux se sont particulièrement intéressés aux applications pour les systèmes temps-réel répartis embarqués (TR^2E), qui doivent respecter un certain nombre de contraintes (temps d'exécution, taille mémoire, ressources disponibles, etc.). Ces contraintes doivent être respectées par tous les composants applicatifs, et en particulier par l'intergiciel.

Les implantations classiques d'intergiciel ne permettent pas la prise en compte complète de telles contraintes. Certains travaux visent à permettre l'intégration de ces considérations dans le processus de conception de l'intergiciel ; cependant, il s'agit en général d'un processus de configuration externe qui ne permet pas la prise en compte automatique des caractéristiques de l'application.

À ces problématiques de configuration s'ajoute le besoin de fiabilité. Il est impératif de pouvoir s'assurer du fonctionnement correct de l'application et de l'intergiciel qui lui est associé. L'étude du comportement des éléments applicatifs sont en général l'objet d'une série de tests, qui bien qu'utiles sont par nature incomplets.

IX-1 Conception conjointe de l'application et l'intergiciel

Afin d'exploiter de façon efficace les caractéristiques d'une application pour la mise en place d'un intergiciel adapté, il est nécessaire de recourir un formalisme permettant d'en décrire tous les aspects.

Nous avons établi que les langages de description d'architecture (ADL) proposent une approche synthétique pour rassembler tous les éléments nécessaires à une description complète des systèmes. Leur utilisation permet notamment de pouvoir exploiter la description de l'application selon différents aspects – documentation, analyse, génération automatique, etc.

Dans le cadre de nos travaux, nous avons choisi d'utiliser AADL comme langage de modélisation. Ce language se distingue par une approche très concrète, centrée sur une identification précise des composants de l'architecture. Il permet ainsi de modéliser à la fois l'application (*threads*, sous-programmes) et son environnement d'exécution (processeurs, bus).

IX-1.1 AADL comme support pour un cycle de conception

Nous utilisons AADL comme langage unificateur pour décrire les différents aspects de l'application, que ce soit les éléments fonctionnels (interfaces, etc.) ou non fonctionnels (temps d'exécution, répartition des nœuds, etc.). AADL fournit une représentation commune de l'application répartie qui peut être exploitée par différents outils. Une modélisation AADL peut ainsi être utilisée par des analyseurs d'ordonnancement, des générateurs de code, ou tout autre outil spécialisé. Nous nous servons du langage pour décrire la structure d'exécution de l'application ; les différents composants AADL définissent les types de données échangés, les interfaces des entités ainsi que la localisation des nœuds sur le réseau, les politiques d'ordonnancement, etc. Les descriptions comportementales des éléments de l'application sont exprimées dans un paradigme différent d'AADL – typiquement, un langage de programmation ou une description plus formelle – et associées aux composants AADL.

La description AADL modélise une application destinée à s'exécuter à travers un exécutif adapté. Cet exécutif prend en charge l'ordonnancement des *threads* du système et les communications inter-nœuds en mettant en place un intergiciel adapté.

Nous avons proposé une méthodologie de conception intégrant la description complète des éléments applicatifs en AADL ; la configuration de l'intergiciel peut être déduite de la description architecturale de l'application et de la description de l'environnement d'exécution matériel. Le processus que nous avons défini est constitué de deux étapes principales, correspondant aux degrés de précision de la configuration de l'exécutif.

Dans la première étape nous considérons l'utilisation d'un intergiciel capable de prendre en charge les différentes fonctionnalités exprimées dans la description AADL de l'application. Dans la seconde étape nous déduisons une description AADL de l'intergiciel lui-même ; il est alors possible d'évaluer précisément les dimensions spatiales et temporelles de l'application complète.

À chaque étape, la description AADL peut être exploitée pour en extraire différentes informations afin de vérifier certaines propriétés architecturales vis-à-vis des spécifications initiales. Par exemple, nous pouvons nous assurer de l'absence de blocages dans les flux d'exécutions, ou vérifier l'ordonnançabilité des différents *threads* de l'application. Il est également possible de générer une application exécutable afin d'effectuer des mesures de performance.

Nous avons raffiné l'interprétation sémantique des composants AADL pour introduire une séparation nette entre la modélisation de l'application elle-même et celle de l'exécutif. Afin de guider et d'assister l'utilisateur dans la description de l'application, nous avons spécifié les capacités de l'exécutif sous la forme de directives architecturales en AADL.

IX-1.2 Exploitation des descriptions AADL

Nous avons établi des règles de traduction pour produire du code exécutable dans différents langages de programmation à partir des constructions AADL, dans le cadre des spécifications d'exécutif que nous avons définies.

L'intergiciel supportant notre exécutif AADL doit offrir une grande flexibilité de configuration afin de prendre en charge les différents paramètres de l'application, exprimés dans la description

AADL. Nous nous reposons sur l'intergiciel schizophrène PolyORB, qui fournit une armature pour construire des intergiciels adaptés aux caractéristiques des applications.

Du fait de sa structuration rigoureuse, l'architecture schizophrène peut servir de structure de base à la modélisation d'un intergiciel en AADL. Nous en appliquons les principes pour modéliser l'intergiciel en AADL, lors de la seconde phase de notre processus de conception.

De façon complémentaire à la génération de code exécutable, nous avons défini un processus de transformation pour générer un réseau de Petri coloré à partir de la description AADL des applications. Les réseaux de Petri ainsi produits reflètent les spécifications de l'exécutif ; il est ainsi possible d'étudier des propriétés structurelles telles que l'absence d'interblocage ou de valeurs indéfinies dans l'architecture.

Afin de valider les règles de traduction que nous avons définies, nous avons réalisé un outil appelé Ocarina. Ocarina peut être utilisé comme un compilateur afin de produire du code source ou des réseaux de Petri à partir d'une description AADL. Il peut également être utilisé comme une bibliothèque pour la manipulation des descriptions AADL et intégré au sein d'une application existante ; il est ainsi possible d'étudier certains aspects que nous n'avons pas spécifiquement traités dans nos travaux – par exemple l'analyse d'ordonnancement avec Cheddar – ou de produire très rapidement un prototype, avec Glade.

Nous avons pu mettre en pratique la première phase de notre cycle de développement, en coordonnant différentes exploitations d'AADL pour vérifier et produire le prototype exécutable issu d'une description architecturale.

IX-2 Perspectives

L'objectif de nos travaux consistait à étudier l'utilisation d'un langage unique – AADL – pour rassembler tous les aspects de la modélisation d'une application afin de produire un système vérifié, respectant les différentes contraintes d'exécution. Nous avons montré que cette approche était viable et permettait de produire effectivement des systèmes répartis vérifiés.

Nos travaux peuvent être poursuivis selon différents axes complémentaires, correspondant aux différents aspects de la méthodologie que nous avons traités.

IX-2.1 Production automatique d'intergiciels en AADL

L'implantation complète de notre processus de construction nécessite la définition et la réalisation d'un expanseur AADL permettant de produire automatiquement les composants AADL de l'intergiciel selon les directives que nous avons proposées. La production automatique des composants autoriserait une configuration plus fine de l'intergiciel ; elle permettrait également l'optimisation des services de communication au niveau architectural – afin, par exemple, de ne pas utiliser le service de protocole si l'application est constituée d'un seul nœud. L'utilisation d'AADL pour la description exhaustive des éléments logiciels faciliterait donc la phase d'optimisation et de configuration de l'exécutif en permettant la production d'un intergiciel spécifique à l'application considérée. Cette approche idéale est traditionnellement irréaliste compte-tenu du coût de la production d'un tel intergiciel sur mesure ; l'utilisation d'AADL permettrait la production d'un intergiciel ad-hoc à moindre coût.

Ces considérations de configuration constituent un problème d'optimisation combinatoire ; il s'agirait en effet de déterminer un compromis entre les fonctionnalités de l'exécutif et ses dimensions, notamment spatiales. Ainsi, l'utilisation de personnalités protocolaires différentes au sein d'un même nœud implique une plus grande empreinte mémoire.

IX-2.2 Raffinement des spécifications de l'exécutif

Les spécifications de notre exécutif peuvent être considérées comme relativement restrictives au regard des fonctionnalités de multiplexage que peut offrir un intergiciel.

Notamment, les *threads* AADL ne peuvent gérer qu'une seule séquence d'appels, associée à l'ensemble de leurs ports. Nous avons établi de telles limitations pour délimiter la complexité de la réalisation. Il serait possible d'enrichir la sémantique de modélisation afin de pouvoir modéliser plusieurs traitements alternatifs au sein des *threads* AADL – correspondant alors à des séquences d'appels différentes.

IX-2.3 Automatisation de la réduction des architectures

Nous avons vu en section VIII-3.2 que les modélisations AADL pouvaient produire des réseaux de Petri générant trop d'états pour être analysables. Nous avons montré deux stratégies pour la réduction des architectures permettant de conserver la sémantique des modèles vis-à-vis de la transformation en réseau de Petri.

Il serait intéressant de poursuivre l'étude des réductions possibles afin de définir un ensemble d'opérations pouvant être appliquées automatiquement. Dans la mesure où la génération de réseau de Petri a pour objectif l'étude des propriétés déterminées, il est envisageable d'isoler des sous-ensembles architecturaux, en assimilant par exemple certains systèmes AADL à un seul *thread*, afin de réduire la complexité des parties non pertinentes de l'architecture.

IX-2.4 Correspondances entre AADL et d'autres représentations

Nous avons défini un certain nombre de directives de transformation pour produire un système exécutable ou une modélisation comportementale à partir d'une description architecturale.

D'autres règles de transformations pourraient être établies afin de couvrir d'autres aspects de vérification. Nous nous sommes focalisés sur la vérification comportementale ; il serait intéressant d'étudier la sémantique des données et l'intégrité de leur traitement.

Des formalismes tels que PVS [Owre et al., 1992] ou B [Habrias, 2001] permettent de formaliser les spécifications d'une application. L'étude de leur transposition en constructions AADL permettrait de les intégrer dans notre processus de développement.

IX-2.5 Reconfiguration dynamique de l'exécutif

Dans les spécifications que nous avons définies pour notre exécutif, nous avons fait abstraction des aspects dynamiques fournis par les modes de configuration d'AADL. Cet aspect du langage ne nous a en effet semblé pas encore assez mature pour permettre une modélisation efficace.

Ces aspects de modélisation pourraient être étudiés afin de modéliser la reconfiguration de l'application et de l'exécutif en fonction d'événements tels que la détection de pannes dans le système. Il est notamment nécessaire de définir précisément quels aspects architecturaux (propriétés, connexions, sous-composants…) peuvent dépendre des modes de configuration. Cela implique l'extension des spécifications de l'exécutif.

La mise en place de l'exécutif pour la prise en charge de tels mécanismes nécessite l'utilisation d'une bibliothèque d'exécution capable de fournir en environnement adaptable, permettant par exemple de changer de politique d'ordonnancement à la volée. Certains travaux portent sur

la conception de telles bibliothèques d'exécution [Ogel et al., 2003]. Leur exploitation, conjointement avec l'utilisation de l'architecture schizophrène, permettrait la construction d'un exécutif capable de se reconfigurer en fonction du contexte d'exécution.

IX-2.6 Intégration dans la démarche MDA

Nos travaux se sont concentrés sur la phase de raffinement et de génération d'une application répartie. Nous nous ainsi sommes intéressés aux dernières étapes du cycle de production. Notre démarche suppose par conséquent l'existence initiale d'une description AADL de l'application à produire. Il serait intéressant de prolonger nos travaux par une étude des phases de conception amont permettant l'obtention de la description AADL.

De part la précision de sa sémantique, AADL peut être considéré comme un langage de pré-implantation. En ce sens, il peut servir à décrire un modèle de l'application associé à un exécutif tel que celui que nous avons spécifié. Nos travaux peuvent donc être intégrés dans un cycle de production plus large, inspiré de l'approche MDA (*Model Driven Architecture*), en se plaçant au niveau de la description des phases de transformation du modèle spécifique à la plate-forme (PSM, *Platform Specific Model*) en l'application exécutable elle-même. Les étapes de conception amont – c'est-à-dire la conception du modèle indépendant de la plateforme (PIM, *Platform Independent Model*) – ainsi que les règles de transformation du PIM au PSM que nous exploitons peuvent constituer un prolongement tout à fait pertinent de nos travaux.

Les règles de transformation du PIM vers le PSM peuvent s'appuyer sur des spécifications formelles des entités du modèle afin de définir un ensemble de propriétés fonctionnelles devant être assurées par l'application. Des méthodes de spécification telles que B ou PVS peuvent alors être utilisées afin de décrire les fonctionnalités algorithmiques attendues. Ces spécifications peuvent guider la sélection des composants AADL du PSM à partir d'une bibliothèque d'éléments prédéfinis et caractérisés vis-à-vis des propriétés attendues. L'analyse de l'architecture finale permet alors de valider l'implantation par rapport aux spécifications initiales.

Bibliographie

D. Abad-Garcia. Génération de code et configuration d'intergiciel à partir d'une description architecturale. Mémoire de master, École nationale supérieure des télécommunications, septembre 2005.

ABLE Group. The Acme Architectural Description Language. http://www.cs.cmu.edu/~acme/.

R. J. Allen. *A Formal Approach to Software Architecture*. Thèse de doctorat, School of Computer Science, Carnegie Mellon University, 1997.

R. J. Allen et D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3) :213–249, 1997.

ASSERT. ASSERT-online.net website. http://www.assert-online.net, juillet 2006.

J. Barnes. *Programmer en Ada 95*. Vuibert, 2e edition, 2000.

F. P. Basso, T. C. Oliveira, et L. B. Becker. Using the FOMDA Approach to Support Object-Oriented Real-Time Systems Development. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 374 – 381, Washington, DC, État-Unis, avril 2006. IEEE Computer Society.

M. Bernardo et P. Inverardi, editors. *Formal Methods for Software Architectures*, volume 2804 of *Lecture Notes in Computer Science*. Springer Verlag, 2003.

L. Besson, S. Lescuyer, J. Mairal, et S. Mir. Connexion d'un éditeur graphique à un outil de génération automatique de code. Rapport de projet, École nationale supérieure des télécommunications, décembre 2005.

X. Blanc. *MDA en action*. Eyrolles, 1ère edition, avril 2005.

J. Bloomer. *Power Programming with RPC*. O'Reilly, février 1992.

P. Boisieau et N. Gianiel. Reconciling the Needs of Architectural Description with UML. Rapport technique 004033.DVT_CS.DVRB.03, ASSERT, avril 2006.

G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Trunbull, et R. Belliardi. *The Real-time Specification for Java*. Addison-Wesley, juillet 2002.

E. Bruneton. *Julia tutorial*. Objectweb, novembre 2003.

E. Bruneton. *Developing with Fractal*. Consortium ObjectWeb, mars 2004.

E. Bruneton, T. Coupaye, et J.-B. Stéfani. Recursive and Dynamic Software Composition with Sharing. In *7th International Workshop on Component-Oriented Programming (WCOP'02)*, juin 2002.

F. Budinsky, D. Steinberg, E. Merks ans R. Ellersick, et T. Grose. *Eclipse Modeling Framework*. Addison-Wesley, août 2003.

A. Burns et A. Wellings. *Real-Time Systems and Programming Languages*. Elsevier, 3e edition, mars 2001.

J. L. Campos, J. J. Gutiérrez, et M. G. Harbour. The Chance for Ada to Support Distribution and Real-Time in Embedded Systems. In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, volume LNCS 3063, pages 91–105, Palma de Mallorca, Espagne, juin 2004. Springer.

C. Canal, J. M. Murillo, et P. Poizat. Software Adaptation. *L'Objet*, 12 :9–31, 2006.

I. Charon. *Le langage Java : concepts et pratique, le JDK 5.0*. Hermès, 3e edition, décembre 2005.

L. Clausen. Dia. http://live.gnome.org/Dia, décembre 2006.

E. M. Dashofy, A. van der Hoek, et R. N. Taylor. A Highly-Extensible, XML-Based Architecture Description Language. In *Working IEEE / IFIP Conference on Software Architecture (WICSA)*, pages 103–112. IEEE Computer Society, 2001.

M. Diaz, editor. *Vérification et mise en œuvre des réseaux de Petri*. Hermès, 2003.

P. Dissaux. HOOD and AADL. In *Proceedings of the Data Systems In Aerospace (DASIA'03)*. ESA Publication Division, juin 2003.

P. Dissaux. Hood and Ada 95. In *Data Systems In Aerospace (DASIA'99)*, Lisbonne, Portugal, mai 1999.

P. Duquesne. Module AADL pour l'éditeur de diagrammes Dia. http://www.gnome.org/projects/dia/, 2005.

J.-P. Elloy et F. Simonot-Lion. An Architecture Description Language for In-Vehicle Embedded System Development. In *15th IFAC World Congress*, Barcelone, Espagne, 2002.

H. Espinoza, H. Dubois, J. Medina, et S. Gérard. A General Structure for the Analysis Framework of the UML MARTE Profile. In *Proceedings of the Modeling and Analysis of Real-Time and Embedded Systems Workshop (MARTES'05)*, Montego Bay, Jamaïque, octobre 2005.

P. Farail et P. Gaufillet. TOPCASED — un environnement de développement open source pour les systèmes embarqués. http://www.topcased.org/, mai 2005.

J.-M. Farines, B. Berthomieu, J.-P. Bodeveix, P. Dissaux, P. Farail, M. Filali, P. Gaufillet, H. Hafidi, J.-L. Lambert, P. Michel, et F. Vernadat. Towards the Verification of Real-Time Systems in Avionics : the COTRE Approach. In *8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03)*, Trondheim, Norvège, juin 2003a.

J.-M. Farines, B. Berthomieux, J.-P. Bodeveix, P. Farail, M. Filali, P. Gaufillet, H. Hafidi, J.-L. Lambert, P. Michel, et F. Vernadat. The COTRE Project : Rigorous Software Development for Real-Time Systems in Avionics. In *27th IFAC/IFIP/IEEE Workshop on Real-Time Programming (WRTP'03)*, Zielona Góra, Pologne, mai 2003b.

S. Faucou, A.-M. Déplanche, et Y. Trinquet. Timing Fault Detection for Safety-Critical Real-Time Embedded Systems. In ACM Press, editor, *10th ACM SIGOPS European Workshop (SIGOPS'02)*, septembre 2002.

P. H. Feiler, B. Lewis, et S. Vestal. Improving Predictability in Embedded Real-Time Systems. Rapport technique CMU/SEI-2000-SR-011, université Carnegie Mellon, décembre 2000. http://la.sei.cmu.edu/publications.

P. H. Feiler, B. Lewis, et S. Vestal. The SAE Architecture Analysis & Design Language (AADL) Standard : A Basis for Model-Based Architecture Driven embedded System Engeineering. In *proceeding of the RTAS 2003 Workshop on Model-Driven Embedded Systems*, Washington, D.C., États-Unis, mai 2003.

P. H. Feiler, D. P. Gluch, et Hudak J. J. *The Architecture Analysis & Design Language (AADL) : An Introduction*. université Carnegie Mellon, février 2006.

Free Software Foundation. Emacs. http://www.gnu.org/software/emacs, novembre 2006.

E. Gamma, R. Helm, R. Johnson, et J. Vlissides. *Design Patterns*. Addison-Wesley, 22e edition, 2001.

D. Garlan et M. Shaw. *Advances in Software Engineering and Knowledge Engineering*, volume 2 of *Series on Software Engineering and Knowledge Engineering*, chapter An Introduction to Software Architecture, pages 1–39. World Scientific Publishing, 1993.

D. Garlan, S.-W. Cheng, et A. J. Kompanek. Reconciling the Needs of Architectural Description with Object-Modeling Notations. *Sci. Comput. Program.*, 44(1) :23–49, 2002. ISSN 0167-6423.

D. Giannakopoulou, J. Kramer, et S.C. Cheung. Behaviour Analysis of Distributed Systems Using the Tracta Approach. *Journal of Automated Software Engineering*, 6(1) :7–35, 1999.

C. D. Gill, J. Gossett, D. Corman, J. P. Loyall, R. E. Schantz, M. Atighetchi, et D. C. Schmidt. Integrated Adaptive QoS Management in Middleware : An Empirical Case Study. In *10th Real-Time and Embedded Technology and Application Symposium (RTAS'04)*, Toronto, Canada, mai 2004. IEEE.

O. Gilles. Génération de réseau de Petri pour la vérification formelle de code généré et configuration d'intergiciel. Mémoire de master, École nationale supérieure des télécommunications, septembre 2006.

F. Gilliers. *Développement par prototypage et génération de code à partir de LfP, un langage de modélisation de haut niveau*. Thèse de doctorat, université Pierre & Marie Curie, septembre 2005.

C. Girault et R. Valk, editors. *Petri Nets for System Engineering*. Springer, 2003.

A. Gokhale, D. Schmidt, T. Lu, B. Natarajan, et N. Wang. CoSMIC : An MDA Generative Tool for Distributed Real-Time and Embedded Applications. 2003.

S. Graf et J. Hooman. Correct Development of Embedded Systems. In *European Workshop on Software Architecture : Languages, Styles, Models, Tools, and Applications (EWSA'04)*, LNCS 3047, pages 241–249, St-Andrews, Royaume-Uni, mai 2004. Springer-Verlag.

R. M. Graham, T. S. Woodall, et J. M. Squyres. Open MPI : A Flexible High Performance MPI. In *6th International Conference on Parallel Processing and Applied Mathematics (PPAM'05)*. Springer, septembre 2005.

W. Gropp et E. Lusk. *User's Guide for mpich, a Portable Implementation of MPI*. université de Chicago, février 1999.

W. Grosso. *Java RMI*. O'Reilly, octobre 2001.

D. Grune, H.E. Bal, C.J.H. Jacobs, et K. Langendoen. *Compilateurs*. Dunod, 2002.

H. Habrias. *Spécification formelle avec B*. Hermès, 2001.

S. Haddad, F. Kordon, et L. Petrucci, editors. *Méthodes formelles pour les systèmes répartis et coopératifs*, chapter 4. Hermès, 2006.

A. Hamez et X. Renault. *PetriScript Reference Manual*. Lip6, 2005.

P. Hnětykna. A Model-driven Environment for Component Deployment. In *3rd ACIS International Conference on Software Engineering, Research, Management and Applications (SERA'05)*, Mount Pleasant, Michighan, États-Unis, août 2005. IEEE Computer Society.

C. Hofmeister, R. L. Nord, et D. Soni. Describing Software Architecture with UML. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 145–160, Deventer, Pays-Bas, 1999. Kluwer, B.V. ISBN 0-7923-8453-9.

J. Hugues. *Architecture et services des intergiciels temps-réel*. Thèse de doctorat, École nationale supérieure des télécommunications, septembre 2005.

J. Hugues. *PolyORB User's Guide*, 2006. http://www.adacore.com.

J. Hugues, Y. Thierry-Mieg, F. Kordon, L. Pautet, S. Baarir, et T. Vergnaud. On the Formal Verification of Middleware Behavioral Properties. In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, volume ENTCS 133, pages 139 – 157, Linz, Autriche, septembre 2004. Elsevier.

J. Hugues, F. Kordon, L. Pautet, et T. Vergnaud. A Factory To Design and Build Tailorable and Verifiable Middleware. In *Workshop on Networked Systems : realization of reliable systems on top of unreliable networked platforms (Monterey Workshop Series, 12e édition, 2005)*, volume 4322 of *LNCS*, pages 123–144, Univ. de Californie, Irvine, États-Unis, 2007. Springer Verlag.

Impr. Nat. *Lexique des règles typographiques en usage à l'Imprimerie nationale*. Imprimerie nationale, 3e edition, 1990.

Ada Reference Manual. ISO, 3e edition, 2005.

B. W. Kernighan et D. M. Ritchie. *Le langage C*. Dunod, 2e edition, 2000.

S. Kleijnen et S. Raju. An Open Web Services Architecture. *Queue*, 1(1) :38–46, 2003. ISSN 1542-7730.

F. Kordon et Luqi. An Introduction to Rapid system Prototyping. In *Transactions on Software Engineering*, volume 28. IEEE, septembre 2002.

F. Kordon et E. Paviot-Adet. Using CPN-AMI to Validate a Safe Channel Protocol. In *International Conference on Theory and Applications of Petri Nets*, Williamsburg, Virginie, juin 1999.

S. Krakowiak. Jonathan. http://kilim.objectweb.org/doc/.

A. S. Krishna, B. Natarajan, A. Gokhale, D. C. Schmidt, N. Wang, et G. Thaker. CCMPerf : A Benchmarking Tool for CORBA Component Model Implementations. *Real-Time Systems*, 29 (2-3) :281–308, 2005. ISSN 0922-6443.

J. W. Krueger, S. Vestal, et B. Lewis. Fitting the Pieces Together : System/Software Analysis and Code Integration Using MetaH. In *17th Digital Avionics Systems Conference (DASC*, volume 1. IEEE, novembre 1998.

F. Loiret, L. Seinturier, et É. Gressier-Soudan. Fractal, Kilim, JAC : une expérience comparative. In *Journée Composants (JC'04)*, 2004.

J. Magee et J. Kramer. Dynamic Structure in Software Architectures. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 3–14, 1996.

J. Magee et J. Kramer. *Concurrency : State Models and Java Programs*. Wiley, 1999.

J. Magee, J. Kramer, et D. Giannakopoulou. Behaviour Analysis of Software Architectures. In *First Working IFIP Conference on Software Architecture (WICSA)*, IFIP Conference Proceedings, pages 35–50. Kluwer, 1999.

N. Medvidovic et R. R. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1) :70–93, 2000.

S. J. Mellor et M. J. Balcer. *Executable UML*. Addison-Wesley, 2nde edition, décembre 2003.

MPI-2 : Extensions to the Message-Passing Interface. MPIForum, novembre 2003. http://www.mpi-forum.org.

S. Mullender, editor. *Distributed Systems*. ACM Press, 1993.

P.-A. Müller et N. Gaertner. *Modélisation objet avec UML*. Eyrolles, 2nde edition, décembre 2003.

C. Nester, M. Philippsen, et B. Haumacher. A more efficient RMI for Java. In *JAVA'99 : Proceedings of the ACM 1999 conference on Java Grande*, pages 152–159, New York, États-Unis, 1999. ACM Press. ISBN 1-58113-161-5.

Kilim 2 Tutorial. ObjectWeb, juillet 2003.

F. Ogel, G. Thomas, I. Piumarta, A. Galland, B. Folliot, et C. Baillarguet. Towards Active Applications : the Virtual Virtual Machine Approach. *New Trends in Computer Science and Engineering*, 2003.

Model Driven Architecture. OMG, 2001a.

Ada Language Mapping Specification. OMG, octobre 2001b.

Common Object Request Broker Architecture. OMG, mars 2004.

Java Language Mapping Specification. OMG, août 2002a.

Minimum CORBA Specification. OMG, août 2002b.

Real-time CORBA Specification. OMG, janvier 2005.

M. Oussalah, editor. *Ingénierie des composants. Concepts, techniques et outils*. Vuibert Informatique, 2005.

S. Owre, J. M. Rushby, et N. Shankar. PVS : A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748 – 752, Saratoga, NY, juin 1992. Springer-Verlag.

L. Pautet. *Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition*. Habilitation à diriger des recherches, université Pierre & Marie Curie (Paris 6), décembre 2001.

L. Pautet et S. Tardieu. *GLADE User's Guide*. Free Software Foundation, 2005.

D. E. Perry et A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 17(4) :40–52, 1992.

F. Plášil, D. Bálek, et R. Janeček. SOFA/DCUP : Architecture for component trading and dynamic updating. In *ICCDS'98*. IEEE, mai 1998.

P. Poizat, J.-C. Royer, et G. Salaün. Formal Methods for Component Description, Coordination and Adaptation. In *First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT)*, pages 89–100, 2004.

I. Pyarali, D. C. Schmidt, et R. K. Cytron. Achieving End-to-end Predictability in the TAO Real-time CORBA ORB. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, page 13, Washington, DC, États-Unis, 2002. IEEE Computer Society. ISBN 0-7695-1739-0.

T. Quinot. *Conception et réalisation d'un intergiciel schizophrène pour la mise en œuvre de systèmes répartis interopérables*. Thèse de doctorat, École nationale supérieure des télécommunications, mars 2003.

D. Regep, Y. Thierry-Mieg, F. Gilliers, et F. Kordon. Modélisation et vérification de systèmes répartis : une approche intégrée avec LfP. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'03)*, IRISA, Rennes, janvier 2002.

J.-M. Rifflet et J.-B. Yunès. *UNIX – programmation et communication*. Dunod, 2003.

A.-E. Rugina, K. Kanoun, et M. Kaâniche. AADL-based Dependability Modelling. Rapport technique 06209, LAAS-CNRS, avril 2006.

B. Rumpe, M. Schoenmakers, A. Radermacher, et A. Schürr. UML + ROOM as a Standard ADL ? In *ICECCS'99*, 1999.

SAE. SAE AADL information site. http://www.aadl.info, avril 2006a.

Architecture Analysis & Design Language (AS5506) 1.0. SAE, 1re edition, septembre 2004.

AS5506, annex A : Error Model. SAE, septembre 2005. http://www.sae.org.

Language Compliance and Application Program Interface. SAE, juillet 2006b. http://www.sae.org.

AS5506, annex D : Programming Language Guidelines. SAE, septembre 2006c. http://www.sae.org.

AS5506, annex D : Unified Modeling Language (UML) Profile for the AADL. SAE, pré-version edition, juillet 2006d.

L. Seinturier, N. Pessemier, et T. Coupaye. *AOKell 2.0 Documentation*. Objectweb, février 2006.

F. Singhoff, J. Legrand, L. Nana, et L. Marcé. Scheduling and Memory Requirements Analysis with AADL. In *SIGAda'05 : Proceedings of the 2005 annual ACM SIGAda international conference on Ada*, pages 1–10, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-185-6.

S. Srinivasan. *RPC : Remote Procedure Call Protocol Specification Version 2*. Sun Microsystems, août 1995a. RFC 1831.

S. Srinivasan. *XDR : External Data Representation Standard*. Sun Microsystems, août 1995b. RFC 1832.

D. Statezni. Analyzable and Reconfigurable AADL Specifications for IMA System Integration. Rapport technique, Rockwell-Collins, 2004.

B. Stroustrup. *Le langage C++*. Campus Press, 2e edition, avril 2003.

Java Remote Method Invocation. Sun Microsystems, 2004. http://java.sun.com/j2se/1.5.0/docs/guide/rmi/.

Runtime Environment Specification : JavaCard Platform, version 2.2.2. Sun Microsystems, mars 2006. http://java.sun.com/products/javacard.

C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

A. S. Tanenbaum. *Réseaux*. Pearson Education, 4e edition, juin 2003.

M. Vadet et P. Merle. Les containers ouverts dans les plates-formes composants. octobre 2001.

A. Vallecillo, J. Hernández, et J. M. Troya. New issues in object interoperability. In *Object-Oriented Technology*, volume 1964 of *Lecture Notes in Computer Science*, pages 256–269. Springer Verlag, 2000.

T. Vergnaud et B. Zalila. Ocarina, a Compiler for the AADL. Rapport technique, École nationale supérieure des télécommunications, 2006. http://ocarina.enst.fr.

T. Vergnaud, J. Hugues, L. Pautet, et F. Kordon. PolyORB : a schizophrenic middleware to build versatile reliable distributed applications. In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, volume LNCS 3063, pages 106 – 119, Palma de Mallorca, Espagne, juin 2004. Springer Verlag.

T. Vergnaud, J. Hugues, L. Pautet, et F. Kordon. Rapid Development Methodology for Customized Middleware. In *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*, pages 111 – 117, Montréal, Québec, juin 2005a. IEEE.

T. Vergnaud, L. Pautet, et F. Kordon. Using the AADL to describe distributed applications from middleware to software components. In *Proceedings of the 10th International Conference on Reliable Software Techologies Ada-Europe 2005 (RST'05)*, volume LNCS 3555, pages 67 – 78, York, Royaume-Uni, juin 2005b. Springer Verlag.

T. Vergnaud, I. Hamid, K. Barbaria, É. Najm, L. Pautet, et S. Vignes. Modeling and Generating Tailored Distribution Middleware for Embedded Real-Time Systems. In *2nd European Congress Embedded Real-Time Software (ERTS'06)*, Toulouse, France, janvier 2006.

S. Vestal. *MetaH User's Manual*. Honeywell, 1998.

S. Vestal. MetaH Support for Real-Time Multi-Processor Avionics. *wpdrts*, 00 :11, 1997.

S. Vinoski. Distributed Object Computing With CORBA. In *C++ Report Magazine*, 1993.

N. M. Vo. Étude et réalisation d'un parseur pour AADL. Mémoire de DEA, École nationale supérieure des télécommunications, 2004.

T. Wall et V. Cahill. Mobile RMI : Supporting Remote Access to Java Server Objects on Mobile Hosts. In *Proceedings of the 3rd International Symposium of Distributed Objects and Applications (DOA'01)*, pages 41–51, septembre 2001.

B. Zalila, J. Hugues, et L. Pautet. An Improved IDL Compiler for Optimizing CORBA Applications. In *Special Interest Group on Ada 2006 (SIGAda'06)*. ACM Press, 2006.

Table des figures

Table des listings

Liste des tableaux