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

Mohamed Yassin Chkouri

Université de Grenoble

T H È S E

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

Spécialité : Informatique

préparée au laboratoire VERIMAG

dans le cadre de l'École Doctorale (Mathématiques et Informatique)

Présentée et soutenue publiquement par

Mohamed Yassin CHKOURI

le 7 Avril 2010

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

---

JURY

Président	**Jean Claude Fernandez**	**Professeur UJF Grenoble**
Rapporteurs	**Jean-Paul Bodeveix**	**Professeur IRIT Toulouse**
	Elie Najm	**Professeur ENST Paris**
Examinateurs	**Iulian Ober**	**Maître de conférence Toulouse**
	Pierre Gaufillet	**Ingénieur AIRBUS Toulouse**
	Marius Bozga	**Ingénieur de recherche CNRS**
Directeur de thèse	**Joseph Sifakis**	**Directeur de recherche CNRS**

REMERCIEMENTS

Je tiens à remercier toutes les personnes qui ont rendu cette thèse possible par leur aide et leurs contributions.

Je tiens tout d'abord à remercier mon directeurs de thèse Joseph SIFAKIS, Directeur de recherche CNRS et fondateur du laboratoire VERIMAG. L'aide et les conseils qu'il n'ont jamais cessé de me donner m'ont été d'une valeur inestimable.

Je voudrais tout particulièrement remercier Marius BOZGA, Ingénieur de recherche CNRS pour avoir suivi mes travaux durant ces années de thèse ainsi que pour l'ensemble des conseils qu'il m'a prodigués. Son expérience m'a été grandement profitable et les discussions que nous avons eues ont été d'une importance indéniable dans mon cursus.

Je remercie Jean Claude Fernandez, professeur à l'Université Joseph Fourier et directeur de l'IMAG, pour m'avoir fait l'honneur de présider ce jury.

Je remercie Elie Najm, professeur à TELECOM Paris et Jean-Paul Bodeveix, professeur à l'Université de Toulouse III pour avoir accepté d'être les rapporteurs de ce mémoire. Les remarques pertinentes qu'ils ont émis ont permis de le consolider.

Je remercie également Iulian Ober, Maître de conférence à l'Université de Toulouse III, Pierre Gaufillet, Ingénieur à AIRBUS de Toulouse pour l'intérêt qu'ils ont porté à mes travaux et pour avoir accepté de faire partie du jury de ma soutenance de thèse.

Enfin, la totalité de ma reconnaissance et de mes pensées vont à mes parents Mohamed et Khadouj, à ma soeur Ijlal et à mes frères Bilal et Houssain. Ils n'ont jamais cessé de m'encourager et de m'apporter du support surtout durant les dernières phases de la rédaction de ce manuscrit. C'est donc tout naturellement que ce document leur soit dédié.

Table des matières

Table des figures

Listings

Chapitre 1

Introduction

Les systèmes embarqués sont des systèmes électroniques intégrant du logiciel et du matériel, inclus dans des objets ou des systèmes qui ne sont pas perçus comme des ordinateurs par leurs utilisateurs. Les systèmes embarqués sont devenus des constituants de base dans de très nombreux champs d'application : avionique, espace, automobile, télécommunication, domotique, etc. Les produits courants intègrent des fonctionnalités de plus en plus sophistiquées, grâce aux systèmes embarqués qu'ils contiennent, eux-mêmes de plus en plus complexes.

Les systèmes embarqués sont d'une importance stratégique pour les économies modernes. Ils sont massivement utilisés dans des produits et services pour assurer des nouvelles fonctionnalités et des nouveaux usages. Le poids économique du logiciel dans la valeur des systèmes embarqués est en progression constante. Le logiciel, lorsqu'il est conçu en parfaite adéquation avec la plateforme matérielle qui le supporte, permet le développement de nouveaux services, et offre une différentiation des produits et des avantages compétitifs. Les technologies de l'embarqué sont le premier secteur de croissance des technologies de l'information.

Les technologies de l'embarqué offrent des avantages nouveaux aux développeurs de systèmes et de services en générant de la valeur ajoutée et en améliorant leur compétitivité. En raison de la rapidité des évolutions technologiques, produire des systèmes compétitifs est un effort permanent. Le faible coût, l'absence d'erreurs, la rapidité de conception de fabrication et de test sont des critères majeurs de réussite. Les systèmes embarqués présentent, en fonction du domaine d'application, une grande variété de formats et de solutions techniques.

Composants Logiciels et Modélisation Les travaux de recherche sur les systèmes embarqués visent le développement de modèles, de langages et d'outils pour la conception de systèmes embarqués. Ces derniers sont construits de manière modulaire à partir de composants. Jusqu'alors, les

composants avaient été envisagés comme une décomposition structurelle du système à concevoir. Le défi actuel est de prendre en compte d'autres types de spécifications que des spécifications structurelles et fonctionnelles. Les systèmes embarqués sont à la fois soumis à des contraintes de capacités (mémoire, bande passante), et à des exigences non fonctionnelles (exécution sous contraintes de temps réel). L'objectif principal dans ce domaine est d'étudier des modèles formels permettant de décrire ces systèmes et leurs contraintes (conception, spécification), de les construire (programmation, simulation, synthèse, exécution), et de les analyser (validation, vérification).

La difficulté se présente, en raison de la variété des modèles formels servant à représenter les caractéristiques non fonctionnelles des composants, et les exigences sur le système complet.

Compilation et Synthèse pour l'Embarqué
Les problèmes de compilation pour systèmes embarqués et de synthèse de systèmes sont au premier plan de la problématique. Parmi les problèmes rencontrés on peut citer la synthèse haut niveau, la conception conjointe logiciel-matériel, l'ordonnancement, la minimisation de la consommation, de la mémoire, de la taille de code et les communications.

Nous distinguons essentiellement deux approches :

- L'adaptation des techniques logicielles au domaine des systèmes embarqués : Cela inclut les techniques de compilation, les systèmes d'exploitation, la gestion des architectures, les techniques de conception logicielle par composant, les techniques de vérification du logiciel. Cette adaptation doit se faire en prenant en compte les contraintes évoquées pour les composants, notamment les contraintes non fonctionnelles.
- La synthèse de haut niveau et la conception conjointe du matériel et du logiciel : Cela inclut la conception de haut niveau d'architectures matérielles, les méthodologies permettant la réutilisation de composants existants, le partitionnement et l'adaptation de l'interface entre le matériel et le logiciel.

Validation des Systèmes Embarqués L'enjeu technologique et scientifique est l'étude de méthodes et outils facilitant le développement rigoureux et à coût maîtrisé des logiciels embarqués. La vérification représente un souci majeur pour un grand nombre de systèmes embarqués. D'où l'importance de la validation de ces systèmes, c'est-à-dire test, vérification et certification. Il y a donc un besoin réel et pressant de développer des méthodes et outils efficaces pour la validation des systèmes embarqués. Car la moindre faille sera exploitée par des malveillants potentiels.

La vérification d'un système embarqué, en dépit des progrès réalisés dans la technologie des logiciels de vérification formelle, demeure longue et coû-

teuse. Les méthodes formelles se développent pour les systèmes temporisés, mais restent limitées à des systèmes de faible complexité.

1.1 Conception fondée sur les modèles

La complexité croissante des systèmes informatiques entraîne des difficultés d'ingénierie pour les systèmes à base de composants, en particulier liées à l'optimisation, la validation et à l'analyse des performances et des exigences concernant la sûreté de fonctionnement. Des approches d'ingénierie guidée par des modèles sont de plus en plus utilisées dans l'industrie dans l'objectif de maîtriser cette complexité au niveau de la conception. Ces approches encouragent la réutilisation et l'automatisation du cycle de développement. Elles doivent être accompagnées de langages et outils capables d'assurer la conformité du système implémenté aux spécifications.

L'Architecture Dirigée par les Modèles (*Le Model Driven Architecture*, MDA) [MDA, Poo01] est une démarche de développement proposée par l'*Object Managment Group* (OMG) [OMG]. Elle permet de séparer les spécifications fonctionnelles d'un système des spécifications de son implémentation sur une plate-forme donnée. A cette fin, le MDA définit une architecture de spécifications structurée en modèles indépendants des plates-formes (*Platform Independent Model*, PIM) et en modèles spécifiques (*Plateform Specific Model*, PSM).

L'approche MDA permet de réaliser le même modèle sur plusieurs plates-formes grâce à des projections standardisées. Elle permet aux applications d'interopérer en reliant leurs modèles et supporte l'évolution des plates-formes et des techniques. La mise en œuvre du MDA est entièrement basée sur les modèles et leurs transformations.

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 1.1 :

1. la définition d'un modèle de très haut niveau, indépendant des contraintes d'implantation : le PIM peut être raffiné afin d'ajouter différentes informations non fonctionnelles telles que la gestion de la sécurité ; ces informations demeurent indépendantes 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 ; le PSM peut être également raffiné afin de prendre en compte différents paramètres liés à l'environnement d'exécution.
3. Le PSM est ensuite utilisé pour générer une application exécutable basée sur les spécifications à partir desquelles le PIM a été construit.

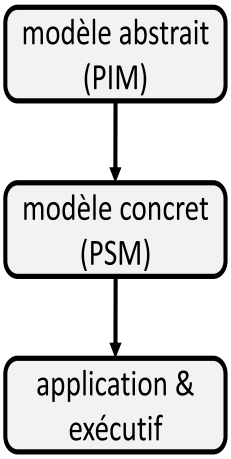


FIG. 1.1 – L'approche MDA

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 de l'application jusqu'à la production de code exécutable.

1.2 Modèles formels des applications

L'approche MDA peut être instanciée en utilisant différents langages. C'est pourquoi un certain nombre de langages permettant cette description sont apparus et offrent à ce jour un certain nombre de fonctionnalités. La plupart des approches guidées par des modèles se basent soit sur :

- le langage de modélisation unifié (*Unified Modeling Language*, UML) [UMLa], qui est un langage de modélisation à usage général et ses profiles comme MARTE (*Modeling and Analysis of Real-Time and Embedded*) [MAR]. Si UML permet une grande expressivité, il ne transporte par en lui-même une sémantique précise pour décrire les architectures [GCK02].
- des langages de description d'architecture (*Architecture Description Language*, ADL), qui sont des langages propres à des domaines particuliers. Un ADL est un langage qui permet la modélisation d'une architecture conceptuelle d'un système logiciel et/ou matériel. Il fourni une syntaxe concrète et une structure générique (*framework*) conceptuelle pour caractériser les architectures.
- des langages spécifiques, conçus pour répondre à des besoins particuliers de modélisation et d'analyse, par exemple *Behavior Interaction Priority* (BIP) [BBS06], Fractal [BCL+06] et Ptolemy [EJL+03].

Parmi les ADL, le langage d'analyse et de description d'architectures (*Architecture Analysis and Design Language*, AADL) [AADb], a fait l'objet d'un intérêt croissant dans l'industrie des systèmes embarqués critiques (comme Honeywell, Rockwell Collins, l'Agence Spatiale Européenne, Astrium, Airbus). AADL a été standardisé par la SAE (*International Society of Automotive Engineers*) en 2004, pour faciliter la conception et l'analyse de systèmes complexes, critiques, temps réel dans des domaines comme l'avionique, l'automobile et le spatial. AADL fournit une notation textuelle et graphique standardisée pour décrire des architectures matérielles et logicielles. Le succès d'AADL dans l'industrie est justifié par son support avancé à la fois pour la modélisation d'architectures reconfigurables et pour la conduite d'analyses. En particulier, le langage a été conçu pour être extensible afin de permettre des analyses qui ne sont pas réalisables avec le langage de base. Dans cette optique, une annexe au standard AADL a été définie (*Annex Behavior Specification*, ABS [ABS]) pour compléter les descriptions d'architecture avec du comportement.

Le langage AADL, pour sa part, nous semble très prometteur pour offrir une description fiable et assez rigoureuse des systèmes embarqués. Cependant, la mise en œuvre de ce langage nous paraît peu immédiate à maîtriser. De plus, les outils utilisant ce langage sont encore pour la plupart au stade du développement et n'implémentent pas tous les éléments spécifiés dans la norme AADL. En outre, AADL n'offre pas une sémantique précise et manque d'outils de vérification robuste.

Nous pensons que les outils d'AADL doivent être capables de manipuler des modèles formels, pour permettre la vérification et le raffinement par transformation des modèles, permettre l'intégration incrémentale de composants hétérogènes, et d'évoluer aisément suivant l'ajout de nouvelles primitives et modèles de plates-formes, pour permettre la synthèse d'implémentations sur différentes plates-formes d'exécution concrètes.

BIP [BBS06] est basé sur une théorie permettant le développement incrémentale de systèmes à base de composants hétérogènes. Trois sources différentes d'hétérogénéité sont considérées : interaction, exécution et abstraction. En BIP, les composants sont la superposition de trois couches décrivant : comportement, interaction, et priorité. Le modèle comportemental étend la théorie des automates temporisés avec l'expression de l'urgence [BST97]. L'interaction concerne la synchronisation entre les composants, avec éventuel transfert de données. La priorité est un mécanisme élémentaire de contrôle pour la résolution de conflits et permet la modélisation et la composition des politiques d'ordonnancement. La chaîne d'outils BIP permet la génération d'une implémentation exécutable sur une machine virtuelle qui supporte le modèle d'exécution BIP, ainsi que la vérification formelle des modèles en utilisant différentes techniques.

1.3 Applications formellement vérifiées

L'activité de test est la méthode de validation préconisée dans les normes de sécurité. Cependant, le test ne peut pas être exhaustif et il ne garantit pas l'absence d'erreurs. Les méthodes formelles, permettent de pallier le problème de l'activité de test en offrant la possibilité de raisonner sur les propriétés d'un système. Pour mettre en évidence la différence entre ces deux méthodes : l'activité de test vérifie si un système, soumis à une situation particulière, a un comportement correct par rapport à sa spécification, tandis que les méthodes formelles vérifient l'adéquation du système à sa spécification pour n'importe quelle situation. Les méthodes formelles permettent de prouver des propriétés mais elles sont plus difficiles à mettre en œuvre.

L'application des techniques d'analyse statique par interprétation abstraite sur des systèmes embarqués ou la vérification des systèmes par techniques de vérification de modèles (*model checking*) jouent en faveur de l'utilisation de telles méthodes dans le cycle de développement. Ces méthodes automatiques fournissent des preuves de propriétés telles que l'absence d'erreurs à l'exécution ou de satisfiabilité de propriétés temporelles.

De nombreux outils AADL ont été proposés ces dernières années : ADELE [ADEb], Stood [STO], OSATE [OSA], TOPCASED [TOP], ADes [Til05, ADEa], Furness [SLC06], ADAPT [RKK08], Ocarina [HZPK07, VPK05] ou Cheddar [SLNM04]. Ces outils permettent :

- d'éditer des modèles AADL (outils Stood, ADELE, OSATE, TOPCASED).
- de générer le logiciel du système à partir d'un modèle AADL (Stood, Ocarina).
- ou de conduire diverses analyses (OSATE, Furness, ADAPT, Cheddar).

En particulier, les outils d'analyse permettent de tester le respect des contraintes temporelles. Par exemple Cheddar, OSATE et Furness permet d'effectuer une analyse de l'ordonnancement d'un système AADL. Cheddar vérifie l'ordonnançabilité de l'architecture et permet d'effectuer des simulations exhaustives.

Pour les langages spécifiques, les outils de vérification sont en général plus performants. La chaîne d'outils de BIP permettent l'exploration exhaustive de l'espace des états du système, la détection des blocages potentiels et la vérification de certaines propriétés dans les modèles. BIP permet de mettre en place des systèmes robustes et sûrs en produisant un contrôleur d'exécution correct par construction, et en fournissant un modèle formel qui peut être utilisé avec divers outils de vérification et de validation. Cela joue en faveur de l'utilisation de telles méthodes dans le cycle de développement. Au cours de nos travaux, nous utiliserons deux techniques de vérification :

Absence de blocage (*Deadlock freedom*) : Il s'agit d'une propriété essentielle car elle caractérise la capacité du système à exécuter des activités

au cours de sa durée de vie. Les outils de BIP permettent la détection de blocages potentiels par analyse statique des connecteurs dans le modèle BIP [GS03]. Cette méthode consiste a utiliser les invariants générés automatiquement pour prouver la non-satisfiabilité des prédicats caractérisant les blocages globaux [BBSN08]. Deux types d'invariants sont générés : (i) Les invariants de composants qui sur-approximent l'ensemble d'états atteignables de ces derniers, et (ii) les invariants d'interaction qui sont les contraintes sur les états des composants liées aux interactions. La méthode est implémentée dans l'outil D-Finder [BBNS09].

Model checking : consiste à construire un modèle fini du système analysé et à vérifier les propriétés souhaitées de ce modèle. La vérification demande une exploration complète ou partielle du modèle. Les avantages principaux de cette technique sont : la possibilité de rendre l'exploration du modèle automatique et de faciliter la production de contre-exemples, lorsque la propriété est violée. Dans nos travaux de thèse nous avons utilisé le model checking avec observateurs pour exprimer et vérifier des exigences et l'outil Aldebaran [BFKM97] pour minimiser et comparer des comportements attendus.

Aldebaran permet de vérifier les systèmes communicants représentés comme des systèmes de transitions étiquetées. Il permet de minimiser et de comparer des systèmes de transitions étiquetées par rapport à des relations d'équivalentes. Aldebaran prend en entrée les LTS générés par l'exploration exhaustive du moteur d'exécution BIP.

En plus des techniques de vérification, nous pouvons simuler ou déboguer les modèles en créant un système exécutable. Nous pouvons utiliser une simulation interactive et un débogueur pour vérifier chaque interaction étape par étape, et ainsi connaître l'état ou les interactions qui sont exécutables. Ces analyses permettent d'évaluer la viabilité du système, d'affiner et de corriger le comportement du système.

1.4 Contribution

Ce travail de thèse s'attache à décrire une méthodologie pour la construction d'applications formellement vérifiées. Nous nous plaçons plus particulièrement dans le cadre de systèmes temps-réel embarqués. Nous nous appuyons pour cela sur les travaux menés autour de l'architecture AADL et les outils fournis par BIP.

AADL manque d'une sémantique opérationnelle rigoureuse. Nous abordons ce problème en fournissant une traduction d'AADL vers BIP, qui a une sémantique opérationnelle formellement définie en termes de systèmes de transitions étiquetés (*Labelled Transition Systems*).

Nous proposons un processus de conception/synthèse permettant de produire automatiquement une application formellement vérifiée. Cette problé-

matique peut se décomposer en plusieurs étapes.

AADL vers BIP La première étape consiste à extraire les caractéristiques de l'architecture que nous devons prendre en compte pour spécifier l'application. Pour cela, il est nécessaire de s'appuyer sur un formalisme permettant de décrire l'application elle-même, afin d'en exprimer toutes les caractéristiques, tant fonctionnelles 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 langages 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.

La description AADL de l'application est ensuite traduite automatiquement vers une description BIP. Il est alors possible d'interpréter la description afin d'en extraire une modèle formel propre à l'analyse ou la vérification comportementale ; nous pouvons alors vérifier formellement l'application qui sera créé. Dans le cadre de nos travaux, nous nous focalisons sur la simulation, la vérification et la validation. 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 intéressons aussi pour la génération vers C/C++. Le système ainsi obtenu peut être testé afin de s'assurer du respect des contraintes non fonctionnelles spécifiées dans la modélisation initiale.

La description de certains éléments dans le standard d'AADL est assez primitive. Afin de pouvoir pleinement exprimer les éléments d'architecture, il est nécessaire d'enrichir la syntaxe AADL et ainsi d'étendre la traduction de AADL vers BIP. Cela facilitera le passage entre AADL et d'autres langages. Nous intéressons plus particulièrement aux instanciation des sous-programmes, intégration du comportement dans les devices, communication du flux de données et la modélisation et la génération d'une application distribuée.

BIP temporisé vers non temporisé Les systèmes complexes incluent souvent des propriétés temporisées, ces derniers prennent une place importante dans le développement de nouvelles technologies en temps-réel.

Les applications temporisées sont modélisés en BIP sous forme d'automates temporisés. Par contre, La chaîne d'outils BIP ne supporte pas l'exécution de ce type d'applications. Sachant que l'application générée à partir

d'AADL est temporisée, une transformation formelle d'une description BIP temporisée à une description non-temporisée (à temps discret) doit avoir lieu. Nous présentons notre méthodologie de transformation d'une description BIP temporisée en une description non-temporisée. Cette transformation doit conserver les propriétés : si un composant temporisé été conçu pour satisfaire une certaine propriété, alors sa transformation vers un composant non temporisé doit satisfaire également cette propriété. Le but de ce processus est de générer un modèle BIP que nous pouvons tester et vérifier en utilisant des méthodes fournies par la chaîne d'outils BIP.

Optimisation Les techniques d'optimisation d'un langage à base de composants est un problème bien identifié. L'objectif de l'optimisation est « d'aplatir le code » pour générer du code monolithique en remplaçant des mécanismes de communication de haut niveau par des instructions simples. Le point de départ des techniques d'optimisation est l'étude de la sémantique opérationnelle pour la composition des composants qui évite l'explosion d'états du système.

La multiplication des connecteurs et des priorités dans les structures d'exécution des environnements à composants peut générer un surcoût significatif en termes de temps d'exécution et d'occupation mémoire. Ce surcoût est d'autant plus élevé que l'architecture logicielle est modulaire. L'évolution dans le domaine va même vers une augmentation de ces structures d'exécution, car on s'oriente progressivement vers des modèles à composants composites.

Le principe général que nous appliquons pour optimiser des modèles BIP consiste à fusionner les composants pour réduire le surcoût induit par l'interprétation des connecteurs et des priorités. Les modèles optimisés sont obtenus par substitution d'un ensemble de composants par un composant produit équivalent.

1.5 Organisation de la thèse

Notre thèse est présentée comme suit. Dans le **chapitre 2**, nous montrons comment les langages de description d'architecture sont apparus indispensables pour modéliser au plus haut niveau un système et constituent une bonne solution à notre problématique. Nous décrivons les éléments constitutifs de la description d'une architecture, les fonctionnalités, les différences entre les ADL et les autres langages, les outils support d'ADL et les langages pour assister la production de systèmes.

Le **chapitre 3** comporte une étude détaillée du langage AADL. Nous montrons comment ce langage peut être exploité pour décrire précisément l'organisation d'une application. Une telle description comporte aussi toutes les informations nécessaires à la mise en place d'une application adaptée.

Nous présentons les différents aspects de ce langage ainsi que les avantages qu'il apporte pour la description des systèmes.

Le **chapitre 4** comporte une étude détaillée du langage et de l'environnement BIP. Nous décrivons les éléments de l'architecture du langage accompagnés par des exemples, ainsi qu'une description détaillée de la chaîne d'outils BIP.

Le **chapitre 5** détaille la façon dont nous traduisons une description AADL en une description BIP. Nous y exploitons les concepts introduits dans les chapitres précédents pour traduire les constructions AADL en langage BIP. Les problématiques que nous abordons ici sont l'interprétation des éléments AADL en terme de construction logicielle, ainsi que l'intégration et la traduction de descriptions comportementales définies par une annexe comportementale au sein de la description. Cette transformation a pour objectifs de fournir une sémantique opérationnelle formellement définie en termes de systèmes de transitions étiquetés, ainsi que de produire automatiquement une application formellement vérifiée.

Dans le **chapitre 6**, nous présentons les extensions de la syntaxe et de la traduction que nous proposons pour faciliter la description de l'assemblage des applications et des communications. Nous décrivons l'instanciation des sous-programmes, l'intégration du comportement dans les devices, la modélisation des communications du flux de données et la modélisation et la génération d'une application distribuée.

Dans le **chapitre 7**, nous étudions deux démarches de transformation de BIP. La première transformation permet de traduire une description BIP temporisée vers une description non-temporisée. La deuxième transformation permet d'optimiser les modèles BIP avec la préservation des propriétés.

Le **chapitre 8** constitue une mise en application de notre méthodologie. Nous décrivons la chaîne d'outils que nous avons implémentée, qui se compose principalement d'un traducteur d'AADL vers BIP, des transformations de BIP vers BIP et d'un simulateur et déboguer. Nous appliquons notre méthodologie sur des études de cas concrets afin d'en valider les performances. Ces études de cas sont classés selon leur catégories.

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

Chapitre 2

Les Langages de Description d'Architectures

Sommaire

Depuis le début des années 90, la communauté de l'ingénierie du logiciel a développé des techniques centrées sur la description de l'architecture des systèmes [IEE]. Celles-ci ont été élaborées à de multiples fins : améliorer la compréhension et la conception des systèmes complexes, favoriser leur évolution et leur réutilisation, procéder à diverses analyses, faciliter la construction et le déploiement, ou encore assister la gestion de configuration. Ces techniques se sont concrétisées par des langages spécifiques, les ADL's (*Architecture Description Language*), et l'outillage associé [TM00].

2.1 Généralités

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.

Le développement des langages de description d'architectures [TM00] 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 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 [TM00]. Elles ont débouché sur un ensemble de critères permettant de classer les ADL.

2.2 Concepts de base de la description d'une architecture

Les ADL sont généralement orientés vers la mise en exergue des constituants majeurs de la structure afin d'en montrer leurs interconnections. Dans le domaine particulier des architectures logicielles, on trouve les notions de composants, de connecteurs et de configuration (i.e. topologie) [MT97] :

2.2.1 Composants

Un composant est un élément central d'une architecture (élément de calcul ou de stockage d'information). Il requiert ses propres données et/ou son propre espace d'exécution qu'il peut partager avec d'autres composants. Un composant peut être simple ou composé ; on parle alors dans ce dernier cas de composite. Sa taille peut aller de la fonction mathématique à une application complète.

Un composant est défini généralement en deux parties. Une première partie, dite extérieure, correspond à son interface. Elle comprend la description des interfaces fournies et requises par le composant. Elle définit les interactions du composant avec son environnement. La seconde partie correspond à son implantation et permet la description du fonctionnement interne du composant. En général, un composant est décrit par son interface, les types, la sémantique, les contraintes, les évolutions et les propriétés non fonctionnelles.

- L'*interface* est un moyen d'expression des liens du composant ainsi que ses contraintes avec l'extérieur. L'interface d'un composant est la description de l'ensemble : des ports en entrée et/ou en sortie, des service offert ou requis, etc.
- Le *type* d'un composant est un concept représentant l'implantation des fonctionnalités fournies par le composant.
- La *sémantique* doit être enrichie par un modèle plus complet et plus abstrait permettant de spécifier les aspects dynamiques ainsi que les contraintes liées à l'architecture. Ce modèle doit garantir une projection cohérente de la spécification abstraite de l'architecture vers la description de son implantation avec différents niveaux de raffinements.
- Les *contraintes* définissent les limites d'utilisation d'un composant et ses dépendances intra composants.
- Les composants sont conçus pour être des éléments de conception qui évoluent. L'*évolution* doit donc être simple et s'effectuer par le biais de techniques comme le sous typage ou le raffinement.
- Les *propriétés non fonctionnelles* (propriétés liées à la sécurité, la performance, la portabilité) doivent être exprimées à part, permettant ainsi une séparation dans la spécification du composant des aspects fonctionnels.

2.2.2 Connecteurs

Les connecteurs permettent de représenter les interactions entre les différents composants ainsi que les règles auxquelles sont soumises ces interactions.

Un connecteur comprend également deux parties. La première correspond à la partie visible du connecteur, c'est-à-dire son interface. La seconde partie correspond à la description de son implantation. Il s'agit là de la définition du protocole permettant la mise en œuvre du protocole associé à l'interaction. Les connecteurs sont généralement décrits par leur interface, les types, la sémantique, les contraintes, les évolutions et les propriétés non fonctionnelles.

- L'*interface* d'un connecteur définit les points d'interactions entre connecteurs et composants.
- Le *type* correspond à la description de son implantation. Il reprend les mécanismes de communication entre composants ou les mécanismes de décision de coordination et de médiation.
- La *sémantique* est définie par un modèle de haut niveau spécifiant le comportement du connecteur (protocole d'interaction).
- Les *contraintes* permettent de définir les limites d'utilisation d'un connecteur, c'est-à-dire les limites d'utilisation du protocole de communication associé.
- L'*évolution* : Le changement des propriétés (interface, comportement) d'un connecteur doit pouvoir évoluer sans perturber son utilisation et son intégration dans les applications existantes (maximiser la réutilisation par modification ou raffinement des connecteurs).
- Les *propriétés non fonctionnelles* spécifient des besoins qui viennent s'ajouter à ceux déjà existants et qui favorisent une implantation correcte du connecteur.

2.2.3 Configuration

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 [TM00].

2.3 Différences entre les ADL et les autres langages

Afin de clarifier la définition d'un ADL, il est important d'en montrer les différences avec les autres notations qui, bien que similaires, ne sont pas considérées comme des ADL.

L'ensemble des langages peut être partagé en 3 catégories [MT97] :

- *Les langages à configuration implicite* où les informations sur la topologie sont distribuées au travers des descriptions des différents composants et connecteurs,
- *Les langages à configuration "in-line"* qui modélisent les configurations explicitement, mais spécifient les interconnections entre composants ainsi que leurs protocoles, "in-line".
- *Les langages à configuration explicite* où les informations sur la topologie se distinguent des informations sur les composants et les connecteurs.

L'exigence d'un modèle de configuration explicite distingue les ADL des autres langages de conception de haut niveau [MT97].

D'autre part, un ADL englobe une théorie sémantique formelle dont la spécification peut s'appuyer sur les réseaux de Pétri, les diagrammes d'états, les machines abstraites, les formalisme algébriques, etc. [MT97].

2.4 Fonctionnalité des ADLs

Les ADL peuvent être utilisés dans de nombreux domaines et leurs fonctionalités sont très diverses [HR97] :

Analyser à partir de plusieurs points de vue

Cette méthode d'analyse, utilisée intensément dans l'ensemble des domaines de l'ingénierie du logiciel, permet de gérer la complexité en séparant les différents aspects de la représentation (par exemple, textuelle, graphique, XML).

Bien que la notion de points de vue multiples soit généralement évoquée dans les méthodes industrielles, beaucoup d'ADL ne supportent pas ce concept comme, par exemple, UML peut le supporter.

Délimiter le problème

Afin de prendre des décisions architecturales avec le client, un concepteur doit généralement estimer différents choix de conception en fonction de leurs implications. Ces choix de conception sont toutefois limités par certaines exigences indépendantes du problème. Les ADL permettent donc de délimiter le problème en fixant les variables architecturales indépendamment du problème en question. Par exemple, il est important de fixer :

- les décisions prises par le client ("exigences") ;
- les décisions fixées par l'environnement (hors de contrôle de l'architecte mais influence fortement le résultat) ;
- les décisions prises par l'architecte en réponse aux exigences du client et de l'influence de l'environnement. Ces décisions doivent être enregistrées de manière à conserver une bonne traçabilité du projet.

Exprimer les choix architecturaux et leurs niveaux de liberté

Le travail d'un architecte s'articule généralement autour de nombreuses caractéristiques du système en question. Ces caractéristiques peuvent être structurelles, de style, d'interactions, etc.

Un architecte a généralement besoin d'exprimer ces caractéristiques au moyen d'un arbre représentant les degrés d'intention, en relation avec les futures activités de conception. L'architecte doit donc décrire les engagements (choix fixés par l'architecte), les obligations (bas niveau de décision donné au concepteur) et les choix libres de décision pour le concepteur.

Définir les contraintes architecturales

Bien que les composants et les connecteurs soient des constructeurs généralement acceptés dans les architectures logicielles, un consensus est récemment né sur l'expression de leurs définitions et de leurs contraintes.

Les contraintes architecturales sont généralement caractérisées par un nom, un type, une expression et un ensemble d'éléments sources et cibles.

Indépendance par rapport à l'implémentation

De nombreux ADL possèdent un modèle d'exécution implicite. Il est pourtant important de conserver une liberté de choix d'implémentation et de pouvoir retarder ce choix au maximum dans le cycle de vie d'un produit. Les ADL doivent donc s'appuyer sur un modèle d'exécution plus abstrait et

plus générique qui permettrait une plus grande indépendance vis-à-vis de la plateforme d'exécution.

Créer des patrons d'architecture

De nombreux travaux actuels dans la conception logicielle et les architectures sont centrés sur l'articulation de patrons et de styles. Pour permettre leurs supports, les ADL doivent être sufisamment modulaires afin de permettre l'expression d'entités individuelles.

2.5 Les outils support d'un ADL

Les ADL sont généralement supportés par un ou plusieurs outils permettant une manipulation aisée du langage. Ces outils, en plus de supporter l'ensemble des fonctionnalités du langage, offrent certaines fonctionnalités supplémentaires [MT97] :

Gestion des erreurs :

Chaque outil propose une gestion des erreurs de manière proactive (tâche permanente vérifiant les erreurs pouvant survenir au moment de la conception) ou réactive (détection des erreurs existantes).

Intéraction de différentes vues :

La plupart des outils proposent les deux vues basiques d'une architecture : textuelle et graphique. La gestion de vues supplémentaires est beaucoup plus rare.

Analyse de la description architecturale :

A partir du modèle sémantique de l'ADL, chaque outil propose la vérification d'un certain nombre de propriétés et le contrôle du respect des contraintes.

Raffinage d'architecture :

Le raffinage architectural trouve son importance à travers l'utilisation des styles et la manipulation de différents niveaux de détails. Les outils supportant cette fonctionnalité sont toutefois rares.

Génération de code :

L'ultime objectif de la conception d'un logiciel est de fournir une application exécutable. C'est pourquoi, un grand nombre d'outils supports d'un

ADL (mais pas tous) permettent la génération de code C++, Java, Ada, etc.

2.6 Langages pour assister la production de systèmes

2.6.1 Présentation d'ADL existants

ACME :

Le projet ACME, commencé en 1995, a pour principal but de fournir un langage commun permettant l'échange de descriptions architecturales entre plusieurs outils de conception d'architectures. Il s'agit d'un langage de description d'architectures logicielles fournissant une base conceptuelle abstraite et suffisamment générale pour permettre la description de nouveaux outils et notations. ACME a été conçu comme langage passerelle pour permettre aux différents ADL's d'intéropérer. Cette vocation l'a conduit à se définir comme langage d'architecture générique. [DGRMDW97] [ACM].

Rapide :

Rapide [LKA+95], développé par David Luckham à Stanford, est un langage de simulation d'objets concurrents basé sur les événements et qui s'appuie sur le principe de posets (Partial Ordered event Sets). Il permet la description et la simulation du comportement de systèmes à objets répartis. Cette simulation est représentée par un ensemble d'événements (posets) ordonnés par rapport au temps et en respectant la causalité inter-événements.

Aesop :

Aesop [GAO94] est un outil permettant de fédérer une famille d'environnements pour la conception et l'analyse d'architectures logicielles. La particularité de Aesop est de permettre la définition de plusieurs *styles* d'architectures, chacun étant propre à un domaine comme par exemple le domaine du temps réel.

Les styles d'architecture regroupent des patrons de conception d'architecture. Il existe plusieurs styles tels que le *Pipe* and *Filter*. Ainsi un style permet de définir un vocabulaire permettant d'identifier les éléments d'une architecture, un ensemble de règles de configuration appliquées, une sémantique particulière et un cadre d'analyse spécifique à chaque architecture. Chaque style créé dérive d'un style générique qui est composé des éléments suivants : le composant, le port, le connecteur, le rôle, la configuration, la représentation et la liaison.

Aesop fournit très peu de moyens pour décrire la sémantique des éléments tels que le composant ou le connecteur. La classe abstraite est le seul moyen

de décrire un type. Celle-ci ne fournit pas un moyen formel et précis pour décrire la sémantique associée à un type.

Darwin :

Darwin [MK96] est un ADL qui permet 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).

Le concept principal de Darwin est le composant. Un composant est décrit par une interface qui contient les services fournis et requis. Ces services s'apparentent plus aux entrées et sorties de flux de communication qu'à la notion de fonction. Deux types de composants existent. Le premier correspond aux composants primitifs qui intègrent du code logiciel, le second aux composites qui sont des entités de configuration décrivant les interconnexions entre composants.

C2 :

C2 [MORT96, Med99] permet de définir une application sous forme de réseau de composants s'exécutant de manière concurrente, liés par des connecteurs et communiquant de manière asynchrone par envoi de messages. Un composant défini dans une architecture C2 est à l'écoute des composants qui sont situés au-dessus de lui et produit des événements pour les composants situés en-dessous de lui. Chaque composant et chaque connecteur possèdent un haut (« top ») et un bas (« bottom »). Le haut d'un composant doit être connecté au bas d'un seul connecteur et le bas d'un composant est connecté au haut d'un connecteur unique. Ainsi, un composant est rattaché à l'architecture d'un système grâce à deux connecteurs dont l'un lui permet d'envoyer à travers sa partie « bottom » des événements à d'autres composants et dont l'autre lui permet d'en recevoir à travers sa partie « top ». Un connecteur peut accepter plusieurs composants. Il n'y a pas de limites du nombre de composants liés à un connecteur.

UniCon

UniCon [SDK+95, SDZ96] (Universal Connector Support) est un langage de description d'architecture crée par l'Université de Carnegie Mellon en Pennsylvanie.

Le composant dans ce langage est une entité d'encapsulation des fonctions d'un module logiciel ou des données. Un composant est caractérisé par une interface qui définit les services requis et fournis par le composant, une implantation qui permet d'établir le lien entre l'interface du composant et sa mise en œuvre.

Le connecteur permet de spécifier les règles d'interconnexion de composants. Un connecteur est caractérisé par un protocole défini par un type de connecteur et une implantation. Le protocole est défini par le type du connecteur choisi dans la liste fournie par UniCon (Par exemple, *RemoteProcCall*).

Wright

Wright [AG97, Wri] est un langage d'architecture logicielle qui se centre sur la spécification de l'architecture et de ses éléments. Il n'y a pas de générateur de code, ni de plate-forme permettant de simuler l'application. Wright décrit formellement le comportement d'une architecture à l'aide de *Communicating Sequential Process* (CSP).

Un composant en Wright est une unité abstraite localisée et indépendante. La description d'un composant contient deux parties importantes qui sont l'interface et la partie calcul (*computation*). L'interface consiste à décrire les ports, c'est-à-dire les interactions auxquelles le composant peut participer. La partie calcul, quant à elle, consiste à décrire le comportement du composant en indiquant comment celui-ci utilise les ports. Un connecteur représente une interaction entre une collection de composants.

SADL

SADL [MR97b, CKSS01] (Structural Architecture Description Language) a été conçu pour décrire la structure architecturale d'un système logiciel en termes de composants et de connecteurs ainsi que pour le raffinement de styles architecturaux. SADL permet de définir des patrons de raffinement réutilisables et pouvant être composés.

AADL :

AADL [AADb] est un langage d'analyse et de conception d'architecture basé sur l'ADL Meta-H [Metb] qui fut développé dans les années 90 pour la description d'architectures de systèmes embarqués avioniques. Il est donc orienté vers la specification de systèmes embarqués temps-réel mais pas uniquement avioniques. C'est un langage textuel et graphique permettant de manipuler les flux de contrôles et de données et de décrire les aspects non-fonctionnels importants liés à ces systèmes (exigences temporelles, fautes, propriétés de sûreté et de certification,etc.). Le **chapitre 3** comporte une étude détaillée du langage AADL.

2.6.2 Utilisation d'UML comme ADL

La description architecturale d'un système peut ce faire à partir de langages spécifiques (ADL) mais également à partir de notations génériques

existantes. Ainsi un certain nombre de travaux [UMLa] s'orientent sur l'utilisation de la syntaxe et du vocabulaire d'UML pour la conception non seulement des applications, mais aussi des architectures matérielles. Il s'agit donc d'utiliser UML comme un ADL. En effet la notation UML, depuis sa nouvelle version 2.0[UMLb], introduit la notion de composant et de diagramme structurel facilitant la modélisation d'architecture. UML2 considère également les notions de description matérielle et de mapping.

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, MARTE (*Modeling and Analysis of Real-Time and Embedded*) [MAR] permet de prendre en compte les contraintes liées aux systèmes embarqués. D'autres profils, comme SysML [Boc06], qui permet de regrouper dans un modèle commun à tous les corps de métiers, les spécifications, les contraintes, et les paramètres de l'ensemble du système. SysML n'aborde plus la conception avec la notion de classes mais avec la notion de blocs qui deviendront des parties mécaniques, électroniques, informatiques ou autres.

2.6.3 Langages spécifiques

Les langages spécifiques sont conçus pour répondre à des besoins particuliers de modélisation et d'analyse. Nous citons dans cette section Fractal [BCL+06] et Ptolemy [EJL+03].

Fractal :

Le modèle de composants Fractal est un modèle général dédié à la construction, au déploiement et à l'administration (e.g. observation, contrôle, reconfiguration dynamique) de systèmes logiciels complexes. Le modèle de composants Fractal a été défini par France Telecom R&D et l'INRIA. Il se présente sous la forme d'une spécification et d'implémentations dans différents langages de programmation comme Java, C, C++, SmallTalk ou les langages de la plate-forme .NET.

La base du développement Fractal réside dans l'écriture de composants et de liaisons permettant aux composants de communiquer. Ces composants peuvent être typés. Un composant Fractal est une entité d'exécution qui possède une ou plusieurs interfaces. Il existe deux catégories d'interfaces : les interfaces *serveurs* qui correspondent aux services fournis par le composant, et les interfaces *clients* qui correspondent aux services requis par le composant.

Fractal définit deux types de liaisons : primitive et composite. Les liaisons *primitives* sont établies entre une interface client et une interface serveur de deux composants résidant dans le même espace d'adressage. Les liaisons

composites sont des chemins de communication arbitrairement complexes entre deux interfaces de composants. Les liaisons composites sont constituées d'un ensemble de composants de liaison (e.g. *stub*, *skeleton*) reliés par des liaisons primitives.

Ptolemy :

Ptolemy II est développé par une équipe du département EECS (*Electrical Engineering and Computer Sciences*) de l'université Berkeley en Californie. Il est principalement utilisé pour implémenter et expérimenter divers techniques et travaux de recherche comme la définition de modèles d'ordonnancement (la façon d'exécuter les tâches dans un workflow), la génération de code à partir d'un modèle ou encore la conception objet.

Ptolemy II introduit la notion de domaine de polymorphisme (où les composants peuvent être conçus pour fonctionner dans des domaines multiples) et des modèles modaux (où des machines à états finies sont combinées hiérarchiquement avec d'autres modèles de calcul).

Ptolemy II possède un système de typage de données avec inférence de type et le polymorphisme de données et une sémantique très riche. Le concept de types comportementaux est alors apparu : composants et domaines peuvent avoir des définitions d'interfaces décrivant statiquement leur structure, mais aussi leur dynamique.

Ptolemy intègre la notion de classes et sous-classes orientés acteurs avec héritage et inclut aussi la modélisation de systèmes sans fil. Il s'est également vu doté de la gestion du cycle de vie des acteurs, de l'évaluation dynamique des composants « *higher-order* », et ajouté de nombreux domaines expérimentaux pour l'exploration du temps réel et du calcul distribué. La vérification des modèles n'est pas possible pour l'instant, un projet vise à ajouter cette possibilité.

2.7 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.

La comparaison entre les langues ADLs est représenter dans les figures 2.1, 2.2. Cette comparaison est basée sur : les composants, les connexions, les priorités entre les composants et la description du comportement.

Tous les langages font des distinctions entre l'interface d'un composant et l'instance d'un composant. Tous les langages fournissent la syntaxe et la sémantique de la spécifications d'interfaces du composant. Tous les langages montrent une spécification de l'interface d'un composant comme la définition d'un type de composant, où l'on peut instancier plusieurs composants qui partagent la même interface. Tous les langages permettent une composition

	Composants		
	Interface	**Implantation**	**Propriétés non-fonctionnelles**
C2	exportée à travers les ports *top* et *buttom* ; *provided* et *required*	composant implantation	–
SADL	ports d'entrée et de sortie (*iports* et *oports*)	composant implantation	exige la modification des composants
Rapide	*provides*, *requires*, *action* et *service*	interface implantation	–
Darwin	services (*provided* et *required*)	composant implantation	–
Unicon	*players*	composant implantation	attributs pour l'analyse d'ordonnançabilité
AADL	ports d'entrée et de sortie (*event* et/ou *data*) ; *provide* and *require* ; paramètres d'entrée et de sortie	composant implantation	contraintes de temps ordonnançabilité propriétés niveau de sécurité

FIG. 2.1 – Comparaison entre les ADLs

hiérarchisée qui permet de décrire les architectures des systèmes logiciels à différents niveaux, en utilisant une collection de sous-composants et les connexions entre ces sous-composants.

Certains de ces langages, en se plaçant à un haut niveau d'abstraction, permettent la spécification formelle et la vérification des descriptions architecturales. Ainsi quelques langages proposent 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 d'autres langages, qui prévoient également d'être associés à UML pour constituer les dernières étapes de la production de l'application. D'autres langages comme MetaH ou AADL se placent à un niveau de modélisation beaucoup plus bas ; ils 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.

Plusieurs chercheurs ont tenté de faire la lumière sur les comparaisons des langages, soit en faisant l'état de l'art de ce qu'ils considèrent des ADL existantes [Cle96], [Ves93] ou par listing des « exigences essentielles » pour un ADL [LV95], [SDK+95] ou pour comprendre et comparer les ADL sur les problèmes au sein d'architectures logiciels pour lesquelles ils sont adaptés [MR97a].

Les ADL's n'ont pas réussi à surmonter les obstacles d'incompatiblilité, de non portabilité, de multiplicité et de complexité, d'absence de standardisation, etc., ce qui leur aurait permis de dépasser le cadre de leur niche

	Connecteurs	**Priorités**	**Comportement**
C2	interface avec chaque composant via un port distinct ; interface élémentaire *provided* et *required*	priorité *low* et *high*	se compose d'un invariant et d'un ensemble d'opérations. L'invariant est utilisé pour spécifier les propriétés qui doit être vrai pour tous les états du composant
SADL	précise les types de données pris en charge	planification du processus utilisant une priorité statique	calcul mathématique
Rapide	connexion ; *in-line*	information prioritaires pour l'analyse d'ordonnançabilité	consiste d'un ensemble de règle de transitions
Darwin	*binding* ; *in-line* ; pas de modélisation explicite des interactions entre les composants	information prioritaires pour l'analyse d'ordonnançabilité	CORBA
Unicon	connecteur	information prioritaires pour l'analyse d'ordonnançabilité	attributs
AADL	connecteur (paramètres, accès aux données, ports)	niveau de sécurité	sous-programmes ; C/C++ ; ADA ; machine d'état

FIG. 2.2 – Comparaison entre les ADLs

d'origine et de s'imposer comme langages de référence.

Une réponse a été trouvée en intégrant la technologie XML et ses dérivés pour la spécification, la manipulation et l'utilisation des langages. C'est ce qui conduit à la respécification de certains ADL's par les méta-langages d'XML (DTD et XML schema), ou à une redéfinition nouvelle générique et hiérarchisée.

2.8 Conclusion

L'étape de description architecturale est de plus en plus fréquente dans la phase de conception d'un logiciel. Elle offre au concepteur un niveau d'abstraction plus élevé en lui permettant de ne pas prendre en compte les détails de l'architecture. Seuls des concepts architecturaux complètent la conception et permettent ainsi de la rendre totalement générique par rapport à une architecture particulière. Les travaux de conception pourront alors être réutilisés ou instantiés sur différentes plateformes.

Pour cela de nombreux travaux de recherche ont abouti sur la création de langages de description d'architecture. Ces langages sont spécifiques à un domaine particulier ou transversaux dans l'activité de description architecturale. Ils peuvent également se baser sur une syntaxe bien particulière ou sur des standards génériques comme UML. Pour améliorer la manipulation de ces langages, ils sont généralement fournis avec des outils qui apportent une certaine productivité dans l'étape de description architecturale.

Chapitre 3

Langage de Description et d'Analyse d'Architectures

Sommaire

AADL [AADb] (Architecture Analysis & Design Language) est un langage de description d'architectures. Il trouve son origine dans MetaH [Metb], qui fut développé dans les années 90 par Honeywell. Ce langage permet la

conception et la description d'architectures temps-réel embarquées, en considérant simultanément tous les aspects logiciels et matériels sous-jascents. C'est la SAE (Society of Automotive Engineers) qui a préparé le standard international d'AADL depuis 2001, dont la version 1.0 est parue en 2004. Le comité de standardisation comporte les principaux acteurs de l'industrie avionique et aérospatiale, en Europe comme aux USA (Honeywell, ESA, Airbus, Rockwell Collins, EADS, etc.). AADL joue un rôle central au sein de plusieurs projets tels que SPICES [SPI], OpenEMBeDD [Ope], ASSERT [ASS] et au sein de plusieurs outils tels que TOPCASED [TOP], OSATE [OSA], OCARINA [OCA], etc. 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 illustrons notre propos par une série d'exemples décrivant la description progressive d'un système.

3.1 Principes du langage

AADL peut être exprimé selon différentes syntaxes. Le standard [AADb] en définit trois : en texte, 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 souci 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 déclarations de composants. Ces déclarations peuvent être instanciées pour former la modélisation d'une architecture.

3.2 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 partagent la même interface.

3.2.1 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 3.1). AADL étant un langage déclaratif, l'ordre des déclarations n'importe pas : il est possible de déclarer une implantation avant le type correspondant.

```
processor myProcessor
end myProcessor;

processor implementation myProcessor.Intel
end myProcessor.Intel;

processor newProcessor
end newProcessor;

processor implementation newProcessor.newIntel
                         extends myProcessor.Intel
end newProcessor.newIntel;
```

Listing 3.1 – 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 3.1). La figure 3.1 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.

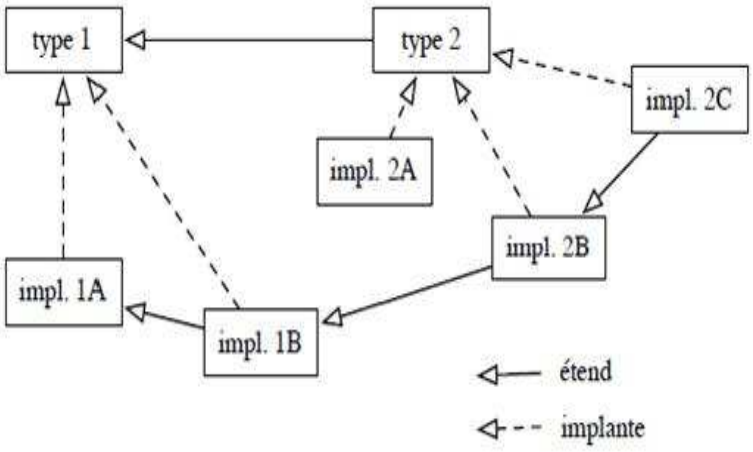


FIG. 3.1 – Possibilités d'extension des composants

L'extension de composant AADL ne correspond pas exactement à la notion d'héritage des langages objet : un composant et son extension con-

stituent 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.

3.2.2 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.

3.2.2.1 Composants logiciels

Il existe quatre catégories de composants logiciels :

- les données ;
- les sous-programmes ;
- les fils d'exécution (threads) ;
- les processus.

(a) : Données (b) : Sous-programmes (c) : Threads (d) : Processus

FIG. 3.2 – Syntaxe graphique des composants logiciels

La figure 3.2 illustre la représentation des différentes catégories de composants logiciels selon la syntaxe graphique d'AADL.

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.

3.2.2.1.1 Données : les données représentent les structures de données qui peuvent être stockées ou échangées entre les composants. Le listing 3.2 illustre un exemple du composant type donnée et son implémentation qui contient d'autres composants de données.

```
data Person
end Person;

data implementation Person.impl
   subcomponents
        Name : data string;
        Adress: data string;
        Age : data integer;
end Person.impl;
```

Listing 3.2 – Exemple du composant donnée

3.2.2.1.2 Sous-programmes : Les sous-programmes représentent un fragment de code séquentiel exécutable, qui est appelé avec des paramètres.

Le listing 3.3 illustre un exemple d'un composant sous-programme, qui prend en entrée deux entiers A et B, et produit un entier *result* en sortie.

```
subprogram operation
   features
        A: in parameter integer;
        B: in parameter integer;
        result: out parameter integer;
end operation;
```

Listing 3.3 – Exemple du composant sous-programme

3.2.2.1.3 Threads : 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 propriétés du thread sont utilisées pour représenter les attributs et d'autres caractéristiques, telles que la période (*period*), la politique de déclenchement (*dispatch protocol*), l'échéance (*deadline*), etc. La propriété *period* spécifie l'intervalle de temps entre deux déclenchements d'un thread. La propriété *dispatch_protocol* est une propriété qui définit le comportement de déclenchement d'un thread.

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

- périodique : Les threads périodiques se déclenchent d'eux-mêmes selon la période indiquée par la propriété *Period*;
- apériodique : Les threads apériodiques se déclenchent sur l'arrivée d'un événement ou d'une donnée/événement sur l'un de leurs ports;

– sporadique : Les threads sporadiques se déclenchent également à l'arrivée d'un événement, mais avec une période de garde indiquée par la propriété *Period* ;
– tâche de fond (background) : Les threads en tâche de fond s'exécutent en continu.

Le listing 3.4 illustre un exemple du composant thread appelé sensor. C'est un thread périodique avec une période de $20ms$. Ce thread reçoit une donnée entière à travers le port *inp* et envoie un événement à travers le port *outp*.

```
thread sensor
   features
        inp : in data port integer;
        outp : out event port;
   properties
        Dispatch_protocol=>Periodic;
        Period => 20ms;
end sensor;
```

Listing 3.4 – Exemple du composant thread

3.2.2.1.4 Processus : Les processus définissent des espaces mémoire dans lesquels s'exécutent les threads. Le listing 3.5 illustre un exemple du composant processus appelé Partition, qui contient des sous-composants threads et de deux types de connexions.

```
process Partition
end Partition;

process implementation Partition.Impl
   subcomponents
        Sensor_A : thread Sensor ;
        Data_Fusion: thread Fusion;
        Alrm : thread Alrm_Thread;
   connections
        data port Sensor_A.outp->Data_Fusion.inpA;
        event port Sensor_A.launch->Alrm.launch A;
end Partition.Impl;
```

Listing 3.5 – Exemple du composant processus

3.2.2.2 Composants de plate-forme

Il existe quatre catégories de composants de plate-forme :
– les processeurs ;

– les mémoires ;
– les bus ;
– les devices (dispositifs).

(a) : Processeurs (b) : mémoires (c) : bus (d) : devices

FIG. 3.3 – Syntaxe graphique des composants matériels

La figure 3.3 illustre la représentation des différentes catégories de composants de plateforme selon la syntaxe graphique d'AADL.

3.2.2.2.1 Processeurs Les processeurs représentent des ensembles constitués d'un microprocesseur combiné à un ordonnanceur ; ils modélisent donc un processeur associé à un système d'exploitation minimal.

3.2.2.2.2 Mémoires Les mémoires représentent tous les dispositifs de stockage : disque dur, mémoire vive, etc.

3.2.2.2.3 Bus Les bus modélisent toutes les sortes de réseaux ou de bus de communication, 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.

3.2.2.2.4 Devices 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.

3.2.2.3 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. Le composant système représente un composant composite comme un ensemble de composants logiciels et de plate-forme d'exécution.

FIG. 3.4 – Syntaxe graphiques des composants système

composant	**sous-composants**
donnée	données
sous-programme	-
thread	donnée
processus	thread, donnée
processeur	mémoires
mémoire	mémoires
bus	-
device	-
système	donnée, thread, processus, processeur, mémoire, bus, device, systèmes

FIG. 3.5 – Compositions légales des composants AADL

La figure 3.4 illustre la représentation des systèmes selon la syntaxe graphique d'AADL.

3.3 Structure interne des composants

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

3.3.1 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 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. La figure 3.5 représente la synthèse des compositions possibles.

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. 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 device 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 , 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.

3.3.2 Appels de sous-programmes

Les appels de sous-programmes 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. Le listing 3.6 illustre un exemple du thread qui appelle deux sous-programmes.

```
subprogram sp_a
end sp_a;

subprogram sp_b
end sp_b;

thread thread_sp
end thread_sp;

thread implementation thread_sp.impl
  calls {
        appel1 : subprogram sp_a;
        appel2 : subprogram sp_b;};
end thread_a.impl;
```

Listing 3.6 – Structure interne des composants AADL

3.4 Les éléments d'interface

Les éléments d'interface (*features*) d'un composant sont déclarés 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

d'élément d'interface : les ports de communication, les paramètres du sous-programme et les accès à des sous-composants.

3.4.1 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.

3.4.1.1 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, processus, devices, systèmes et processeurs. Dans le cas des sous-programmes, on parle de paramètres.

3.4.1.2 Groupes de ports

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 ports 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.

3.4.2 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 devices et les mémoires.

3.4.3 Les paramètres de sous-programme

Les paramètres décrivent les transmissions d'information entre les sous-programmes. Ils sont déclarés en entrée, en sortie ou en entrée/sortie. Les paramètres ont une sémantique équivalente aux ports de données ou d'événement/données.

3.5 Connexions des composants

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

3.5.1 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.

Une connexion AADL est un lien qui représente la communication de données et de contrôle entre les composants, plus précisement, entre les ports de différents threads ou entre les threads et les processeurs ou les devices. Il existe trois types de connexions :

3.5.1.1 Les connexions de port

La figure 3.6 représente les connexions de port entre le thread d'origine et le thread de destination en passant par une connexion hiérarchique à travers les processus.

AADL définit trois type de connexions du port :

3.5.1.1.1 Connexions de donnée (*data port connection*) : une connexion de donnée est un lien qui représente la communication de données entre les composants. Cela peut être la transmission de données entre un émetteur et un récepteur. AADL définit deux catégories de connexions de donnée :

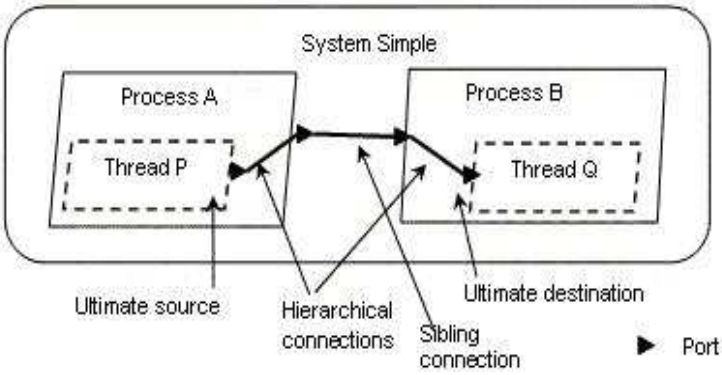


FIG. 3.6 – Les connexions de port

– **Connexion Immédiate** (*Immediate Connection*) : La transmission de données est déclenchée lorsque le thread émetteur termine le calcul (*computation*). Une Connexion Immédiate est représenté dans la figure 3.7, Thread A et Thread B sont deux threads qui s'exécutent périodiquement à un taux de 10Hz, c'est-à-dire, ils sont logiquement déclenchés toutes les 100 ms. Pour les deux threads illustrés dans la figure 3.7, une spécification textuelle est montrée dans le listing 3.7.

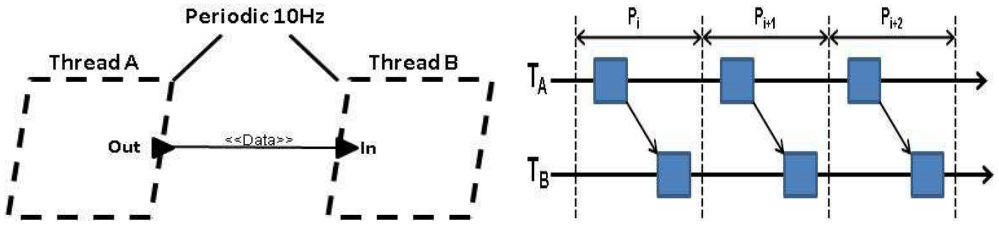


FIG. 3.7 – Connexion Immédiate

– **Connexion Retardée** (*Delayed Connection*) : La transmission de données est déclenchée à la date limite (*deadline*) du thread émetteur. La figure 3.8 montre le même système que la figure 3.7 avec la différence de la déclaration de type de connexion (immédiate ou retardée).

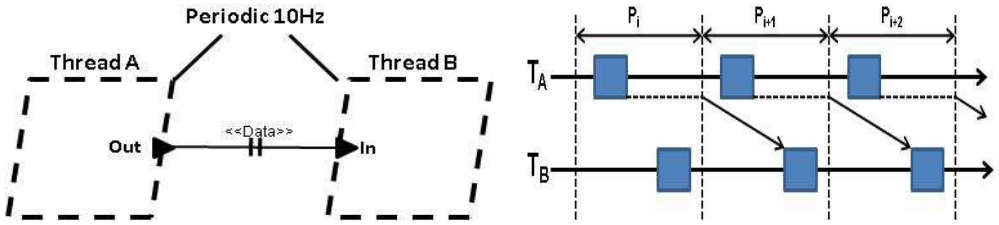


FIG. 3.8 – Connexion Retardée

```
thread ThreadA
   features
        outp : out data port integer;
   properties
        Dispatch_protocol=>Periodic;
        Period => 100ms;
end ThreadA;

thread ThreadB
   features
        inp : in data port integer;
   properties
        Dispatch_protocol=>Periodic;
        Period => 100ms;
end ThreadB;

process Partition
end Partition;

process implementation Partition.Impl
   subcomponents
        T1 : thread ThreadA;
        T2 : thread ThreadB;
   connections
        cnx : data port T1.outp->T2.inp;
end Partition.Impl;
```

Listing 3.7 – Connexion Immédiate

3.5.1.1.2 Connexions d'événement : Représentent la transmission d'événement, elles peuvent aussi déclencher l'exécution des threads à travers le port d'événement.

3.5.1.1.3 Connexions d'événement et de donnée : Représentent la transmission d'événement et de donnée, elles peuvent aussi déclencher l'exécution des threads à travers le port d'événement.

3.5.1.2 Les connexions de paramètre

Les connexions de paramètre (*Parameter Connections*) représentent le flux des données entre les paramètres d'une séquence d'appels de sous-programme dans un thread. La figure 3.9 illustre un exemple de connexions de paramètre entre un thread et un sous-programme, entre des sous-

programmes de mêmes et de différents niveaux.

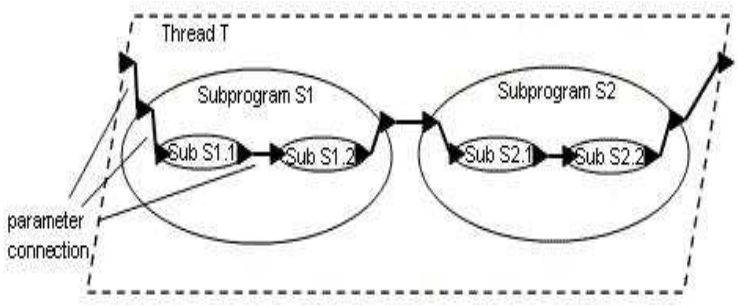


FIG. 3.9 – Connexions de parametère

3.5.1.3 Les connexions d'accès

Les connexions d'accès (*Access connections*) représentent l'accès à des éléments de données simultanément par des threads d'exécution ou par des sous-programmes d'exécution au sein du thread. Ils indiquent également la communication entre les processeurs, la mémoire et des périphériques en accédant à un bus partagé. figure 3.10 illustre un exemple des connexions d'accès.

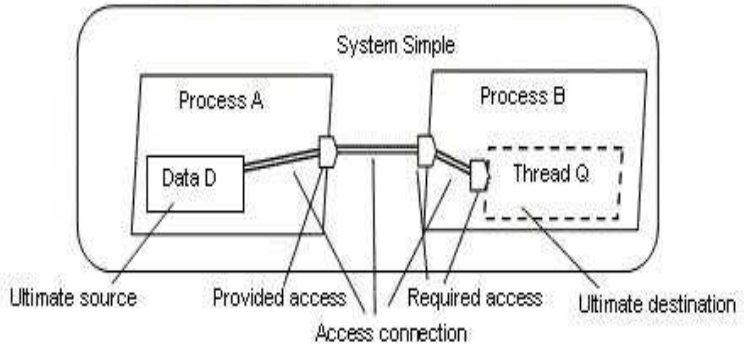


FIG. 3.10 – Connexions d'accès

3.5.2 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 fonction 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.

3.5.3 Matérialisation des connecteurs

AADL est conçu pour refléter l'architecture de systèmes réels. Les composants constituent donc les principales entités du langage, 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.

3.6 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.

3.6.1 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.

3.6.2 Les modes

AADL décrit des architectures aux dimensions clairement définies : il n'est pas possible par exemple 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 événement (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.

3.7 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.

3.7.1 Déclarations de propriétés

Les déclarations 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.

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.

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.

3.7.1.1 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* comme décrit dans la section 3.2.2.1.3.

3.7.1.2 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, devices et processeurs, permettant ainsi de décrire les composants logiciels (sous-programmes, par exemple en C).

3.7.1.3 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* permet d'exprimer les contraintes en taille mémoire.

3.7.1.4 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 utilisées pour indiquer respectivement par quel bus, mémoire ou processeur est porté une connexion, une donnée ou processus. Ces propriétés peuvent être associer à des propriétés pour exprimer des contraintes permettant la synthèse du bindings.

3.7.2 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).

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.

3.8 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 exprimées dans une syntaxe indépendante. L'utilisation des annexes permet donc d'étendre la syntaxe standard d'AADL afin de spécifier le comportement des composants [ABS].

Le comportement spécifié par l'annexe est décrit par un système de transition (un automate étendu). La syntaxe de l'annexe comportementale est décrite dans le listing 3.8.

```
annex behavior_specification {**
  [ state variables
     (Identifier : data_type;)+ ]
  [ initial (<assignment> ; )+ ]
   states
     (state_id : [initial] [complete] state;)+
   transitions
     (  <state_id> -[ <guard> ]
           -> <state_id> { <action>* }; )+
**};
```

Listing 3.8 – La syntaxe de l'annexe comportementale

L'annexe comportementale est définie par les éléments suivants :

- Les Variables d'état (*state variables*) : Cette section permet de déclarer les variables. Elles doivent être initialisées dans la section d'initialisation (*initialization*).
- Les états (*States*) : Cette section permet de déclarer les états.
- Les *Transitions* : Cette section permet de définir les transitions de l'état source vers l'état de destination. La transition peut avoir une garde (une condition booléenne par exemple) et une action à exécuter si la garde est vraie.

La garde d'annexe comportementale peut prendre trois valeur :

- la garde est toujours vraie ;

– la garde sous forme d'une *expression* ;
– la garde sous forme d'une *expression* et d'un port (*event*).

La syntaxe de la garde est décrite dans le listing 3.9.

```
<guard> ::=
        [on <expression> -->] <event>
        | <expression>
        |
```

Listing 3.9 – La syntaxe des gardes

La syntaxe de l'action est décrite dans le listing 3.10.

```
action ::=
          computation ( expression ,expression );
        | delay ( expression ,expression );
        | communication ;
        | assignment ;
```

Listing 3.10 – La syntaxe des actions

Le listing 3.11 illustre un exemple d'annexe comportementale d'AADL.

```
thread Producer
 features
   Data_Source: out event data port Behavior::integer;
end Producer;

thread implementation Producer.Impl
 properties
   Dispatch_Protocol => Periodic;
   Period => 10 ms;
 annex behavior_specification {**
   state variables
        c : Behavior::integer;
   initial c := 0;
   states
     s0 : initial complete state;
     transitions
     s0 -[Data_Source!(c)]-> s0 { c:=c+1; };
  **};
end Producer.Impl;
```

Listing 3.11 – Exemple d'annexe comportementale

Cette exemple représente le comportement d'un thread nommé *Producer*. C'est un thread périodique avec une période de $10ms$. Ce thread contient une annexe comportementale, qui défini une variable c de type entier avec

une valeur initial $c = 0$. L'état initial de l'annexe comportementale est l'état *s0* à partir duquel il peut se déplacer vers l'état *s1* à travers une interaction sur le port *Data_Source*. Lors de la transition la valeur de c est incrémentée.

3.9 Conclusion

AADL permet de décrire les architectures avec une approche très concrète ; le langage offre un ensemble de catégories de composant, réparties en trois grandes familles : logiciels, matériels et systèmes.

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.

Chapitre 4

BIP, un langage à base de composants hétérogènes

Sommaire

BIP [Bas08] [BBS06] (*Behavior*, *Interaction*, *Priority*) est un cadre théorique pour la construction et l'analyse des systèmes temps réel à partir de composants hétérogènes. Ce langage permet la conception et la description comportementale d'architectures temps-réel embarquées. BIP trouve son inspiration dans la chaîne d'outils IF [BGO+04] (Platforme pour la modélisation et la validation des systèmes temps-réel) et dans l'outil PROMETHEUS [Gößl01] (Outil de modélisation des systèmes temps-réel) qui sont développés aussi à VERIMAG. BIP est développé à VERIMAG, la version 2.0 est parue en 2008. Dans ce chapitre, nous présentons la version 2.0 du langage qui remplace la version 1.0.

4.1 Principes du langage

Les modèles BIP sont obtenus par la superposition de trois couches de modélisation représentées sur la figure 4.1 :

- la couche inférieure décrit le comportement d'un composant comme un ensemble de transitions (c'est-à-dire un automate à états finis étendus avec des données) ;
- la couche intermédiaire inclue les connecteurs décrivant les interactions entre les transitions de la couche de comportement ;
- la couche supérieure est constituée d'un ensemble de règles de priorité utilisées pour décrire les politiques de planification des interactions.

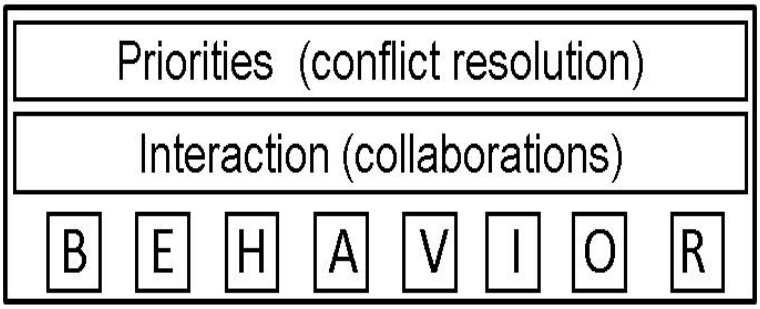


FIG. 4.1 – Architecture de BIP

Le modèle BIP offre une séparation claire entre le comportement des composants et la structure d'un système (interactions et priorités). Le modèle BIP permet une construction hiérarchique [GG05] des composants composites à partir des composants atomiques en utilisant les connecteurs et les priorités.

4.2 Le langage BIP

Le langage BIP fournit des constructions syntaxiques structurelles pour la définition du comportement de composants, en précisant la coordination par l'intermédiaire des connecteurs et des priorités. BIP permet d'exprimer la concurrence et le comportement séquentiel des systèmes, comme l'interconnexion de composants.

Un système BIP peut être décrit hiérarchiquement et peut également exprimer des informations temporisées dans la même description.

Les principales constructions du langage BIP sont :

- *atome* : pour préciser le comportement avec une interface composée de ports. Le comportement est décrit comme un ensemble de transitions. Les transitions sont étiquetées par des ports.
- *connecteur* : pour spécifier les coordinations entre les ports des composants et les actions.

- *priorité* : pour restreindre les interactions possibles basées sur des conditions, en fonction de l'état des composants intégrés.
- *composite* : pour spécifier la hiérarchie du système à partir des atomes ou des composites, avec l'utilisation des connecteurs et des priorités.
- *modèle* : pour spécifier le plus haut niveau du système.

4.2.1 Composants atomiques

Les atomes représentent les feuilles dans la hiérarchie des composants. Un type atomique définit un atome, il est caractérisé par ses ports et une liste des paramètres optionnels. La syntaxe d'un composant atomique est la suivante :

atomic type *atom-type-name* [**(** *parameter-list* **)**]
variable-definition
port-definition
place-definition
[*initial-block*]
transition-definition
[*priority-definition*]
[*export-definition*]
end

Nous allons décrire ci-dessous les éléments de la syntaxe.

Les variables

Les variables sont utilisées dans les atomes. Les variables sont typées comme dans le langage C et une valeur initiale peut leur être attribuée.

Les ports

Le langage BIP permet la définition des types des ports (*Port Type*). La définition du type de port est caractérisée par le nombre et le type de ses données associées. Voici un exemple de déclaration de type de port en BIP :

port type `intPort(` **int** `i)`

Le type du port est nommé par `intPort`, il définit un port qui est associée à une variable entière i.

Les ports sont des instances de types de port, ils peuvent être associés à des variables. Un port peut faire référence à plusieurs variables. Une variable peut être référencée par plusieurs ports.

Un port peut être interne, c'est-à-dire, qu'il représente une action interne qui n'a pas besoin de synchronisation explicite pour son déclenchement. Comme il peut être exporté pour le rendre visible dans l'interface du composant.

Les états (Places)

Le comportement d'un composant atomique peut être représenté soit comme un automate soit comme un réseau de Pétri. Les places représentent les états de contrôle pour les automates, ou les places pour le réseau de Pétri. Par exemple : la ligne 6 du listing 4.1 décrit les trois états `idle`, `empty` et `full`.

Bloc initial

Le bloc initial (où l'initialisation) est utilisé pour initialiser les états de contrôle et les variables. A titre d'exemple, la ligne 7 du listing 4.1 décrit l'état `idle` comme état initial, et initialise une variable $x = 0$.

Les transitions

Le comportement est donné par une série de transitions qui permet de modéliser les étapes de calcul atomique. Une transition est définie par une étiquette (un nom de port) après le mot clé **on**. La(les) source(s) et la(les) destination(s) d'une transition sont suivis respectivement par les mots clés **from** et **to**. BIP définit deux types de garde : une garde non-temporisée et une garde temporisée. Une garde non-temporisée est une expression booléenne construite à partir des variables atomiques. Une garde temporisée est utile pour la modélisation des systèmes temporisés, c'est une expression des variables temporisées. La garde temporisée est détaillée dans la section 7.2. La syntaxe d'une transition est la suivante :

on *port-name* **from** *place-list* **to** *place-list*
[**provided** *guard-expression*]
[*timed-guard-expression*]
[**do** *statement-list*]

Les priorités dans les atomes

Le langage offre la possibilité de spécifier une priorité entre les ports d'un atome. Supposons que nous avons deux ports `in` et `out` comme décrit dans le listing 4.1. Nous pouvons spécifier que le port `out` est prioritaire par rapport au port `in` comme suit :

priority P `in` $<$ `out`

Exemple

Le listing 4.1 montre un exemple d'un composant atomique nommé *Reactive*. Ce composant contient trois ports `in`, `out`, `awake` ; deux variables

`x`, `y` ; et trois états de contrôle `Idle`, `Empty` et `Full`. L'état initial du composant *Reactive* est `Idle`, à partir duquel il peut se déplacer vers l'état `Empty` à travers une interaction sur le port `awake`. Lorsque le composant est dans l'état `Empty`, une transition à travers le port `in` peut avoir lieu si et seulement si `0<x`. Dans l'état `Full`, une transition à travers le port `out` peut se produire et permettra de se déplacer vers l'état `Idle`. La représentation graphique de cet exemple est décrite dans la figure 4.2.

```
1  atomic type Reactive
2      data int x, y
3      export port intPort in(x) = in
4      export port intPort out(y) = out
5      port ePort awake
6      place Idle, Full, Empty
7      initial to Idle do x=0;
8      on awake
9          from Idle to Empty
10     on in
11         from Empty to Full provided 0<x do y=f(x);
12     on out
13         from Full to Idle
14 end
```

Listing 4.1 – Description du composant Reactive

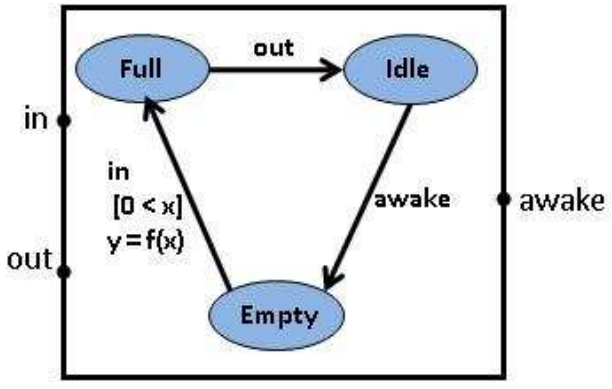


FIG. 4.2 – Représentation graphique du composant Reactive en BIP

4.2.2 Connecteurs

Un connecteur définit l'ensemble des interactions possibles entre les ports des composants et le transfert des données entre les variables associées avec les ports. Un type de connecteur définit une famille de connecteur paramétré

par une liste de ports. Il pourra être instancié plusieurs fois. La syntaxe est la suivante :

connector type *connector-type-name* **(** *port-list* **)** [**(** *parameter-list* **)**]
  define *port-expression*
  variable-definition
  (**on** *interaction* **provided** *guard* **up {** *up-statement-list* **}**
  down { *dn-statement-list* **}**)+
  [*export-definition*]
end

Un connecteur peut contenir des variables locales et une liste d'interactions gardées avec les transferts de données. La garde est limitée aux expressions booléennes sur les variables associées avec les ports.

Pour chaque interaction, le transfert de données est définie par deux actions `up` et `down`. L'action `up` est censée mettre à jour les variables locales du connecteur à partir des valeurs des variables associées avec les ports. Inversement, l'action `down` est censée mettre à jour les variables associées aux ports à partir des valeurs des variables du connecteur. Les actions (`up` et `down`) et les gardes sont respectivement des instructions et des expressions du langage C.

Nous utilisons une syntaxe graphique pour représenter les connecteurs. La figure 4.3 montre deux exemples de deux types de connecteur BIP (Rendezvous et Broadcast).

(a) : Rendezvous (b) : Broadcast

FIG. 4.3 – Types du connecteur

Le type de connecteur Rendezvous est paramétré par trois ports p_1, p_2 et p_3. Les ports p_1 et p_2 ont le même type $intPort$ tandis que le port p_3 est de type $int2Port$ qui est associé à deux variables entières. La représentation textuelle de la figure 4.3 (a) est donnée dans le listing 4.2 avec la prise en compte que le connecteur contient une garde et une fonction de transfert.

Le connecteur Rendezvous déclare une variable locale x de type entier. L'interaction entre les trois ports peut se produire lorsque la garde (`p1.i + p2.i != p3.i1 + p3.i2`) est vraie. L'action `up` décrit que le MAX des variables associées avec les ports est calculé et stocké dans la variable x. Dans l'action `down`, la valeur de x est affectée à chaque variable de port.

Le deuxième exemple définit un type de connecteur Broadcast comme le montre la figure 4.3 (b). Il est paramétré par deux ports p_1 et p_2. Le port p_1 est un port déclencheur (*trigger*, représentation graphique : ▲). La représentation textuelle de la figure 4.3 (b) est décrite dans le listing 4.3.

```
connector type Rendezvous (intPort p1,
                           intPort p2, int2Port p3)
   define [p1 p2 p3]
   data int x
   on p1 p2 p3 provided (p1.i + p2.i != p3.i1 + p3.i2)
      up {x = MAX(p1.i, p2.i, p3.i1, p3.i2);}
      down {p1.i = p2.i = p3.i1 = p3.i2 = x;}
end
```

Listing 4.2 – Description du connecteur Rendezvous

```
connector type Broadcast (intPort p1, intPort p2)
   define [p1' p2]
   data int x
   on p1
      up {x = p1.i;}
      down {}
   on p1 p2
      up {x = p1.i;}
      down {p2.i = x;}
end
```

Listing 4.3 – Description du connecteur Broadcast

Contrairement à d'autres formalismes, BIP ne fait pas de distinction entre les ports d'entrée et de sortie.

Un connecteur a une option pour définir un port et de l'exporter. Cela permet à un connecteur d'être utilisé comme un port dans d'autres connecteurs, et de créer des connecteurs structurés. Les variables locales du connecteur peuvent être associés au port exporté. Cela permet de gérer le mécanisme de transfert de données hiérarchiques.

Nous reprenons le même exemple d'écrit dans le listing 4.3, et nous rajoutons la déclaration de l'exportation d'un port comme décrit dans le listing 4.4.

```
connector type Broadcast (intPort p1, intPort p2)
   ...
   export intPort p(x)
end
```

Listing 4.4 – Description de l'exportation d'un port du connecteur Broadcast

La déclaration de l'exportation d'un port permet de créer un port p de type *intPort*, et d'associé à ce port la variable x du connecteur. Ceci conduit à deux possibilités :

- le connecteur de type Broadcast peut être utilisé comme un *intPort* dans un autre connecteur, conduisant à un connecteur structuré,
- il peut être exporté comme un port de type *intPort* dans un type de composant composite (les ports du composite sont décrits dans la section 4.2.4).

4.2.3 Priorités

Les priorités sont exprimées par un ensemble de règles. Chaque règle définit un ordre entre une paire d'interactions ou de connecteurs. Il y a deux types de priorités :

- *dynamique* : lorsque une priorité est exprimée entre deux connecteurs avec une garde (condition sur l'état du composant).
- *statique* : la garde de la priorité est toujours vraie.

Deux exemples de règles de priorité sont montrés dans le listing 7.5.

```
1 priority P1 C1:R1.in < C2:R1.out, R2.in
2 priority P2 C2 < C4 provided (R2.x > 0)
```

Listing 4.5 – Exemple de priorités

La priorité nommée par $P1$ (ligne 1) est une priorité statique entre deux interactions. La priorité nommée par $P2$ (ligne 2) est une priorité dynamique entre deux connecteurs (toute interaction du connecteur $C2$ est moins prioritaire par rapport à toute interaction du connecteur $C4$, si la condition est vrai).

4.2.4 Composants composite

Un type composite définit un nouveau type de composant à partir de composants existants (atome ou composite) par la création de leurs instances, l'instanciation des connecteurs et en précisant les priorités. Ainsi, le type du composant définit un modèle structurel de composants existants. Un composite offre la même interface que l'atome, c'est pour cela qu'il n'y a pas de différence entre un composite et un atome vu de l'extérieur. La syntaxe du composant composite est la suivante :

compound type *compound-type-name* [**(** *parameter-list* **)**]
component-instantiation
connector-instantiation
priority-definition
export-definition
end

Pour instancier un composant, tout d'abord il doit être défini soit comme un type d'atome, soit comme un type du composite.

Une instance de connecteur associe les ports des composants instanciés à travers les interactions définies par le type de connecteur. Les ports spécifiés dans l'instanciation du connecteur sont associés aux paramètres du port dans la définition du type de connecteur.

Pour exporter un connecteur comme étant un port, le connecteur lui-même doit avoir un port exporté. Le type du port à exporter du composite est alors le même que le type de port à exporter du connecteur.

Le listing 4.6 fournit un exemple de composant composite nommé *SendRecv*. Il contient deux instanciations du composant atomique *Reactive* et un connecteur de type *Broadcast* qui relie les deux instances. La représentation graphique est montrée dans la figure 4.4.

```
compound type SendRecv
    component Reactive Sender
    component Reactive Receiver
    connector Broadcast C0 (Sender.s, Receiver1.r)
    export intPort p is C0
end
```

Listing 4.6 – Exemple de composant composite

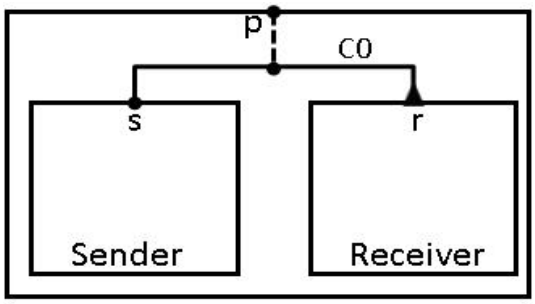


FIG. 4.4 – Représentation graphique du composite *SendRecv*

4.2.5 Packages et Systèmes

La déclaration des packages permet de créer une bibliothèque de composants et de types de connecteurs. Cette bibliothèque est réutilisable dans différentes applications. Le listing 4.7 montre une définition d'un package nommé `my_pack`. Il définit un *typedef* en C, les types de port, les types de connexion et les types de composant.

Un modèle peut inclure des packages, des composants, des connecteurs, des structures de données et des fonctions comme un package. Le modèle

```
package my_pack
   {# typedef char* String; #}
   port type IntPort(int x)
   port type ePort
   connector type SendRec(IntPort s, IntPort r)
        define [ s r ]
        data int val
        on s r up {val = s.x;} down {r.x = val;}
        export IntPort p(val)
   end
   atomic type Producer
        data int i
        export port IntPort comm(i)
        port ePort work(i)
        place idle
        place working
        initial to idle do i=0;
        on work
           from working to idle
           do i=i+1;
        on comm
           from idle to working
   end
end
```

Listing 4.7 – Exemple d'utilisation de package

représente aussi le haut niveau de la conception d'une architecture comme un composant racine.

Nous fournissons la description de l'ensemble d'un système dans le listing 4.8. Le système contient deux composants atomiques Producer et Consumer. Le système est synchronisé par un rendezvous avec le transfert de données (entier) du Producer vers le Consumer. Le système montre aussi l'utilisation du package.

4.2.6 Expressions et instructions

Les expressions en BIP sont essentiellement des expressions C, avec le même ensemble d'opérateurs arithmétiques et logiques que le langage C. Les expressions qui ne sont pas supportées par le langage BIP sont exprimées entre $\{\# \cdots \#\}$. La seule restriction dans ce cas est que les expressions ne sont pas analysées par le compilateur BIP, et elles sont directement intégrées dans le code de l'application générée.

```
model ProdCons
    use my_pack
    atomic type Consumer
        data int j
        export port IntPort comm(j)
        port ePort work
        place idle
        place working
        initial to idle
        on work
            from working to idle
            do printf("j=%d\n", j);
        on comm
            from idle to working
    end
    compound type System
        component my_pack.Producer P
        component Consumer C
        connector my_pack.SendRec cnx(P.comm,C.comm)
    end
    component System S
end
```

Listing 4.8 – Exemple de modèle Producteur Consomateur

Les instructions (*statements*) sont un sous-ensemble des instructions en C. Les instructions qui ne sont pas supportées par le langage BIP, sont exprimées entre $\{\# \cdots \#\}$. Les instructions d'appel de fonctions C sont supportées par le compilateur, mais les paramètres de fonctions sont validés au niveau de la compilation du code de l'application.

4.3 Chaîne d'outils

La chaîne d'outils permet l'exploration de l'espace d'états et fournit un accès aux outils de model checking [QS82] de IF-toolset [BGO+04], Aldebaran [BFKM97], utilisation des observateurs [Tre94], ainsi que l'outil D-Finder [BBSN08]. Ceci permet de valider les modèles BIP et d'assurer qu'il vérifie des propriétés telles que l'absence du blocage, les invariants d'états et l'ordonnançabilité.
La figure 4.5 représente la chaîne d'outils BIP qui contient les éléments suivants :

- un éditeur pour la description textuelle d'un système en BIP ;

- un parseur pour analyser des programmes BIP, et pour générer des modèles conformes au méta-modèle [Meta] BIP, à partir d'une description textuelle ;
- un reconstructeur de code (*deparser*) pour produire, à partir d'un modèle, une description textuelle associée en BIP ;
- un générateur de code pour générer du code C++ exécutable pour le backend de BIP ;
- des outils de transformation au niveau du modèle, pour l'optimisation des modèles ;
- des outils d'analyse structurelle comme D-Finder [BBNS09] ;
- des outils de model checking de la chaîne d'outils IF [BGO+04] ou Aldebaran [BFKM97].

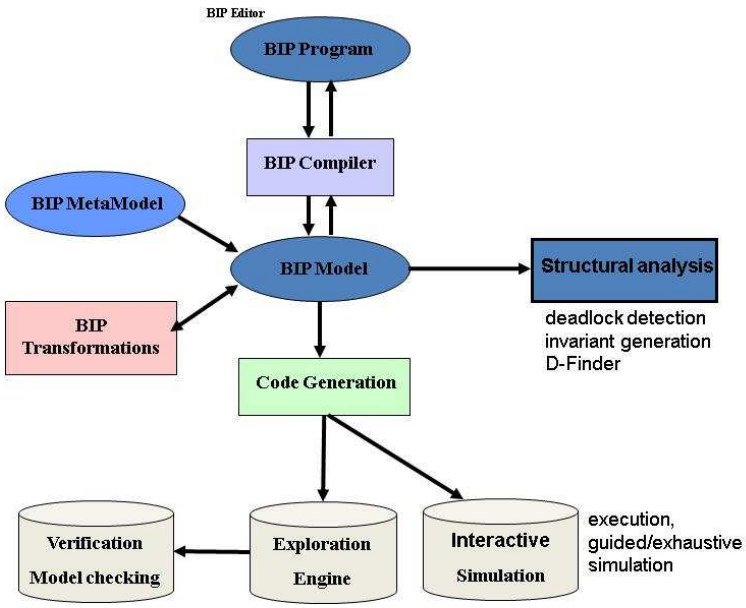


FIG. 4.5 – Chaîne d'outils

Le frontend de la chaîne d'outils est composé d'un éditeur, d'un parseur, d'un reconstructeur de code BIP à partir du modèle et d'un générateur de code. Le backend est composé d'une infrastructure logicielle pour l'exécution et la validation du modèles BIP.

4.3.1 Frontend

Le frontend transforme les programmes BIP en un modèle intermédiaire. Il se compose d'un parseur et d'un générateur de code. Le parseur effectue l'analyse syntaxique du programme et génère une représentation intermédiaire du programme sous la forme d'un modèle qui est conforme au méta-modèle BIP. Le générateur de code prend le modèle généré par le

parseur en entrée et produit un code d'application C++. L'application est un modèle exécutable du programme original BIP. Le code est généré de manière modulaire et préserve la structure du modèle initial, pour chaque composant atomique, connecteur et priorité.

4.3.2 Backend

Le backend de la chaîne d'outils BIP fournit une plate-forme pour l'exécution et l'analyse de l'application C++ générée par le frontend. Le backend comprend un moteur (*engine*) d'exécution et une infrastructure de logiciels associés.

BIP a deux types de plates-formes :

- *single-threaded* : le générateur de code pour la plateforme single-threaded a des options pour générer deux types d'exécution : effectuer l'exploration exhaustive ou lancer une exécution.
- *multi-threaded* : le code généré est instrumenté avec des structures de données supplémentaires et des routines qui permet de stocker les états des composants, et de surveiller l'espace d'état global nécessaire pour la simulation exhaustive.

Le moteur joue le rôle de coordinateur dans la sélection et l'exécution des interactions entre les composants. Il observe l'état des composants avec la prise en compte des interactions. Le moteur cherche les interactions actives et applique ensuite les règles de priorités pour éliminer les interactions avec une faible priorité. Enfin, il choisit une interaction parmi les interactions maximales actives pour l'exécution.

Il existe trois implantations différentes pour le moteur d'exécution de BIP :

- Un moteur énumératif centralisé (*centralized enumerative engine*) [BBS06], qui décide de la sélection et de l'exécution du système d'interactions fondées sur l'ensemble de la connaissance de l'état du système. Il représente les interactions par énumération.
- Un moteur symbolique centralisé (*centralized symbolic engine*) [BBJ09], où la précédente représentation énumérative d'une interaction est remplacée par une représentation symbolique en utilisant des arbres de décision binaires (*Binary Decision Diagrams*, BDDs). Cette implémentation permet une exécution efficace pour certaines classes spécifiques de systèmes.
- Un moteur d'exécution distribuée [BBBS08], qui calcule les interactions possibles basées sur des informations partielles sur l'état du système.

4.4 Conclusion

BIP permet de modéliser et d'analyser des systèmes temps réel à partir de composants hétérogènes. Le langage offre la conception et la description comportementale d'architectures comme un ensemble de transitions, les connecteurs décrivent les interactions entre les transitions et un ensemble de règles de priorité pour décrire les politiques de planification des interactions. BIP possède une grande puissance d'expression et une sémantique formelle qui assure que pour chaque modèle, aucune ambiguïté ou erreur d'interprétation n'est possible. Plusieurs études de cas ont été menées comme MPEG4 encoder [PPRS06], TinyOS [BMP+07], et DALA [BBG+08].

Les outils de vérification BIP permettent l'exploration exhaustive de l'espace des états du système, la détection des blocages potentiels et la vérification de certaines propriétés dans les modèles. Ces derniers éléments jouent en faveur de l'utilisation de telles méthodes dans le cycle de développement. BIP joue un rôle central au sein de plusieurs projets tels que ITEA/Spices [1], OpenEMBeDD [2], TAPIOCA [3].

Pour ces raisons que nous avons choisi d'utiliser BIP qui offre plusieurs avantages considérables par rapport à d'autres langages de modélisation.

[1] http ://www.spices-itea.org
[2] http ://openembedd.org
[3] http ://www-verimag.imag.fr/ yovine/projects/tapioca/tapioca.html

Chapitre 5

Traduction d'AADL vers BIP

Sommaire

Une modélisation en AADL permet de décrire les éléments architecturaux d'une application à base de composants. Dans ce chapitre nous étudions la démarche de transformation d'un modèle AADL vers un modèle formel avec une sémantique opérationnelle précise. Les problématiques que nous abordons ici sont l'interprétation des éléments AADL, ainsi que l'intégration de descriptions comportementales au sein de la description. Nous étudions les différents cas à considérer dans le cadre de la description des composants en AADL.

5.1 Cycle de développement d'une application formellement vérifiées

Le développement d'un système embarqué temps-réel nécessite la vérification régulière du respect des spécifications. Une approche de développement itérative permet une rétroaction entre l'application et ses spécifications initiales. Cette approche consiste en une démarche de conception pas à pas permettant la validation 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.

Les modèles AADL sont exploités de différentes manières, comme illustré sur la figure 5.1 :

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

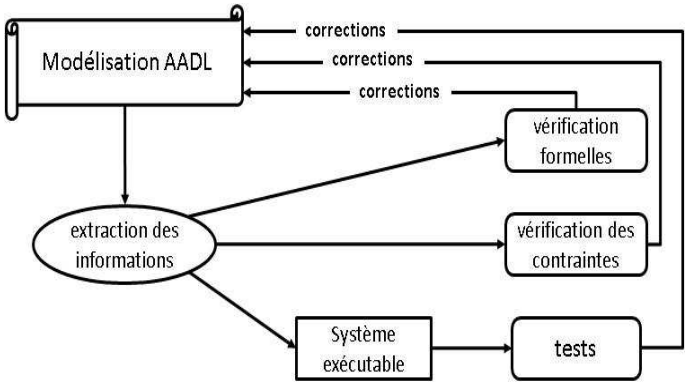


FIG. 5.1 – Cycle de développement

Dans une première phase nous nous basons sur une description AADL

de l'application que nous voulons construire. Ensuite, la description de l'application est complétée par l'addition du comportement. Dans une seconde phase, nous appliquons à la description AADL la technique de la transformation de modèle. La description peut alors être analysée afin d'en évaluer les dimensions exactes, s'assurer de sa fiabilité et de son adéquation avec les besoins.

5.2 Utilisation d'AADL pour le cycle de développement

Nous avons vu qu'AADL se focalise essentiellement sur les descriptions architecturales, et permet de modéliser les architectures avec une représentation centrale. 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. En contrepartie, cette liberté entraîne une dispersion des éléments de description qui limite les moyens de vérification et d'analyse de l'architecture.

Notre objectif final est de décrire tous les éléments architecturaux d'une application avec AADL pour la transformer automatiquement vers une application BIP, qui a une sémantique opérationnelle formellement définie en termes de systèmes de transitions étiquetées. La production d'un système exécutable à partir de sa description architecturale nécessite la prise en compte de plusieurs paramètres qui fournirons les fonctionnalités nécessaires à la bonne exécution du code généré à partir des composants AADL.

5.3 Identification des éléments d'AADL

Dans le chapitre 3, nous avons établi qu'AADL définit un certain nombre de composants pour formaliser la partie logiciel d'un système. La figure 5.2 montre la génération d'une application à partir d'une description AADL, qui consiste à traduire les constructions AADL en ajoutant les descriptions comportementales que nous avons identifiées au chapitre 3. Nous nous intéressons aussi à la traduction vers un langages de programmation impératif (C/C++).

Nous étudions dans ce chapitre une méthodologie générale pour traduire automatiquement AADL vers BIP. Cette traduction permet l'analyse formelle des systèmes AADL et l'application à ces systèmes des techniques de vérification formelle développées pour BIP comme la détection du blocage.

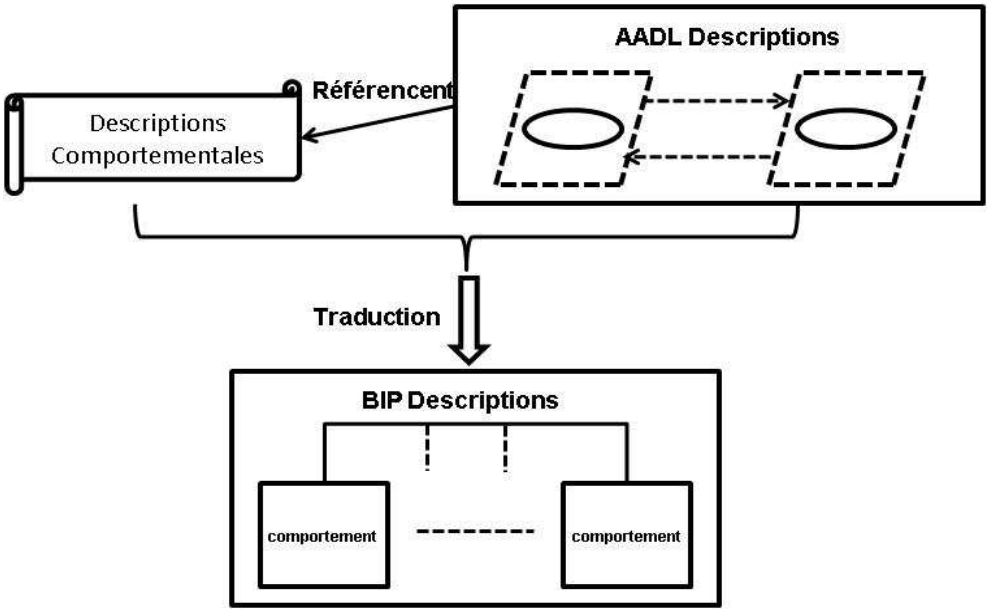


FIG. 5.2 – Génération de l'application BIP

5.4 Syntaxe abstraite

Avant de commencer la traduction des éléments du langage AADL vers le langage BIP, Nous rappelons la syntaxe abstraite d'AADL décrite dans le listing 5.1, pour faciliter la compréhension de la hiérarchie des composants.

```
system_component       := system_component*
                          data_component*
                          process_component*
                          hardware_component*
process_component      := thread_component*
                          data_component*
thread_component       := data_component*
                          subprogram_component*
subprogram_component   := data_component*
                          subprogram_component*
data_component         := data_component*
hardware_component     := processor_component*
                          device_component*
                          memory_component*
                          bus_component*
```

Listing 5.1 – Syntaxe Abstraite d'AADL

La syntaxe abstraite d'AADL permet de montrer une suite de déclarations afin d'obtenir une description architecturale. Pour cela, il faut définir un composant initial qui servira de racine à l'arborescence. Ce composant doit être un système, il représente l'ensemble de l'architecture.

5.5 Traduction des composants logiciels

Dans cette section, nous allons présenter la traduction des composants logiciels vers BIP : 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.

5.5.1 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.

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 dans le listing 5.2 ne donne aucune indication sur la nature de la donnée déclarée.

```
data une_donnee
end une_donnee;
```

Listing 5.2 – 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.

La sémantique d'un composant de donnée est indiquée par les propriétés AADL : *Source_Data_Size : Size* ; *Type_Source_Name : aadlstring* ; *Source_Name : aadlstring* et *Source_Text :* **inherit list of** *aadlstring*.

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 (entiers, flottants, caractères, chaînes de caractères, etc).

BIP permet de manipuler les données C/C++. Par conséquent, BIP pourrait représenter toutes les données AADL correspondant aux source texte C/C++. La traduction des composants de donnée suit donc les mêmes principes.

```
data (int|bool|float|string) une_donnee
```

Listing 5.3 – Déclaration d'une donnée BIP

5.5.1.1 Structure de Données

Les structures de données constituées de plusieurs champs (aussi appelées enregistrements dans certains langages) constituent des données complexes. Leur organisation peut être représentée par une construction AADL à l'aide de sous-composants, comme illustré sur le listing 5.4. 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.

```
data implementation une_donnee.structure
  subcomponents
      i : data une_donnee.entier;
      c : data une_donnee.chaine;
end une_donnee.structure;
```

Listing 5.4 – Déclaration d'une structure de données en AADL

La traduction d'une structure de données du listing 5.4 vers BIP est exprimée en C/C++ comme le montre le listing 5.5.

```
struct une_donnee.structure {
   une_donnee.entier i;
   une_donnee.chaine c;
};
```

Listing 5.5 – Déclaration d'une structure de données en BIP

5.5.2 Traduction des sous-programmes

Les sous-programmes AADL constituent les unités structurelles abritant les descriptions comportementales. Un sous-programme AADL possède des paramètres en entrée, sortie, ou entrée/sortie.

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 flux 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 BIP nécessite donc la prise en compte de trois paramètres principaux :

- présence de séquences d'appel dans l'implantation du sous-programme ;
- présence de comportements spécifiés par l'annexe dans l'implantation du sous-programme ;
- référence à un fichier (C/C++) contenant la description comportementale du sous-programme.

Nous avons étudié les différentes combinaisons possibles de ces trois paramètres afin de définir des règles générales concernant d'une part le code BIP à générer à partir des sous-programmes 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 comportementale ;
- implantation opaque ;
- séquence d'appel ;

5.5.2.1 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.

La traduction consiste à produire un composant atomique en BIP comme le montre la figure 5.3. Ce composant a deux ports *call* et *return* pour exprimer l'appel du sous-programme à partir d'un autre sous-programme ou thread, ces deux ports permettent le transfert de données (paramètres). Il a aussi deux états *IDLE* et *RETURN*, et deux transitions.

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.

5.5.2.2 Implantation comportementale

Dans cette situation, l'implantation du sous-programme est décrite en utilisant le comportement spécifié par une annexe comportementale. L'utilisation des annexes permet donc d'étendre la syntaxe du sous-programmes

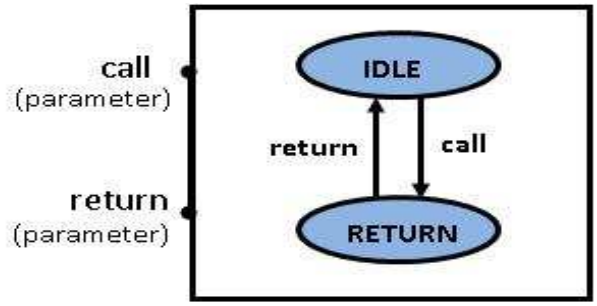


FIG. 5.3 – Sous-programme sans implantation

afin de spécifier le comportement en utilisant les machines d'états.

La traduction du sous-programme vers BIP est décrite dans la figure 5.4.

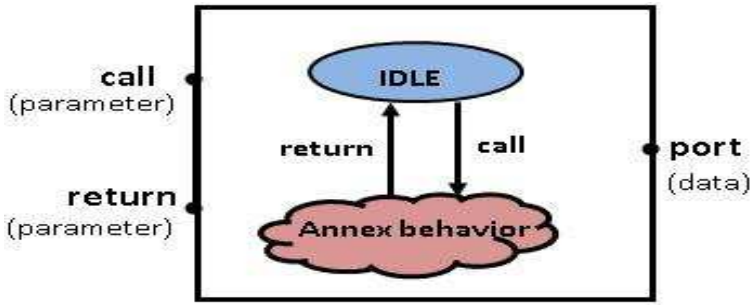


FIG. 5.4 – Implantation comportementale

Lors de l'exécution, le sous-programme est appelé à partir d'un thread ou à partir d'un autre sous-programme à travers le port *call* avec le passage des paramètres. L'interaction à travers ce port permet au sous-programme de passer de l'état *IDLE* à l'état initiale du comportement spécifier par l'annexe. Lorsque l'exécution du comportement arrive à sa fin, le sous-programme effectue une transition à travers le port *return* pour transmettre les nouveaux paramètres et pour rentrer à l'état *IDLE* pour attendre un nouveau appel. La déclaration du *port* dans le composant atomique décrit dans la figure 5.4 fait référence à l'annexe comportementale, pour échanger des donnée avec le sous-programme.

5.5.2.3 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 : *Source_Name*, *Source_Text* et *Source_Language* tels qu'elles sont définies dans le standard AADL.

La traduction vers BIP doit faire le lien avec le code source que l'u-

tilisateur 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 données AADL. La figure 5.5 correspond à la traduction d'un sous-programme opaque en BIP.

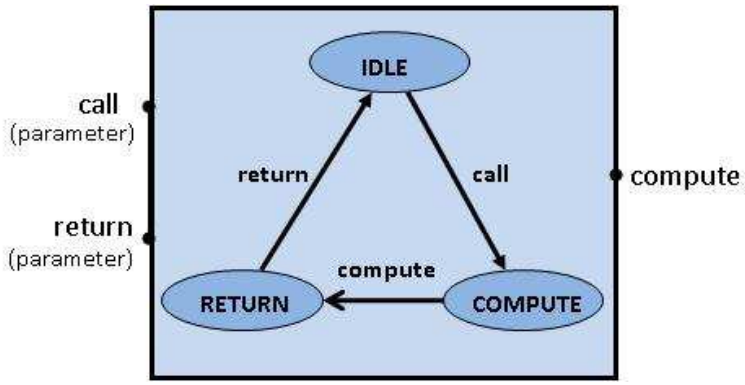


FIG. 5.5 – Sous-programme opaque en BIP

Le listing 5.6 donne un exemple de sous-programme AADL, appelé *Sunseekerplant* qui prend en entrée *Controllerinput* de type flottant, et produit en sortie *Controllerinput* de type flottant. L'implantation du sous-programme *Sunseekerplant* est décrite dans un fichier externe de type *C*. Il fait appel à une fonction *user_sunseekerplant* dans le fichier *sunseekerplant.c*.

```
subprogram Sunseekerplant
  features
    Controllerinput : in parameter Behavior::float;
    Outputfeedback  : out parameter Behavior::float;
end Sunseekerplant;

subprogram implementation Sunseekerplant.Beacon
  properties
    Source_Language          => C;
    Source_Name              => "user_sunseekerplant";
    source_text  => ("sunseekerplant.c");
end Sunseekerplant.Beacon;
```

Listing 5.6 – Exemple de sous-programme AADL avec implantation opaque

Le listing 5.7 correspond à la traduction du sous-programme AADL vers BIP. Le lien vers le ficher externe est décrit par un appel de fonction en faisant correspondre les paramètres du sous-programme lors d'une transition

```
atomic type Sunseekerplant_Beacon
    data float Controllerinput
    data float Outputfeedback
    export port floatPort call(Controllerinput)
    export port floatPort return(Outputfeedback)
    port aadllib.Port compute()
    place IDLE
    place RETURN
    place COMPUTE
    initial to IDLE
    on call
      from IDLE to COMPUTE
    on compute
      from COMPUTE to RETURN
      do
        {#
            user_sunseekerplant(Controllerinput,
                                &Outputfeedback);
        #}
    on return
      from RETURN to IDLE
end
```

Listing 5.7 – Traduction du sous-programme AADL vers BIP

de l'état *COMPUTE* vers l'état *RETURN* à travers une interaction sur le port *compute*.

5.5.2.4 Séquences d'appel

AADL permet de modéliser le comportement d'un sous-programme comme étant une séquence d'appel à d'autres sous-programmes AADL.

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 un code BIP comme le montre la figure 5.6.

L'exécution d'une séquence d'appel de sous-programmes est séquentielle, elle est représentée en BIP sous forme d'un composant atomique appelé *call_sequence*. Ce composant a pour but d'assurer l'exécution séquentielle des sous-programmes, il contient des états $wait\_call_1$ ... $wait\_call_n$ et $wait\_return_1$ ... $wait\_return_n$, des transitions étiquetées par les ports $call_1$ ... $call_n$ et $return_1$ ... $return_n$ (où n est le nombre de sous-programmes appelés sub_1 ... sub_n).

Pour faire respecter le bon ordre d'exécution et le transfert de paramètres, deux ports *call* et *return* sont utilisés pour exprimer des appels du sous-programme par d'autres sous-programmes ou threads.

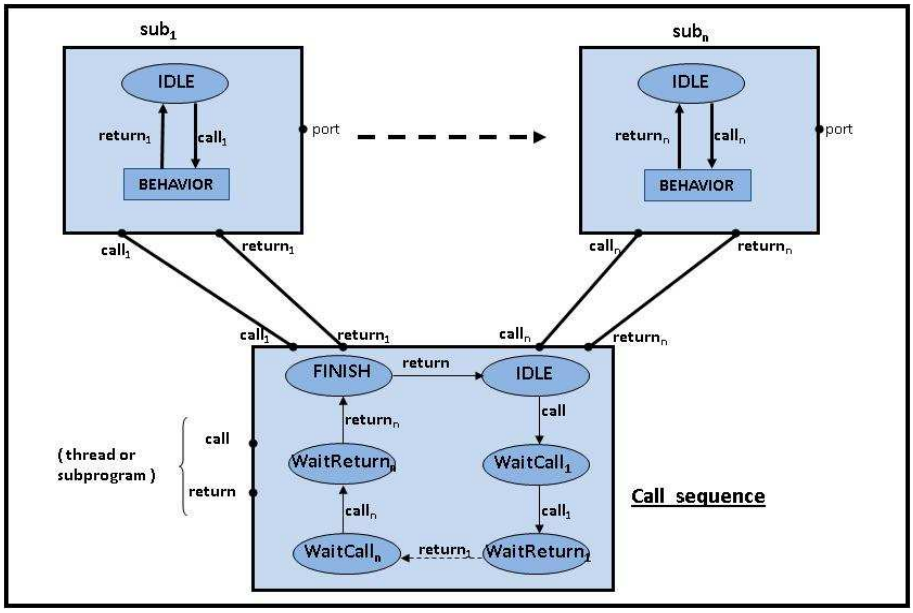


FIG. 5.6 – Traduction du séquences d'appel vers BIP

La modélisation des séquences d'appel en utilisant les sous-programmes est illustrée sur le listing 5.8

5.5.3 Traduction des Threads

Le standard AADL définit quatre protocoles 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 *Period*. Un thread apériodique est déclenché par les données arrivant sur un port de donnée/événement déclencheur. 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.

Un thread AADL est traduit en BIP sous forme d'un composant atomique comme le montre la figure 5.7. L'état initial du thread est *HALTED*. Le thread est initialisé à travers une interaction sur le port *load*. Une fois l'initialisation terminée le thread passe à l'état *READY*, si le thread est prêt

```
subprogram spA
  features
    e : in parameter Behavior::integer;
    s : out parameter Behavior::integer;
end spA;

subprogram spB
  features
    e : in parameter Behavior::integer;
    s : out parameter Behavior::integer;
end spB;

subprogram spC
  features
    e : in parameter Behavior::integer;
    s : out parameter Behavior::integer;
end spC;

subprogram implementation spA.impl
  calls
   {
     appel1 : subprogram spB;
     appel2 : subprogram spC;
   };
  connections
     cnx1 : parameter appel1.s -> appel2.e;
     cnx2 : parameter appel2.s -> s;
end spA.impl;
```

Listing 5.8 – Exemple de séquence d'appel

pour une interaction à travers le port *exec_req*. Sinon, il passe à l'état *SUSPENDED*. Lorsque le thread est dans l'état *SUSPENDED*, il ne peut pas être dispatché pour l'exécution.

Lorsque le thread est dans l'état *SUSPENDED*, il est en train d'attendre un événement et/ou une période pour être activé en fonction du protocole de déclenchement du thread (périodique, apériodique, sporadique). Dans l'état *READY*, un thread est en attente d'être déclenché à travers une interaction dans le port *get_exec*. Lorsque le thread est déclenché, il entre dans l'état *COMPUTE* pour exécuter le comportement spécifique associé. En cas de réussite d'achèvement du calcul, le thread passe à l'état *OUTPUTS*. S'il y a des ports à expédier, le thread retourne à l'état *OUTPUTS*. Sinon, il passe à l'état *FINISH*.

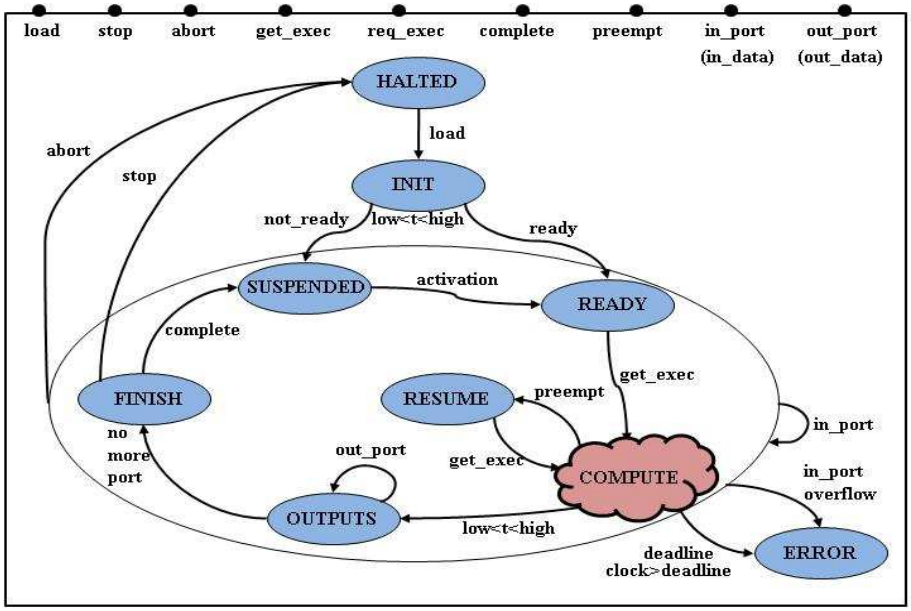


FIG. 5.7 – Traduction du thread vers BIP

Pour que la traduction d'un thread AADL vers BIP soit complète, nous devons prendre en compte trois paramètres principaux de l'implantation du thread :

- présence de séquences d'appel dans l'implantation du thread ;
- présence de comportement spécifié par l'annexe dans l'implantation du thread ;
- référence à un fichier (C/C++) contenant la description comportementale du thread.

Nous avons étudié les différentes combinaisons possibles de ces trois paramètres afin de définir des règles générales concernant d'une part le code BIP 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 threads AADL différents, selon leur implantation :

- pas d'implantation ;
- implantation opaque ;
- implantation comportementale ;
- séquence d'appel ;

Un thread sans implantation ne fournit aucune information quand à sa structure interne. Il s'agit d'un type ne possédant aucune implantation, ou possédant une implantation vide. La traduction du thread en BIP est montré dans la figure 5.7.

5.5.3.1 Implantations opaques

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

Cette traduction est similaire à ce que nous avons vu pour les sous-programmes, comme le montre la figure 5.8. Les propriétés exprimées en AADL sont traduites sous forme d'un appel de fonction (*Function_call*) en BIP.

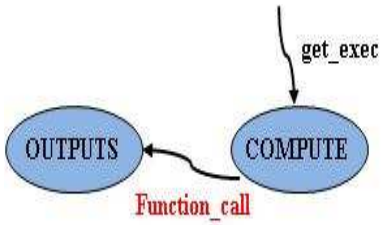


FIG. 5.8 – Implantations opaques du thread en BIP

5.5.3.2 Implantations comportementale

L'implantation comportementale du thread est décrite en utilisant le comportement spécifié par l'annexe. La traduction du thread AADL vers BIP en utilisant l'annexe comportementale est décrite dans la figure 5.9. Comme le montre la figure, le comportement est toujours relié à l'état *COMPUTE*, pour permettre à cette dernière de faire une interaction sur le port *preempt* pour accéder à l'état *RESUME*.

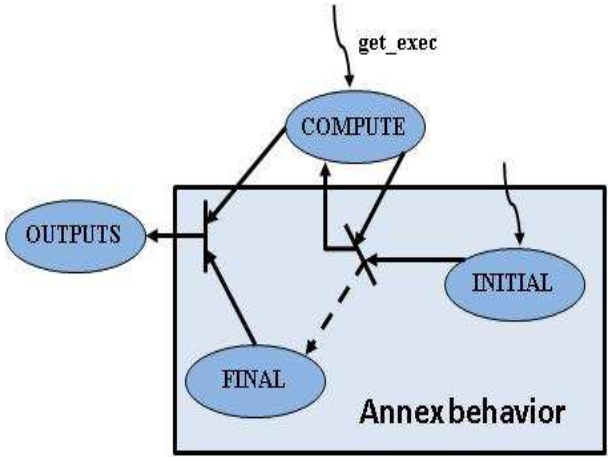


FIG. 5.9 – Implantations comportementale du thread en BIP

5.5.3.3 Séquences d'appel

En l'absence de description comportementale, nous considérons que le comportement du thread consiste à exécuter la séquence d'appel du sous-programme. Un appel au sous-programme dans le thread AADL est modélisé en BIP comme le montre la figure 5.10. Cette figure montre que le thread appelle un sous-programme à travers le port *call* qui exprime l'appel et l'envoi des paramètres au sous-programme et le port *return* exprime la fin de l'exécution et le retour des paramètres du sous-programme.

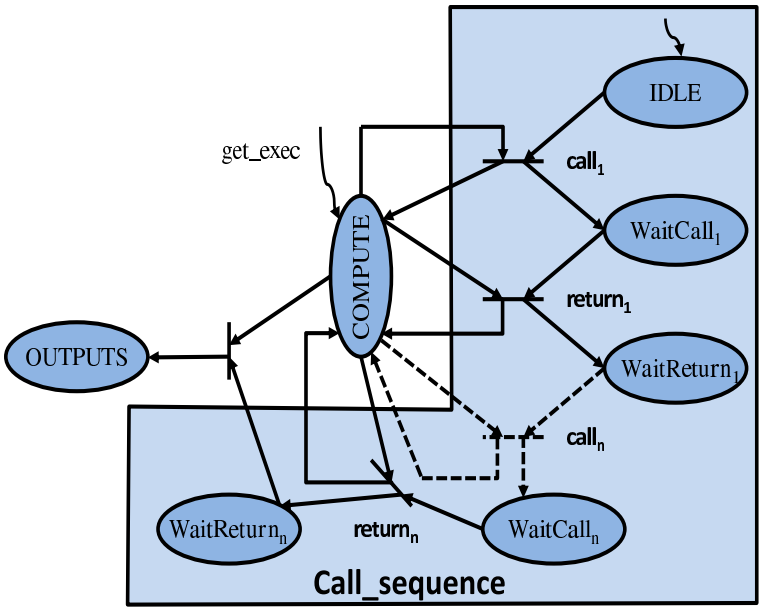


FIG. 5.10 – Séquences d'appel du thread en BIP

5.5.4 Traduction des processus

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 et les données. Afin de structurer la génération de code, nous conservons l'organisation des entités exprimée dans la description AADL.

Les processus définissent des espaces de mémoire dans lesquels les threads s'exécutent. Les processus AADL peuvent être vus comme des « conteneurs de *threads* ». C'est-à-dire, ce composant organise hiérarchiquement l'architecture logicielle des threads modélisés avec AADL. La façon la plus naturelle de traduire les processus vers BIP en conservant la sémantique du processus décrite dans le standard AADL est présentée dans la figure 5.11.

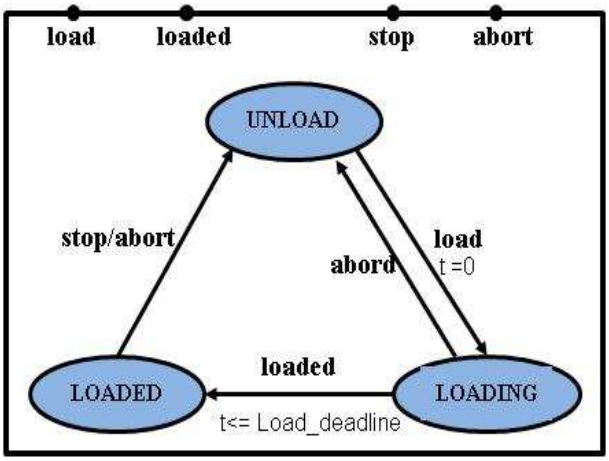


FIG. 5.11 – Modélisation du processus en BIP

L'état initial du processus est *UNLOAD*. Le processus passe à l'état *LOADING* via le port *load* et il est prêt à être chargé.

Un processus est considéré comme arrêté au moment où tous les threads du processus sont interrompus. Lorsqu'un processus est arrêté à travers une interaction sur le port *stop*, chacun de ses threads ont une chance de terminer leur exécution.

Un processus peut être avorté en utilisant le port *abort*. Dans ce cas, tous les threads contenus terminent leur exécution immédiatement et libèrent toutes les ressources.

Les connexions BIP entre le processus et les threads sont décrites dans la figure 5.12.

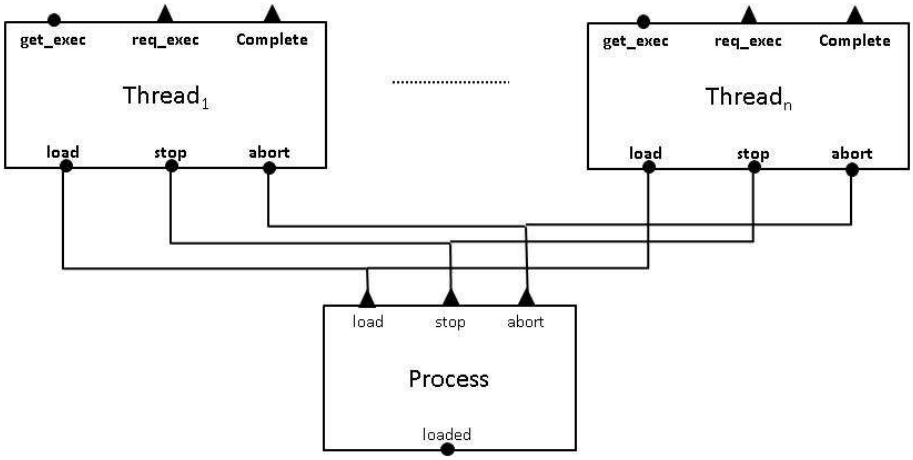


FIG. 5.12 – Représentation en BIP des connexions entre le processus et les threads

5.6 Traduction des composants matériels

Les composants matériels sont représentés par des composants BIP abstraits. Nous ne décrivons pas en détails leurs comportements, mais à un certain niveau d'abstraction, l'influence qu'ils peuvent avoir sur la partie logicielle du système (e.g. temps d'exécution, ordonnancement, etc)

5.6.1 Traduction des processeurs

Le processeur AADL représente un ensemble constitué d'un microprocesseur combiné à un ordonnanceur. Il est responsable de l'ordonnancement (scheduling) et l'exécution de threads.

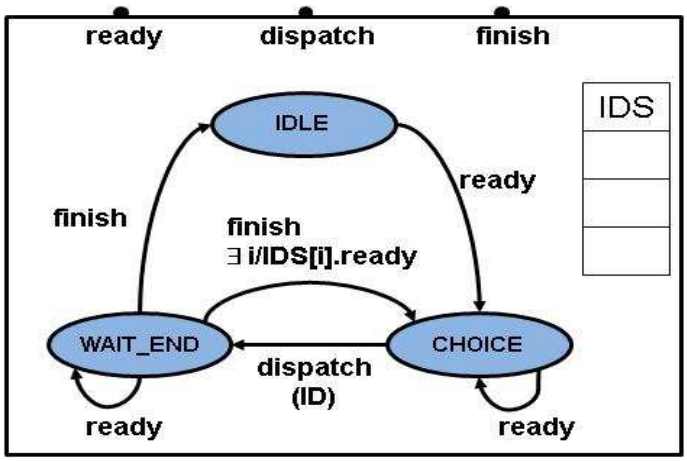


FIG. 5.13 – Ordonnanceur (Scheduler)

L'ordonnanceur est modélisé en BIP sous forme d'un composant atomique comme le montre la Figure 5.13. L'état initial de l'ordonnanceur est *IDLE*. L'ordonnanceur interagit avec l'ensemble des threads du processeur. Il reçoit les demandes d'exécution. Dans ce cas, l'ordonnanceur sélectionne le thread et passe à l'état *CHOICE* à travers une interaction sur le port *ready*. Dans cet état, l'identificateur (ID) du thread est stocké dans la mémoire (IDS) de l'ordonnanceur. Lorsque le thread est prêt à être exécuté, l'ordonnanceur sélectionne son identificateur et passe à l'état *WAIT_END* à travers une interaction sur le port *dispatch*. S'il y a plusieurs threads à expédier, l'ordonnanceur rentre dans l'état *CHOICE* pour choisir un autre thread, sinon, il passe à l'état *IDLE*.

Les connexions BIP entre l'ordonnanceur et les threads sont décrites dans la figure 5.14.

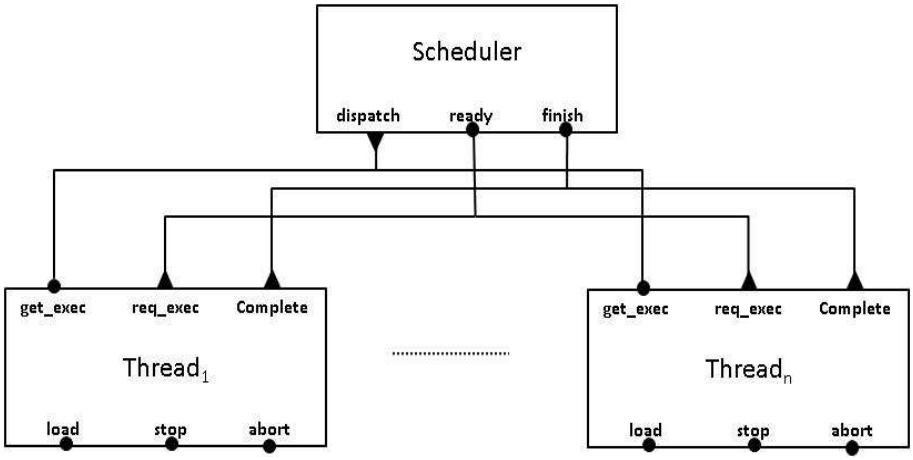


FIG. 5.14 – Représentation en BIP des connexions entre l'ordonnanceur et les threads

5.6.2 Traduction des bus

Les connexions entre les entités logicielles d'un système peuvent être associées à un bus donné. Dans ce cas, le flot de données passe par le bus en question. Le bus est traduit en BIP en utilisant un composant atomique avec un port d'entrée et un port de sortie de données. La figure 5.16 montre l'utilisation d'un bus en AADL et la figure 5.15 montre la traduction vers BIP. Les données qui arrivent sur le port *in* du composant *BUS* sont mémorisées en file d'attente. Le composant *Bus* peut avoir des propriétés qui permettent de spécifier le temps entre le début et la fin de la transmission des données et la taille de la file d'attente.

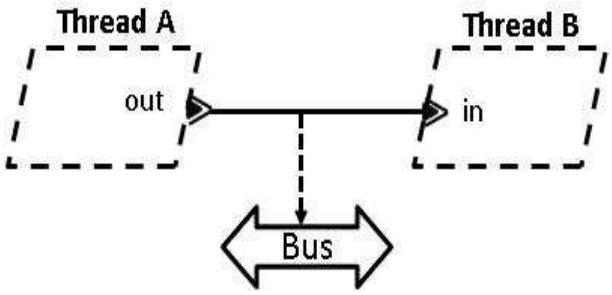


FIG. 5.15 – Représentation du bus en AADL

5.6.3 Traduction des devices

Les devices peuvent être vus comme des boites noires et leur structure interne n'est pas décrite en AADL. Seule l'interface du device est visible par

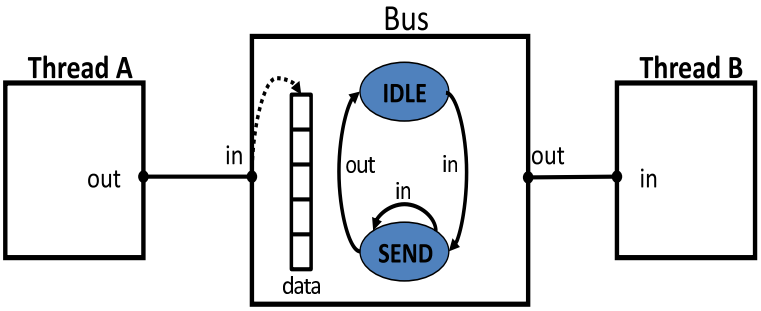


FIG. 5.16 – Modélisation du bus en BIP

les autres composants du système. Cette interprétation du device ne permet pas d'avoir un système exécutable à cause d'absence du comportement. La traduction du device vers BIP est représentée par un composant atomique comme le montre la figure 5.17. Ce composant permet d'envoyer et de recevoir des données à travers les ports *in* et *out*.

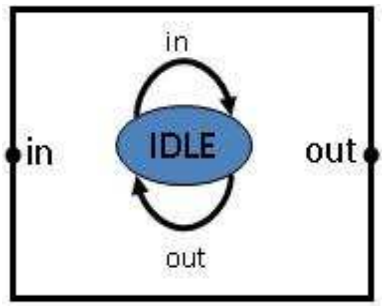


FIG. 5.17 – Modélisation des device en BIP

5.7 Traduction des composants systèmes

Un système AADL est modélisé comme un composant composite en BIP. La figure 5.18 montre un composant BIP qui représente un système constitué d'un processus, un ordonnanceur, des threads, des sous-programmes et des connexions entre les threads et les sous-programmes. Dans cette figure, nous avons représenté un processus et un ordonnanceur, mais notre système peut supporter plusieurs processus et ordonnanceurs.

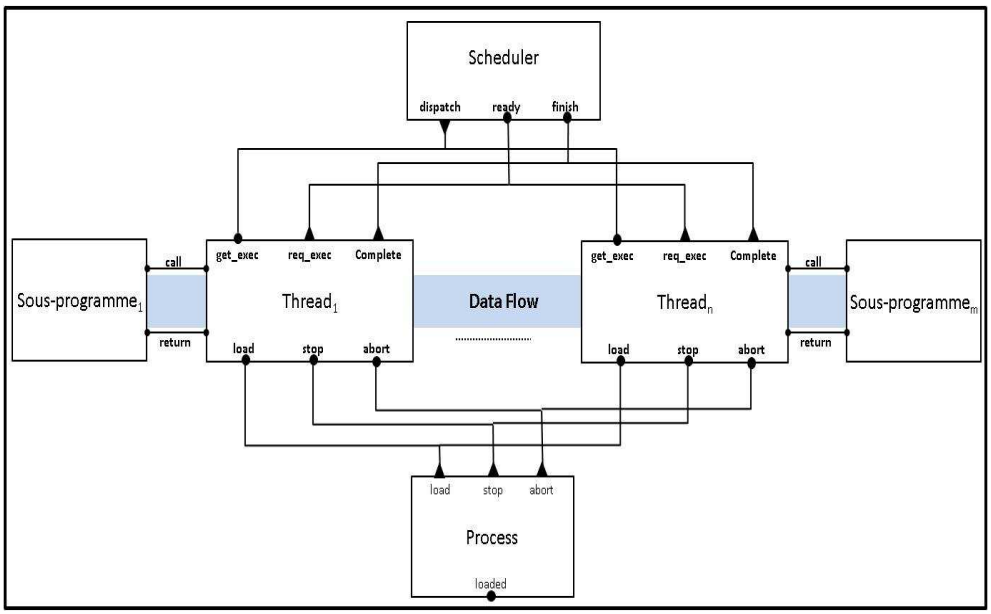


FIG. 5.18 – Représentation graphique d'un système en BIP

5.8 Traduction des ports et des paramètres

5.8.1 Traduction des ports

Les ports d'AADL représentent une interface de communication des composants concernés pour échanger des données, des événements ou des événements associés à des données entre composants.

La traduction d'un port AADL vers BIP prend en compte les trois types de port :

- Un port d'événement permette de modéliser la transmission d'un signal simple entre deux composants AADL, il est traduit en BIP sous forme d'un port simple sans aucune donnée. Ce type de port permet le déclenchement du thread.
- Un port de donnée ne comporte pas de file d'attente, il est traduit en BIP sous forme d'un port attaché à une donnée. Ce type de port ne permet pas le déclenchement du thread.
- Un port d'événement et de donnée est traduit en BIP sous forme d'un port attaché à une donnée. Ce type de port permet le déclenchement du thread et permet aussi la transmission des données. Lors de la traduction nous ajoutons un file d'attente pour qu'une donnée soit consommée pour chaque période du thread récepteur.

Le listing 5.9 montre un exemple d'un port de donné en AADL et sa traduction dans le listing 5.10 en BIP.

```
inp : in data port une_donnee ;
```

Listing 5.9 – Déclaration du port de donnée en AADL

```
port type inp(une_donnee)
```

Listing 5.10 – Traduction du port de donnée en BIP

5.8.2 Traduction des paramètres

Les paramètres d'AADL sont représentés en BIP sous forme de variables. le listing 5.11 représente la traduction d'un paramètre AADL vers BIP.

```
        AADL                          —>        BIP
A: in parameter integer ; —>     data int A
```

Listing 5.11 – Traduction des paramètres

5.9 Traduction des connexions

Une connexion permet de relier deux ports, soit les ports de deux sous-composants, soit le port d'un sous-composant avec le port du composant le contenant. Les connexions AADL se traduisent de différentes façon, selon leur nature.

5.9.1 Les connexions de port

Les connexions AADL se traduisent par des connecteurs et/ou des composants particuliers, de la manière suivante.

5.9.1.1 Connexions d'événement

Les connexions d'événement AADL sont traduit en BIP sous forme de connecteur. Cette traduction prend en compte la transmission d'événement et aussi le déclenchement de l'exécution des threads à travers le port qui reçoit l'événement.

5.9.1.2 Connections d'événement et de donnée

Les connexions d'événement et de donnée sont traduites en BIP sous forme de connecteurs. Cette traduction prend en compte la transmission d'événement, la transmission de donnée, et aussi le déclenchement de l'exécution des threads à travers le port qui reçoit l'événement.

5.9.1.3 Connexions de donnée

Les connexions de donnée sont traduites en BIP selon leur nature : une connexion immédiate (->) ou retardée (-»). Le sens intuitif est le suivant :

Connexion Immédiate Les connexions immédiates dans le standard AADL suivent une règle très précise, dans le cas où la période du thread expéditeur est égale à la période du thread récepteur. Cette règle permet de définir un cadre de communications déterministe : l'exécution du thread récepteur sera retardée après l'exécution du thread expéditeur. Par contre dans le cas contraire, c'est à dire, la période du thread expéditeur égale à $r\times$ la période du thread récepteur (tel que r est un réel) nous trouverons aucune sémantique précise.

Dans le cas où la période du thread expéditeur égale la période du thread récepteur, une connexion immédiate est traduite sous forme de connecteur BIP, avec la prise en compte du transfert de données. Dans cette traduction, le thread expéditeur s'exécute avant le thread récepteur.

Dans le cas contraire, nous fournirons plus de détails dans le chapitre suivant.

Connexion Retardée Dans le cas où la période du thread expéditeur est égale à la période du thread récepteur. Une connexion retardée est traduite en un composant atomique qui prend en entrée une donnée du thread expéditeur et attend la fin de sa période pour envoyer cette donnée au thread récepteur. Ce composant atomique assure la connexion retardée entre les deux composants. Dans le cas où les deux périodes ne sont pas égales, nous fournirons plus de détails dans le chapitre suivant.

Le listing 5.12 présente un exemple de connexion de port de donnée en AADL. Il permet de transmettre les données à partir du port *outp* vers le port *inp*.

```
connections
    cnx : data port Sender.outp -> Receiver.inp;
```

Listing 5.12 – connexion de port de donnée

La traduction de l'exemple précédent en BIP est représentée dans le listing 5.13.

```
connector
        typePortCnx cnx(Sender.outp, Receiver.inp)

connector type typePortCnx(outp, inp)
    define  inp outp
    on inp outp
    down inp.i = outp.i;
```

```
end
```

Listing 5.13 – Traduction d'une connexion de port de donnée

5.9.1.4 Les connexions hiérarchiques

Le standard AADL permet de définir un connecteur hiérarchique, comme décrit dans la figure 3.6 du chapitre 3. Ce type de connecteur relie le port d'un sous-composant avec le port du composant le contenant. Nous traduisons ce type de connexion en BIP sous forme d'un port exporter d'un sous composant vers le composant le contenant.

5.9.1.5 Cas particulier des connexions

AADL permet de modéliser un connecteur qui a un port d'entrée et un port de sortie du même composant, comme décrit dans la figure 5.19. L'intérêt de ce connecteur est de contrôler la réactivation du thread par l'envoi d'un événement.

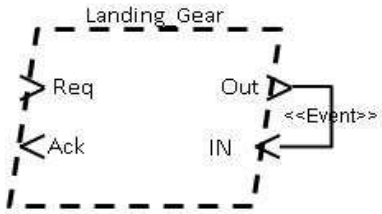


Fig. 5.19 – Connecteur AADL

Le langage BIP ne supporte pas ce type de connecteur, pour cela la traduction nécessite la prise en compte de ce paramètre avec la préservation de la sémantique de ce dernier. Nous avons proposé une modélisation du connecteur qui permet la réactivation du thread comme le montre la figure 5.20. La synchronisation entre les deux composants (*langing_Gear* et *Reactivation*) permet d'assurer la réactivation périodique du thread *langing_Gear*.

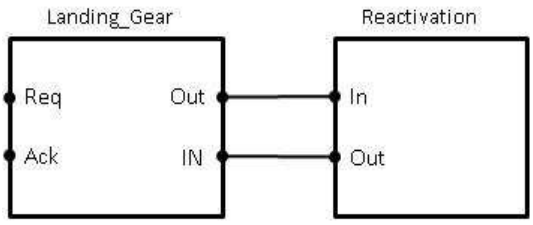


Fig. 5.20 – Modélisation de la réactivation du thread

5.10 Traduction de l'annexes comportementale

Une annexe comportementale est définie pour donner la possibilité d'indiquer, au niveau d'AADL, le comportement des threads et des sous-programmes sous forme de machines à états. Ce comportement est pris en compte dans la traduction des modèles en BIP.

Certains éléments de l'annexe comportementale peuvent être directement traduits en BIP alors que pour d'autres la traduction est complexe.

La traduction de l'annexe comportementale vers BIP se décompose en quatre étapes :

1. Les variables d'état (*state variables*) correspondent à la déclaration des variables de donnés en BIP.
2. La section d'initialisation correspond à l'initialisation des variables en BIP.
3. La section de déclaration des états correspond à la section de déclaration des états (*places*) en BIP.
4. La traduction des transitions de l'annexe comportementale vers BIP est plus complexe. Elle nécessite de prendre en compte plusieurs paramètres. Pour cela nous avons décidé de traiter les gardes et les actions séparément.

Le listing 5.14 montre un exemple d'utilisation des trois premières sections. La traduction vers BIP est montré dans le listing 5.15.

```
annex behavior_specification {**
   state variables
      var_1 : integer;
   initial var_1 = 0 ;
   states
      state_1 : initial state;
      state_2 : state;
**};
```

Listing 5.14 – Exemple d'utilisation

```
data int var_1
place state_1 , state_2
initial to state_1 do var_1 = 0;
```

Listing 5.15 – Traduction en BIP

5.10.1 Traduction des gardes

Nous avons étudié les différentes combinaisons possibles des gardes de l'annexe comportementale à fin de définir des règles générales du code BIP à générer à partir des descriptions comportementales.

La garde peut avoir trois formes :
– la garde est toujours vraie ;
– la garde sous forme d'une *expression* ;
– la garde sous forme d'une *expression* et d'un port (*event*).
La traduction de la garde vers BIP est décrite dans le listing 5.16 :

```
  on <event> [provided <expression>]
| on internal_port provided <expression>
| on internal_port
```

Listing 5.16 – Traduction des gardes en BIP

En BIP, une transition doit avoir un port pour effectuer une interaction. Pour cela, nous avons ajouté un port interne appeler *internal_port* pour que la transition peut avoir lieu.

5.10.2 Traduction des actions

Nous distinguons cinq cas d'utilisation :
– l'action est une affectation (*assignment*) ;
– l'action est une construction **if then else** ;
– l'action est une *computation* ;
– l'action est un *delay* ;
– l'action est une *communication.*
Dans le cas où l'action contient des affectations. La traduction vers BIP correspond à la déclaration des instructions dans la partie d'action.

Dans le cas où l'action prend la forme d'une construction **if***(condition)* **then** *{action1}* **else** *{action2}*. L'état initial de la transition est S_i et l'état final est S_j. La traduction vers BIP est décrite dans la figure 5.21. Une transition doit avoir un port pour effectuer une interaction. Dans le cas où le port est spécifié dans la transition AADL, nous le mentionnons dans la transition BIP, sinon, nous ajoutons un port interne.

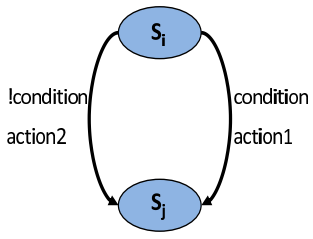


FIG. 5.21 – Traduction de la construction **if then else** en BIP

Dans le cas de la *computation(min,max)*, nous ajoutons un état intermédiaire *state_int* et une transition pour incrémenter le temps. La traduction est décrit dans le listing 5.17.

Dans le cas de *delay*, nous avons appliqué la même methode de traduction en ajoutant un état intermédiaire. Nous retardons l'interaction avec le délais spécifié entre le minimum et le maximum (*delay(min, max)*).

```
on begin_com
    from state_initial to state_int
    do { t=0; }
on tick
    from state_int to state_int
    do {t = t +1 ;}
on begin_com
    provided (t>min & t<max)
    from state_int to state_final
    do {expression; }
```

Listing 5.17 – Traduction du computation en BIP

Les actions sous forme de *communication* sont traduites en BIP comme le montre le listing 5.18. Dans le cas où aura plusieurs communications dans une seule action, nous ajoutons pour chaque communication un état intermédiaire.

```
on port_communication
    from state_initial to state_final
```

Listing 5.18 – Traduction d'une communication en BIP

5.11 Conclusion

Le langage AADL permet 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. A partir de cette application AADL, nous avons défini un processus de traduction des constructions AADL vers le langage BIP. Notre approche se base sur une séparation nette des constructions architecturales et du code des algorithmes : les composants sont définis par la description AADL, tandis que le comportement des composants AADL est fournie par l'utilisateur sous forme de code source ou d'annexe comportementale.

La richesse de BIP permet de retranscrire les descriptions AADL. Ceci fournit une grande flexibilité pour la réutilisation et le recomposition des composants, et fournit tous les éléments nécessaires pour décrire les composants.

L'avantage principale de notre méthodologie de traduction d'AADL vers BIP est de fournir une sémantique formelle définie en termes de systèmes de transitions étiquetés avant la génération du code. Contrairement à d'autre méthodologies de traduction d'AADL qui génèrent directement du code C/C++, Java ou ADA.

La traduction d'AADL vers BIP est effectuée pour les deux version de BIP v.1 et v.2. Dans ce chapitre, nous avons représenté la traduction d'AADL vers la v.2 de BIP.

Chapitre 6

Extension de la syntaxe d'AADL et de la traduction vers BIP

Sommaire

La première version du standard AADL s'inspire très largement de son ancêtre MetaH ; l'une des principales conséquences de cela est que la description de certains éléments est assez primitive. Afin de pouvoir pleinement exprimer les éléments d'architecture, il est nécessaire d'enrichir la syntaxe.

Cela facilitera le passage entre AADL et d'autres langages. Dans ce chapitre nous décrivons des extensions du langage AADL que nous proposons, ainsi que leurs traduction vers BIP.

6.1 Modélisation des appels de sous-programmes

Afin de pouvoir pleinement exprimer les éléments d'architecture logicielle, il est nécessaire d'enrichir la syntaxe d'AADL, principalement les sous-programmes.

6.1.1 Limitation du standard

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

6.1.2 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.

Les processus peuvent intégrer des sous-programmes comme sous-composants. Les processus étant des espaces de mémoire, la signification d'une telle composition est relativement simple. Nous introduisons une construction syntaxique supplémentaire pour les sous-programmes. De cette façon, nous différencions l'appel « traditionnel » du sous-programme, de celui que nous ajoutons.

La nouvelle syntaxe facilite la sélection du sous-programme qui doit effectivement être appelé, comme l'illustre le listing 6.1. 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.

```
subprogram sp_a
  features
     a: out data port integer ;
end sp_a ;

thread thread_a
  features
     p: in data port integer ;
```

```
    end thread_a ;

    process implementation processus_a.impl
     subcomponents
        sp1 : subprogram sp_a;
        thread1 : thread thread_a;
     connections
        cx1 : data port sp1.a -> thread1.p;
    end processus_a.impl;
```

Listing 6.1 – Instanciation du sous-programme

La traduction du sous-programme vers BIP est décrite dans le listing 6.2. Nous avons gardé la même hiérarchie de composition. Le sous-programme est traduit en BIP sous forme d'un composant atomique. Il est instancié au niveau du composant composite (*processus_a.impl*).

```
compound type  processus_a.impl
    component sp_a  sp1
    component thread_a thread1
    connector typePortCnx cx1(sp1.a, thread1.p)
    ...
end

atomic type sp_a
   data int a_data
   export port intPort a ( a_data )
   ...
end

atomic type thread_a
   data int p_data
   export port intPort p ( p_data )
   ...
end

port type intPort(int i)

connector type typePortCnx (intPort outp ,intPort inp)
   define inp outp
   on inp outp
   down inp.i = outp.i ;
end
```

Listing 6.2 – Traduction du sous-programme en BIP

6.2 Intégration du comportement dans les devices

6.2.1 Limitation du standard

Les devices représentent des boîtes noires dont on ignore le comportement et la structure interne, contrairement aux autres composants. Cette sémantique réduit l'utilisation des devices AADL dans le cas où nous voulons définir le comportement et produire un système exécutable. Il est nécessaire d'enrichir la syntaxe des devices dont on ignore la structure interne.

6.2.2 Extension de la syntaxe

La solution que nous proposons est de promouvoir les devices AADL au même rang que les threads et les sous-programmes, et ainsi de pouvoir les instancier et définir leur structure interne. Soit par la référence à une annexe comportementale, soit par la description du comportement à l'aide d'un fichier externe. De cette façon, nous différencions les devices décrits par AADL de ceux que nous ajoutons. Cette extension permet aux utilisateurs de définir des nouveaux composants qui sont différents de ceux décrits dans le standard AADL et qui ont une sémantique bien définie par l'utilisateur.

La nouvelle syntaxe facilite la description du comportement du device, comme l'illustre le listing 6.3. Dans cet exemple, le composant device contient une annexe pour spécifier le comportement du composant.

```
device implementation dev.Impl
  annex behavior specification {**
    state variables
      c : Behavior::integer;
    states
      INIT : initial state;
      END : return state;
    transitions
      INIT -[ ]-> END { c=c+1 };
**};
end dev.Impl;
```

Listing 6.3 – Modélisation du device

La traduction vers BIP est décrite dans le listing 6.4. Le composant device est traduit en BIP sous forme d'un composant atomique. Nous remarquons que les deux ports *internalPort0 et internalPort1* ne sont pas déclarés dans l'annexe comportementale. Les ports dans les transitions en annexe comportementale sont optionnels, c'est pour cela que la transition de l'état *INIT* vers l'état *END* ne contient pas de port. Par contre, les transitions en BIP doivent contenir un port, ce qui explique l'apparition du port *internalPort0*.

Dans l'annexe comportementale, l'état *return* indique le retour à l'état initial du comportement. Ce cas est représenté en BIP sous forme d'une transition de l'état *END* à l'état *INIT*, ce qui explique l'apparition du port *internalPort1*.

```
atomic type dev.Impl
    data int c
    port intPort internalPort0
    port intPort internalPort1
    place INIT
    place END
    initial to INIT
    on internalPort0
      from INIT to END
      do { c=c+1; }
    on internalPort1
      from END to INIT
end
```

Listing 6.4 – Traduction du device en BIP

6.3 Communication du flux de données

6.3.1 Limitation du standard

Une des propriétés que les systèmes critiques doivent généralement exhiber est le déterminisme : toute exécution du système avec les mêmes entrées doit produire les mêmes résultats. En fait, l'exécution déterministe est l'un des pré-requis pour la création des applications de haute intégrité, telles que les systèmes de contrôle ou embarqués. La raison principale pour cela est que la validation est généralement assurée par des tests de l'implémentation du système et tester une implémentation non-déterministe est difficile. Cela est dû au fait que le non-déterminisme peut rendre les tests non-reproductibles et la couverture par le test (*test coverage*) est difficile à définir et à mesurer.

Malheureusement, la communication des flux de données entre les composants AADL n'est pas déterministe, dans le cas où la période du thread expéditeur est différente de la période du thread récepteur. Cela conduit à une exécution non-déterministe en général, et empêche l'utilisation de AADL pour la plupart des systèmes critiques.

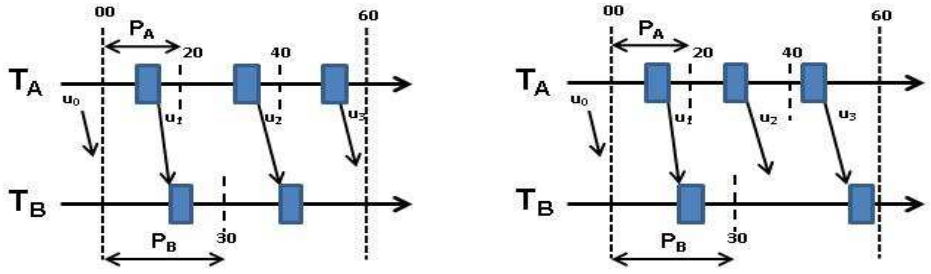
L'approche existante pour résoudre le problème du non-déterminisme du flux de données en AADL s'appuie sur l'utilisation des données de la dernière hyperpériode [IE07]. L'inconvénient de cette approche est que, si l'hyperpériode est très longue, alors les données envoyés par le thread expédi-

teur sont rarement utilisées par le thread receveur, surtout dans la première hyperpériode où le thread receveur ne reçoit aucune donnée. En outre, la méthode est limitée aux cas où le rapport entre la période longue P_l et la période courte P_s est un nombre entier $r = P_l/P_s$. Par exemple, le cas où : $P_l = 30$ et $P_s = 20$, conduisant à $r = 1.5$, n'est pas accepté par [IE07].

6.3.2 Non-déterminisme des communications du flux de données

Nous allons illustrer le non-déterminisme des communications du flux de données par un exemple simple. Considérons deux threads T_A et T_B, avec des périodes $P_A = 20ms$ et $P_B = 30ms$ respectivement. Nous avons étudié les différentes combinaisons possibles où le non-déterminisme peut se produire, afin de définir des règles générales. Nous distinguons quatre cas de non-déterminisme selon le type de connexions et selon la période. Dans cette section, nous nous intéressons plus particulièrement aux cas de non-determinisme produit par les connexions immédiates.

Dans le cas d'une connexion immédiate nous distinguons les deux cas suivants :

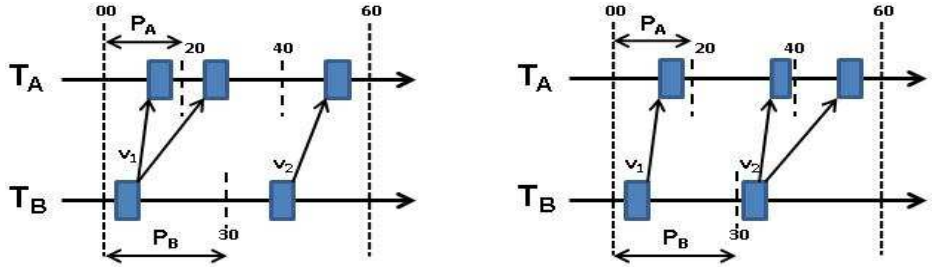


(a) : Scénario 1 (b) : Scénario 2

FIG. 6.1 – Non-determinisme : Connexion immédiate

- **La période du thread expéditeur est inférieure à la période du thread receveur :** Le thread T_A produit des données u qui sont consommées par le thread T_B, puis le non-déterminisme, comme indiqué dans la figure 6.1 peut se produire : dans le premier scénario (a), la deuxième exécution du thread T_B est lancée juste après que T_A termine sa deuxième exécution et produit une donnée u_2, qui est consommée par T_B en tant que donnée du flux de données. Au cours du deuxième scénario (b), la deuxième exécution du thread T_B consomme la dernière donnée u_3 produite par le thread T_A.
- **La période du thread expéditeur est supérieure à la période du thread receveur :** Le thread T_B produit des données v qui sont consommées par le thread T_A, puis le non-déterminisme, comme indiqué dans la figure 6.2 peut se produire : dans le premier scénario (a),

la deuxième exécution du thread T_A consomme la première donnée v_1 produite par la première exécution du thread T_B. Au cours du deuxième scénario, T_B est lancé avant la deuxième exécution du thread T_A et produit une donnée v_2, qui est consommée par le thread T_A comme une donnée du flux de données.



(a) : Scénario 1 (b) : Scénario 2

FIG. 6.2 – Non-determinisme : Connexion immediate

6.3.3 Protocole de communication déclenché par le temps

Nous avons expliqué en détail le non-déterminisme de la communication des données en AADL, et nous allons fournir un protocole de communication déclenché par le temps qui assure le déterminisme. Nous précisons que nous considérons les systèmes de la forme suivante :

- Le thread expéditeur et le thread receveur ont la propriété *Dispatch_Protocol => Periodic*.
- La connexion doit être une connexion de données, pas une connexion d'événement ou une connexion de données et d'événement.

Par manque de formalisation et d'approche formelle pour résoudre le non-déterminisme entre les threads en AADL, sans l'utilisation de l'hyperpériode, nous proposons un nouveau protocole déclenché par le temps, qui consiste en les deux règles suivantes :

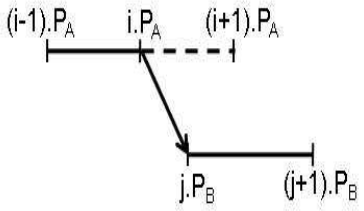


FIG. 6.3 – Deuxième règle

1. Une données produite par le thread expéditeur est disponible uniquement à la fin de sa période.
2. Les données sont consommées par le thread récepteur à la prochaine période qui se produit pendant ou après la date limite du thread expéditeur, comme le montre la figure 6.3.

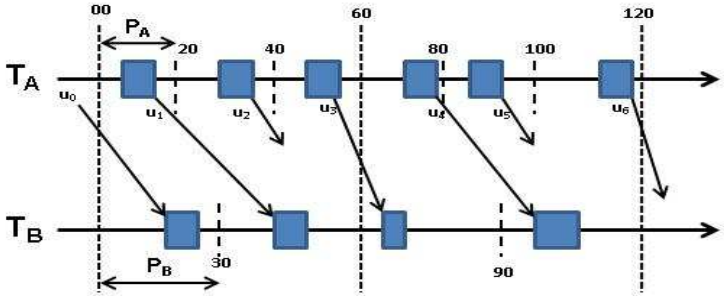


FIG. 6.4 – Protocole de communication pour une connexion immédiate

La figure 6.4 montre notre protocole de communication appliqué à la connexion de données immédiate. Chaque donnée produite par le thread T_A est disponible à la fin de sa période. Le thread T_B consomme une donnée au début de sa période qui se produit pendant ou après le date limite du thread T_A.

Theorem 6.3.1 *Le protocole de communication déclenché par le temps défini par les règles (1) et (2) assure une communication du flux de données déterministe entre les threads AADL.*

6.3.4 Correction du protocole de communication

Nous fournissons un argument formel pour la correction de notre protocole. Nous allons introduire tout d'abord quelques notations. La notation $T_A(i) \rightarrow u(t)$ signifie que la $i^{ème}$ activation du thread T_A produit des données u disponibles au moment t. De même, la notation $u(t) \rightarrow T_B(j)$ signifie que les données u disponibles au moment t seront consommées par la $j^{ème}$ activation du thread T_B. Avec cette notation, notre protocole stipule que :

$$\begin{array}{ll} \forall i \geq 1: & T_A(i) \rightarrow u(i \cdot P_A) \\ \forall j \geq 1: & u((j-1) \cdot P_B) \rightarrow T_B(j) \end{array}$$

À partir de ces deux règles, nous obtenons que :

$$\forall j \geq 1. \exists! i \geq 1: \quad T_A(i) \rightarrow u(i \cdot P_A) = u(j \cdot P_B) \rightarrow T_B(j)$$

En fait, l'unique activation i satisfaisant la condition est telle que :

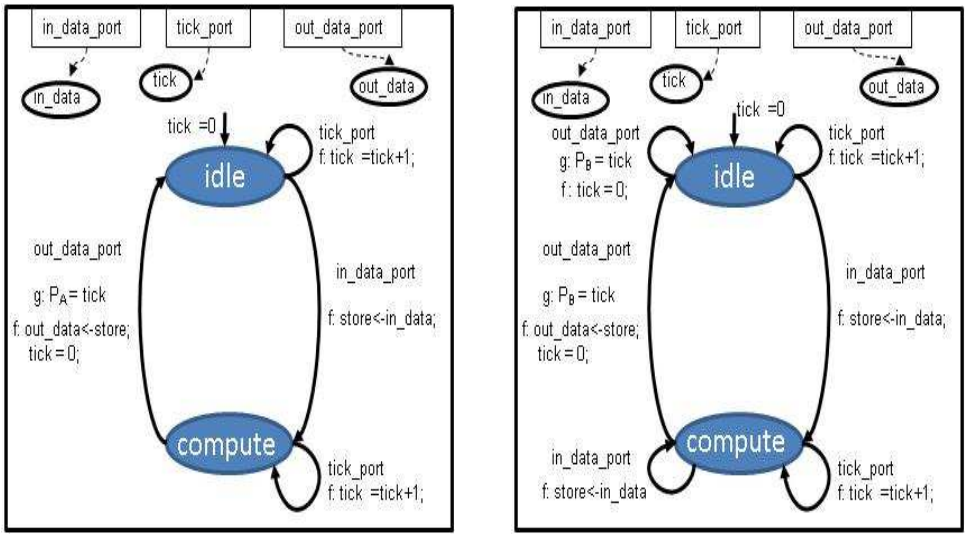
$$(j-1) \cdot \frac{P_B}{P_A} < i \leq j \cdot \frac{P_B}{P_A}$$

Cette relation assure une communication du flux de donnée déterministe entre les deux threads : chaque donnée consommée par une activation (j) du thread récepteur correspond à une unique activation (i) du thread producteur.

6.3.5 Modélisation des connexions de données en BIP

Nous avons vu dans la section 5.9.1.3 la modélisation des connexions de port de donnée en BIP. Cette modélisation n'assure pas le déterminisme. Afin d'obtenir une communication déterministe entre les threads périodiques, nous nous appuyons sur deux composants atomiques BIP supplémentaires :

- Le premier composant atomique, nommé *get_data*, est présenté dans la figure 6.5 (a). Le rôle de ce composant est de retarder le transfert de données à la fin de la période du thread expéditeur. Ce composant représente la première règle définie dans la section 6.3.3.
- Le deuxième composant atomique, nommé *set_data*, est présenté dans la figure 6.5 (b). Le rôle de ce composant est de retarder le transfert de données lors de la prochaine période qui se produit pendant ou après la date limite du thread receveur. Ce composant représente la deuxième règle définie dans la section 6.3.3.



(a) : *get_data* (b) : *set_data*

FIG. 6.5 – Modélisation du protocole de communication en BIP

La description syntaxique des composants *get_data* et *set_data* de la figure 6.5 est disponible en [AADa].

À l'aide de ces composants atomiques, nous assurons la communication déterministe entre les threads AADL comme suit :

Connexion immédiate :

Nous appelons **D***eterministic* **B***IP* **C***ommunications for* **I***mmediate connections* (**DBCI**), les deux composants atomiques (*get_data* et *set_data*),

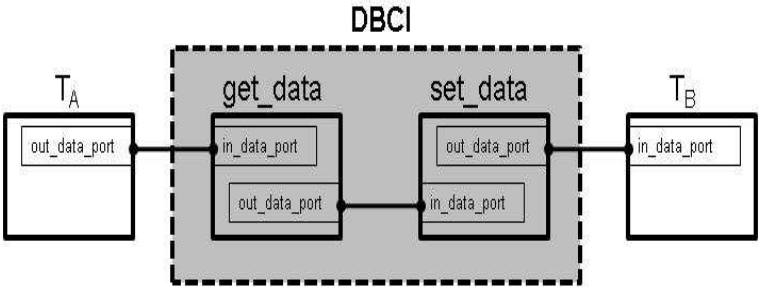


FIG. 6.6 – Communications entre les threads et DBCI

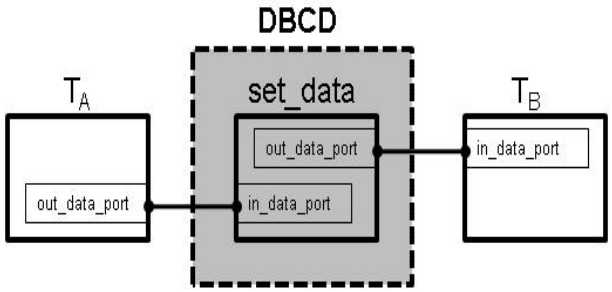


FIG. 6.7 – Communications entre les threads et DBCD

qui implémentent le flux de données entre deux threads. Les communications entre les deux threads et DBCI sont représentés dans la figure 6.6

Connexion retardée :

Nous appelons ***D**eterministic **BIP** **C**ommunications for **D**elayed connections* par (**DBCD**), le composant atomique *set_data* qui implémente le flux de données entre deux threads. Les communications entre les deux threads et DBCD sont représentées dans la figure 6.7. Nous avons ajouté un seul composant dans ce cas, parce que le premier composant (*get_data*) est inclus dans la sémantique d'une connexion retardée.

Le type de connecteurs BIP entre les threads et DBCI/DBCD est une synchronisation forte avec le transfert de données à partir du port *out_data_port* vers le port *in_data_port*.

6.4 Modélisation des systèmes répartis en AADL

6.4.1 Généralités

La complexité croissante des systèmes logiciels force à développer de nouvelles approches pour faire face aux défis des nouvelles applications gérant les systèmes répartis. L'usage des systèmes répartis s'est aujourd'hui généralisé,

tant auprès du grand public (téléphonie mobile, ...) qu'au sein de systèmes complexes (aérospatiale, automobile, ...). Ces systèmes sont caractérisés par leur aspect distribué, sur des réseaux hétérogènes, mettant en jeu des technologies internet et intranet, tout en devant répondre à des impératifs fonctionnels et non-fonctionnels. En conséquence, il est de plus en plus difficile de concevoir, développer et gérer de façon cohérente ces systèmes.

6.4.2 Utilisation d'AADL pour décrire une application répartie

Nous exposons dans cette section une façon d'utiliser AADL pour décrire le déploiement d'une application répartie. Pour décrire la répartition des différents noeuds de l'application, nous nous plaçons à un niveau d'abstraction assez élevé, dans lequel la structure des noeuds n'est pas explicitée. Nous nous focalisons sur la description du déploiement et des relations entre les noeuds. 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 associés 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.

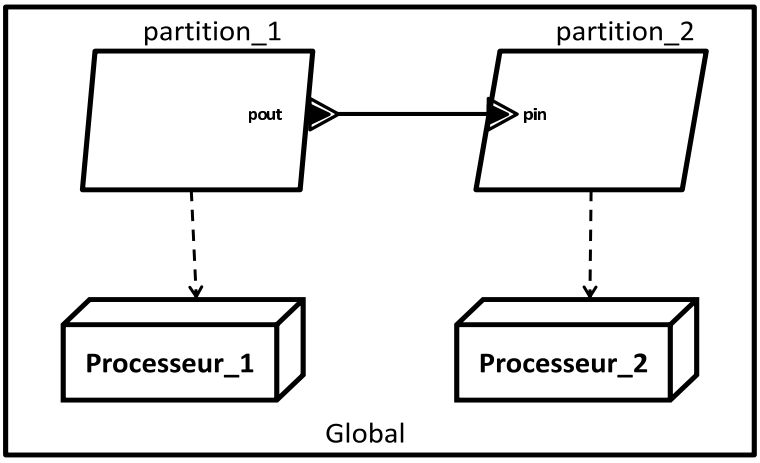


FIG. 6.8 – Représentation graphique d'une application sur deux processeurs

Le listing 6.5 représente deux partitions communicantes, chacune s'exé-

cutant sur un processeur. Les ports des processus sont connectés pour modéliser les communications entre les deux partitions. La représentation graphique du listing 6.5 est montrée dans la figure 6.8.

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 des réseaux de communication ne sont pas prises en compte dans cette approche. Les éventuelles informations concernant la transmission des données entre les partitions devraient être indiquées au moyen du composant bus associé à la connexion.

```
process Partition_1
  features
    pout : out event data port;
end Partition;
process Partition_2
  features
    pin : in event data port;
end Partition;
processor Machine
end Machine;

system Global
end Global;

system implementation Global.implem
  subcomponents
     partition_1 : process Partition_1;
     partition_2 : process Partition_2;
     processeur_1 : processor Machine;
     processeur_2 : processor Machine;
  connections
     event data port partition_1.pout -> partition_2.pin;
  properties
     actual_processor_binding => reference
        (processeur_1) applies to partition_1;
     actual_processor_binding => reference
        (processeur_2) applies to partition_2;
end Global.implem;
```

Listing 6.5 – Modélisation d'une application sur deux processeurs

Intégration des informations de déploiement

Une connexion entre deux noeuds doit être mappée sur un bus pour permettre de spécifier une connexion à travers un réseau de communication.

Le listing 6.6 montre l'exemple précédent en ajoutant un bus pour modéliser un réseau de communication entre deux noeuds.

```
bus Ethernet_Bus
end Ethernet_Bus;

system implementation Global.implem
  subcomponents
    ETH : bus Ethernet_Bus;
    ...
  connections
    event data port partition_1.pout -> partition_2.pin
       {Actual_Connection_Binding => (reference ETH);};
  ...
end Global.implem;
```

Listing 6.6 – Binding d'une connexion sur un bus

L'utilisation d'AADL fait apparaître la distinction entre partition logicielle et composant matériel supportant l'exécution des partitions. Les informations de localisation des partitions ne sont pas supportées par AADL. Cependant, nous proposons d'ajouter les informations de localisation qui seront associées directement aux composants AADL sous forme de nouvelles propriétés. Ces propriétés décrivent l'adresse IP et le port de communication inter processus de la machine.

La propriété qui décrit l'adresse IP de la machine sera intégrée dans le processeur AADL, et la propriété qui décrit le port de communication sera intégrée dans le port AADL. Le listing 6.7 illustre un déploiement d'une application répartie sur deux machines. Ces propriétés vont permettre la génération de code sans l'intervention de l'utilisateur pour ajouter ces informations à la main.

6.4.3 Prototypage d'une application répartie

AADL permet au concepteur de décrire les différents aspects d'une application répartie (nombre de processeurs, le nombre de threads dans chacun des processeurs, la connexion entre les threads ...). Pour construire une application répartie, nous commençons avec un modèle construit par le concepteur de l'application, qui relie ses entités d'application sur une architecture matérielle. Ensuite, nous utilisons la traduction d'AADL vers BIP pour générer un modèle BIP conforme à la sémantique AADL. Enfin, cette architecture est testée pour la solidité, toute disparité dans l'application est rapportée par la chaîne d'outils et d'analyse de BIP.

La figure 6.9 montre les étapes de la génération à partir de la description d'un système répartie AADL vers une application exécutable répartie comme suit :

```
process Partition_1
  features
    pout : out event data port {PORT_Number => "6000"};
end Partition;

process Partition_2
  features
    pin : in event data port {PORT_Number => "6000"};
end Partition;

bus Ethernet_Bus
end Ethernet_Bus;

processor Machine
end Machine;

system Global
end Global;

system implementation Global.implem
  subcomponents
      partition_1 : process Partition_1;
      partition_2 : process Partition_2;
      processeur_1 : processor Machine
            {IP_adress => "129.88.107.53"};
      processeur_2 : processor Machine
            {IP_adress => "129.88.107.54"};
     ETH : bus Ethernet_Bus;
  connections
    event data port partition_1.pout -> partition_2.pin
      {Actual_Connection_Binding => (reference ETH);};
  properties
      actual_processor_binding => reference
        (processeur_1) applies to partition_1;
      actual_processor_binding => reference
        (processeur_2) applies to partition_2;
end Global.implem;
```

Listing 6.7 – Déploiement des partitions en AADL

1. identifier chaque système et chaque connecteur mappé au bus.
2. générer pour chaque système AADL sa description correspondante en BIP, et pour chaque connecteur mappé au bus un protocole de com-

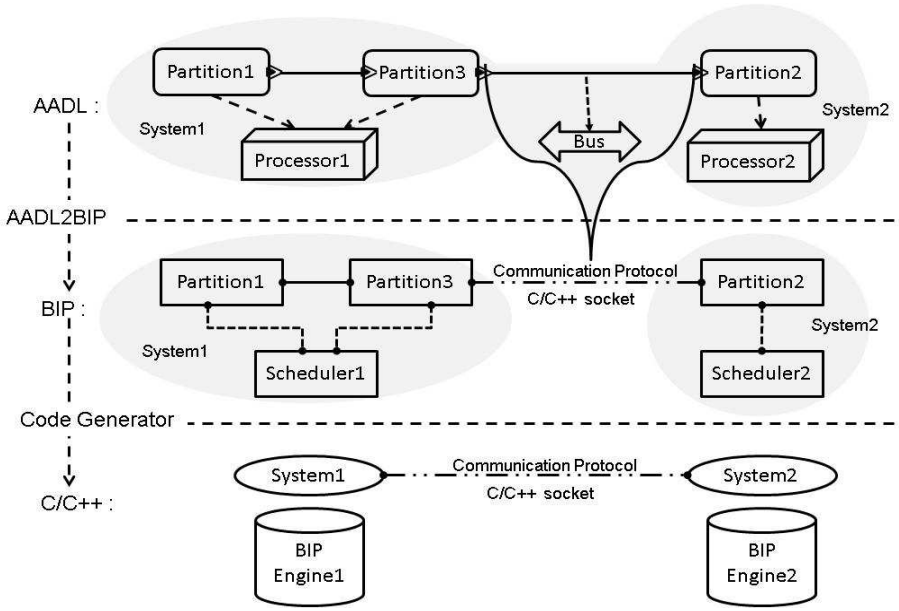


FIG. 6.9 – Déploiement d'une application répartie

munication.

3. compiler les systèmes BIP générés et produire un exécutable pour chaque système avec le protocole de communication.
4. exécuter et déboguer l'application distribuée.

Notre protocole supporte les communications entre deux ou plusieurs ordinateurs. Il fournit un canal de communication full-duplex entre les processus qui ne sont pas forcément sur le même ordinateur. De même, nous considérons que les canaux d'échange de données entre plusieurs threads dans un ou plusieurs processus sont gérés par le moteur d'exécution de BIP, si seulement si, les processus sont exécutés sur le même ordinateur. Sinon, si les processus sont exécutés sur plusieurs ordinateurs connectés par un réseau, nous utilisons un protocole de communication réseau (*network communication protocol*).

La plupart des protocoles de communication réseau utilisent le modèle client/serveur. Ces termes font référence à deux machines qui communiqueront l'une avec l'autre. L'une des deux machines est le client qui se connecte à l'autre machine, le serveur, généralement pour faire une demande d'information. Notez que le client a besoin de connaître l'existence de l'adresse du serveur, mais le serveur n'a pas besoin de connaître l'adresse du client avant que la connexion soit établie.

Notre protocole de communication utilise les sockets. Les sockets représentent le moyen de programmer la communication entre des applications locales ou distantes. Elles sont généralement utilisées dans tout ce qui touche

le réseau. Les sockets servent à établir une transmission de flux de données entre deux machines ou applications.

Les étapes nécessaires pour établir un protocole de communication côté client sont les suivantes : (1) Initialisation du protocole de communication. (2) Connexion au serveur en passant en paramètre l'adresse IP et le numéro de port du serveur. (3) Dialogue avec le serveur par envoi et réception de données.

Les étapes nécessaires pour établir un protocole de communication côté serveur sont les suivantes : (1) Initialisation du protocole de communication. (2) Récupération de l'adresse IP et du numéro de port du serveur. (3) Prêt à accepter les requêtes des clients. (4) Acceptation d'une connexion d'un client (5) Envoi et réception de données.

La structure de données utilisée doit être codée avant de passer sur le réseau pour être décodée après son récupération. Pour cette étape, nous avons utilisé des fonctions pour le codage/décodage des données.

Bien que primitive, cette approche permet d'obtenir de 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 utilisé dans certains intergiciels plus évolués, tels que MPI (Message Passing Interface) [GWS05].

Le code BIP généré fournit un framework qui permettra d'appeler directement le code utilisateur si nécessaire. Ceci permet une conception rapide et flexible du système distribué et ne restreint pas les implémentations de l'utilisateur.

6.5 Conclusion

Dans ce chapitre nous avons introduit des extensions à la syntaxe du standard AADL 1.0 afin de permettre l'instanciation des sous-programmes, la définition du comportement interne des devices, la modélisation des communications déterministes et la modélisation des systèmes répartis. 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 la recomposition des composants, et fournit tous les éléments nécessaires pour décrire des composants logiciels.

Chapitre 7

Transformation de modèles BIP

Sommaire

Dans ce chapitre nous étudions deux démarches de transformation de BIP :

Une transformation d'une description BIP temporisée vers une description non-temporisée (à temps discret). La problématique que nous abordons ici est l'interprétation des éléments BIP temporisée en une description non-temporisée. Nous avons vu dans le chapitre précédent la transformation de AADL vers BIP, cette transformation (Comme d'autres transformations vers BIP) génère automatiquement une application BIP temporisée. Sachant que le moteur d'exécution de BIP ne supporte pas l'exécution des applications temporisées, une transformation formelle vers des applications non-temporisées doit avoir lieu.

Une transformation de modèles BIP pour l'optimisation et la préservation des propriétés. Les modèles BIP sont obtenus par la superposition de trois couches de modélisation : Comportement ; Interaction entre les transitions de la couche de comportement ; Règles de priorité utilisée pour choisir les interactions possibles. Ces deux derniers permettent d'ajouter des traitements qui sont exécutés lors des interactions entre les composants. L'évolution dans le domaine va vers une augmentation de ces structures d'exécution, car on s'oriente progressivement vers des modèles à composants composites. La multiplication de ces derniers peut générer un surcoût significatif en terme de temps d'exécution et d'occupation de la mémoire. Le principe général que nous appliquons pour optimiser des modèles BIP consiste à fusionner les composants pour réduire le surcoût induit par l'interprétation des connecteurs et des priorités.

7.1 Généralités

Dans cette section nous abordons deux démarches différentes. Une liée aux systèmes temporisés et l'autre pour l'optimisation des systèmes à base de composants.

(1) Au début des années 1990, plusieurs modèles ont été proposés, qui prennent en compte des contraintes quantitatives sur les délais séparant les événements, l'un les plus utilisés encore aujourd'hui étant le modèle des automates temporisés [AD94].

Les systèmes complexes, tels que les systèmes embarqués qui sont largement utilisés de nos jours (télécommunication, transport, automatique), sont souvent temporisés (incluant des contraintes temporisées), et ouverts (interagissant avec leur environnement). Un défi, à l'heure actuelle difficile, est d'analyser les systèmes en prenant en compte simultanément ces aspects.

Le formalisme des automates temporisés (*Timed Automata*, TA) (une extension des automates finis avec des horloges) est clairement reconnu comme adapté à la spécification et l'analyse des systèmes temps-réel. Les automates temporisés peuvent être décrits comme des automates finis auxquels ont été

rajoutées des variables, appelées horloges. Ces horloges évoluent au cours du temps, toutes à la même vitesse. Chaque transition d'un automate temporisé est étiquetée d'une part par l'action qui peut être effectuée, d'autre part par une contrainte d'horloges (i.e. une comparaison entre une horloge et une constante ou entre deux horloges) qui doit être vérifiée pour pouvoir franchir la transition, et enfin par un ensemble d'horloges qui devront être remises à zéro. La sémantique classique de ces automates est celle des mots temporisés. Ces derniers correspondent à une suite de couples (action,date) qui indiquent chacun quelle action est effectuée et à quelle date.

(2) La complexité croissante des systèmes informatiques entraîne des difficultés d'ingénierie de système à base de composants, en particulier liées à l'optimisation et la validation. Des approches d'ingénierie guidée par des modèles sont de plus en plus utilisées dans l'industrie dans l'objectif de maîtriser cette complexité au niveau de la conception. Ces approches encouragent la réutilisation et l'automatisation du cycle de développement. Elles doivent être accompagnées de langages et outils capables d'assurer la conformité du système implémenté aux spécifications.

Les techniques d'optimisation d'un langage à base de composants est un problème bien identifié des langages à base de composants. L'objectif de l'optimisation est « d'aplatir le code » pour générer du code monolithique en remplaçant des mécanismes de communication de haut niveau par des instructions simples. Le point de départ des techniques d'optimisation est l'étude de la sémantique opérationnelle pour la composition des composants qui évite l'explosion des états du système.

7.2 Systèmes temporisés

Les propriétés temporisées prennent une place importante dans le développement de nouvelles technologies en temps-réel. Le but de ce travail est de s'assurer que la mise en œuvre d'un système respecte les spécifications c'est-à-dire, que le système fonctionne correctement une fois qu'il a été mis en œuvre.

Les systèmes temporisés sont modélisés en BIP sous forme d'automates temporisés utilisant l'*urgence* [BS00, BST97], où l'exécution d'une transition est une action définissant un changement de l'état de contrôle. L'*urgence* est exprimée au moyen d'un attribut d'*urgence* sur les transitions. Cet attribut peut prendre les valeurs :

- *Lazy :* Une transition est considérée comme *lazy* à un état, si elle reste activée lors de la prochaine unité de temps discret, sinon, elle est *eager*.
- *Eager :* Les transitions *eager* sont exécutés à un moment donné au cours de laquelle elles deviennent activée, si elles ne sont pas désactivées par une autre transition.
- *Delayable :* Les transitions ne peuvent pas être désactivées par l'a-

vancée du temps.

Intuitivement, on ajoute un ensemble des variables d'horloge au composant atomique. Chaque horloge a une valeur qui croît avec le temps, et qui peut être réinitialisée ou prendre une autre valeur lors d'une transition. La valeur des horloges peut être testée par une garde sur chaque transition.

Les variables d'horloge local permettent la spécification de contraintes temporelles, telles que la durée des tâches (modélisée par le temps qui passe dans un état de contrôle associé à la tâche), et des échéances (*deadlines*) pour des actions dans un processus.

7.3 Le fondement théorique

Nous allons définir les éléments de BIP qui représentent la base de nos transformations. Ces définitions sont des représentations formelles des principaux éléments du langage BIP.

7.3.1 Définition des composants atomiques temporisés

Les composants temporisés sont conçus à partir des composants atomiques temporisés. La définition d'un composant atomique temporisé est inspiré de [AGS02].

Definition 7.3.1 (Composant atomique temporisé) *Un composant atomique temporisé est défini par $a = (S, P, V_t, V_u, T)$, tel que :*

- *S est un ensemble d'états (ou places).*
- *P est un ensemble de ports.*
- *V_t est un ensemble de variables temporisées (clocks).*
- *V_u est un ensemble de variables non-temporisées.*
- *T est un ensemble de transitions.*

Definition 7.3.2 (Transition) *Une transition est définie par $t = (s, p, g, f, s')$, telle que :*

- *s est un ensemble d'états initiaux.*
- *p est un port.*
- *f est une fonction.*
- *g est une garde.*
- *s' est un ensemble d'états finaux.*

Definition 7.3.3 (Garde) *Une garde g est définie par un triplet $(g_u, g_t, urgency)$, telle que :*

- *g_u est une garde non-temporisée.*
- *g_t est une garde temporisée.*
- *$urgency$ peut être lazy, eager ou delayeable.*

La définition des composants atomiques non-temporisés est similaire à la définition des composants atomiques temporisés sans les variables et les gardes temporisées.

7.3.2 Définition des connecteurs

Les connecteurs représentent la deuxième couche du modèle BIP. Ils permettent de décrire les interactions entre les transitions. Nous utilisons le symbole Γ pour représenter un ensemble de connecteurs. Un connecteur est défini de la façon suivante :

Definition 7.3.4 (Connecteurs) *Un connecteur γ est défini par un triplet (P, C, T), tel que :*

- *P est un ensemble de ports.*
- *$C \subseteq 2^P$ est un ensemble d'interactions complètes.*
- *T est un ensemble de transitions (t_α) telle que $\alpha \in C$*

Definition 7.3.5 (Transition) *Une transition t_α est définie par un triplet $(\alpha, G_\alpha, F_\alpha)$, telle que :*

- *α est une interaction (ensemble de port).*
- *G_α est une garde de l'interaction α.*
- *F_α est une fonction de l'interaction α.*

7.3.3 Définition des priorités

Les priorités représentent la troisième couche du modèle BIP. Nous utilisons le symbole Pr pour représenter un ensemble de priorités, telle qu'une priorité est définie de la façon suivante :

Definition 7.3.6 (Priorités) *Une priorité pr est définie par un triplet (G, α_l, α_h), telle que :*

- *G est une garde.*
- *α_l est une interaction low (non prioritaire).*
- *α_h est une interaction high (prioritaire).*

7.3.4 Définition des composants composites

Un composite regroupe tout les éléments de l'architecture, il permet de représenter une ou plusieurs hiérarchies. La définition d'un composant composite est la suivante :

Definition 7.3.7 (composants composites) *Un composant composite est défini par un triplet (A, Γ, Pr), tel que :*

- *A est un ensemble de composants atomiques temporisés.*

- Γ *est un ensemble de connecteurs, dont les ports appartient au composants atomiques.*
- Pr *est un ensemble de priorités sur les interactions de* Γ.

7.4 Transformation des descriptions BIP temporisées en non temporisées

Dans cette section, nous présentons notre méthodologie de transformation d'une description BIP temporisée vers une description non-temporisée. Cette transformation consiste à discrétiser et rendre explicite le progrès du temps dans le modèle. Elle doit conserver les propriétés : si un composant temporisé a été conçu pour satisfaire une certaine propriété, alors sa transformation vers un composant non temporisé doit satisfaire également cette propriété. Le but de ce processus est de générer un modèle BIP que nous pouvons le tester et le vérifier en utilisant des méthodes fournit par la chaîne d'outils BIP.

Nous allons définir les règles de transformation pour chaque élément accompagnés par des exemples pour en faciliter la compréhension.

7.4.1 Transformation des composants atomiques temporisés

La transformation des composants atomiques temporisés consiste à mettre en œuvre pour chaque composant atomique, une boucle de transition avec une garde égale à *true* et une fonction pour incrémenter les variables temporisées. Cette transition est marquée par un port spécial appelé *tick*, pour la synchronisation avec d'autres composants temporisés. La traduction est paramétrée par une fonction *inc*, qui doit être la même pour tous les atomes du système. Le principe de la transformation est expliqué dans la figure 7.3.

La transformation formelle d'un composant temporisé en composant non temporisé est décrite de la manière qui suit :

Definition 7.4.1 *Nous transformons un composant atomique temporisé défini par* $a = (S, P, V_t, V_u, T)$*, en un composant atomique non temporisé défini par :* $a' = (S', P', V', T')$*, où :*

- $P' = P \cup \{p_{tick}\}$
- $V' = V_u \cup V_t$
- $S' = S$
- $T' = T'' \cup T_{tick}$
  - $T'' = \{t'' = (s, g_u \wedge g_t, f, s') \mid \forall t = (s, p, (g_u, g_t, urgency), s') \in T\}$
  - $T_{tick} = \{t_{tick}(s_i) = (s_i, p_{tick}, true, inc, s_i) \mid \forall s_i \in S\}$
    - $inc :: v_t = v_t + 1 \ \forall v_t \in V_t$

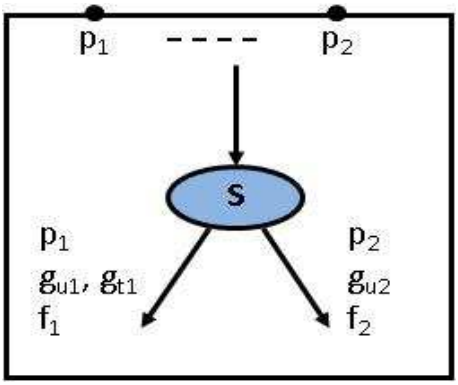


FIG. 7.1 – Composant temporisé

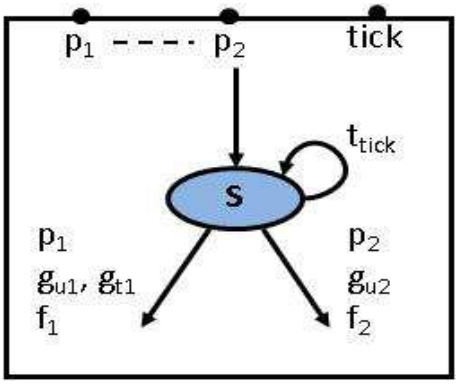


FIG. 7.2 – Composant non temporisé

FIG. 7.3 – Transformation d'un composant temporisé en un composant non temporisé

7.4.2 Transformation des connecteurs

Dans la composition des composants BIP résultants, une forte synchronisation est nécessaire entre tous les ports p_{tick}, comme indiqué dans l'architecture de la figure 7.4. Cela permet d'augmenter le temps discret à chaque interaction. γ_{tick} est un connecteur reliant tous les port p_{tick}. Il est défini comme un connecteur de type *Rendezvous* entre les différents ports p_{tick}.

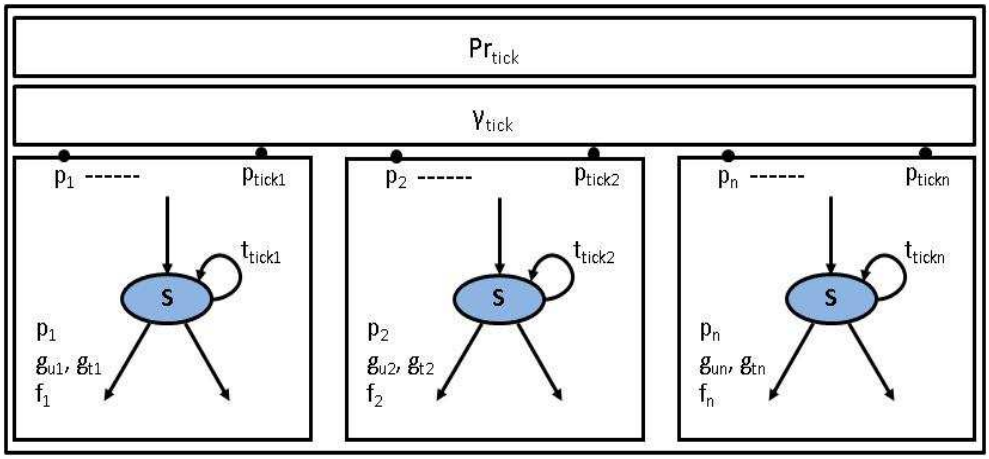


FIG. 7.4 – Composition du composants non temporisé

La définition du nouveau connecteur est la suivante :

Definition 7.4.2 *Le nouveau connecteur γ_{tick} est défini par un triplet (P_{tick}, $\{\alpha_{tick}\}$, T_{tick}) tel que :*

- $P_{tick} = \{p_{tick} \mid p_{tick} \in a'\}_{a' \in A'}$

- $T_{tick} = (\alpha_{tick}, true, \phi)$.

7.4.3 Transformation des priorités

Pour tenir compte de l'*urgence* des transitions, nous utiliserons les priorités. Le sens de la règle de priorité est que, s'il y a une transition *eager* activée dans un composant, le temps est moins prioritaire que cette transition. Ainsi, le temps peut progresser dans le cas ou il n'y a pas de transitions *eager* actives.

La définition de la nouvelle priorité est la suivante :

Definition 7.4.3 *Le symbole Pr_{tick} représente un ensemble de priorités appliqués sur les interactions α_{tick}, telle que une priorité pr_{tick} est définie par un triplet $(G', \alpha_{tick}, \alpha_h)$, où :*

- $\alpha_h = \{p_i\}_{i=1}^{n} \in \gamma$. *Obtenus à partir des transitions* $(s_i, p_i, (g_{u_i}, g_{t_i}, urgency_i), f_i, s'_i)_{i=1}^{n}$
- G' *est définit comme suit :*
  - $G' = False$, *si toutes les transistions sont lazy.*
  - $G' = \{\bigwedge_{i=1}^{n} g'_i\}$ *tel que :*

$$g'_i = \begin{cases} g_{t_i} & si\ urgency_i = eager \\ g_{t_i}\downarrow & si\ urgency_i = deleyable \\ True & si\ urgency_i = lazy \end{cases}$$

Pour la garde, nous la notons par une front descendant $g\downarrow$. Obtenue en remplaçant toutes les contraintes de la forme $x \leq k$ par $x = k$.

7.4.4 Transformation des composants composites

Après avoir transformé les composants atomiques et avoir défini les nouveaux connecteurs et les règles de priorités, nous allons rassembler tous ces éléments dans un composant. Nous définissons ce nouveau composant composite de la manière suivante :

Definition 7.4.4 (Composite temporisé vers un composite non temporisé) *Nous transformons un composant composite temporisé défini par (A, Γ, Pr), vers un composant composite non temporisé défini par (A', Γ', Pr') tel que :*

- $A' = \{a'_i\}_{i=1}^{n}$ *est un ensemble de composants atomiques non-temporisés.*
- $\Gamma' = \Gamma \cup \{\gamma_{tick}\}$.
- $Pr' = Pr \cup Pr_{tick}$

7.4.5 Exemple de transformation

Dans cette section, nous allons présenter un exemple de transformation d'une description BIP temporisée vers une description non-temporisée. Le

modèle BIP de *PeriodicTask*, comme le montre la figure 7.5, contient deux composants atomiques :

- *Task* : contient deux ports *start* et *finish*, deux états *READY* et *EXEC*, une donnée de type entier *execTime* et une donnée temporisée *clock*. Le comportement du composant *Task* est décrit dans le listing 7.1. La transition à travers le port *finish* contient une garde temporisée avec *urgency eager*.
- *Trigger* : contient un port *trigger*, une donnée de type entier *period* et une donnée temporisée *clock*. Le comportement du composant *Trigger* est décrit dans le listing 7.2.

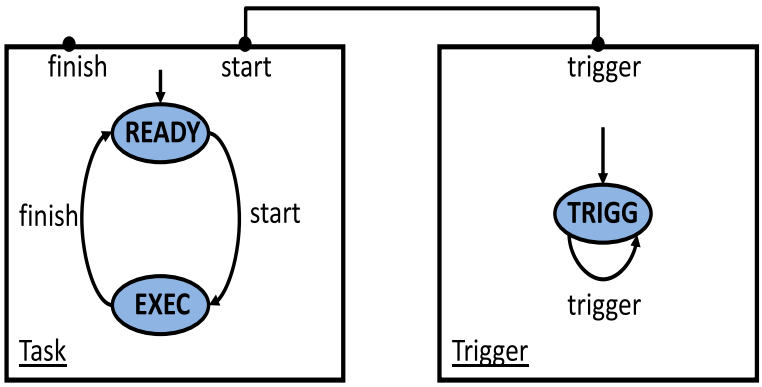


FIG. 7.5 – Architecture du PeriodicTask

```
behavior initial do
      execTime = ET;
      clock = 0;
      to READY
   state READY
      on start do clock = 0;
      to EXEC
   state EXEC
      on finish when (clock == execTime, eager)
      to READY
end
```

Listing 7.1 – Comportement du composant Task

```
behavior initial do
      period = P;
      clock = 0;
      to TRIGG
    state TRIGG
      on trigger when (clock == period, eager)
                  do clock = 0;
      to TRIGG
```

```
end
```

Listing 7.2 – Comportement du composant Trigger

Le modèle BIP de *PeriodicTask* contient aussi deux connecteurs : un connecteur de type *Rendezvous* qui relie les deux composants et un connecteur de type *Singleton* sur le port *finish*. Les connecteurs utilisés dans cet exemple permettent de connecter les ports des composants et de les rendre synchrones.

La transformation du modèle *PeriodicTask* temporisé vers un modèle *PeriodicTask* non-temporisé est décrite de la façon suivante :

Au niveau des composants atomiques : Pour chaque composant atomique du modèle, nous ajoutons une boucle de transition avec une garde égale à *true* et une fonction pour incrémenter les variables temporisées. Cette transition est marquée par un port spécial appelé *tick*, pour la synchronisation avec d'autres composants temporisés.

Le nouveau comportement du composant *Task* est décrit dans le listing 7.3. Nous appliquons la même chose pour le composant *Trigger*.

```
behavior initial do
    execTime = ET;
    clock = 0;
    to READY
  state READY
    on tick  do clock = clock + 1; to READY
    on start do clock = 0; to EXEC
  state EXEC
    on tick  do clock = clock + 1; to EXEC
    on finish provided (clock == execTime) to READY
end
```

Listing 7.3 – Comportement du composant Task

Au niveau des connecteurs : Comme nous avons indiqué précédemment, une forte synchronisation est nécessaire entre tous les ports *tick* des composants temporisés. La ligne 11 du listing 7.4 montre le connecteur appelé *cn_tick* produit lors de la transformation.

Au niveau des priorités : Les lignes de 14 à 19 du listing 7.4 montrent les nouvelles prioritées générés lors de la transformation. L'interaction du connecteur *cn_tick* est la moins prioritaire.

```
1  component System
2    contains Task T(2) // Task with execution time 2 units
3    contains Trigger Trigg(4) // Trigger with period of 4 units
4    connector finishConn = T.finish
5      behavior
6      end
7    connector triggerTask = Trigg.trigger, T.start
8      behavior
9        // Empty
10     end
11   connector cn_tick = Trigg.tick, T.tick
12     behavior
13     end
14   priority finishConn_tick
15     if T.clock == T.execTime
16     cn_tick : Trigg.tick, T.tick < finishConn: T.finish
17   priority triggerTask
18     if Trigg.clock == Trigg.period
19     cn_tick: Trigg.tick,T.tick < finishConn: Trigg.trigger,T.start
20 end
```

Listing 7.4 – Composant composite *System*

7.5 Optimisations du modèle

7.5.1 Approche

Notre objectif général est de mettre en œuvre des techniques d'optimisation des composants BIP. Ces optimisations visent à supprimer dans la mesure du possible les connecteurs et les priorités, sans toutefois perdre les bénéfices de ces derniers.

La transformation de modèles BIP nous permettra d'obtenir à partir d'un modèle BIP d'autres modèles qui sont optimaux et préservent des propriétés données. Notre travail nécessite une étude approfondie de la sémantique opérationnelle du langage et des techniques et technologies de transformation des modèles (*meta-modelling*).

Le principe général que nous appliquons pour optimiser des modèles BIP consiste à fusionner les composants pour réduire le surcoût induit par l'interprétation des connecteurs et des priorités. Les modèles optimisés sont obtenus par substitution d'un ensemble de composants par un composant produit équivalent.

7.5.2 Techniques d'optimisation

La figure 7.6 représente les trois techniques d'optimisation que nous utiliserons pour optimiser les programmes BIP :

1. *Composition des composants* : c'est la première étape pour l'optimisation des composants BIP, qui permet d'obtenir à partir d'un ensemble de composants un composant produit équivalent.
2. *Élimination des priorités* : c'est la deuxième étape pour l'optimisation des composants BIP, qui permet de remplacer les priorités par des fonctions simples sans changer le comportement des composants.
3. *Élimination des connecteurs* : c'est la troisième et la dernière étape pour l'optimisation des composants BIP, qui permet d'exécuter les composants BIP sans utiliser les connecteurs.

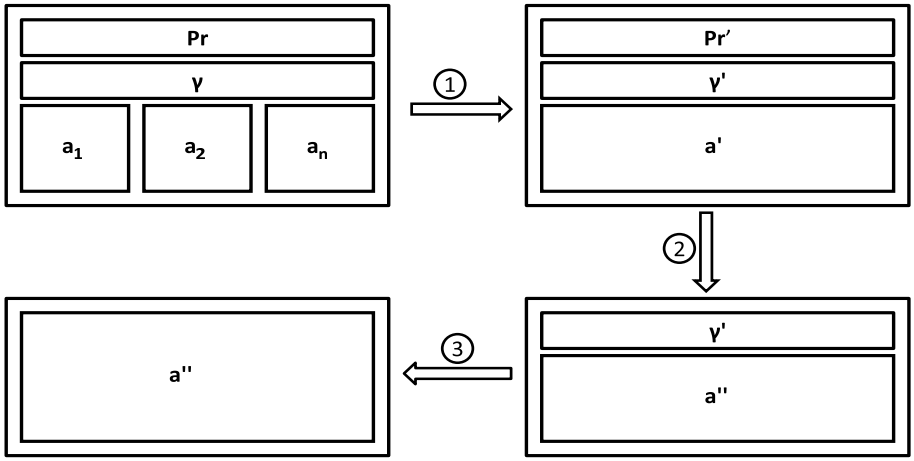


FIG. 7.6 – Les techniques d'optimisation

La difficulté de ces transformations est que le système doit préserver le comportement des composants et par conséquent leur propriétés.

7.5.3 Composition des composants

La composition des composants a pour but d'obtenir a partir d'un ensemble de composants un composant produit équivalent avec la préservation de la sémantique. Nous avons développé deux techniques de la composition des composants : binaire et n-aire. Le principe de la composition binaire est de sélectionner et de fusionner deux composants atomiques à la fois à partir d'un ensemble de composants jusqu'à l'obtention d'un seul composant atomique. Parcontre, dans le cas de la fusion n-aire, nous pouvons sélectionner et fusionner plus de deux composants atomiques. Nous allons présenter par la suite la technique de la composition n-aire.

Nous décomposons cette transformation en trois étapes : fusion des composants, fusion des connecteurs et fusion des priorités.

La figure 7.7 représente un exemple simple appelé système. Il contient deux composants atomiques, instanciés par P et C respectivement, qui peuvent communiquer grâce aux connecteurs de type Rendez-vous.

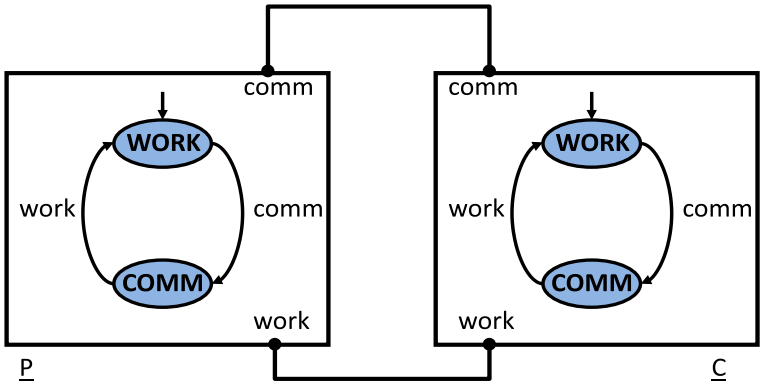


FIG. 7.7 – Architecture du système

Au niveau des composants atomiques : La première étape de la fusion concerne les composants atomiques. La figure 7.8 représente l'architecture du système après la fusion de ces deux composants atomiques. Nous remarquons l'apparition de deux transitions sous forme d'un réseau de Pétri qui peuvent avoir des interactions à travers les deux nouveaux ports. Dans cette figure nous n'avons pas représenté les anciens transitions et ports. Comme vous pouvez le constater par la suite, nous effectuons un renommage des ports, des variables et des états. Le renommage est effectué seulement dans le cas où il y a une collision.

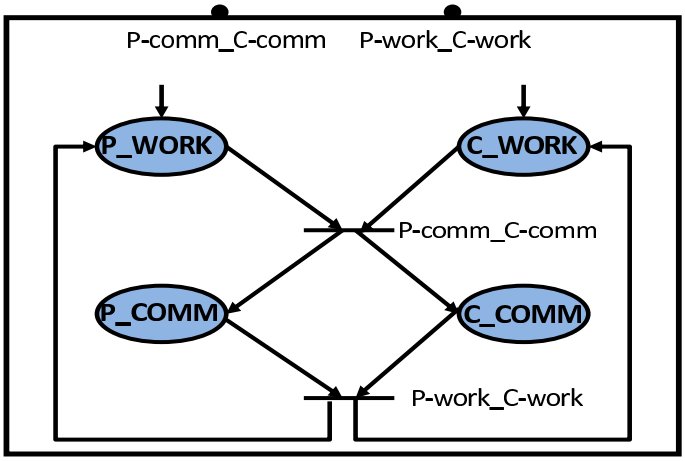


FIG. 7.8 – Architecture du système après la fusion

La définition du nouveau composant atomique est comme suit :

Definition 7.5.1 (Fusion des composants (1/2)) *Nous définissons la composition d'un ensemble de composants atomiques* $a_i = (P_i, V_i, S_i, T_i)$ *comme un composant atomique* a'*, défini par un quadruplet* (P', V', S', T')*, tel que :*

- $P' = P'' \cup P'''$
  - $P'' = \bigcup_{i=1}^{n} P_i$, *tel que* n *est le nombre de composants atomiques à fusionner.*
  - $P''' = \bigcup_{j=1}^{m} p(\alpha_j)$, *tel que* m *est le nombre d'interactions possible. Sachant que :* $p(\alpha) = p_1 p_2 ... p_m$, *tel que* $p_1, p_2, ..., p_m \in \alpha$
- $V' = \bigcup_{i=1}^{n} V_i$.
- $S' = \bigcup_{i=1}^{n} S_i$.
- $T' = T'' \cup T'''$, *où :*
  - $T'' = \bigcup_{i=1}^{n} T_i$, *tel que* n *est le nombre de composants atomiques à fusionner.*
  - $T''' = \bigcup_{j=1}^{m} T(\alpha_j)$, *tel que* m *est le nombre d'interactions possible.*

Definition 7.5.2 (Fusion des composants (2/2)) *La transition* Tr *représente un ensemble de transitions de* tr, *sachant que* Tr *est le produit cartésien entre toutes les transitions de l'interaction. La transition* $T(\alpha)$ *est définie par* $\bigcup$ $\{tr(\alpha, t_1...t_l) | \alpha$ *synchronise* $t_1...t_l$ $\}$, *telle que la transition* tr *est définie par* $(S_\alpha, p(\alpha), G_\alpha, F_\alpha, S'_\alpha)$, *où :*

- $S_\alpha = \bigcup_{k=1}^{l} S_k$ *est un ensembles des états initiaux.* l *représente le nombre de transitions sur une interaction.*
- $G_\alpha = g_1 \wedge g_2 \wedge ... \wedge g_l \wedge g_\alpha$, *où :*
  - g_k *est la garde de la transition, telle que* $g_k \in G$.
  - g_α *est la garde de l'interaction* α.
- $F_\alpha = f_1, f_2, ..., f_l; f_\alpha$, *où :*
  - f_α *est la fonction de l'interaction* α.
  - f_k *est la fonction de la transition, telle que* $f_k \in F$.
- $S'_\alpha = \bigcup_{k=1}^{l} S_k$ *est un ensembles des états finaux.*

Comme vous pouvez le constater dans la dernière définition, nous avons ajouté les gardes et les fonctions des interactions dans les transitions pour diminuer le rôle des connecteurs dans le modèle produit.

Au niveau des connecteurs : La deuxième étape de la fusion concerne les connecteurs. Cette transformation a pour but de transformer un connecteur (*Rendezvous*) qui relie plusieurs ports de différent composants vers un connecteur *Singleton*, c'est à dire, de diminuer le rôle des connecteurs dans le modèle BIP. Par exemple : Avant la fusion, nous avions un connecteur qui relie les deux ports *comm* des deux composants (figure 7.7). Après la fusion, ce connecteur relie un seul port *P-comm_C-comm* (figure 7.8).

La définition du nouveau connecteur est la suivante :

Definition 7.5.3 (Fusion des connecteurs) *Nous définissons un ensemble de connecteurs fusionner* $\{\gamma'_i\}_{i=1}^{n}$ *comme* $\Gamma' = \bigcup_{i=1}^{n} \gamma'_i$ *(*n *représente le nombre de connecteurs), tel que* $\gamma' = (p(\gamma), -, -)$ *est un connecteur de type Singleton, et :*

– $p(\gamma') = p_1 p_2 ... p_n / \ p_1, p_2, ..., p_n \in \gamma$.

Au niveau des priorités : La troisième étape de la fusion concerne les priorités. Cette transformation a pour but de réduire un ensemble d'interactions low/high en une interaction low/high. La définition de la nouvelle priorité est comme suit :

Definition 7.5.4 (Fusion des priorités) *Nous définissons un ensemble de priorités produit par la fusion par* $\{pr'_i\}_{i=1}^{n}$ *(n représente le nombre de priorités), telle que une priorité* pr' *est définie par un triplet* $(G, p(\alpha_l), p(\alpha_h))$*, où :*

– $p(\alpha_l) = p_1 p_2 ... p_n / \ p_1, p_2, ..., p_n \in \alpha_l$.
– $p(\alpha_h) = p_1 p_2 ... p_n / \ p_1, p_2, ..., p_n \in \alpha_h$.

Par exemple : supposons que l'interaction du connecteur qui relie les deux ports *work* des deux composants est moins prioritaire que le connecteur qui relie les deux ports *comm* des deux composants (figure 7.7), comme le montre le listing 7.5.

```
priority P1 Wr:P.work, C.work < Cm:P.comm, C.comm
```

Listing 7.5 – Priorité avant la fusion

Après la fusion, la nouvelle priorité est décrite dans le listing 7.6.

```
priority P1 Wr:P_C.P–work_C–work < Cm:P_C.P–comm_C–comm
```

Listing 7.6 – Priorités après la fusion

Après avoir effectué la première technique d'optimisation que nous avons décomposé en trois étapes, c'est à dire, fusion des composants, des connecteurs et des priorités, nous allons rassembler tous ces éléments dans un composant. Nous définissons ce nouveau composant composite de la manière suivante :

Definition 7.5.5 (Composant Composite) *Nous transformons un composant composite défini par un triplet* (A, Γ, Pr)*, après avoir effectué la première technique d'optimisation, vers un composant composite défini par triplet* (a', Γ', Pr').

7.5.4 Élimination des priorités

Après une étude approfondie des rôles des priorités dans le modèle BIP, nous avons trouvé une solution pour les remplacer par des fonctions simples sans changer le comportement du modèle. Pour cela, nous savons que la priorité après la fusion est définie par un triplet $(G, p(\alpha_l), p(\alpha_h))$. Nous notons par la suite le port $p(\alpha_l)$ par p_l et le port $p(\alpha_h)$ par p_h.

Nous remplaçons la garde $g(p_l)$ de la transition qui accompagne le port p_l par cette formule :

$$g(p_l) = g(p_l) \wedge (\overline{g(p_h) \wedge G \wedge S}).$$

telle que G représente la garde de la priorité et S représente les états initiaux de la transition à travers le port p_h.

En général, le port p_l peut exister plusieur fois dans les règles de priorités. Pour cela, nous fournirons une définition générale pour l'élimination des priorités de la manière suivante :

$$g(p_{li}) = g(p_{li}) \bigwedge (\overline{g(p_{hi}) \wedge G \wedge S})_{i=1..n}$$

Après avoir effectué la deuxième technique d'optimisation qui consiste à éliminer les priorités et les remplacer par les gardes dans le comportement des composants. Nous définissons ce nouveau composant composite de la manière suivante :

Definition 7.5.6 (Composant Composite) *Nous transformons un composant composite défini par un triplet (a', Γ', Pr'), après avoir effectué la deuxième technique d'optimisation, par un composant composite défini par $(a'', \Gamma', -)$.*

7.5.5 Élimination des connecteurs

Après avoir montré la technique du fusion des composants et l'élimination des priorités, nous intéressons plus particulièrement dans cette section par l'élimination des connecteurs. Cette dernière, permet d'exécuter les composants BIP sans utiliser les connecteurs.

Les connecteurs ne joue aucun rôle dans le modèle BIP obtenu après la fusion des connecteurs, le déplacement de leurs fonctions dans le comportement des composants et l'élimination des priorités. Pour cela, nous avons éliminé les connecteurs du modèles. Cependant, cette transformation doit être suivie par une modification du moteur d'exécution de BIP pour permettre l'exécution des composants sans faire appel aux interactions. Cette étape nécessite une transformation de modèle et quelques modifications dans le moteur de BIP.

7.6 Évaluation des performances

Nous avons effectué des tests sur plusieurs études de cas afin d'évaluer les gains des techniques d'optimisations que nous avons développées. Dans cette section, nous allons présenter une étude de cas du WaterFlow du projet européen SPEEDS [1].

[1] http ://www.speeds.eu.com

Le modèle du WaterFlow est conçu à l'aide de quatre éléments : un conteneur qui contient de l'eau, un capteur qui détecte le niveau d'eau, un régulateur qui contrôle l'ensemble du système et d'une valve qui contrôle le débit d'entrée d'eau dans le récipient.

7.6.1 Le modèle WaterFlow

Composants atomiques

Le modèle BIP de Waterflow, comme le montre la figure 7.9, contient quatre composants atomiques :

- *Container* : contient deux ports *tick* et *send*. Le port *tick* est utilisé pour la synchronisation et le port *send* est utilisé pour envoyer le niveau d'eau. Il contient aussi deux états *Compute* et *Sync*. À travers une interaction sur le port *tick*, le container passe de l'état *Sync* à l'état *Compute* et effectue une mise à jour des données. À travers une interaction sur le port *send*, le container passe de l'état *Compute* à l'état *Sync*.
- *Sensor* : est utilisé pour détecter le niveau d'eau dans le récipient (*Container*). Il dispose de trois ports *tick*, *send* et *getwl*. Le port *tick* est utilisé pour la synchronisation, le port *send* exporte le niveau d'eau au *Controller* et le port *getwl* reçoit le niveau d'eau à partir du *Container*.
- *Controller* : est utilisé pour contrôler la valve par l'envoi de commandes. Ces commandes ont pour but d'ouvrir ou de fermer la valve. Il dispose de trois ports *tick*, *send* et *getwl*. Le port *tick* est utilisé pour la synchronisation, le port *send* exporte la valeur de l'état à *Valve* et le port *getwl* reçoit le niveau d'eau à partir du *Sensor*.
- *Valve* : agit comme la valve dans le débit d'entrée du conteneur (*Container*). Il dispose de trois ports *tick*, *send* et *getwl*. Le port *tick* est utilisé pour la synchronisation, le port *getwl* reçoit la valeur de l'état à partir du *Controller* et le port *send* effectue une mise à jour des données.

Connecteurs

Les connecteurs utilisés dans cette étude de cas permettent de connecter les ports des composants et de les rendre synchrones. Les connecteurs utilisés sont de type *Rendezvous*.

Composite

Le composant composite utilisé dans cette étude de cas représente le haut niveau, et permet de rassembler les différents composants atomiques. Il contient une instance C du composant *Container*, une instance S du composant *Sensor*, une instance Ct du composant *Controller* et une instance V

du composant *Valve*. Il dispose également des connecteurs synchrones qui synchronisent toutes les instances. Il contient aussi des priorités entre les interactions. Par exemple, le connecteur qui relie le port *send* du composant *Controller* et le port *getwl* du composant *Valve* est moins prioritaire que le connecteur qui relie le port *send* du composant *Sensor* et le port *getwl* du composant *Controller*.

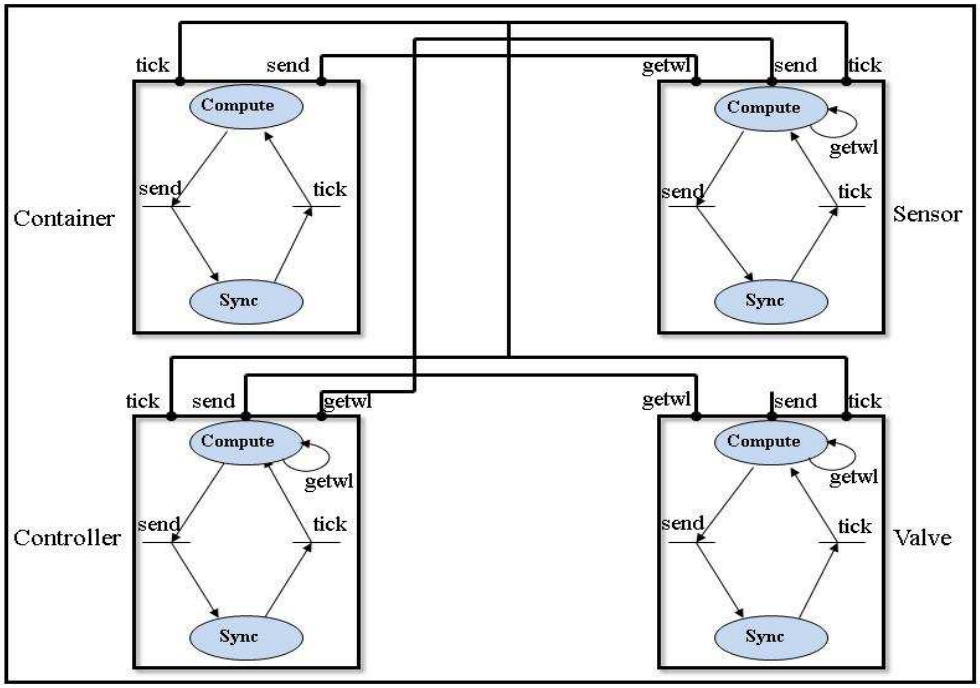


FIG. 7.9 – Le modèle du WaterFlow

7.6.2 Optimisation du modèle WaterFlow

Nous avons développé deux techniques de fusion de composants : binaire et n-aire. Le principe de la fusion binaire est de sélectionner et de fusionner deux composants atomiques jusqu'à l'obtention d'un seul composant atomique. Parcontre, dans le cas de la fusion n-aire, nous pouvons sélectionner et fusionner plus de deux composants atomiques.

Comme vous pouvez le constater par la suite, nous effectuons un renommage des ports, des variables et des états. Le renommage est effectué seulement dans le cas où il y a une collision. Par exemple, dans le modèle du WaterFlow, les quatre composants contient un port nommé *send*. Lors de la fusion des composants, nous obtiendrons quatre port *send* dans le même composant, ce qui provoque une collision.

Fusion binaire

Nous appliquons cette technique sur le modèle BIP comme le montre la figure 7.10. Pour chaque transformation, nous choisissons deux composants à fusionner, jusqu'à obtenir un seul composant atomique.

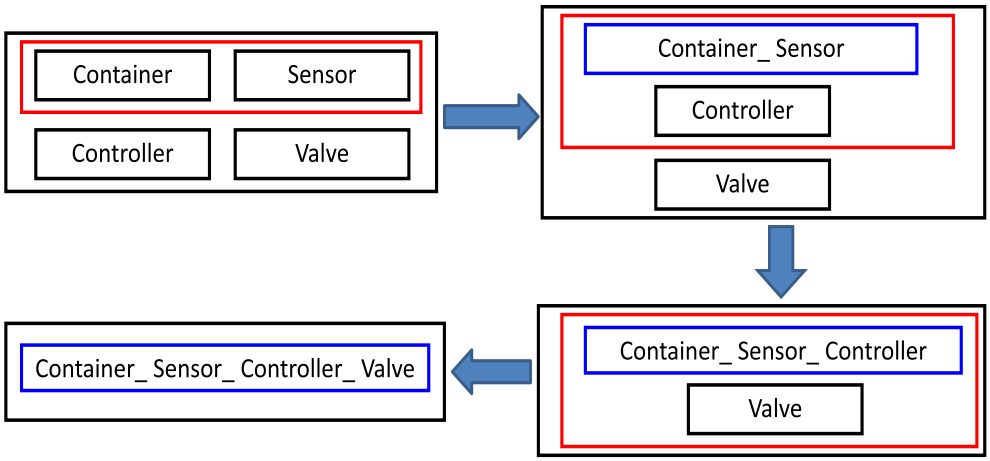


FIG. 7.10 – Fusion binaire

La figure 7.11 montre un exemple de fusion entre les deux composants *Container* et *Sensor*. Nous obtenons un nouveau composant atomique nommé *Container_Sensor*, qui contient les mêmes éléments de base. Il contient aussi deux nouvelles transitions à travers les ports *C_send_S_getwl* et *C_tick_S_tick*, qui sont produits par les interactions. Le connecteur qui reliait les deux ports *send* du composant *Container* et le port *getwl* du composant *Sensor* est transformé sous forme d'un connecteur de type *Singleton* avec un seul port.

Fusion n-aire

Nous appliquons cette technique sur le modèle BIP du WaterFlow. Nous obtenons un seul composant atomique comme le montre la figure 7.12. Dans cette figure nous n'avons pas représenté les anciennes transitions pour que le modèle soit compréhensible. Les connecteurs du modèle original sont transformés sous forme des connecteurs de type *singleton*. À travers les ports produits, nous remarquons que le comportement du modèle obtenu est sous forme de réseau de Pétri.

7.6.3 Tests

Nous avons effectué des tests sur l'étude de cas du WaterFlow afin d'évaluer les gains en temps d'exécution des techniques d'optimisation que nous avons développées. Le temps d'exécution représenté par la suite représente

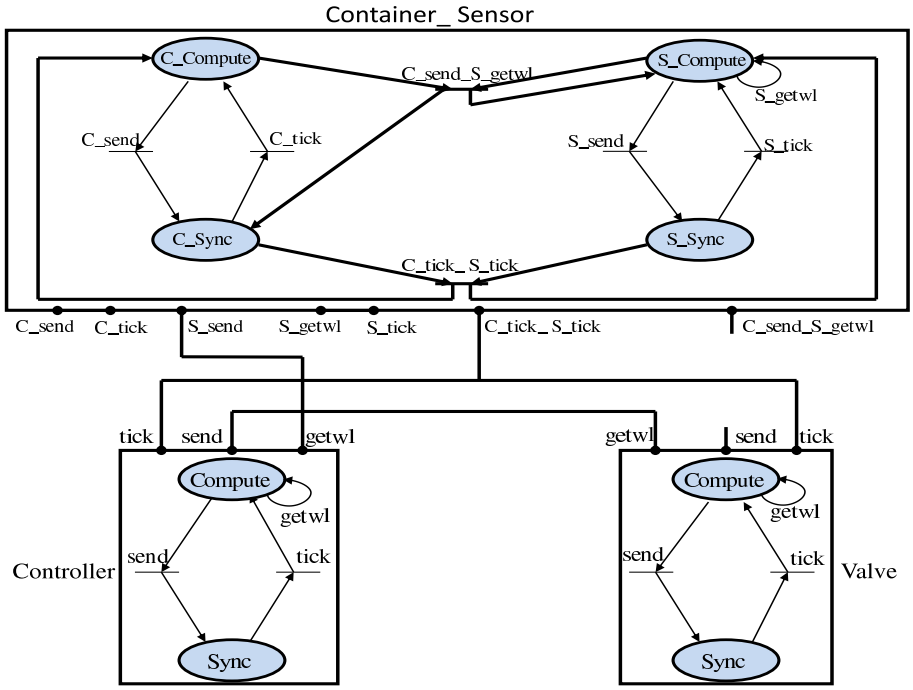


FIG. 7.11 – Fusion de deux composants

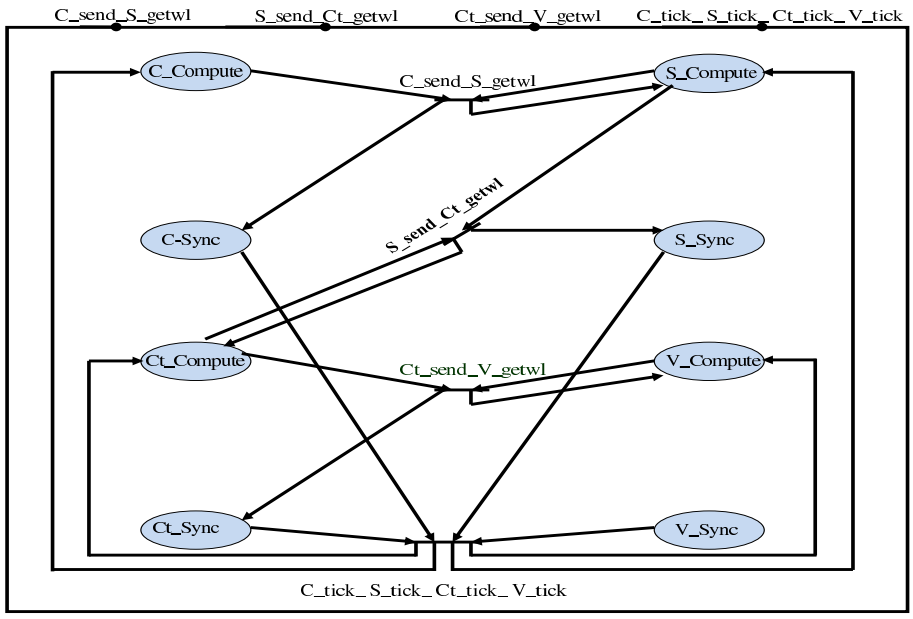


FIG. 7.12 – Fusion n-aire

le temps moyen d'exécution après une série de teste. Ce temps d'exécution correspond à une limite que nous avons fixé pour calculer le temps moyen.

Cette limite correspond à trois cents mille pas.

Nous avons fusionné progressivement les composants de l'étude de cas du WaterFlow, ce qui a donné successivement des modèles à trois, deux puis à un seul composant(s). Les gains en temps d'exécution sont donnés dans la figure 7.13. la première colonne représente le temps d'exécution du WaterFlow. La deuxième colonne représente le gain en temps d'exécution après la fusion de deux composants. La troisième colonne représente le gain en temps d'exécution après la fusion de trois composants. La quatrième colonne représente le gain en temps d'exécution après la fusion de quatre composants.

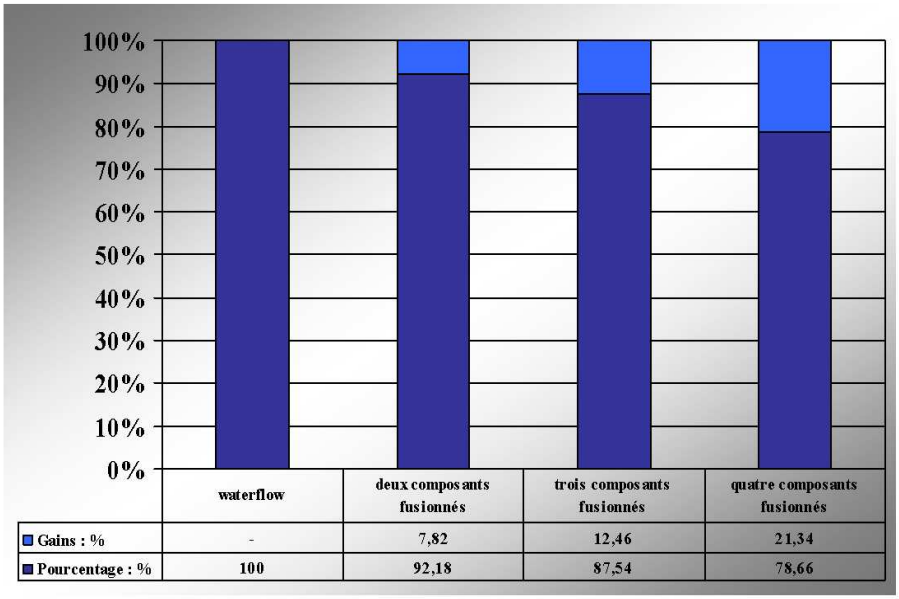


FIG. 7.13 – Évaluation des gains par fusion de composants

La figure 7.14 représente le temps d'exécution moyen avant et après la fusion de deux, trois et quatre composants. Nous remarquons que le temps d'exécution diminue avec la réduction du nombre des composants.

	WaterFlow	**Deux composants fusionnés**	**Trois composants fusionnés**	**Quatre composants fusionnés**
Temps d'exécution	3.0568 s	2.8176 s	2.676 s	2.4044 s

FIG. 7.14 – Temps d'exécution par fusion de composants

Sur la même étude de cas, après avoir fusionné les quatre composants, nous avons appliqué les deux techniques d'optimisations restantes. C'est à dire, l'élimination des priorités et des connecteurs. La figure 7.15 représente le gain en temps d'exécution par rapport au modèle original.

La première colonne représente le temps d'exécution pour WaterFlow. La deuxième colonne représente le gain en temps d'exécution après la fusion des quatre composants. La troisième colonne représente le gain en temps d'exécution après la fusion des composants et l'élimination des priorités, et la quatrième colonne représente le gain en temps d'exécution après l'application des trois techniques d'optimisations.

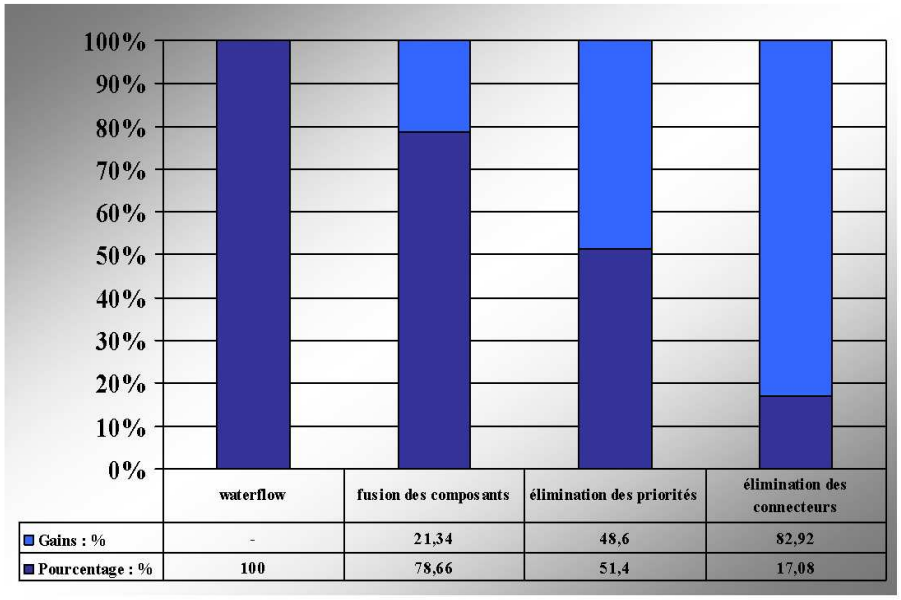


FIG. 7.15 – Évaluation des gains des techniques d'optimisations

La figure 7.16 représente le temps d'exécution moyen avant et après l'application des techniques d'optimisations. Nous constatons que le temps d'exécution du modèle optimisé est très réduit par rapport au temps d'exécution du modèle initial.

	WaterFlow	**Fusion des composants**	**Élimination des priorités**	**Élimination des connecteurs**
Temps d'exécution	3.0568 s	2.8176 s	1.5712 s	0.522 s

FIG. 7.16 – Temps d'exécution des techniques d'optimisations

7.7 Conclusion

Dans ce chapitre nous avons défini une représentation formelle des principaux éléments du langage BIP. Cette formalisation, nous a permis de définir

par la suite, les transformations que nous avons appliquées pour le langage BIP.

Nous avons présenté notre méthodologie de transformation d'une description BIP temporisée en une description non-temporisée. Cette transformation a pour but de générer un modèle BIP que nous pouvons tester et vérifier. Nous n'avons pas calculer le gain en temps d'exécution apporté par la discrétisation du temps car le moteur d'exécution de BIP ne supporte pas l'exécution des applications temporisées.

Ensuite, nous avons mis en œuvre les techniques d'optimisation des composants BIP. Ces optimisations visent à supprimer dans la mesure du possible les connecteurs et les priorités, sans toutefois perdre les bénéfices de ces derniers.

Nous avons effectué des tests sur plusieurs études de cas afin de montrer la validité de notre méthodologie et d'évaluer les gains en temps d'exécution des techniques d'optimisations que nous avons développées. Le gain en temps d'exécution est trés important et dépend de la structure des applications (composants, connecteurs, priorités).

Chapitre 8

Mise en Œuvre

Sommaire

Dans les chapitres précédents nous avons présenté un processus de conception basé sur AADL. Nous avons montré comment exploiter AADL pour produire une application BIP qui a une sémantique opérationnelle formellement définie. Nous avons aussi montré comment transformer une application temporisée vers une application non-temporisée et comment réduire le temps d'exécution des applications BIP.

Dans ce chapitre nous appliquons notre méthodologie sur des études de cas concrets afin d'en valider les propriétés et d'évaluer la performance des implantations réalisées. Nous présentons AADL2BIP, un traducteur pour AADL vers BIP que nous avons développé pour démontrer la validité de notre approche. Nous présentons également comment exploiter une description AADL pour la vérification formelle, en utilisant la chaîne d'outils fournis par BIP. Nous présentons aussi le simulateur et le débogueur développé spécialement pour AADL.

8.1 Chaîne d'outils

La chaîne d'outils que nous avons développée au cours de la thèse est décrite dans la figure 8.1. Elle contient les éléments suivants :

- *Éditeur textuel* : Permet la description du système en langage BIP. Nous l'avons développé en utilisant le langage Java et nous l'avons intégré dans la plateforme d'Eclipse.
- *Méta-modèle* : Nous avons contribué au développement du méta-modèle de BIP via la plate-forme Eclipse/EMF [EMF]. Dans ce contexte, il est recommandé, par exemple pour la génération automatique d'éditeurs de modèles, d'avoir un tel élément de base.
- *Parseur* : Pour analyser les programmes BIP et pour générer des modèles conformes au méta-modèle BIP à partir d'une description textuelle s'il n y a pas d'erreur dans le programme. Sinon, la ligne d'erreur est indiquée pour faciliter la correction. Nous avons développé le parseur en utilisant ANTLR [ANT].
- *Reconstructeur de code (deparser)* : Pour produire à partir d'un modèle BIP une description textuelle associée. Nous l'avons développé en utilisant ATL [ATL].
- *Transformations BIP vers BIP* : Divers outils de transformation de modèles BIP, comme la transformation de modèle BIP temporisé vers non-temporisé, et les transformations pour l'optimisation du temps d'exécution. Nous avons développé ces transformations en utilisant ATL et nous les avons intégré dans la platforme d'Eclipse.
- *Traduction d'AADL vers BIP* : Un outil de génération de BIP à partir d'AADL. Il est implémenté en Java [Jav], sous forme d'un plugin pour la plateforme open source *Eclipse*. Il prend comme entrée un modèle AADL (. aaxl) conforme au méta-modèle AADL et génère un modèle BIP conforme au méta-modèle BIP.
- *Simulateur et débogueur* : Pour effectuer la simulation des programmes AADL qui sont transformés en BIP. Il offre la possibilité à l'utilisateur de déboguer son programme au cours de l'exécution.

Toute la chaîne d'outils décrite précédemment est intégrée à Eclipse sous forme de plugins. Eclipse est un environnement de développement intégré libre, extensible, universel et polyvalent, permettant de créer des projets de développement mettant en œuvre n'importe quel langage de programmation. La spécificité d'Eclipse vient du fait de son architecture totalement développée autour de la notion de plugin, toutes les fonctionnalités de cet atelier logiciel sont développées en tant que plug-in.

Mise en œuvre avec ATL : *Atlas Transformation Language* (ATL) été conçu pour réaliser des transformations dans le cadre du Framework MDA proposé par l'OMG. Sa syntaxe abstraite a été décrite comme un métamodèle MOF [BDJ+03]. Sa syntaxe concrète textuelle quant à elle, a été définie

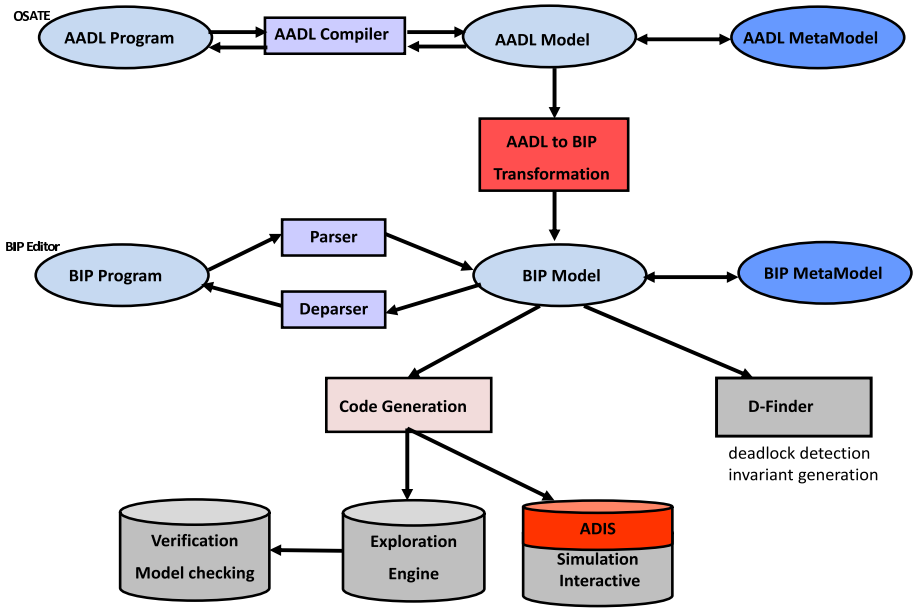


FIG. 8.1 – Architecture générale

en correspondance avec ce métamodèle. Un modèle de transformation ATL peut transformer un ensemble de modèles sources en un ensemble de modèles cibles. ATL est un langage de transformation de modèles « semi-déclaratif » : il permet de définir une transformation par le biais de règles déclaratives (*rule*) pouvant faire appel à des fonctions auxiliaires (*helper*) qui peuvent être récursives.

8.1.1 Méta-modèle de BIP

Un métamodèle est une description formelle de toutes les constructions syntaxiques d'un langage. Le langage de modélisation est représenté par un méta-modèle qui définit les éléments et la structure d'un modèle ainsi que sa sémantique. En d'autres termes, le méta-modèle exprime le lien existant entre un modèle et le système modélisé. Ces modèles appartiennent au langage représenté par le méta-modèle. Dans notre cas, tous les éléments du langage BIP ont une représentation dans le modèle. Les diagrammes de classes sont utilisés pour définir la relation entre les éléments. Les classes peuvent être liées entre elles grâce au mécanisme d'héritage qui permet de mettre en évidence des relations de parenté. D'autres relations sont possibles entre des classes, chacune de ces relations est représentée par un arc spécifique dans le diagramme de classes. Le méta-modèle de BIP a été conçu d'une manière efficace en termes de représentation compacte et réutilisable. La dernière version du méta-modèle à été développée en utilisant l'outil de modélisation d'UML PAPYRUS [PAP]. La figure 8.2 représente le diagramme de classe

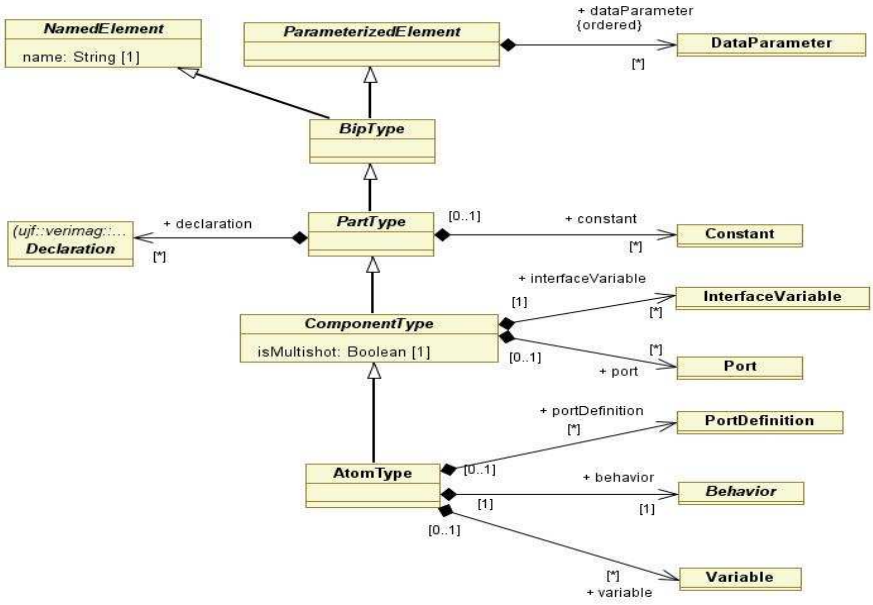


FIG. 8.2 – La représentation des atomes dans le méta-modèle

pour les composants atomiques de BIP.

8.1.2 Transformation de modèles

Traduction d'AADL vers BIP

Nous avons utilisé le langage Java [Jav] pour implanter l'outil de traduction d'AADL vers BIP (AADL2BIP). Cet outil est sous forme d'un plugin dédié à la plateforme open source *Eclipse*. Il prend comme entrée un modèle AADL (. aaxl) conforme au métamodèle AADL et génère un modèle BIP conforme au métamodèle BIP.

Nous avons développé l'outil AADL2BIP pour les deux versions de BIP. Le nombre de ligne de code pour la deuxième version est environ 6000 lignes de code.

Transformations BIP vers BIP

Nous avons utilisé le langage ATL pour les transformations de modèles BIP décrites dans le chapitre 7. Nous avons développé deux techniques pour exécuter ces transformations, soit en ligne de commande, soit en utilisant Eclipse.

Dans le cas d'utilisation en ligne de commande, nous avons implanté les commandes suivantes :

$--$**trans** <*transformation url*>

−−**in** <*id*>=<*model*> <*id*>=<*metamodel*> <*MDR/EMF*>
−−**out** <*id*>=<*model*> <*id*>=<*metamodel*> <*MDR/EMF*>
[−−**lib** <*id*>=<*library url*>]
[−−**params** <*param1*>, <*param2*>, ...]
[−−**next** −−**trans** ...] ;

L'appel à une transformation est spécifié par −−**trans**, suivi par son url. Ensuite, les éléments d'entrée sont spécifiés par la commande −−**in**, suivi par le modèle source, son métamodèle et par le type du métamodèle. Puis, Les éléments de sortie sont spécifiés par la commande −−**out**, suivi par le nom du modèle cible, son métamodèle et par le type du métamodèle. L'appel à une bibliothèque est spécifiée par −−**lib** suivi par son url, l'utilisation de la bibliothèque est optionnelle.

Dans le cas, d'utilisation de la transformation de fusion binaire ou n-aire, la spécification des paramètres est très utile pour que la transformation soit possible. Les paramètres représentent les noms des composants à fusionner. Dans le cas d'une fusion binaire, l'utilisateur peut spécifier deux paramètres pour chaque transformation. Dans le cas d'une fusion n-aire, l'utilisateur peut spécifier plus de deux paramères. Dans le cas où les paramètres ne sont pas spécifiés, tous les composants seront fusionnés.

Enfin, l'utilisateur peut lancer plus d'une transformation en utilisant la commande −−**next** suivi par la deuxième transformation et ainsi de suite.

Le nombre de ligne de code pour l'ensemble de transformations BIP vers BIP est environ 7000 lignes de code.

8.1.3 Génération

Après le lancement de la traduction d'AADL vers BIP, nous générons plusieurs fichiers automatiquement :

- Une représentation textuelle et XML du BIP temporisé.
- Une représentation textuelle et XML du BIP non-temporisé.
- Un système exécutable.
- Des fichier C/C++ à l'aide du générateur du code fourni par la chaîne d'outils BIP.

La génération de ces derniers ne nécessitent aucune intervention de l'utilisateur. Cela facilite largement le travail des utilisateurs, la réutilisation des fichiers BIP générés, et aide dans le cycle de développement d'une application.

8.1.4 Simulateur & Débogueur

ADIS est un logiciel de simulation du comportement d'architectures systèmes décrites avec le langage AADL après sa traduction vers BIP. Le nom de cet outil signifie d'ailleurs **A***rchitecture* **D***escription* **I***nteraction* **S***imulation*. Cet outil permet d'évaluer et d'analyser le comportement d'un système au

cours de l'exécution, par exemple en laissant le choix aux utilisateurs de choisir l'interaction à exécuter ou de laisser le choix au simulateur, choisir le composant ou le connecteur pour accéder à ses données, savoir le statut d'un port et l'état d'un composant.

ADIS se présente comme un plug-in Eclipse. Ce choix technique assure une simulation interactive et permet de déboguer le programme au cours de l'exécution.

Le déroulement d'une simulation se fait en plusieurs phases :

1. Analyse syntaxique et sémantique de la description AADL : cette phase est fournie directement par le plugin OSATE. Elle consiste à vérifier que la description fournie en entrée est correcte, et signaler les éventuelles erreurs à l'utilisateur, puis à construire une représentation interne de l'architecture utile aux phases suivantes.
2. Instanciation du modèle : les éléments de l'architecture décrite sont instanciés dans un fichier XML en utilisant l'outil OSATE.
3. Lancement de la simulation : dès la sélection du modèle instancié, un bouton dans le menu d'Eclipse est activé pour permettre le lancement de la simulation. Cette étape se compose de : la traduction du modèle AADL vers un modèle BIP temporisé, la transformation de ce dernier vers un modèle BIP non-temporisé, la génération du code C/C++ et la création d'un exécutable.
4. Simuler et déboguer : La fenêtre de simulation s'ouvre directement dès le lancement, qui permettra de simuler et de déboguer le comportement de l'architecture.

La figure 8.3 représente un exemple de simulation d'un modèle AADL traduit en BIP. ADIS contient le bouton *Next* pour effectuer l'interaction suivante, *Display* pour imprimer le statut du port et de l'état (*state*), un menu déroulant pour choisir l'interaction à effectuer ou laisser le choix aléatoire, un menu déroulant pour choisir le connecteur ou le composant pour accéder à ses données à l'aide du bouton *Goto*, et un menu déroulant pour choisir le port ou l'état. Par exemple, l'état *LOADED* égal à *true* signifie que cet état est actif à ce moment de l'exécution.

8.1.5 Vérification formelle de la structure des applications

La génération de code BIP à partir d'une description AADL a pour objectif la production d'un système exécutable, et la production d'une représentation formelle qui doit correspondre à un objectif de vérification défini. 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 la période des threads.

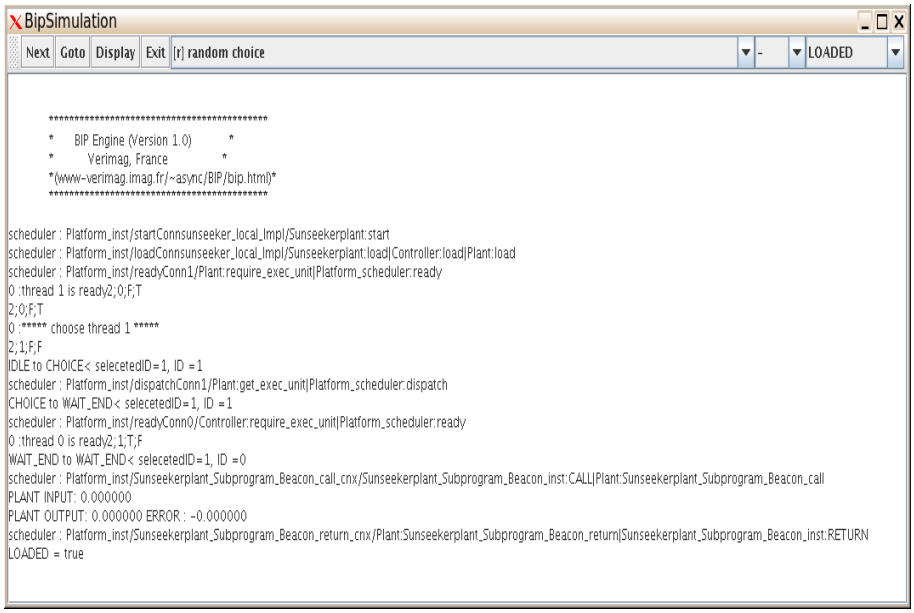


FIG. 8.3 – Exemple de simulation

8.1.5.1 Les outils proposés

Nous proposons les outils de vérification BIP pour la vérification et l'analyses des modèles AADL. La chaîne d'outils de BIP permet l'exploration exhaustive de l'espace d'états du système, la détection des blocages potentiels et la vérification de certaines propriétés dans les modèles. BIP permet de mettre en place des systèmes robustes et sûrs en produisant un contrôleur d'exécution correct par construction, et en fournissant un modèle formel qui peut être utilisé avec divers outils de vérification et de validation. Cela joue en faveur de l'utilisation de telles méthodes dans le cycle de développement. Une fois le modèle BIP généré à partir d'un modèle AADL, deux techniques de vérification sont appliquées :

Absence de blocage (*Deadlock freedom*) : Il s'agit d'une propriété essentielle car elle caractérise la capacité du système à exécuter des activités au cours de sa durée de vie. Les outils de BIP permettent la détection de blocages potentiels par analyse statique des connecteurs dans le modèle BIP [GS03]. Cette méthode consiste à utiliser les invariants générés automatiquement pour prouver la non-satisfiabilité des prédicats caractérisant les blocages globaux [BBSN08]. Deux types d'invariants sont générés : (i) les invariants de composants qui sur-approximent l'ensemble d'états atteignables de ces derniers, et (ii) les invariants d'interaction qui sont les contraintes sur les états des composants liées aux interactions. La méthode est implémentée dans l'outil D-Finder [BBNS09].

Model checking : Cela consiste à construire un modèle fini du système

analysé et à vérifier les propriétés souhaitées de ce modèle. La vérification demande une exploration complète ou partielle du modèle. Les avantages principaux de cette technique sont : la possibilité de rendre l'exploration du modèle automatique et de faciliter la production de contre-exemples, lorsque la propriété est violée. Dans nos travaux de thèse nous avons utilisé deux techniques de model checking : les observateurs pour exprimer et vérifier des exigences et l'outil Aldebaran [BFKM97].

Aldebaran permet de vérifier les systèmes communicants représentés comme des systèmes de transitions étiquetées. Il permet de minimiser et de comparer des systèmes de transitions étiquetées par rapport aux relations équivalences. Aldebaran prend en entrée les LTS générés par l'exploration exhaustive du moteur d'exécution BIP, et vérifie l'absence de blocage et d'autres propriétés temporelles.

8.1.5.2 Propriétés

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 génériques, telles que :

- l'assemblage des composants n'engendre pas d'interblocage,
- l'architecture n'engendre pas systématiquement de débordement de file d'attente, surtout dans le cas d'utilisation des ports de type *event data*,
- le temps d'exécution et la période des threads doivent être respectés.

En plus, nous pouvons utiliser des observateurs pour vérifier le comportement du système ou certaines propriétés.

8.2 Études de cas

Plusieurs études de cas ont été testés pour prouver le bon fonctionnement de notre processus de production. Ces études de cas ont été choisies pour valider leur adéquation aux problématiques posées et pour valider les solutions que nous avons réalisées.

8.2.1 Flight Computer

Nous présentons une étude de cas pour illustrer notre approche. Nous décrivons un système qui possède trois capteurs. Un capteur met à jour le *climb-rate* (taux de monté) du système toutes les 20 ms. Un autre capteur donne l'angle d'attaque (AOA angle-of-attack) – l'angle entre l'axe longitudinal du fuselage, et la direction de airflow – toutes les 20 ms. Le troisième capteur déclenche une interruption en cas de panne de moteur. Le système

dispose d'un sous-système de train d'atterrissage. Ce sous-système peut envoyer un événement, ce qui provoque le fait de sortir ou de rentrer le train d'atterrissage.

Modèle AADL

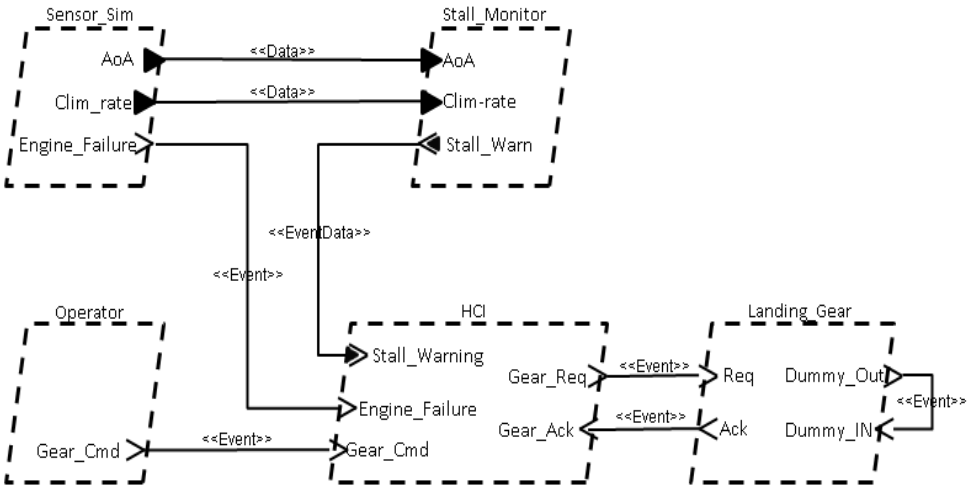


FIG. 8.4 – Architecture du Flight Computer en AADL

La représentation graphique du système *Flight Computer* est montrée dans la figure 8.4. *Flight Computer* contient un thread appelé Sensor_Sim qui envoie périodiquement des données de type entier à travers les ports AoA et Climb_Rate et un événement en cas de panne du moteur. Il a également un thread appelé Stall_Monitor, qui est périodique et surveille les données reçues par les capteurs AoA et Climb_Rate. Stall_Monitor envoie un événement si certaines conditions sont remplies. Le thread Operator simule le pilote, il envoie périodiquement un événement Gear_Cmd. Le thread Landing_Gear simule le sous-système de train d'atterrissage, c'est un thread sporadique avec une période de 3 secondes. Il reçoit un événement pour démarrer l'opération *landing gear*. Le thread HCI représente l'interface homme-machine, c'est un thread sporadique avec une période de 10ms. Il reçoit : Stall_Warning comme un événement de donnée de type entier, un événement Engine_Failure, et un événement à travers le port Gear_Ack (Ack : acknowledgement) du thread Landing_Gear. Le thread HCI envoie aussi un événement à travers le port Gear_Req.

Le listing 8.1 représente le processus AADL qui contient des sous composants et des connexions. Le reste du code AADL est décrit sur la page Web [AADa].

```
process implementation Mgmt_T.RS
  subcomponents
    Sensor_Sim: thread Sensor_Sim_T.RS;
    Stall_Monitor: thread Stall_Monitor_T.RS;
    HCI: thread HCI_T.RS;
    Landing_Gear: thread Landing_Gear_T.RS;
    Operator: thread Operator_T.RS;
  connections
    DataConnection1:
        data port Sensor_Sim.AoA -> Stall_Monitor.AoA;
    DataConnection2:
        data port Sensor_Sim.Climb_Rate -> Stall_Monitor.Climb_Rate;
    EventConnection1:
        event port Sensor_Sim.Engine_Failure -> HCI.Engine_Failure;
    EventDataConnection1:
        event data port Stall_Monitor.Stall_Warn -> HCI.Stall_Warning;
    EventConnection2:
        event port HCI.Gear_Req -> Landing_Gear.Req;
    EventConnection3:
        event port Landing_Gear.Ack -> HCI.Gear_Ack;
    EventConnection4:
        event port Landing_Gear.Dummy_Out -> Landing_Gear.Dummy_In;
    EventConnection5:
        event port Operator.Gear_Cmd -> HCI.Gear_Cmd;
end Mgmt_T.RS;
```

Listing 8.1 – Les sous composants du processus

Modèle BIP

Le modèle AADL du *Flight Computer* est transformé automatiquement en BIP, en utilisant notre outil de traduction d'AADL vers BIP. La figure 8.5 représente les composants atomiques et les connexions dans le modèle BIP obtenu après la transformation. Dans cette figure nous n'avons pas représenté les connexions entre les processus, les threads, et le scheduler.

Au cours de la traduction nous avons gardé la structure de la composition. La figure 8.5 contient un composant atomique nommé *Dummy_In_Out*. Ce composant a pour rôle d'assurer la réactivation périodique du thread *landing_Gear*.

Évaluation

Une fois que le modèle BIP a été généré, deux techniques de model checking ont été appliquées :

- La première technique de vérification utilisée est la détection du blocage en utilisant l'outil Aldebaran. L'exploration exhaustive par le moteur d'exploration BIP génère les LTS qui peuvent être analysés par le model checking. Nous avons vérifié que le modèle est deadlock-free.
- La seconde technique de vérification utilise des observateurs BIP pour exprimer et vérifier les exigences. Nous avons appliqué cette technique pour vérifier deux propriétés essentielles :

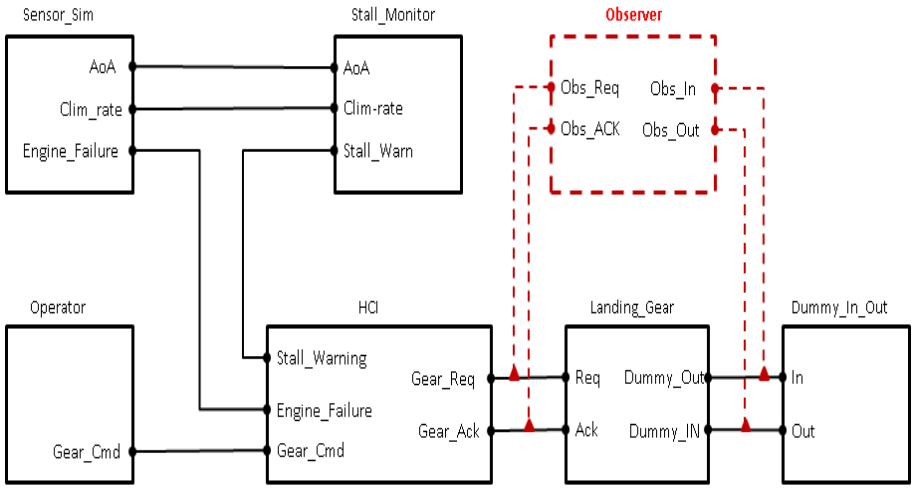


FIG. 8.5 – Le modèle BIP du Flight Computer (comportant le composant observateur en pointillés)

- Vérification de la date limite des threads en utilisant un observateur qui garde une trace du temps d'exécution du thread. Si le temps d'exécution de chaque thread est supérieur à sa date limite, l'observateur se déplace vers un état d'erreur.
- Vérification des synchronisations entre les composants : *landing_Gear* est activé de manière sporadique par le thread HCI à travers le port Req. Lorsqu'il est activé, il renvoie un événement à travers le port ACK. Il peut aussi se réactiver lui-même par la réception d'un événement du composant *Dummy_In_Out*. Cette propriété peut être vérifiée par un observateur qui surveille les interactions entre les composants *landing_Gear*, *HCI* et *Dummy_In_Out*. La figure 8.5 montre l'intégration du composant dans le système généré. La figure 8.6 montre le comportement du composant *Observer*.

8.2.2 Système de Contrôle

Nous allons utiliser une étude de cas AADL pour vérifier la faisabilité de notre méthodologie appliquée sur les communications de données, qui est décrite dans la section 6.3 du chapitre 6.

Le Système de Contrôle est une application composée de trois capteurs, un composant pour fusionner les données reçues des trois capteurs, et d'un autre composant pour actualiser et afficher les données du fusionneur. Cette étude de cas est inspirée de TELECOM ParisTech [1].

[1] http ://aadl.enst.fr/arc/doc/

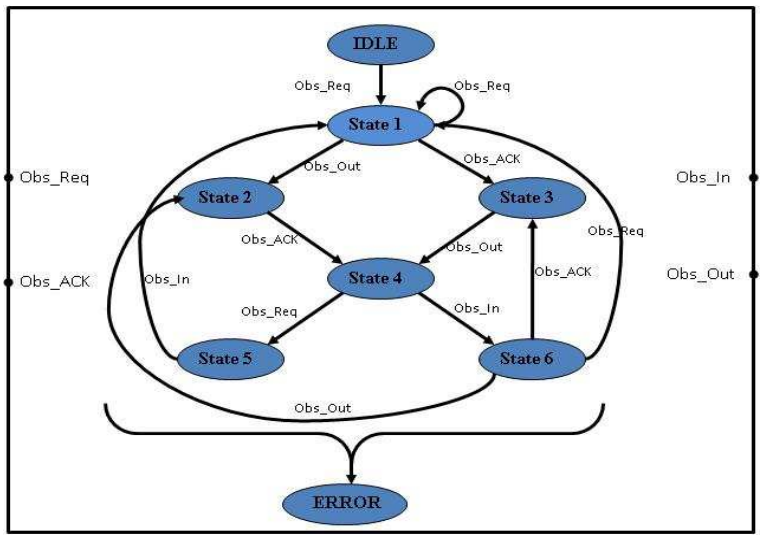


FIG. 8.6 – Le comportement du composant observateur

Modèle AADL

La figure 8.7 montre l'architecture de l'application modélisée en AADL. Le système de contrôle contient trois threads périodiques qui jouent le rôle du capteur avec une période égale à $20ms$, un thread appelé *Data_Fusion* avec une période de $30ms$, et un thread appelé *Actuator* avec une période de $20ms$. Le comportement des threads est décrit en langage C/C++.

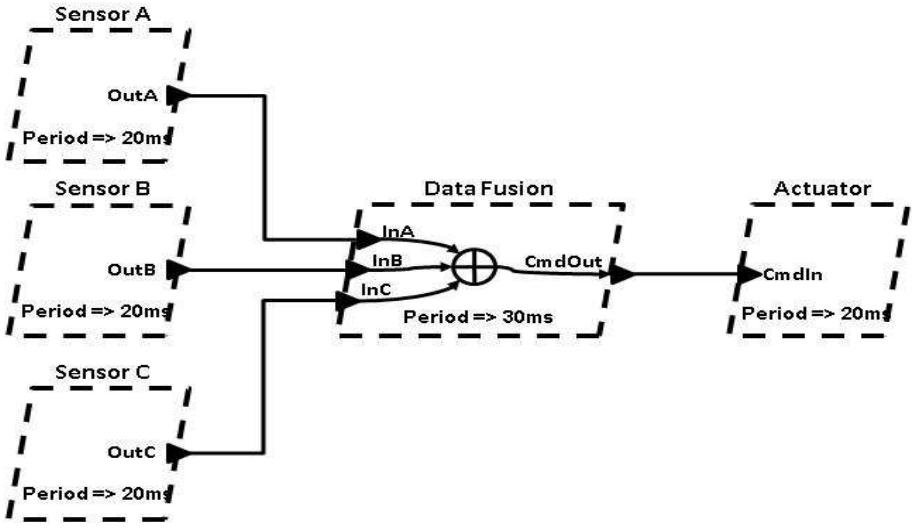


FIG. 8.7 – Système de contrôle

- Les threads *Sensor_A*, *Sensor_B* et *Sensor_C* incrémentent péri-

odiquement une variable entière et les envoient au thread *Data_Fusion* à travers les ports de données. Le listing 8.2 montre la représentation textuelle du thread *Sensor_A*. Ce dernier appelle une fonction décrite en langage C appelée *sensorA(&outp)*.

- Le thread *Data_Fusion* lit les valeurs reçues sur ses ports de données. Il les ajoute, et il envoie la somme périodiquement au thread *Actuator*. Le thread *Data_Fusion* contrôle ses entrées. Lorsque les variables reçues à partir des threads *Sensor_A*, *Sensor_B* et *Sensor_C* ne sont pas égales, le thread affiche un message d'erreur. Sinon, il continue l'exécution.
- Le thread *Actuator* lit et imprime la valeur de son port de données à chaque expédition. Le Listing 8.3 montre le code AADL du thread *Actuator*. Lors de l'exécution le thread appelle une fonction appelée *actuator(cmd)* à partir du fichier *actuator.c*.

```
thread Sensor_Thread
features
        outp : out data port Behavior::float;
properties
        Dispatch_Protocol => Periodic;
        Period => 2ms;
end Sensor_Thread;

thread implementation Sensor_Thread.A
properties
        Compute_Entrypoint => "sensorA(&outp)";
        source_text  => ("sensorA.c");
end Sensor_Thread.A;
```

Listing 8.2 – La description du thread Sensor_A en AADL

```
thread Actuator
   features
      cmd : in data port Behavior::float;
end Actuator;

thread implementation Actuator.Impl
   properties
      Dispatch_Protocol => Periodic;
      Period => 2ms;
      Compute_Entrypoint => "actuator(cmd)";
      source_text  => ("actuator.c");
end Actuator.Impl;
```

Listing 8.3 – La description du thread Actuator en AADL

Le processus AADL contient tous les threads ainsi que la description des connexions de type données entre ces threads, comme le montre le listing 8.4.

```
process implementation Partition.Impl
subcomponents
        Sensor_A : thread Sensor_Thread.A;
        Sensor_B : thread Sensor_Thread.B;
        Sensor_C : thread Sensor_Thread.C;
        Data_Fusion : thread Fusion_Thread.Impl;
        Act : thread Actuator.Impl;

connections
        data port Sensor_A.outp -> Data_Fusion.inpA;
        data port Sensor_B.outp -> Data_Fusion.inpB;
        data port Sensor_C.outp -> Data_Fusion.inpC;
        data port Data_Fusion.cmdout -> Act.cmd;
end Partition.Impl;
```

Listing 8.4 – La description du processus Actuator

Modèle BIP

Le modèle AADL du système de contrôle est transformé en BIP automatiquement en utilisant l'extension de notre outil de traduction de AADL vers BIP, qui assure une communication du flux de donnée déterministe entre deux threads. Nous avons généré deux modèles BIP à partir de notre étude de cas pour pouvoir étudier l'impact causé par le non-déterminisme comme décrit dans la section 6.3. Au cours de la génération nous avons gardé la même architecture et le même nombre de composants sauf dans le cas où on ajoute les composants DBCI pour assurer le déterminisme.

Le listing 8.5 représente le traduction du thread *Sensor_A* décrit dans le listing 8.2.

```
atomic type Sensor_Thread_A(int ID)
  data int threadID=ID
  timed data int t=0
  timed data int clock=0
  data int period=2
  data int deadline=2
  data bool inEventOccurs=false
  data float outp
  data bool outp_writed
  export port aadllib.intPort get_exec_unit(threadID)
  export port aadllib.Port load()
  export port aadllib.Port stop()
  export port aadllib.Port abort()
  export port aadllib.intPort require_exec_unit(threadID)
  export port aadllib.intPort release_exec_unit(threadID)
  port aadllib.Port BIPinternalPort0()
  port aadllib.Port BIPinternalPort1()
```

```
export port floatPort outp_port(outp)
port aadllib.Port BIPinternalPort2()
port aadllib.Port BIPinternalPort3()
export port aadllib.Port tick()
place HALTED
place INIT
place SUSPENDED
place READY
place FINISH
place COMPUTE
place ERROR
place OUTPUTS
place time
initial to HALTED,time
on load
  from HALTED to INIT
  eager
  do {
    t = 0;
  }
on require_exec_unit
  from INIT to READY
  eager
on require_exec_unit
  from SUSPENDED to READY
  provided clock>=period
  eager clock in (period,_)
  do {
    clock = 0;
  }
on get_exec_unit
  from READY to COMPUTE
  do {
    t = 0;
  }
on BIPinternalPort0
  from SUSPENDED to ERROR
  provided clock>deadline
  do printf("Sensor_Thread_A: Error deadline missed \n");
on BIPinternalPort1
  from COMPUTE to ERROR
  provided clock>deadline
  do printf("Sensor_Thread_A: Error deadline missed \n");
on release_exec_unit
  from FINISH to SUSPENDED
  eager
  do {
    inEventOccurs = false;
    outp_writed = false;
  }
on BIPinternalPort2
  from OUTPUTS to FINISH
  provided true&&outp_writed
  do {
  }
on BIPinternalPort3
  from COMPUTE to OUTPUTS
  do  {#
     sensorA(&outp);
    #}
on outp_port
  from OUTPUTS to OUTPUTS
```

```
        provided !outp_writed
        eager
        do outp_writed = true;
      on tick
        from time to time
        do {
          t = t+1;
          clock = clock+1;
        }
    end
```

Listing 8.5 – La description du thread Sensor_A en BIP

Analyses

Nous exécutons le modèle BIP. Le thread *Data_Fusion* imprime un message d'erreur à chaque fois qu'il rencontre une situation inattendue : les valeurs reçues de *sensorA*, *sensorB*, et *sensorC* ne sont pas égales.

Ces erreurs sont dues au non-déterminisme des communications de flux de données entre les threads. Afin de résoudre ce problème, nous générons le modèle BIP en utilisant le composant DBCI. Cette fois, les erreurs à l'exécution sont totalement éliminées.

Les figures 8.9 et 8.8, décrivent les données reçues par le thread appelé *Data_Fusion* de *sensorA*, *sensorB*, et *sensorC* chaque 30ms. On voit clairement la différence entre ces deux figures. La figure 8.8 montre un comportement correct lorsque nous utilisons un composant DBCI : les trois entrées du thread *Data_Fusion* ont toujours les mêmes valeurs. Par contre, la figure 8.9 montre un comportement incorrect : les trois entrées du thread *Data_Fusion* ont des valeurs différentes dans plusieurs périodes.

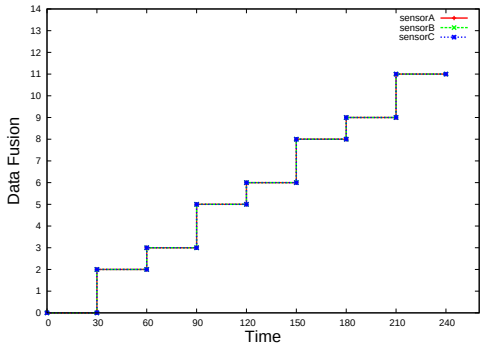


FIG. 8.8 – Avec DBCI

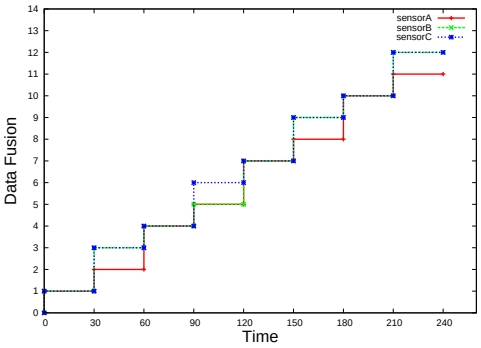


FIG. 8.9 – Sans DBCI

8.2.3 Multi-Platform Cooperation (MPC)

Nous allons utiliser cette étude de cas pour vérifier la faisabilité de notre extension pour la génération des applications distribuées, qui est décrite dans la section 6.4 du chapitre 6.

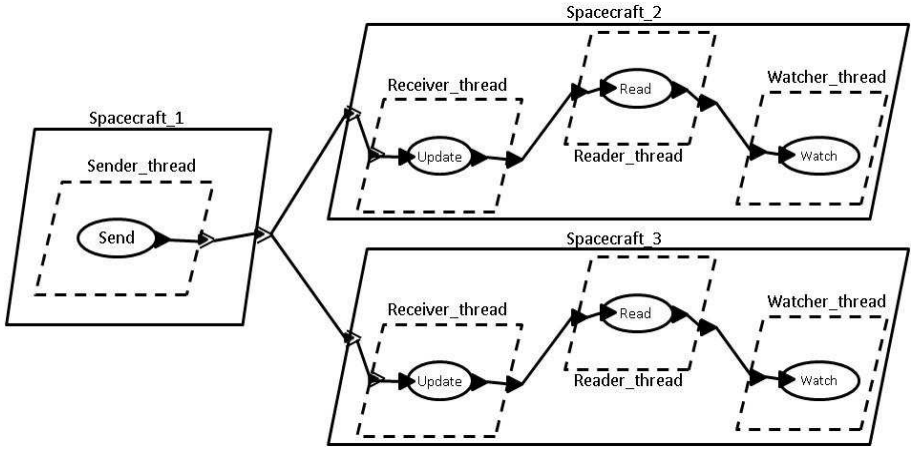


FIG. 8.10 – Vue logicielle de MPC

Cette étude de cas a été inspirée de J. Hugues [HZPK08]. La figure 8.10 montre la partie logicielle de notre étude de cas. Ce modèle contient trois partitions, chacune étant un spacecraft avec son propre rôle :

- *Spacecraft_1* est un leader spacecraft contenant un thread périodique, qui transmet sa position à *Spacecraft_2* et *Spacecraft_3*.
- *Spacecraft_2* et *Spacecraft_3* reçoivent les positions transmises par *Spacecraft_1* avec un thread sporadique (*Receiver_thread*), qui met à jour leur propre position et l'envoie au thread *Reader_thread*. Le thread *Reader_thread* lit périodiquement la valeur de la position à partir du thread *Receiver_thread* et il la stocke dans un objet protégé. Le troisième thread "watches and reports" tous les éléments à cette position (par exemple, l'observation de la terre).

Ce modèle rassemble les éléments typiques des systèmes distribués, avec un ensemble de tâches périodiques consacrées au traitement des données entrantes (*Watcher_thread*), au stockage des données (*Reader_thread*), et à l'échanger des données (*Receiver_thread*). Ces entités travaillent à des rythmes différents et doivent toutes respecter les délais afin que le thread *Watcher_thread* puisse traiter toutes les données d'observation en temps voulu.

Le point de vue logiciel représente la façon dont le traitement des données est distribué sur les différentes threads et de rassemblées ces derniers dans le processus AADL pour former des partitions. La prochaine étape est d'associé ce point de vue logiciel sur une vue matérielle, afin que les ressources du processeur puissent être associées à chaque partition.

La figure 8.11 est une représentation graphique du système global en AADL, qui représente le déploiement de la vue logicielle et de la vue matérielle. Il montre aussi l'architecture globale de l'application, c'est-à-dire le nombre de partitions et leurs associations aux composants matériels. Il indique com-

ment la communication entre les partitions se produit en utilisant les bus.

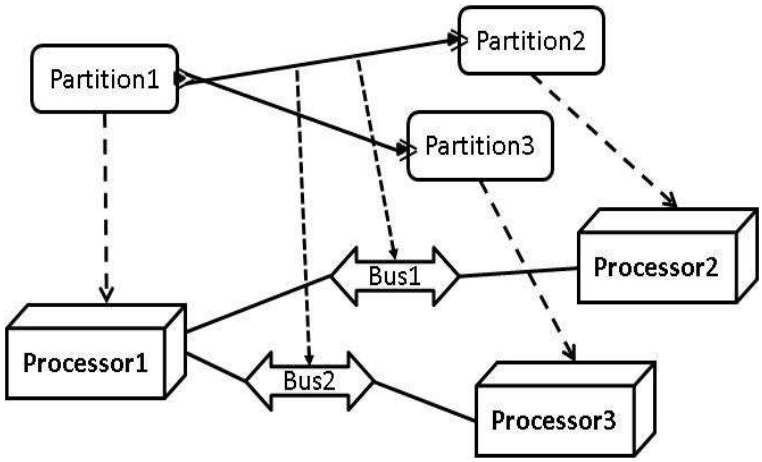


FIG. 8.11 – Vue matérielle de MPC

8.2.3.1 Modèle AADL

L'étude de cas MPC est construite par la création de composants logiciels et l'association de ces derniers à une architecture matérielle. La flexibilité de AADL nous permet de définir partiellement les composants et de les utiliser dans d'autres composants. Ceci est très utile pendant les étapes de prototypage, où chaque détail du système n'est pas encore clair. Les détails peuvent être ajoutés à ces composants soit au moyen des propriétés AADL, soit par extension du composant, sans avoir à redéfinir tous les autres composants.

Types de données Les composants de données AADL modélisent les messages qui sont échangés entre les partitions d'une application distribuée ou à l'intérieur d'une partition.

```
data Record_Type
end Record_Type;

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

Listing 8.6 – Déclaration du type de données

Pour exprimer la nature d'un type de données, nous utilisons les composants de données comme illustré dans le listing 8.6.

Sous-programmes Dans cette étude de cas, les sous-programmes encapsulent les aspects du comportement d'une application. Ils sont modélisés à l'aide du composant sous-programme AADL. La mise en œuvre d'un sous-programme est entièrement décrite par l'utilisateur en indiquant le fichier source ou les bibliothèques qui contiennent l'implantation. Le listing 8.7 montre un des sous-programmes appelé *Update* de notre étude cas. Il appelle une fonction nommé *Update* décrite dans un fichier extérieur *mpc.cpp*.

```
subprogram Update
  features
    Data_Sink: in parameter Record_Type;
    Protected_Local: out parameter Record_Type;
end Update;

subprogram implementation Update.impl
  properties
    Source_Language => C;
    Source_Name => "Update";
    Source_Text => "mpc.cpp";
end Update.impl;
```

Listing 8.7 – Le sous-programme Update

Le comportement des sous-programmes AADL peut être modélisé de plusieurs façons. L'outil de transformation de AADL vers BIP permet trois types d'implantation des sous-programmes : en utilisant une référence à un fichier (C/C++) contenant la description comportementale du sous-programme, ou en utilisant le comportement spécifié par l'annexe, ou en utilisant des séquences d'appels à d'autres sous-programmes. Dans cette étude, le comportement est décrit dans le fichier *mpc.cpp* comme le montre le listing 8.7. Tout cela donne une grande flexibilité aux utilisateurs lors de la modélisation et du prototypage de son système.

Threads Les threads représentent la partie active d'une application distribuée. Une partition (ou processus) doit contenir au moins un thread. L'interface du thread est constituée de ports. Dans cette étude de cas, nous utilisons deux types de threads :

- les threads périodiques : Le listing 8.8 montre le modèle AADL du thread périodique nommé *Sender_thread* qui se trouve dans la Partition1. Ce thread envoie des données de type *Record_Type* à travers le port *Data_Source*. Le protocole de l'expédition du thread et sa période sont spécifiées en utilisant les propriétés du standard AADL.

Dans l'implantation du thread, nous décrivons son comportement en spécifiant l'appel au sous-programme qui modélise son activité.

- les threads sporadiques : Le modèle AADL du thread sporadique nommé *Receiver_thread* est présent dans les partitions *Spacecraft_2* et *Spacecraft_3*. Il est déclenché par la réception de la position envoyée par le thread *Sender_thread* de la partition *Spacecraft_1* .

```
thread Sender_Thread
features
  Data_Source : out  event data port Record_Type;
  Data_activate : in event data port Record_Type;
properties
  Dispatch_Protocol => Periodic;
  Period            => 100 Ms;
end Sender_Thread;

thread implementation Sender_Thread.Impl
calls Main: {
  Wrapper : subprogram Sender_Thread_Wrapper.impl;
 };
connections
  parameter Wrapper.Data_Source -> Data_Source;
  parameter Data_activate -> Wrapper.Data_activate;
end Sender_Thread.Impl;
```

Listing 8.8 – Description du thread *sender_thread*

Processus Les processus sont les composants AADL utilisés pour modéliser les partitions d'applications distribuées. Le listing 8.9 montre le modèle AADL du processus appelée *Sender_Process*, qui contient un seul thread comme sous composant.

```
process Sender_Process
features
  Data_Source : out event data port Record_Type;
end Sender_Process;

process implementation Sender_Process.Impl
subcomponents
  Sender : thread Sender_Thread.Impl;
connections
  event data port Sender.Data_Source -> Data_Source;
end Sender_Process.Impl;
```

Listing 8.9 – Description du Processus : Spacecraft_1

	AADL	BIP		
		Spacecraft_1	Spacecraft_2	Spacecraft_3
Components	20	4	8	8
Connectors	21	8	18	18
Lines of code	350	250	600	600

FIG. 8.12 – Comparaison entre AADL & BIP

8.2.3.2 Modèle BIP

La génération du code BIP nous permet de prototyper l'étude de cas MPC en une application distribuée utilisant le protocole de communication décrit dans le chapitre 6 entre chaque partition. Cette génération nous a permis d'analyser l'étude de cas sur des ordinateurs (PC) afin de la déboguer et de l'évaluer.

La séparation entre le logiciel et le matériel en AADL permet au programmeur de modéliser toutes les parties logicielles de son application et de les tester sur une plate-forme PC.

Nous générons pour chaque partition AADL associée à un processeur un fichier qui correspond à sa description en BIP, et pour chaque connexion associée au bus un protocole de communication réseau.

La figure 8.12 résume la taille des lignes de code, le nombre de composants et de connecteurs dans AADL et respectivement le code BIP pour l'étude de cas MPC. Nous avons généré trois fichiers, c'est à dire un pour chaque *Spacecraft* contenant sa description correspondante en BIP.

Types de données Le composant de données décrit en AADL dans le listing 8.6 représente une structure de données, qui est composée de trois entiers X,Y et Z. La traduction de ce composant en BIP est représentée comme une structure de données décrite en langage C, comme le montre le listing 8.10.

```
struct Record_Type.Impl {
   int X;
   int Y;
   int Z;
};
```

Listing 8.10 – Déclaration d'une structure de données en BIP

Codage/Décodage de données La structure utilisée doit être codée avant de passer sur le réseau pour être décodée après son récupération. Pour cette étape nous avons utilisé des fonctions pour le codage/décodage des données.

Envoi/Réception de données à travers le réseau

- côté client (*Spacecraft_1*) : Pour que les données soient envoyées du *Spacecraft_1* vers *Spacecraft_2* à travers le réseau, il faut que *Spacecraft_1* connaisse l'adresse IP du serveur et le numéro du port. L'envoi d'une donnée est réalisé par un appel à la fonction suivante :
  $send(sock, buffer, strlen(buffer), 0)$;
- côté serveur (*Spacecraft_2*) : *Spacecraft_2* doit savoir le nom du port où il va recevoir les données. La réception d'un message est réalisée par un appel à la fonction suivante :
  $recv(csock, buffer, sizeofbuffer - 1, 0)$;

Un numéro de port sert donc à différencier plusieurs voies de communication aboutissant à une même machine afin que les différentes applications qui s'y exécutent ne mélangent pas leurs données les unes avec les autres.

Communication de données

- Les communications s'exécutant sur un ordinateur sont gérées par le moteur BIP.
- Les communications s'exécutant entre plusieurs ordinateurs reliés par un réseau, sont gérées par le protocole de communication réseau.

8.2.3.3 Analyse

La traduction de AADL vers BIP permet de donner une sémantique formelle à l'architecture avant d'effectuer la génération de code C/C++.

Nous compilons les partitions BIP et nous générons un modèle exécutable pour chaque partition. Puis, nous mettons chaque exécutable dans une machine qui contient le moteur BIP. Premièrement, nous lançons l'exécutable récepteur, puis l'exécutable expéditeur. Lorsque le protocole de communication réseau est initialisé entre l'émetteur et le récepteur, l'échange de données peut démarrer.

Une fois que le modèle exécutable est lancé, la simulation interactive et le débogage sont utiles pour comprendre le fonctionnement de cette application distribuée. Cela nous a permis de vérifier étape par étape, l'interaction entre les composants, de connaître l'état ou le port actif, et de voir la valeur des données reçues/envoyées. En plus, nous utilisons des observateurs qui se déplacent à un état d'erreur si la durée d'exécution d'un thread est supérieure à sa limite et effectuent les analyses décrites dans la section 8.1.5.

Les figures 8.13 et 8.14 montrent un fragment de la simulation de *Spacecraft_1* et de *Spacecraft_2* sur deux machines différentes.

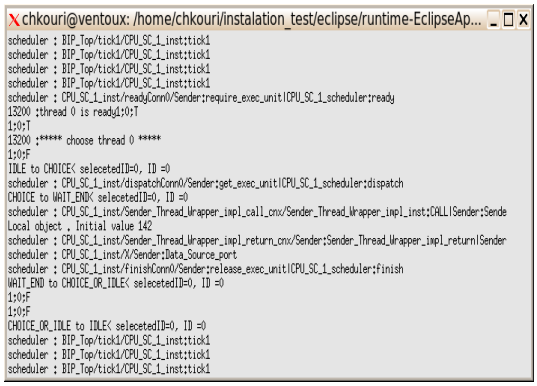


FIG. 8.13 – Spacecraft_1

FIG. 8.14 – Spacecraft_2

8.3 Conclusion

Dans ce chapitre, nous avons présenté la chaîne d'outils développée au cours de notre thèse. Nous avons intégré toute la chaîne sous forme de plugins à Eclipse. Cette intégration facilite le travail des utilisateurs, et ne leur demande aucun effort après la modélisation de l'application jusqu'à la simulation.

Nous avons utilisé des outils de vérification pour la détection des blocages potentiels, l'exploration exhaustive de l'espace des états du système et la vérification de certaines propriétés dans les modèles BIP obtenus.

Nous avons utilisé plusieurs études de cas en AADL pour tester et prouver le bon fonctionnement de notre processus de production. Ces études de cas ont été choisies pour valider leur adéquation aux problématiques posées et pour valider les solutions que nous avons réalisés. Nous avons effectué des analyses de vérification sur les modèles BIP obtenus pour tester les applications modélisés en AADL. Ensuite, nous avons produit du code et exécuté nos applications avec succès.

Ceci nous permet de dire que nous avons accompli les objectifs de nos travaux de thèse qui consistent à construire automatiquement des transformateurs de modèles et un simulateur dédié aux besoins des applications embarquées.

Chapitre 9

Conclusions et Perspectives

L'objectif de nos travaux consistait à étudier l'utilisation du langage AADL pour rassembler tous les aspects de la modélisation d'une application afin de fournir une sémantique formelle au langage et de produire une application vérifiable formellement respectant les différentes contraintes d'exécution. Nous avons montré que cette approche était viable et permettait de produire effectivement des applications vérifiées.

Nous rappelons les réalisations du travail effectué au cours de la thèse. Ensuite, nous présentons les perspectives pour poursuivre nos travaux selon différents axes.

Réalisations

Afin d'exploiter de façon efficace les caractéristiques d'une application, il est nécessaire de recourir à 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, validation, etc.

Nous avons choisi AADL, un langage de description d'architecture, pour décrire les systèmes temps-réel embarqués. Nous l'avons utilisé 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, etc.). Ce langage 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 la partie logiciel (threads, sous-programmes) et son environnement d'exécution (processeurs, bus).

Les descriptions comportementales des éléments de l'application sont exprimées dans un paradigme différent d'AADL. Nous avons utilisé au cours

de nos travaux deux type de descriptions comportementales des éléments. Il s'agit d'un langage de programmation C/C++ et de l'annexe comportementale qui utilise un système de transition (un automate étendu). Ces descriptions comportementales sont associées aux composants AADL.

Le chapitre 3 décrit les aspects du langage AADL utiles pour notre travail de thèse. Dans le chapitre 4 nous avons présenté le langage BIP, qui permet de mettre en place des systèmes robustes et sûrs, en produisant un contrôleur d'exécution correct par construction, et en fournissant un modèle formel qui peut être utilisé avec divers outils de vérification et de validation. Nous avons ensuite proposé un processus automatique de production décrit dans le chapitre 5. Nous avons établi des règles de traduction pour produire une application en BIP à partir des constructions AADL.

Sémantique formelle : Contrairement à d'autres méthodologies de traduction d'AADL qui génèrent directement du code C/C++, Java ou ADA. Nous avons ajouté une étape très importante, la traduction d'AADL vers BIP avant la génération du code. Le but de cette étape est de fournir à AADL une sémantique formelle définie en termes de systèmes de transitions étiquetés.

Nous avons présenté les extensions de la syntaxe AADL et de la traduction vers BIP que nous avons proposé pour faciliter la description de l'assemblage des applications et des communications. Nous avons décrit l'utilité de l'instanciation des sous-programmes, de l'intégration du comportement dans les devices, de la modélisation des communications des données et de la modélisation et la génération d'une application distribuée.

Nous avons présenté les techniques d'optimisation qui consiste à fusionner les composants BIP pour réduire le surcoût induit par l'interprétation des connecteurs et des priorités, en préservant le comportement des composants. Les modèles optimisés sont obtenus par substitution d'un ensemble de composants par un composant produit équivalent. Ces techniques d'optimisation permettent de réduire le temps d'exécution, la génération de code et d'éviter l'explosion des états du système.

Validation : Nous avons adapté et utilisé la chaîne d'outils de BIP, qui permet l'exploration exhaustive de l'espace des états du système, la détection des blocages potentiels et la vérification de certaines propriétés dans les modèles BIP obtenus. Cela jouent en faveur de l'utilisation de telles méthodes dans le cycle de développement.

Nous pouvons simuler ou déboguer les modèles en créant un système exécutable. Nous pouvons utiliser un simulateur interactif et un débogueur pour vérifier chaque interaction, étape par étape, et ainsi connaître l'état ou les interactions qui sont exécutables. Ces analyses permettent d'évaluer la fiabilité du système, d'affiner et de corriger son comportement.

Génération de code : Nous avons décrit notre méthodologie de production automatique d'applications formellement vérifiées. Cette production permet dans un premier temps de fournir une sémantique opérationnelle rigoureuse. Puis de valider, de vérifier et enfin d'analyser les systèmes temps réel embarqués.

Implantation/Expérimentation : Nous avons implanté toute la chaîne d'outils d'écrite dans notre thèse sous forme de plugins Eclipse. Plusieurs études de cas ont été testées durant notre travail de thèse. Il s'agit, pour la plupart, d'exemples de systèmes embarqués temps réel. Ces études de cas ont été choisies pour valider leur adéquation aux problématiques posées et pour valider les solutions que nous avons réalisés.

Perspectives

Nos travaux peuvent être poursuivis pour envisager plusieurs pistes de recherches.

Extension d'AADL

AADL v2 : La nouvelle version du standard fournit quelques évolutions de la syntaxe et de la sémantique des entités existantes, telle que les connexions et les éléments de l'interface. La version v2 du standard AADL propose trois nouvelles catégories de composants : composants abstraits, processeurs virtuels et bus virtuels. Ces deux derniers composants permettent de raffiner la spécification de la partie matérielle de l'application. Nous devons intégrer ces nouveaux éléments dans notre outils de traduction d'AADL vers BIP pour permettre une utilisation durable.

Intégration de la notion de partitionnement : La norme ARINC 653 [ARI] impose de stricts critères de fonctionnement du système d'exploitation, comme le partitionnement temporel. Ce dernier est très important dans le domaine de l'avionique, où on partitionne les ressources pour faire tourner plusieurs applications sur le même processeur. Pour cela, nous voulons intégrer la notion de partitionnement au sein d'AADL. Cette intégration nécessite une étude approfondie de l'impact de ce type de partitionnement sur les systèmes AADL et leurs sémantiques.

Optimisation des modèles AADL

Dans le chapitre 7, nous avons proposé des techniques d'optimisation pour les modèles BIP. Cependant, le modèle AADL lui même devrait être optimisé en fonction des besoins de l'application. C'est à dire que nous pouvons appliquer la technique de la fusion des composants pour les composants

de la même catégorie et du même niveau, comme les threads ou les sous-programmes. Cette technique d'optimisation aura pour but d'optimiser le nombre de tache dans une application et d'optimiser la génération de code.

La production d'intergiciel pour les applications distribuées

La mise en place d'une application distribuée repose en général sur la construction d'une couche applicative particulière (l'intergiciel) pour prendre en charge les communications inter-partitions. L'intergiciel doit pouvoir fournir tous les mécanismes de communication requis par l'application.

Les applications temps réel distribuées doivent respecter un certain nombre de contraintes (temps d'exécution, taille mémoire, ressources disponibles, etc.). Ces contraintes doivent être respectées par tous les composants, et en particulier par l'intergiciel.

Dans le chapitre 6, nous avons proposé un prototype pour la génération des applications distribuées à partir d'AADL qui utilise les sockets comme intergiciel. Nous devons l'améliorer pour prendre en compte plusieurs type de communications et prendre en compte d'autres types contraintes.

Bibliographie

[AADa] *http ://www-verimag.imag.fr/∼async/AADL2BIP.php.*

[AADb] *SAE. Architecture Analysis & Design Language (standard SAE AS5506), September 2004, available at http ://www.sae.org.*

[ABS] *Annex Behavior Specification SAE AS5506.*

[ACM] *ABLE Group. the Acme Architectural Description Language. http ://www.cs.cmu.edu/ acme/.*

[AD94] Rajeev Alur and David L. Dill, *A theory of timed automata*, Theoretical Computer Science **126** (1994), 183–235.

[ADEa] *Ades : Architecture description simulation. http ://www.axlog.fr/aadl/ades_fr.html.*

[ADEb] *Ellidiss. adele : a versatile system architecture graphical editor based on aadl. http ://gforge.enseeiht.fr/projects/adele,.*

[AG97] Robert Allen and David Garlan, *A formal basis for architectural connection*, ACM Transactions on Software Engineering and Methodology (1997).

[AGS02] K. Altisen, G. Gössler, and J. Sifakis, *Scheduler modeling based on the controller synthesis paradigm*, Real-Time Syst. **23** (2002), no. 1/2, 55–84.

[ANT] *ANTLR, http ://www.antlr.org/.*

[ARI] *ARINC653. Airlines electronic engineering committee (aeec), Avionics Application Software Standard Interface (arinc specification 653-1). arinc, inc., 2003.*

[ASS] *ASSERT : http ://www.assert-project.net/.*

[ATL] *http ://www.eclipse.org/m2m/atl/doc/.*

[Bas08] Ananda Shankar Basu, *Component-based modeling of heterogeneous real-time systems in bip*, Ph.D. thesis, UJF-Verimag, 2008.

[BBBS08] Ananda Basu, Philippe Bidinger, Marius Bozga, and Joseph Sifakis, *Distributed semantics and implementation for systems with interaction and priority*, FORTE, 2008, pp. 116–133.

[BBG+08] A. Basu, S. Bensalem, M. Gallien, F. Ingrand, C. Lesire, T.H. Nguyen, and J. Sifakis, *Incremental component-based construction and verification of a robotic system*, Proceedings of ECAI'08, Patras, Greece, 2008.

[BBJ09] Ananda Basu, Simon Bliudze, and Mohamad Jaber, *Symbolic implementation of connectors in bip*, 2nd Interaction and Concurrency Experience, 2009, To appear.

[BBNS09] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis, *D-finder : A tool for compositional deadlock detection and verification*, CAV, 2009, pp. 614–619.

[BBS06] A. Basu, M. Bozga, and J. Sifakis, *Modeling heterogeneous real-time components in bip*, Proceedings of SEFM '06, Pune, India, IEEE Computer Society, 2006, pp. 3–12.

[BBSN08] S. Bensalem, M. Bozga, J. Sifakis, and T.H. Nguyen, *Compositional verification for component-based systems and application*, Proceedings of ATVA'08, Seoul, South Korea, 2008, pp. 64–79.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani, *The fractal component model and its support in java : Experiences with auto-adaptive and reconfigurable systems*, Softw. Pract. Exper. **36** (2006), no. 11-12, 1257–1284.

[BDJ+03] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui, *First experiments with the atl model transformation language : Transforming xslt into xquery*, 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture, 2003.

[BFKM97] M. Bozga, J-C. Fernandez, A. Kerbrat, and L. Mounier, *Protocol verification with the aldebaran toolset*, STTT **1** (1997), 166–183.

[BGO+04] M. Bozga, S. Graf, Il. Ober, Iul. Ober, and J. Sifakis, *The if toolset*, Proceedings of SFM'04, Bertinoro, Italy, LNCS, vol. 3185, September 2004, pp. 237–267.

[BMP+07] A. Basu, L. Mounier, M. Poulhiès, J. Pulou, and J. Sifakis, *Using bip for modeling and verification of networked systems – a case study on tinyos-based networks.*, Proceedings of NCA'07, Cambridge, MA USA, 2007, pp. 257–260.

[Boc06] Conrad Bock, *Sysml and uml 2 support for activity modeling*, Syst. Eng. **9** (2006), no. 2, 160–186.

[BS00] Sébastein Bornot and Joseph Sifakis, *An algebraic framework for urgency*, Inf. Comput. **163** (2000), no. 1, 172–202.

[BST97] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis, *Modeling urgency in timed systems*, In International Symposium : Compositionality - The Significant Difference, Springer, 1997, pp. 103–129.

[CKSS01] Valentin Crettaz, Mohamed Mancona Kandé, Shane Sendall, and Alfred Strohmeier, *Integrating the concernbase approach with sadl*, In Proceedings 4th International Conference on Modeling Languages, Concepts, and Tools .Toronto, Canada, 2001, pp. 166–181.

[Cle96] P. C. Clements, *A survey of architecture description languages*, In Proceedings of the Eighth International Workshop on Software Specification and Design, Paderborn, Germany, 1996.

[DGRMDW97] David Garlan, Robert Monroe, and Dave Wile, *Acme : An architecture description interchange language*, Proceedings of CASCON 97, 1997, pp. 169–183.

[EJL+03] Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, and Yuhong Xiong, *Taming heterogeneity - the ptolemy approach*, Proceedings of the IEEE **91** (2003), no. 1, 127–144.

[EMF] *EMF : Eclipse modeling framework project, http ://www.eclipse.org/modeling/emf.*

[GAO94] David Garlan, Robert Allen, and John Ockerbloom, *Exploiting style in architectural design environments*, SIGSOFT Softw. Eng. Notes **19** (1994), no. 5, 175–188.

[GCK02] David Garlan, Shang-Wen Cheng, and Andrew J. Kompanek, *Reconciling the needs of architectural description with object-modeling notations*, Sci. Comput. Program. **44** (2002), no. 1, 23–49.

[GG05] J. Sifakis G. Gossler, *Composition for component-based modeling*, Science of Computer Programming **55** (2005), 161–183.

[Gößl01] G. Gößler, PROMETHEUS — *a compositional modeling tool for real-time systems*, Proc. Workshop RT-TOOLS 2001 (P. Pettersson and S. Yovine, eds.), Technical report 2001-014, Uppsala University, Department of Information Technology, 2001.

[GS03] Gregor Gössler and Joseph Sifakis, *Component-based construction of deadlock-free systems*, proceedings of FSTTCS 2003, Mumbai, India, LNCS 2914, 2003, pp. 420–433.

[GWS05] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres, *Open MPI : A flexible high performance MPI*, Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics (Poznan, Poland), September 2005.

[HR97] Rich Hilliard and Tim Rice, *Expressiveness in architecture description languages*, Proceedings of the 3rd International Software Architecture Workshop, ACM Press, 1997, pp. 65–68.

[HZPK07] Jérôme Hugues, Bechir Zalila, Laurent Pautet, and Fabrice Kordon, *Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina*, Proceedings of the 18th IEEE International Workshop on Rapid System Prototyping (RSP'07) (Porto Alegre, Brazil), IEEE Computer Society Press, May 2007, pp. 106–112.

[HZPK08] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, *From the prototype to the final embedded system using the ocarina aadl tool suite*, ACM Trans. Embed. Comput. Syst. **7** (2008), no. 4, 1–25.

[IE07] Hamid I. and Najm E., *Real-time connectors for deterministic data-flow*, Proceedings of RTCSA 2007, Seoul, Korea, IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2007.

[IEE] *IEEE Architecture Working Group : "Recommended Practice for Architectural description of Software-Intensive Systems", IEEE Std 1471-2000, IEEE, 2000.*

[Jav] *Java, http ://www.java.com/.*

[LKA+95] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, *Specification and analysis of system architecture using rapide*, IEEE Transactions on Software Engineering, vol. 1 no.4, 1995, pp. 336–335.

[LV95] David C. Luckham and James Vera, *An event-based architecture definition language*, IEEE Trans. Softw. Eng. **21** (1995), no. 9, 717–734.

[MAR] *Omg, uml profile for modeling and analysis of real-time and embedded systems (marte) rfp, realtime/05-02-06.*

[MDA] *OMG Model-Driven Architecture Home Page, http ://www.omg.org/mda.*

[Med99] Nenad Medvidovic, *A language and environment for architecture-based software development and evolution*, In Proceedings of the 1999 International Conference on Software Engineering, 1999, pp. 44–53.

[Meta] *http ://www-verimag.imag.fr/∼async/bipMetamodel.php.*

[Metb] *Vestal s (1998) software programmer's manual for the honeywell aerospace compiled kernel (metah language reference manual). honeywell technology center, minneapolis.*

[MK96] J. Magee and J. Kramer, *Dynamic structure in software architectures.*, In Proceedings of ACM SIGSOFT'96 : Fourth Symposium on the Foundations of Software Engineering (FSE4), 1996, pp. 3–14.

[MORT96] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor, *Using object-oriented typing to support architectural design in the c2 style*, In Proceedings of ACM SIGSOFT'201996 : Fourth Symposium on the Foundations of Software Engineering (FSE4), ACM Press, 1996, pp. 24–32.

[MR97a] Nenad Medvidovic and David S. Rosenblum, *Domains of concern in software architectures and architecture description languages*, DSL'97 : Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL) (Berkeley, CA, USA), USENIX Association, 1997, pp. 16–16.

[MR97b] M. Moriconi and R. A. Riemenschneider, *Introduction to sadl 1.0 : A language for specifying software architecture hierarchies*, Technical Report SRI-CSL-97-01, SRI International, 1997.

[MT97] N. Medvidovi and R. M. Taylor, *A framework for classifying and comparing architecture description languages*, In Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97), 1997, pp. 60–76.

[OCA] *OCARINA - http ://ocarina.enst.fr/.*

[OMG] *Object managment group home page, http ://www.omg.org.*

[Ope] *OpenEMBeDD - https ://openembedd.org.*

[OSA] *SEI. Open Source AADL Tool Environment. http ://la.sei.cmu.edu/aadlinfosite/ OpenSourceAADL-ToolEnvironment.html.*

[PAP] *PAPYRUS, http ://www.papyrusuml.org/.*

[Poo01] John D. Poole, *Model-driven architecture : Vision, standards and emerging technologies*, In In ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models, 2001.

[PPRS06] M. Poulhiès, J. Pulou, C. Rippert, and J. Sifakis, *A methodology and supporting tools for the development of*

component-based embedded systems, 13th Monterey Workshop, Paris, France, LNCS, vol. 4888, 2006, pp. 75–96.

[QS82] Jean-Pierre Queille and Joseph Sifakis, *Specification and verification of concurrent systems in cesar*, Proceedings of the 5th Colloquium on International Symposium on Programming, 1982, pp. 337–351.

[RKK08] Ana-Elena Rugina, Karama Kanoun, and Mohamed Kaâniche, *The adapt tool : From aadl architectural models to stochastic petri nets through model transformation*, CoRR **abs/0809.4108** (2008).

[SDK+95] Mary Shaw, Robert Deline, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik, *Abstractions for software architecture and tools to support them*, IEEE Transactions on Software Engineering **21** (1995), 314–335.

[SDZ96] Mary Shaw, Robert Deline, and Gregory Zelesnik, *Abstractions and implementations for architectural connections*, In Proceedings of the Third International Conference on Configurable Distributed Systems, 1996.

[SLC06] Oleg Sokolsky, Insup Lee, and Duncan Clarke, *Schedulability analysis of aadl models*, IPDPS, 2006.

[SLNM04] Frank Singhoff, Jérôme Legrand, Laurent tchamnda Nana, and Lionel Marcé, *Cheddar : a flexible real time scheduling framework*, ACM Ada Letters journal, 24(4) :1-8, ACM Press, ISSN :1094-3641 (2004).

[SPI] *SPICES - http ://www.spices-itea.org/public/news.php.*

[STO] *P. dissaux. using the aadl for mission critical software development. 2nd European Congress ERTS, EMBEDDED REAL TIME SOFTWARE Toulouse, January 2004.*

[Til05] J. F. Tilman, *Building tool suite for aadl*, IFIP International Federation for Information Processing, vol. 176, 2005, pp. 197–207.

[TM00] R. M. Taylor and Nenad Medvidovic, *A classification and comparison framework for software architecture description languages*, IEEE Transactions on Software Engineering **26** (2000), 70–93.

[TOP] *TOPCASED- Un environnement de développement open source pour les systèmes embarqués. http ://www.topcased.org/.*

[Tre94] Jan Tretmans, *A formal approach to conformance testing*, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI, 1994, pp. 257–276.

[UMLa] *A. Cuccuru, P. Marquet, and J.-L. Dekeyser, "UML2 as an ADL hierarchical hardware modeling," Tech. Rep., Apr. 2004.*

[UMLb] *O. M. Group, "UML 2.0 specication," Tech. Rep.*

[Ves93] S. Vestal, *A cursory overview and comparison of four architecture description languages*, Technical Report, Honeywell Technology Center, 1993.

[VPK05] Thomas Vergnaud, Laurent Pautet, and Fabrice Kordon, *Using the aadl to describe distributed applications from middleware to software components*, Ada-Europe, 2005, pp. 67–78.

[Wri] *R. j. allen. a formal approach to software architecture. thèse de doctorat, school of computer science, carnegie mellon university, 1997.*

Résumé

Le langage d'analyse et de description d'architectures (AADL) fait l'objet d'un intérêt croissant dans l'industrie des systèmes embarqués temps-réel. Il définit plusieurs catégories de composants, réparties en trois grandes familles (logiciel, matériel, système). Le travail réalisé durant cette thèse exploite les fonctionnalités offertes par AADL pour spécifier les besoins exacts d'une application et exprimer toutes les caractéristiques tant fonctionnelles que non fonctionnelles (dimensions temporelle et spatiale), afin de la produire automatiquement. La méthodologie de production que nous proposons génère automatiquement, à partir d'une application décrite en AADL, une application décrite en BIP. BIP permet de mettre en place des systèmes robustes et sûrs en produisant un contrôleur d'exécution correct par construction et en fournissant un modèle formel. Les objectifs de ce processus de production sont : (1) fournir à AADL une sémantique formelle définie en termes de systèmes de transitions étiquetés ; (2) permettre l'analyse et la validation, c'est à dire, l'exploration exhaustive de l'espace des états du système, la détection des blocages potentiels et la vérification de certaines propriétés ; (3) permettre la génération d'une application exécutable pour simuler et déboguer les modèles AADL. Ces trois derniers points jouent en faveur de l'utilisation de méthodes formelles dans le cycle de développement.

Abstract

The Architecture Analysis & Design Language (AADL) is the subject of increasing interest in the industry of real-time embedded systems. It defines several categories of components, grouped into three major categories (software, hardware, systems). The work realized in this thesis exploits the features offered by AADL to specify the exact requirements of an application and to express all the features both functional and nonfunctional (temporal and spatial dimensions) required to produce automatically the application. The production methodology that we propose generates automatically from an application described in AADL, an application described in BIP. BIP allows to implement robust and safe systems by producing a correct execution controller design and providing a formal model. The objectives of this production process are : (1) provide to AADL a formal semantics defined in terms of labeled transition systems ; (2) allow analysis and validation, i.e. exhaustive exploration of the state space of the system, the detection of potential deadlocks and verification of certain properties ; (3) allow the generation of an executable application to simulate and debug the AADL models. These last three step play a cental rule for the use of formal methods in the development cycle.