Définition d'un Modèle de Déploiement d'une Architecture logicielle à base de BRS

Nadira Benlahrache

▶ **To cite this version:**

Nadira Benlahrache. Définition d'un Modèle de Déploiement d'une Architecture logicielle à base de BRS. Informatique [cs]. Université Constantine 2 Abdel Hamid Mehdi Algérie, 2014. Français. NNT : . tel-02385545

HAL Id: tel-02385545
https://theses.hal.science/tel-02385545

Submitted on 28 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL
archives-ouvertes.fr

Definition d'un Modèle de Deploiement d'une Architecture logicielle à base de BRS

Nadira Benlahrache

▶ To cite this version:

Nadira Benlahrache. Definition d'un Modèle de Deploiement d'une Architecture logicielle à base de BRS. Informatique [cs]. Université Constantine 2 Abdel Hamid Mehdi Algérie, 2014. Français. tel-02385545

HAL Id: tel-02385545
https://tel.archives-ouvertes.fr/tel-02385545

Submitted on 28 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Constantine 2, Faculté NTIC
Laboratoire LIRE, Équipe GLSD

Thèse

présentée en vu d'obtenir le grade de Docteur, spécialité « Informatique »

par

Nadira Benlahrache-Benslama

Définition d'un Modèle de Déploiement d'une Architecture Logicielle à Base de BRS

Thèse soutenue le 08/06/2014 devant le jury composé de :

Mme.	Z. Boufaida	Professeur, Université Constantine 2	(Président)
Mme.	F. Belala	Professeur, Université Constantine 2	(Rapporteur)
Mr.	DJ. Meslati	Professeur, Université Badji Mokhtar, Annaba	(Examinateur)
Mme.	T. Tebibel	Professeur, ESI, Alger	(Examinateur)
Mme.	S. Meshoul	Maitre de Conférences, Université Constantine 2	(Examinateur)
Mr.	K. Barkaoui	Professeur, CNAM, Paris France	(Invité)

À ma famille…

Remerciements

Je voudrais tout d'abord exprimer mes plus profonds remerciements au Pr. Faiza Belala pour m'avoir encadrée, conseillée, dirigée, rassurée et surtout soutenue durant ces années de thèse.

Mes sincères remerciements s'adressent à Mesdames et Messieurs les membres du jury :
Au Pr. Z. Boufaida pour avoir accepté d'être présidente du jury. Je remercie également Pr. T. Tebibel, Pr. Dj. Meslati et Dr. S. Meshoul pour avoir accepté de juger ce travail. Je les remercie pour leur remarques pertinentes qu'ils ont faites à ce travail.

Je tiens aussi à manifester ma profonde gratitude envers le professeur K.Barkaoui, Responsable de l'équipe *CEDRIC* du *CNAM* Paris, de m'avoir bien accueillie dans ce laboratoire à plusieurs reprises et m'avoir fait part de ses compétences et de son savoir faire.

Grand merci au Dr C. Bouanaka pour sa lecture attentive de ce manuscrit et pour ses remarques. Merci à Mr F. Latreche, pour son aide précieuse. Mes remerciements vont, également, à tous les membres de l'équipe GLSD (Aicha, Meriem, Malika, Nadia Z., Nadia C., K&H Boukhelfa, H. Sebih, M. Chihoub, L. Derdouri et M.T. Cherfia) avec qui j'ai partagé le quotidien de la recherche avec ses périodes de crises et d'échanges.

Merci aux membres du laboratoire LIRE particulièrement à son directeur Pr. M. Boufaida pour m'avoir accueillie dans son laboratoire.

J'adresse mes remerciements à celles et ceux (Frères, Sœurs, Amies et Collègues), qui par leurs actions, leurs témoignages, leurs encouragements ont permis la réalisation de ce travail.

Je conclurai en remerciant de tout cœur mon époux Salim, merci pour tout et d'être simplement à mes côtés.

Table des matières

Liste des figures

Liste des tables

Introduction Générale

Sommaire

Nous ne pouvons pas prédire où nous conduira la Révolution Informatique. Tout ce que nous savons avec certitude, c'est que, quand on y sera enfin, on n'aura pas assez de RAM.

Dave Barry

Contexte et Problématique

Les applications émergentes récentes telles que les applications de géolocalisation, de prévention et d'alerte face aux catastrophes naturelles, l'Internet participatif, la nanotechnologie, la biotechnologie et le cloud computing sont de plus en plus distribuées, ouvertes et à architecture complexe. De telles applications sont composées d'un nombre, généralement important, d'entités logicielles dispersées sur le réseau coopérant pour fournir à leurs utilisateurs, dans les délais et avec une qualité acceptable, les services qu'ils requièrent. Cette complexité est liée, par exemple, à l'extension des réseaux de communication, à l'hétérogénéité matérielle et logicielle, au fort degré d'interaction et aux exigences d'une évolution nécessaire et permanente.

La conception de tels logiciels, qui consiste à déterminer les fonctionnalités qu'ils devront offrir, est devenue elle aussi si complexe et difficile à maitriser et provoquant parfois des dérapages budgétaires. C'est à ce stade que le génie logiciel intervient, en particulier, pour définir les procédures systématiques qui permettent d'arriver à ce que des logiciels de grande taille correspondent aux attentes du client, soient fiables, aient un coût d'entretien réduit et de bonnes performances tout en respectant les délais et les coûts de construction.

L'architecture logicielle est une discipline récente dans le génie logiciel ayant comme objectif d'apporter des solutions pour gérer la complexité au niveau conceptuel. Elle concerne la conception et la modélisation des systèmes à un niveau d'abstraction qui révèle leur structure brute et permet de raisonner sur les propriétés clés du système, telles que la performance, la fiabilité et la sécurité (Garlan 2003). La modélisation architecturale sert généralement à décrire un système comme un ensemble d'éléments en interaction, où les détails d'implémentation de bas niveau sont cachés mais les propriétés de haut niveau du système décrivant ses qualités de services (comme les débits attendus, les latences, et fiabilités) sont exposées.

L'architecture logicielle a été largement préconisée comme une abstraction efficace pour la modélisation, la mise en œuvre et l'évolution des systèmes logiciels complexes tels que les systèmes distribués, décentralisés et ceux évoluant dans des environnements hétérogènes et ambiants (Allen et al. 1998, Xihong et al. 2010).

En proposant un modèle de haut niveau pour la structure d'un système,

permettra de la comprendre en termes beaucoup plus simples que ceux accordés par les structures au niveau code, telles que les classes, variables, méthodes, etc. En outre, si c'est bien spécifiée, une description architecturale doit, en principe, permettre d'affirmer que la conception d'un système satisfait aux exigences clés en faisant appel à des raisonnements abstraits sur la structure.

Malheureusement, la description d'une architecture logicielle est généralement informelle, elle est en grande partie basée sur des dessins de boîtes et de lignes qui sont souvent ambiguës, incomplets, incohérents, et non-analysables. Ce n'est pas nécessairement ce qui est recherché (Garlan 2003). À cet effet, plusieurs formalismes et approches ont été utilisés pour décrire convenablement une architecture.
L'OMG (Object Management Group) (OMG 2003) a proposé l'approche dirigée par les modèles (MDA : Model Driven Architecture). Dans celle-ci, les modèles constituent la couche nécessaire pour fournir le niveau d'abstraction demandé. L'approche MDA préconise l'utilisation d'UML (Unified Modeling Language) (OMG 2005) pour l'élaboration de ces modèles. Par ailleurs, ce sont les graphes qui ont été considérés comme le meilleur formalisme pour décrire les architectures vu leur base formelle et leur aspect visuel et facile à appréhender. On y trouve essentiellement l'utilisation des réseaux de Petri ou les systèmes de transitions. Par contre, et par souci de vérification et d'analyse, se sont les logiques qui ont servi de formalisme pour spécifier les architectures logicielles tel est le cas de l'approche *Z*. Ces formalismes, malgré leurs avantages, sont souvent inadéquats pour décrire convenablement les spécificités des architectures logicielles (Zhang et al. 2010).

Les Langages de Description d'Architectures (ADLs) sont définis afin de spécifier précisément une architecture logicielle constituée principalement de composants fonctionnels décrits en termes de leurs comportements, interfaces et intercommunications. Les ADLs existants dans la littérature possèdent souvent des caractéristiques spécifiques liées à leur motivation, à leur usage et éventuellement à la sémantique formelle associée (Medvidovic et Taylor 2000, Zhang et al. 2010).

En raison de la complexité croissante de l'interaction matériel/logiciel, ces ADLs et malgré leur nombre et diversité, deviennent difficilement utilisables pour certains aspects du cycle de développement des systèmes logiciels, notamment la spécification des mécanismes d'installation et de reconfiguration, activités principales constituant le processus de déploiement des applications

logicielles.

D'autant plus que des recherches antérieures ont montré que le déploiement d'un système logiciel peut avoir un impact significatif sur les propriétés non-fonctionnelles du système c'est à dire, il aura une incidence sur la QoS du système livré (Medvidovic et Sam 2007). De par leur complexité, les activités de déploiement d'une application, étroitement dépendantes des contraintes technologiques de la plateforme cible, sont rarement spécifiées formellement ceci constitue une entrave pour l'analyse et la vérification de la correction du déploiement d'une application logicielle.

Partant du principe que l'abstraction réduit la complexité, spécifier le processus de déploiement au niveau architectural permettra de réfléchir en amont sur les contraintes liées à ce processus. La spécification du déploiement d'une architecture logicielle suit quasiment les principales étapes du processus de déploiement d'un logiciel, qui demeure un processus complexe, traduit par une suite d'activités allant de la production d'une nouvelle version de l'application jusqu'à sa désinstallation (Parrish et al. 2001, Xihong et al. 2010).

Peu de travaux dans la littérature se sont intéressés à ce processus de déploiement au niveau architectural. Cependant, le langage de modélisation *UML* (OMG 2005) offre un diagramme dit de "déploiement" représentant la disposition physique des entités matérielles qui composent la plateforme et la répartition des composants logiciels sur ces entités. L'inconvénient est que ce langage ne possède pas une sémantique formelle. En plus, il ne permet pas une définition claire du processus de déploiement.

Le langage *AADL*, standard promu par la *SAE* [1] (SAE 2008) pour l'analyse et la conception des architectures logicielles, se distingue des autres ADLs par sa capacité à rassembler au sein d'une même notation l'ensemble des informations concernant l'organisation de l'application et sa plateforme d'exécution. Il possède formulation intéressante de description du déploiement de composants logiciels sur les composants matériels (par des propriétés). Néanmoins, il n'est pas doté d'une notation formelle décrivant l'opération de déploiement en particulier.

Bien que ce langage a bénéficié durant ces dernières années des travaux de formalisation (Vergnaud 2006, Benammar 2011). Ces travaux ne considèrent que quelques aspects de ses éléments syntaxiques et aucun d'eux ne s'est intéressé à la formalisation de l'opération de déploiement.

1. SAE : Society of Automotive Engineers

Par ailleurs, les systèmes réactifs bigraphiques (BRS) introduits dans Jensen et Milner (2004) et communément appelés Bigraphes, constituent un modèle graphique capable de spécifier formellement les applications distribuées à code mobile. Ils fédèrent dans un même modèle deux graphes. L'un de ses graphes décrit la structure et la distribution spatiale des éléments d'un bigraphe appelé *graphe des places* et l'autre est dédié à la représentation des interactions, appelé *graphe des liens*. Les premiers travaux visant à concrétiser l'utilisation des *BRS* dans le domaine des architectures logicielles et styles architecturaux ont été publiés dans Chang et al. (2007; 2008).

Objectifs

L'objectif de ce travail de thèse est de proposer une solution générique pour la formalisation du processus de déploiement d'une architecture logicielle sans se limiter à un ADL particulier. Ensuite, nous projetons cette définition formelle à base des bigraphes sur le langage AADL.

La description formelle de l'application logicielle est donnée par un bigraphe, où les nœuds du bigraphe sont les composants et les connecteurs de l'architecture et les liens représentent les interactions entre ces éléments architecturaux. En parallèle et de façon indépendante, nous utilisons un autre bigraphe pour décrire les éléments architecturaux de la plateforme d'exécution.
Ensuite, la corrélation entre les deux bigraphes est exploitée dans le cadre de cette thèse pour décrire formellement :

- d'une part, l'opération d'*installation* des entités logicielles sur les entités matérielles,
- d'autre part, l'opération de *reconfiguration* de l'application pour s'adapter aux changements du contexte de l'exécution.

L'approche de formalisation à base des bigraphes que nous proposons dans cette thèse, permet de dégager et d'élaborer un modèle unificateur (méta-modèle), générique et supportant la sémantique d'une application logicielle à base de composants, de sa plateforme d'exécution ainsi que du processus de déploiement. En particulier, il modélise les activités d'installation et de reconfiguration du processus de déploiement, souvent considérées séparément dans la plus part des approches existantes. Le modèle bigraphique proposé dans ce travail, avec son aspect graphique, rend la description du processus de déploiement très expressive.

L'approche de déploiement proposée sera mise en œuvre sur un système temps-réel et critique (Patient Monitoring System : PMS) décrit en langage AADL, où nous identifions un couplage entre les entités logicielles et matérielles, matérialisé à l'aide d'une propriété AADL.

Nous précédons à la vérification du modèle bigraphique obtenu selon deux scénarios. Le premier utilise le model-checker (*BigMc*) élaboré par Perrone et al. (2012) dédié aux bigraphes. Nous montrons que cet outil souffre de quelques lacunes concernant la prise en charge des spécificités des bigraphes telle que l'opération de composition. Le second scénario de vérification utilise la logique de réécriture (Meseguer 1997) à travers son environnement d'exécution *Maude* et son model-checker LTL.

Organisation du Document

Ce manuscrit est organisé comme suit. Le premier chapitre est consacré à la présentation du concept d'*architecture logicielle* et les travaux connexes concernant le déploiement des logiciels et particulièrement le déploiement architectural. Ensuite, nous introduisons dans le chapitre 2 le langage AADL et ses spécificités en tant que langage de description d'architecture. Le chapitre 3 est dédié à la présentation des concepts fondamentaux des BRS (bigraphes) et leurs opérations de manipulation inhérentes. Dans le chapitre 4 nous détaillons la description formelle des deux structures d'une architecture AADL à savoir l'application et la plateforme d'exécution tout en utilisant le formalisme des bigraphes. La modélisation proposée est illustrée à travers un cas d'exemple décrivant un système réparti temps-réel *PMS*, où l'architecture logicielle et son environnement d'exécution sont modélisés par des bigraphes indépendants. Nous décrivons par la suite dans le chapitre 5, la formalisation des deux activités les plus critiques du processus de déploiement à savoir l'*installation* et la *reconfiguration* et ce à travers des opérations de manipulation des bigraphes. Le chapitre 6 est consacré à la validation et la vérification des modèles bigraphiques générés. Nous présentons deux scénarios de validation. Le premier, appelé *manipulation directe des bigraphes* où l'on fait subir à une description bigraphique une vérification par le model-checker *BigMc*. Dans le deuxième scénario, la validation sera à travers le système Maude et son model-checker basé sur la logique *LTL*. Dans le même chapitre, nous présentons un ensemble d'outils pratiques développés afin de mettre en œuvre notre contribution à savoir *AADL2Bigraph*, *BigraphOperations* et *Bigraph2Maude*. Enfin, une conclusion résume notre contribution et présente quelques perspectives.

1 Déploiement d'une Architecture Logicielle

Sommaire

Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled. Eoin Woods

1.1 Introduction

Concevoir une architecture logicielle d'un système complexe est une étape cruciale pour assurer le bon fonctionnement de celui-ci et garantir que le système produit satisferait ses principaux objectifs. Malheureusement, la description de cette architecture logicielle, généralement, est en grande partie basée sur des dessins informels en formes de boîtes et de lignes qui sont souvent ambiguës, incomplets, incohérents, et non-analysables (Garlan 2003). Les Langages de Description d'Architecture (ADL), en plein essor, ont pour principale vocation de remédier à ses faiblesses très tôt dans le processus de développement.
Dans ce chapitre, nous présentons la définition d'une architecture logicielle à base de composants. Nous nous appuyons ici sur différents travaux autour des langages de description d'architecture, des modèles de composants académiques et essentiellement sur des travaux issus des organismes de normalisation. Ce chapitre étudie les principales caractéristiques, les éléments structurels du modèle de composants, l'expression de la dynamique et la prise en compte du déploiement. L'évaluation des langages de description d'architectures, sera dirigée ici, par leur niveau d'abstraction, leur indépendance vis-avis de l'infrastructure physique sur laquelle est déployé le logiciel, leur capacité à exprimer les différentes étapes du processus de déploiement, et la prise en compte de la dynamique du système.

1.2 Architecture Logicielle

Pendant des décennies, les concepteurs de logiciels ont appris à construire des systèmes en se basant exclusivement sur les exigences techniques. Conceptuellement, le cahier des charges est scotché sur le mur de la cabine du concepteur, ce dernier doit sortir avec un "design" satisfaisant. Les exigences engendrent la conception, qui à son tour engendre le système. Heureusement, les méthodes modernes de développement de logiciels reconnaissent la naïveté de ce modèle et essayent de fournir toute sorte de boucles de rétroaction du concepteur à l'analyste.

Ainsi, l'architecture logicielle joue un rôle essentiel dans ce processus et le rend plus efficace en permettant à une organisation d'atteindre ses objectifs d'automatisation. Élaborer une architecture logicielle exige un prix (le coût de son développement), mais ce coût sera grassement payé en permettant à l'organisation d'atteindre ses objectifs et d'étendre ses capacités logicielles.

L'architecture est un atout qui détient une valeur tangible pour l'organisation au-delà de l'élaboration du projet pour lequel elle a été créée (Bass et al. 2003).

1.2.1 Concepts Généraux

Le terme architecture logicielle désigne une esquisse bien bâtie du système futur. Il n'existe aucune norme de la définition universellement acceptée du terme, l'architecture logicielle est un domaine à ses débuts, bien que ses racines sont profondément ancrées dans le génie-logiciel.

Définitions

La littérature foisonne des définitions concernant ce terme qui vient booster le monde informatique et les techniques de développement des logiciels. Nous allons exposer dans les paragraphes suivants quelques définitions données par des pionniers dans ce domaine.

Nous reprenons une définition donnée par Shaw et Garlan (1996) :
"*Software architecture encompasses the set of significant decisions about the organization of a software system including the selection of the structural elements and their interfaces by which the system is composed ; behavior as specified in collaboration among those elements ; composition of these structural and behavioral elements into larger subsystems ; and an architectural style that guides this organization. Software architecture also involves functionality, usability, resilience, performance, reuse, comprehensibility, economic and technology constraints, tradeoffs and esthetic concerns*."
L'architecture logicielle rassemble l'ensemble des décisions importantes au sujet de l'organisation d'un système logiciel, y compris la sélection des éléments structurels qui composent ce système ainsi que les interfaces qui assurent sa composition ; son comportement spécifié à travers les interactions entre ces éléments ; la composition de ces éléments structurels et comportementaux dans des sous-systèmes plus larges, et d'un style architectural qui guide cette organisation. L'architecture logicielle spécifie également la fonctionnalité, l'ergonomie, la résilience, la performance, la réutilisation, la compréhensibilité, les contraintes économiques et technologiques, les compromis et les préoccupations d'esthétiques.

Dans Medvidovic et Sam (2007), l'auteur donne la définition suivante : "*l'architecture logicielle est une collection de modèles qui capturent les principales décisions de la conception d'un système logiciel sous la forme de composants (foyers de système de calcul et de gestion des données), connecteurs (foyers d'interaction*

entre les composants), et les configurations (dispositions spécifiques de composants et de connecteurs destinés à résoudre des problèmes spécifiques)"
Une architecture logicielle est donc :

- un ensemble de décisions importantes relatives à l'organisation d'un système logiciel,
- une sélection d'éléments structurels et de leurs interfaces par lesquelles le système est composé,
- le comportement de ces éléments structurels décrit par les interactions entre ces éléments,
- la composition de ces éléments structurels et comportementaux en sous systèmes progressivement plus grands,
- un style architectural qui guide cette organisation (autrement dit, ces éléments et leurs interfaces, leurs interactions et leur composition).

L'idée que veut véhiculer cette définition est qu'une architecture logicielle doit abstraire certaines informations du système mais en même temps fournir suffisamment d'informations pour en constituer une base pour l'analyse, la prise de décision, et par conséquent la réduction des risques.

Dans (Bass et al. 2003), les auteurs ont défini une architecture logicielle comme suit : "*The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*" : L'architecture logicielle d'un programme ou d'un système informatique est la ou les structures de ce système, y compris ses éléments logiciels, leurs propriétés visibles, et les relations entre eux.

Une autre définition a été donnée plus tard par les mêmes auteurs Bass et al. (2012) "*The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of them.*" : L'architecture logicielle d'un système est l'ensemble des structures nécessaires pour raisonner sur le système, qui comportent des éléments logiciels, les relations entre eux et leurs propriétés.

Nous remarquons que les deux dernières définitions semblent être identiques, mais la deuxième marque une prise de conscience concernant l'aspect analyse et raisonnement. Donc, il ne suffit pas de décrire une architecture mais il parait important et crucial de pouvoir analyser et raisonner voire vérifier cette architecture.

Il faut retenir que l'architecture doit donc :
- Exposer la structure du système, mais masquer les détails d'implémentation.
- Réaliser tous les cas d'utilisation et les scénarios.
- Essayer de répondre aux exigences des différentes parties prenantes.
- Gérer à la fois les exigences fonctionnelles et non fonctionnelles.

Composant

Dans (Szyperski et al. 2003), nous retrouvons la définition suivante concernant un composant logiciel : *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software composent can be deployed independently and is subject to composition by third parties".*

Un composant est défini comme étant une unité de composition avec des interfaces contractuellement spécifiées et des dépendances de contexte explicites. Un composant logiciel peut être déployé de façon indépendante, il est soumis à la composition par une partie tierce.

Dans la spécification UML2 (OMG 2005), un composant est défini comme une partie d'un système modulaire qui *encapsule* son contenu et dont la manifestation est remplaçable dans son environnement. Le composant offre une meilleure structuration de l'application. Il définit son comportement en termes d'interfaces fournies et requises (Dearle 2007).

Connecteur

Le connecteur correspond à un élément d'architecture qui modélise les interactions entre deux ou plusieurs composants en définissant les règles qui gouvernent ces interactions (ACCORD 2002). Un connecteur peut être considéré comme un composant de communication. On y trouve des connecteurs de communication assurant la transmission des données entre composants, ou ceux de coordination assurant le transfert de contrôle d'un composant à un autre (Shaw et Garlan 1996). Les connecteurs spécifiques ont le rôle de convertir l'interaction fournie par un composant vers l'interaction requise par un autre composant.

Port

Un composant interagit avec son environnement par l'intermédiaire de points d'interactions appelés ports. Les ports (et par conséquent les interfaces) peuvent être fournis ou requis. Un port représente un point d'accès à certains services du composant. Le comportement interne du composant ne doit être ni visible, ni accessible autrement que par ses ports (Hadj Kacem 2008).

Interface

L'interface est un moyen d'expression des liens du composant ainsi que ses contraintes avec l'extérieur. C'est le point de communication qui permet au composant d'interagir avec l'environnement. On la retrouve donc associée aux composants et aux connecteurs. Elle spécifie des ports dans le cas des composants et des rôles dans le cas des connecteurs (ACCORD 2002).

Pour un composant, il existe deux types d'interfaces :
- Les interfaces fournies décrivent les services proposés par le composant,
- Les interfaces requises décrivent les services que les autres composants doivent fournir pour le bon fonctionnement du composant dans un environnement particulier.

Configuration

Une configuration appelée aussi topologie définit les propriétés architecturales de connectivité et de conformité aux heuristiques de conception, ainsi que des propriétés de concurrence et de répartition.
La configuration structurelle d'un système correspond à un graphe connexe de l'ensemble des composants et des connecteurs le formant, elle vérifie la correspondance entre les interfaces des composants et des connecteurs. Tandis que la configuration comportementale modélise le comportement en décrivant l'évolution des liens entre composants et connecteurs, ainsi que l'évolution des propriétés non fonctionnelles comme les propriétés spatio-temporelles ou la qualité de service (ACCORD 2002).
Plus précisément, une configuration décrit les instances de composants/connecteurs intervenants et les relations qu'elles entretiennent entre elles (Hadj Kacem 2008). Une configuration représente en fait une instance possible d'un *style architectural*.

1.2.2 Rôle des Architectures Logicielles

L'architecture logicielle à base de composants contient des informations sur la manière dont les composants interagissent les uns avec les autres. Cela signifie que l'architecture omet spécifiquement les informations concernant les composants qui ne se rapportent pas à leur interaction.
Les spécifications architecturales nous informent sur plusieurs aspects concernant une application tels que :

- Quels sont les sous-systèmes ou composants du logiciel ?
- Quelles sont les fonctions d'un composant ?
- Quelles interfaces sont fournies et requises par un composant ?
- Quels sous-systèmes seraient touchés par un changement particulier ?
- Quels éléments sont impliqués dans l'instauration de ce changement ?
- Quelles sont les parties du système qui seront distribuées physiquement ?
- Quel est l'impact d'un changement sur les performances du système ?
- Quelles équipes de développement sont concernées par ce changement ?
- Quel niveau d'effort est impliqué dans le développement de cette fonctionnalité ?

Dans chacune des phases du cycle de vie d'un logiciel, l'architecture logicielle à base de composants joue un rôle particulier (Garlan 2003, Bass et al. 2012; 2003). Elle suit un cycle de vie (figure 1.1) analogue à celui d'une application logicielle depuis la spécification des exigences (cahier des charges) jusqu'au déploiement.

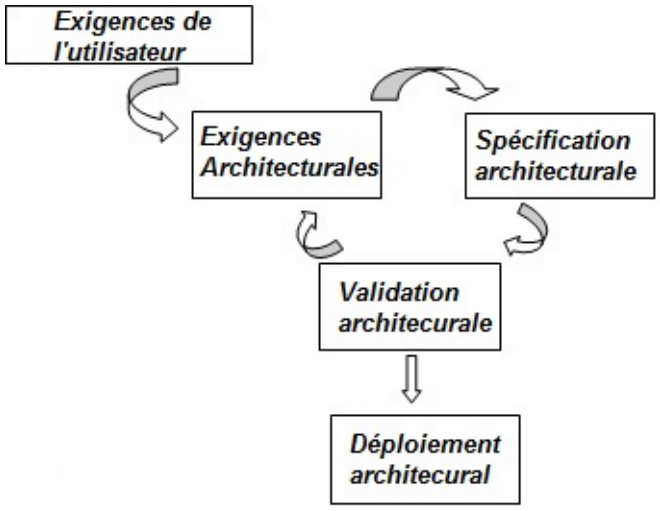

FIGURE 1.1 – *Cycle de Vie d'une Architecture Logicielle.*

Le cycle de vie de développement d'un logiciel suit un certain nombre d'étapes qui peuvent être exprimées au niveau architectural comme suit :

Cahier des charges : La conception architecturale permet de déterminer ce que l'on peut bâtir, et quelles exigences des utilisateurs sont raisonnables (figure 1.1). Souvent, une esquisse architecturale est nécessaire pour évaluer la viabilité du produit. Par exemple, un avant-projet architectural peut dire si un temps de réponse subséquent est une exigence réalisable sur un nouveau système client-serveur.

Conception : L'architecture logicielle est une forme de spécification ou de conception de haut niveau. Elle détermine généralement la première étape, et la plus critique, la décomposition du système. Le système, à ce stade, est vu comme une collection d'entités ou de modules communicants. Les éléments, à considérer ici, sont des composants d'exécution, structures des composants et des connecteurs, y compris la communication des processus, la concurrence (parallélisme), la production et la consommation de données partagées ainsi que la communication client-serveur. Un système, sans une architecture bien conçue, est probablement voué à l'échec.

Implémentation : Comme indiqué précédemment, une architecture est souvent un modèle pour la conception détaillée et la mise en œuvre. Les composants dans une architecture représentent généralement les sous-systèmes au niveau implémentation, où les interfaces architecturales correspondent aux interfaces fournies par cette implémentation. Les modules composant le système sont définis ici en termes de fichiers, classes..etc. L'architecture guide l'affectation de tâches et le partage du travail selon/entre les équipes de développement.

Déploiement : Cette phase définit les structures d'allocation nécessaires au déploiement du système. Ces structures de déploiement sont des composants matériels, logiciels et de communication. Au niveau architectural, le déploiement s'intéresse aux agencements logiciels-matériels possibles de ces composants ainsi qu'à la satisfaction des dépendances exprimées.

Réutilisation : La plupart des systèmes présente des structures régulières qui sont des instances de styles réutilisables. À titre d'exemple, les systèmes d'information centrés sur les données sont souvent conçus comme systèmes client-serveur 3-tiers. Plus généralement, les architectures logicielles sont un élément clé pour la réutilisation d'éléments logiciels. Ces systèmes exploitent les régularités d'architecture (et de codage) à travers une famille de systèmes pour permettre de concevoir et de créer de nouveaux systèmes à faible coût

en spécialisant un cadre général pour créer un produit particulier. Une architecture peut être vue comme un modèle réutilisable qui constitue le cœur d'une ligne de produits (Bass et al. 2012).

Maintenance : Les architectures logicielles facilitent la maintenance en clarifiant la conception du système et elles permettent aux responsables de comprendre l'impact des changements. Comme la maintenance peut représenter plus que la moitié du coût durant la vie d'un système, et puisque une partie importante de la maintenance consiste tout simplement en la compréhension du système dans le but de faire un changement souhaité, les architectures logicielles peuvent jouer un rôle important dans la maintenance de ces systèmes.

Adaptation durant l'exécution : De plus en plus, les systèmes sont censés fonctionner en continu. Des mécanismes automatisés pour détecter et réparer les défauts du système pendant son exécution apporteront probablement plus de capacités pour les systèmes futurs. L'architecture logicielle peut jouer un rôle clé dans l'auto-adaptation, en fournissant un modèle de réflexion qui peut être utilisé comme une base pour la réparation automatique.

Dans Bass et al. (2012), les auteurs ont cité une douzaine de raisons et intérêts incitant à utiliser une architecture logicielle en montrant l'importance et les atouts de ce concept. Certaines de ces raisons ont été déjà abordées dans les paragraphes précédents, nous résumons les plus importantes dans ce qui suit :

1. L'analyse d'une architecture permet une prédiction précoce de la qualité d'un système.
2. Une architecture documentée améliore la communication entre les parties prenantes.
3. Une architecture définit un ensemble de contraintes sur la mise en œuvre ultérieure.
4. L'architecture sert à dicter la structure d'un organisme, ou vice versa.
5. Une architecture peut servir de base pour un prototypage évolutif.
6. Une architecture est l'artefact clé qui permet à l'architecte logiciel et le gestionnaire de projet de raisonner convenablement sur le coût et les délais.
7. Un développement basé-architecture, focalise l'attention sur l'assemblage des composants, plutôt que sur leur création.

8. En limitant les alternatives de conception, l'architecture permet de canaliser la créativité des développeurs, ce qui réduit la complexité de la conception et du système.

D'après Medvidovic (1996), les architectures logicielles sont des entités multidimensionnelles qui peuvent être bien comprises lorsqu'elles sont vues à partir des quatre niveaux d'abstraction : la fonctionnalité interne d'un composant, les interfaces exportées par le composant vers le reste du système, l'interconnexion des éléments architecturaux et les règles du style architectural correspondant.

Un formalisme de description d'architecture logicielle idéal, selon Garlan et M.Shaw (1994), doit satisfaire quatre critères essentiels, il doit être :

- Expressif : le formalisme doit permettre une modélisation simple, sans difficultés ni bricolages, des caractéristiques d'une architecture logicielle,
- Vérifiable : certaines descriptions d'architecture logicielle notamment dans les systèmes critiques où la correction doit être garantie par des techniques de vérification ou de preuve, le formalisme de modélisation utilisé doit d'une part avoir une sémantique rigoureusement définie, et d'autre part posséder de bonnes propriétés de composition et d'abstraction,
- Exécutable : le formalisme doit avoir un aspect exécutable pour pouvoir générer des programmes à partir des modélisations architecturales. Ainsi, l'architecture peut être traitée par des outils de génération automatique de code et des tests pour obtenir une application exploitable,
- Évolutif : le formalisme doit permettre la description de l'aspect statique de l'architecture mais doit aussi offrir des moyens pour suivre son évolution et sa dynamique.

Plusieurs Langages de Description d'Architectures (ADLs : *Architecture Description Languages*) ont vu le jour suite à des nombreux travaux de recherches qui visent à offrir un formalisme et un cadre adéquat pour la description d'une architecture et également un moyen pour profiter des avantages de ce concept. Le nombre élevé de ces ADLs ne facilite pas le choix et la sélection d'un ADL parmi bien d'autres.

1.3 Langages de Description d'Architecture

C'est vers les années 90 que les premiers travaux sur la description des architectures ont commencé à émerger. Ces recherches ont donné naissance à un nouveau type de langages : langages de description d'architectures. Ces langages ont donc la vocation de décrire une architecture avec toutes ses spécificités. En raison de la nature spécifique du domaine de certains ADLs, leur approche dépend de la façon dont les concepteurs du langage perçoivent les besoins sémantiques de la famille d'architectures logicielles qu'ils modélisent. D'un autre côté, les ADLs existants décrivent l'architecture mais chacun met l'accent sur l'une des étapes du processus de développement d'une architecture, à savoir, *spécification*, *implémentation* et *déploiement*. Dans ce qui suit, nous présentons la classification la plus utilisée des ADLs, fondée sur le critère d'abstraction.

1.3.1 ADLs Abstraits

L'idéal consiste à proposer à l'architecte du logiciel un modèle de haut niveau du système qui se concentre sur les éléments caractéristiques d'une description d'architecture. Cette abstraction doit lui permettre de travailler avec un formalisme simple, facilement compréhensible par l'ensemble des acteurs d'un projet informatique y compris l'architecte lui même. Néanmoins, cette abstraction du langage est généralement traduite par une abstraction de la plateforme d'exécution. Donc, aucune information n'est fournie sur les éléments de la plateforme.

Les ADLs classés dans cette catégorie sont aussi appelés ADLs formels. Ils supportent les notions de composants et de connecteurs de manière abstraite sans indiquer à quoi ils correspondent dans le système réel. Ils sont caractérisés par leur capacité de formaliser la description du fonctionnement d'un système. Cependant, ces ADLs ont du mal à s'intégrer dans une démarche de génération automatique et leur usage peut s'avérer rapidement contestable lorsque que l'on veut passer à la phase de tests.
Comme exemple de ce type d'ADL, nous pouvons citer : *Wright* (Allen et Garlan 1996), *Rapide* (Luckham et al. 1995), et *ACME* (Garlan et al. 1997).

1.3.2 ADLs Concrets

Contrairement aux ADLs abstraits, les ADLs concrets ont la faculté de décrire l'architecture logicielle dans le but de la générer automatiquement. Généralement, des outils sont développés autour de ces langages, permettant

d'exploiter l'architecture obtenue et générer facilement un système fonctionnel dans un langage de programmation comme Ada, C ou Java à partir de sa description. Ces langages traitent également la phase déploiement de l'architecture, aspect souvent ignoré par les ADLs abstraits ou formels. Comme exemple de ce type d'ADL, nous pouvons citer : *UML2* (OMG 2005) et *AADL* (SAE 2008).

1.3.3 ADLs Restreints

Les ADLs restreints, comme leur nom l'indique, permettent de décrire uniquement l'assemblage de composants logiciels et la cohérence de l'application sans sémantique opérationnelle forte.
Dans cette catégorie, nous recensons les langages : *ArchJava* (Feiler et al. 2006), *Fractal* (Bruneton et al. 2006). Ils existent des outils autour de Fractal qui permettent la génération automatique du code tels que : *Julia* (Java) et *Cecilia* (langage C).

1.3.4 Critères d'évaluation des ADLs

Plusieurs travaux dans la littérature se sont penchés sur la comparaison et la classification des ADLs existants selon différents critères. On y trouve ceux qui les comparent selon leur capacité à décrire les éléments d'une architecture logicielle en termes d'interface, sémantique et contraintes d'utilisation d'un composant (Medvidovic et Taylor 2000), ou selon l'intégration d'un model-checker pour l'évaluation de l'architecture modélisée (Zhang et al. 2010) ou bien, selon l'aspect formel qu'ils engendrent (Choutri 2011).

En plus de ces critères, d'autres éléments peuvent aider à faire un choix réussi d'ADL pour un contexte bien défini. Mais au préalable, il faudra trouver des réponses à plusieurs questions pour faciliter la prise de décision, telles que :

- Le langage que l'on désire utiliser, est-il destiné à un domaine spécifique ayant des particularités bien définies ? ou simplement à un domaine général (générique) ? Il est important de savoir pour quel type de domaine l'architecture sera décrite. Les ADLs spécifiques, tel que : Fractal, permettent de décrire correctement et pleinement les spécificités du domaine auquel ils sont dédiés. Les ADLs génériques tel que UML, ont l'avantage d'être extensibles à plusieurs domaines.
- Est-il préférable d'utiliser un langage *standard*, *ouvert* ou plutôt *propriétaire* ? Un ADL standard est normalisé, il bénéficie d'outils développés

et disponibles facilitant son exploitation tels que UML et AADL. Les ADLs ouverts ont les caractéristiques publiées et dont l'acquisition et l'extension sont faciles tel que : AADL. Les ADLs propriétaires sont par contre commercialisés, coûteux mais souvent efficaces.

– Est-il préférable d'utiliser un langage graphique ou textuel (suffisamment de détails) ? Une architecture décrite par un ADL graphique est rapidement compréhensible par rapport à celle décrite via un ADL textuel. Toutefois, les ADLs textuels sont plus expressifs et permettent de décrire le moindre détail de l'architecture, particulièrement les propriétés et les contraintes.
– Est ce qu'il prend en compte l'aspect évolutif de l'architecture (architecture dynamique) ? Les ADLs peuvent offrir des mécanismes qui prennent en charge la dynamique d'une architecture à travers la définition de différentes configurations dans une seule spécification.
– Est-il capable de prendre en charge la description des mécanismes de déploiement (les composants de la plateforme d'exécution) ? Enfin, le dernier critère concerne la prise en compte du déploiement d'une architecture logicielle sur une infrastructure physique. Ce critère, souvent négligé dans les modèles de composant, est une information importante pour permettre le bon déroulement de l'application et l'extension des modèles de composant vers l'expression de la qualité de service (Barais 2005).

Les réponses à ces questions permettront de bien cerner les caractéristiques de l'ADL qui répondra le mieux aux exigences de l'architecte et de l'architecture désirée.

1.4 Tâches d'un Processus de Déploiement Architectural

Le déploiement d'un logiciel est un processus assez complexe. Certaines descriptions le réduisent à une simple opération d'installation sur une machine cible d'un logiciel fini et mis dans un paquetage, généralement décrit par le producteur du logiciel (Carzaniga et al. 1998).
Le déploiement de logiciels est un processus composé d'un certain nombre d'activités interdépendantes dont la réalisation d'une version du logiciel, sa configuration, son installation dans l'environnement d'exécution, et son activation (voir 1.2). Il comprend également les activités de post-installation, y compris le contrôle (suivi), la désactivation, la mise à jour, la reconfiguration

ou l'adaptation, le redéploiement et même le reploiement du logiciel éventuellement après une désactivation (Carzaniga et al. 1998).

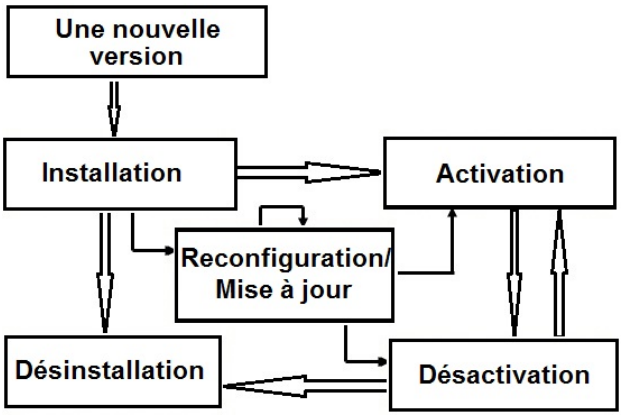

FIGURE 1.2 – *Tâches du Processus de Déploiement d'un Logiciel.*

Un administrateur de déploiement d'une application à base de composants doit souvent faire face à une triple hétérogénéité (Dubus et al. 2008) qui peut se manifester par :

1. Une hétérogénéité matérielle qui peut être traduite par l'hétérogénéité du système d'exploitation, protocoles d'accès à distance, ou du mécanisme de transfert de fichiers,
2. Une hétérogénéité des paradigmes utilisés tels que : *Objet*, *Aspect*, *Composant*, *Service*, etc.
3. Une hétérogénéité des implémentations : *EJB*[1], *CCM*[2] et autres pour les composants.

Le processus de déploiement d'une architecture logicielle suit, en général, les principales étapes du processus de déploiement d'un logiciel, pourtant il est tout à fait différent : il opère à un niveau d'abstraction élevé. Les préoccupations ne sont pas les mêmes et les actions aussi. Le fait de ne pas prendre en considération les détails technologiques fins d'un déploiement d'architecture offre plus de liberté et rend le processus moins complexe.
Le déploiement architectural fournit une base pour la compréhension et l'analyse de la distribution physique du système à travers un ensemble de nœuds de traitement dans le système, y compris la distribution physique des processus et des threads. Il peut aussi s'avérer nécessaire pour se concentrer sur les activités de structuration d'un système. Différentes solutions architecturales ont tendance à donner des résultats, de différents degrés d'ajustement, aux exigences de divers systèmes. Évaluer les alternatives et

1. Enterprise JavaBeans
2. Corba Component Model

analyser les compromis architecturaux sont un complément important à la phase de structuration.

Dans ce qui suit nous exposons les principales tâches d'un processus de déploiement architectural (voir figure 1.3). Nous ne pouvons pas nous empêcher de s'inspirer des tâches du processus de déploiement d'un logiciel pour définir celles décrivant le déploiement architectural. D'ailleurs, les experts décrivent le cycle de vie architectural en utilisant les mêmes phases du cycle de vie d'un logiciel (voir figure 1.1). Nous essayons, toutefois, de mettre en évidence le caractère architectural dans chaque phase du cycle de vie de l'opération de déploiement d'une architecture.

1.4.1 Sélection

Cette phase consiste à sélectionner les composants et les connecteurs devant être combinés pour construire l'application logicielle à installer sur une plateforme d'exécution. Le choix doit être fait de telle sorte que les interactions de ses éléments produisent le comportement attendu du système. Les dépendances doivent être décrites explicitement pour pouvoir gérer les étapes de déploiement et détecter les situations de conflit. Par exemple, dans le cas d'une application Client-Serveur, cette phase consiste en la sélection des composants de type *Client* et ceux du type *Serveur* ainsi que le type canal pour acheminer leurs communications.

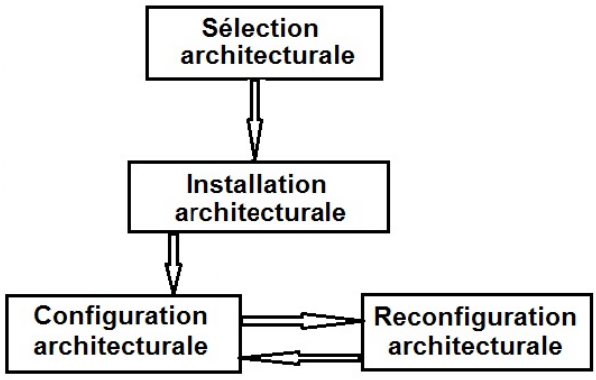

FIGURE 1.3 – *Tâches du Processus de Déploiement Architectural*

1.4.2 Installation

Cette phase est souvent confondue avec le processus de déploiement global. Elle consiste à choisir les sites d'installation et vérifier les dépendances architecturales. Les dépendances et les conflits entre les composants doivent être évalués, vérifiés et corrigés, si possible, pour mener à bien le déploiement

et garantir le bon fonctionnement des entités existantes et celles nouvellement installées.
Par exemple, pour le cas d'une application Client-Serveur, cette phase consiste à spécifier les sites physiques responsables de l'exécution des deux parties de l'application, étudier les dépendances logicielles/logicielles, logicielles/matérielles et les exigences en termes de capacité mémoires, vitesse des processeurs (particulièrement pour le *Serveur*), délai de propagation des communications, etc.

1.4.3 Configuration

La phase de configuration d'un logiciel a lieu entre les activités de création d'instances de sous-composants et de démarrage de l'exécution. Au niveau architectural, cette phase décrit la topologie du système considéré en termes des connexions entre les composants et les connecteurs intervenants via leurs interfaces. Une configuration est une disposition spécifique de composants et de connecteurs destinés à résoudre un problème spécifique. Elle est nécessaire pour le bon fonctionnement d'une application. Il est nécessaire de préciser que la spécification des instances de composants et connecteurs n'est pas suffisante pour définir une configuration architecturale. En effet, le même ensemble de composants/connecteurs peut constituer plusieurs topologies différentes et par conséquent former autant de configurations en fluctuant les connexions.
Une configuration décrit une instance possible d'un style architectural. Plus précisément, le même ensemble d'instances de composants inter-connectées différemment peut représenter des instances de styles architecturaux différents. À ce niveau, c'est la topologie de l'application qui est esquissée. Les connexions entre composants sont établies. Un schéma de l'application est alors défini. Cette étape permet d'évaluer la taille de l'application et sa structure hiérarchique.

1.4.4 Reconfiguration

Cette tâche caractérise le fait qu'une architecture logicielle peut changer en cours de fonctionnement pour s'adapter à un nouveau contexte d'exécution. Chaque changement décrit une nouvelle configuration de l'application. Il peut être défini au préalable et exécuté à la demande. Une reconfiguration définit les changements à apporter sur l'architecture par l'ajout, la suppression ou le remplacement d'un composant ou d'un connecteur (Allen 1997).Par exemple, pour l'exemple de l'application Client-Serveur, l'ouverture d'une

connexion avec un nouveau client est considérée comme une reconfiguration où l'on ajoute une nouvelle instance du composant Client et du connecteur associé. De même, pour la fermeture d'une connexion avec un client existant.

1.5 Approches pour le Déploiement d'un Logiciel

Quelques années auparavant, les applications logicielles étaient des systèmes autonomes (sans aucun lien avec d'autres applications logicielles), leur déploiement sur un seul ordinateur se fait facilement.
Ces dernières années, et avec des avancées significatives en matière de technologies de développement logiciel, il est maintenant possible d'avoir des applications logicielles complexes, qui comprennent un grand nombre de composants logiciels hétérogènes répartis sur un vaste réseau d'ordinateurs avec différentes capacités de calcul (Heydarnoori et Binder 2011). Habituellement, l'environnement dans lequel ces systèmes sont déployés est dynamique : n'importe quelle machine du réseau peut se bloquer, les liaisons réseau peuvent temporairement échouer, et ainsi de suite.

Un processus de déploiement doit assurer des opérations d'installation et de configuration correctes pour le bon fonctionnement du système. La qualité de ce processus peut être mesurée qualitativement par (Talwar et al. 2005) :

- la possibilité d'automatiser le processus de déploiement, y compris sa capacité d'adaptation aux changements (tels que les échecs ou la charge),
- la robustesse, exprimée en termes de configurations incorrectes,
- la capacité d'exprimer des contraintes, des dépendances et des modèles, et
- la facilité de compréhension à la première utilisation de l'outil de déploiement.

La spécification du déploiement d'un logiciel sur une plateforme d'exécution est un processus complexe qui peut être traduit par une suite d'activités allant de la production d'une nouvelle version de l'application jusqu'à sa désinstallation (voir figure 1.2). Il ne se réduit pas uniquement, à l'action d'installation des entités logicielles sur une plateforme matérielle pour assurer un bon fonctionnement de l'application (Parrish et al. 2001).
Pour pouvoir exécuter des applications, leurs composants doivent être instanciés sur des ressources matérielles adéquates dans leurs environnements cibles, tout en préservant les besoins des utilisateurs et les contraintes spéci-

fiées.

Plusieurs approches ont été adoptées durant les dernières années pour décrire sans ambiguïté le processus de déploiement d'une application à base de composants sur une plateforme d'exécution. Certaines sont à intérêt technologique et englobent tous les travaux qui spécifient le processus de déploiement de façon ad-hoc pour des plateformes particulières, telles que *J2EE*, *.NET* et *CORBA*, elles ont un aspect plutôt industriel. Les travaux à caractère académique supportent plus les aspects d'abstraction et de généricité dans la description des applications et des plateformes considérées telles que Sofa (Bures et al. 2006) et D&C (OMG 2006).

Nous recensons, dans ce travail, quatre approches de description du processus de déploiement d'un logiciel existent. Elles se différencient par la façon d'exécuter ce processus : manuelle, basée-script, basée-langage et basée-modèle.

1.5.1 Manuelle

Dans cette approche, les actions de déploiement consistent à distribuer les paquetages du logiciel ; se connecter à chaque nœud ; configurer manuellement l'application en spécifiant ses paramètres et celles du système cible puis tester et vérifier le bon déroulement. L'administrateur chargé de cette activité doit :

1. maitriser le déploiement de chaque logiciel,
2. l'adapter aux propriétés des machines (qui peuvent être plusieurs),
3. l'exécuter en respectant les contraintes et l'ordre de ses dépendances,
4. le tester pour s'assurer du bon déroulement du logiciel et
5. en cas d'erreur, la détecter et la corriger si possible.

Pour les grandes applications distribuées à base de composants avec de nombreuses contraintes et exigences, il est difficile de réaliser manuellement le processus de déploiement. Cette approche peut être rapidement abandonnée, par exemple, dans le cas des *grids*(grilles informatiques) qui peuvent contenir des milliers de machines inter-connectées. Des techniques et outils automatisés deviennent alors nécessaires, car le déploiement de tels systèmes met en jeu, généralement, de nombreuses technologies hétérogènes.

1.5.2 Basée script ou code

Cette approche consiste à développer des modèles d'installation et réaliser des scripts de démarrage. Au moment du déploiement, l'administrateur renseigne l'application avec des attributs spécifiques du client et lance des workflows qui invoqueront le module de distribution, d'installation et de lancement. La découverte des conflits et des erreurs ainsi que la réaction aux changements se fait à ce moment là d'une façon ad-hoc. La caractéristique principale de cette approche est qu'elle est plus directe et ne requiert aucun outils.

Les technologies des services de déploiement se sont toujours basées sur des scripts et des fichiers de configuration avec une capacité minimale d'expression des dépendances, de documentation et de vérification des configurations. Il en résulte parfois des difficultés à identifier et des configurations erronées.

Actuellement, la majorité des opérations de déploiement se font en exécutant des scripts. Souvent ses scripts sont empaquetés avec le logiciel ou téléchargés en cas d'une installation en ligne. L'utilisateur ne se rend compte du résultat de déploiement qu'à la fin. Soit l'application est bien installée et prête à être utiliser, soit elle ne l'est pas et parfois elle peut même provoquer des erreurs sur la machine cible dans le cas de remplacement de fichiers existants par de nouvelles versions ou simplement leur écrasement.

1.5.3 Basée Langage

Elle consiste à développer des modèles de déploiement via des langages spécifiques. Au moment de déploiement, l'administrateur est chargé de renseigner le modèle par des attributs spécifiques du client et le lancement d'un workflow d'installation et de vérification. En cas d' erreur, c'est l'administrateur qui décide sur le chargement du composant adéquat.

Certains langages, dédiés à cette activité, offrent des directives qui permettent de réaliser le déploiement. Il existe de nombreux langages utilisés par la technologie des procédés, ces langages sont regroupés sous le terme de langage de modélisation de procédés, les PMLs (Process Modeling Languages) (Lestideau 2003). Les PMLs sont des langages exécutables décrivant un processus logiciel comme un programme informatique en utilisant un langage de programmation tels que : Ada, Prolog, etc.
Une version récente de ce type de langage, et en guise d'une meilleure com-

préhensibilité, les nouveaux PMLs utilisent une notation graphique tel que le standard *SPEM* d'*OMG*. Le processus de déploiement donc est décrit par un PML comme étant un plan de procédé où l'on peut vérifier les contraintes et les dépendances. La politique de déploiement définie exactement qui doit installer quoi, et elle est décrite comme un ensemble d'assertions de la forme suivante (Merle 2005) :

IF (condition) INSTALL «nom-fichier»

où «nom-fichier» est un fichier qui contient le code du module à installer, ou par exemple :

IF (license == «evaluation») INSTALL «nom-fichier1»

qui indique que pour une certaine version d'évaluation, installer un fichier particulier ; soit «nom-fichier1».

Malgré l'aspect automatique que révèle l'usage d'un script ou un langage particulier facilitant la prise en charge des spécificités du processus de déploiement, la politique est totalement décrite par le producteur de l'application ou l'administrateur du système, c-à-d, ce processus reste toujours sous la responsabilité d'un humain.

1.5.4 Basée Modèle

Les tâches de déploiement sont assez complexes et fortement propices aux erreurs. D'après Medvidovic et Sam (2007), devant des situations complexes de déploiement, une solution évidente consiste à prendre le problème des mains d'un architecte humain et construire des modèles puissants et complets de l'opération de déploiement d'un système. Formaliser les modèles de déploiement permet leur traitement automatisé. Ces modèles devront décrire au moins :

- les éléments du système (hôtes matériels et liaisons de réseau, des composants logiciels et des connecteurs) et leurs multiples paramètres ;
- les fonctions ou d'autres façons de quantification des paramètres du système ;
- les fonctions ou d'autres façons de quantification des dimensions de qualité de service désirées ;
- les utilisateurs et leurs préférences, et
- des contraintes sur les éléments du système et/ou de leurs paramètres.

Cette approche consiste à élaborer des schémas de gestion des modèles de déploiement, créer des instances de modèles de dépendances et de ressources. Au moment du déploiement, une sélection des packages du modèle

concernant la meilleure pratique, analyser et mettre à jour un modèle unifié, puis lancer l'installation et vérifier les événements de notification. En cas de changement ou d'erreur, la détection est automatique et le lancement de l'auto-adaption ou l'auto-réparation.

Il y a plusieurs travaux qui se sont intéressés à la résolution des problèmes de déploiement au niveau modèle et même méta-modèle. Le but étant d'atteindre un niveau d'abstraction qui permet de mieux raisonner sur les contraintes de déploiement.

Un méta-modèle pour l'automatisation du déploiement des applications logicielles fut proposé par Merle (2004). L'auteur a présenté le modèle "*ORYA*" basé sur la réutilisation et l'intégration des outils de déploiement existants. Ce travail traite plus particulièrement l'activité de sélection avec l'utilisation d'un modèle de composant générique. Il met en place un framework basé sur un système d'annotations et de règles.

Le langage UML2 (OMG 2005), en tant que langage de modélisation, dispose d'un diagramme de déploiement qui sert à décrire la disposition des entités logicielles ou composants sur des sites représentant l'infrastructure d'exécution. Le diagramme de déploiement peut être considéré comme un modèle de déploiement de l'application associée.

Flissi et Merle (2005) se sont appuyés sur l'usage des intergiciels à composants afin d'automatiser le déploiement des applications. Un intergiciel, étant une couche qui vient s'interposer entre l'application et le matériel. Cette couche joue comme rôle principal de masquer les hétérogénéités matérielles. Les auteurs de ce travail proposent une démarche dirigée par les modèles (à la MDA [3] de l'OMG) pour la construction de *machines de déploiement* des intergiciels à composants. Cette démarche introduit un profil *UML* de workflow permettant de définir des modèles de déploiement indépendants des intergiciels à composants cibles. De tels modèles sont ensuite raffinés pour chaque intergiciel cible, pour être à la fin projetés, via des transformations, vers diverses plateformes d'exécution. Les différents modèles de cette démarche sont illustrés à travers la construction d'une machine de déploiement de composants *CORBA* mise en œuvre avec le modèle de composants *Fractal*.

Dubus et al. (2008) présentent le modèle *DeployWare*, une approche à base de modèles pour le déploiement de systèmes distribués complexes.

3. MDA : Model Driven Architecture

Cette approche, en deux parties, repose sur un méta-modèle . La première partie permet de décrire les propriétés, les dépendances et actions à effectuer pour déployer des logiciels. La seconde permet d'assembler des instances de logiciels. Ces deux parties sont réalisées de manière à rendre possible la vérification des procédures de déploiement des systèmes. Les modèles *DeployWare* sont projetés vers une plateforme d'exécution à base de composants qui gère automatiquement l'hétérogénéité des machines et l'orchestration des dépendances.

Dans leur travail de Heydarnoori et Binder (2011), et partant du principe que les applications deviennent de plus en plus larges, complexes et distribuées, et que leur besoin en communication ne cesse d'accroitre, ils ont présenté alors une approche à base de graphes pour le traitement du déploiement des communications requises par les composants de l'application sur les moyens de communication disponibles sur les hôtes de l'environnement cible. L'idée consiste à décrire les interactions des composants logiciels d'une application par un premier graphe. Un deuxième graphe sera utilisé pour représenter le réseau des canaux de communication de l'environnement distribué représentant l'infrastructure matérielle réelle. Une opération de correspondance entre les deux graphes sera ensuite effectuée. Si la correspondance est parfaite alors le déploiement peut être jugé comme adéquat.
La planification du déploiement est alors définie comme la cartographie du graphe d'application au graphe de l'environnement cible de sorte que le paramètre de QoS désiré est maximisé. Comme exemple de cette approche, ils montrent comment cette cartographie peut être efficacement utilisée pour déterminer le coût minimum et maximum du déploiement.

1.6 Synthèse

Il n'existe pas actuellement des travaux de comparaison sur l'état de l'art concernant le processus de déploiement d'une architecture logicielle. La figure 1.4 montre le coût d'automatisation de ce processus pour un logiciel selon différentes approches. Pour un déploiement basé modèle, il est très élevé lors des premières phases de développement en comparaison avec les autres approches. Développer les outils de déploiement, l'apprentissage, et la création de modèles viennent à un coût initial élevé. Ce coût sera vite payé dans les déploiements répétés et complexes ou lors du déploiement des systèmes à grande échelle.

La table 1.1 présente une comparaison faite dans le but de relever les différences entre les approches de déploiement d'un logiciel et elle est extraite

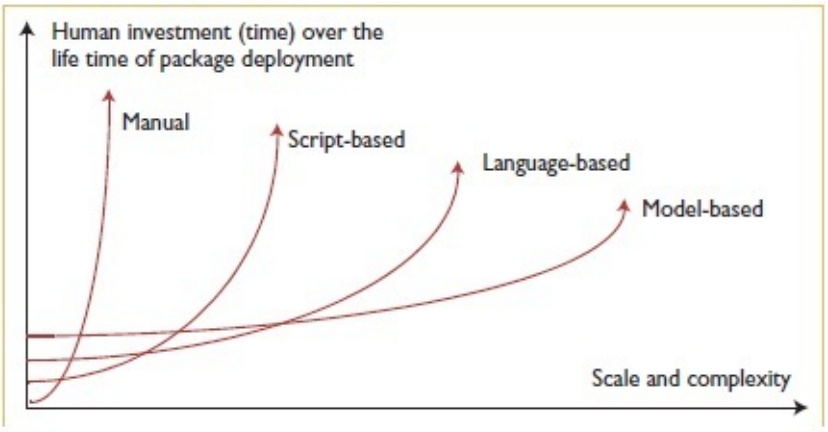

FIGURE 1.4 – *Coût du Déploiement par Approche (Talwar et al. 2005).*

essentiellement des travaux de Talwar et al. (2005).
Les approches basées scripts sont bien adaptées pour les déploiements à grande échelle. Celles basées langage se montrent plus appropriées aux applications des services de grande complexité. Enfin, l'approche basée modèle est considérée une approche automatisable, générique, auto-réparatrice, la plus expressive et la plus compréhensible. L'approche basée modèle est la plus adéquate pour modéliser le déploiement des systèmes dynamiques et pouvant subir des changements dynamiques en cours d'exécution.

Approches de déploiement				
Caractéristiques	Manuelle	Basée-script	Basée-Langage	Basée-Modèle
Concept de base	Langage humain	Configuration, fichiers, script	Langage déclaratif	Modèles et politiques
Automatisation	Néant	Basée événement	Gestion cycle de vie	Modèle automatisé
Auto-réparation	Néant	Minimale	Redéploiement + dépendances	+ Changement du modèle
Expressivité	Néant	Partielle	Significative	Complète
Compréhension	Néant	Basse	Élevée	Très élevée

TABLE 1.1 – *Comparaison des Approches de Déploiement*

D'après les résultats de comparaisons présentées dans la section précédente, le déploiement *basé-modèle* offre plus d'atouts et d'avantages par rapport aux autres approches particulièrement lorsqu'il est destiné à des applications dynamiques et à grande échelle.
Pratiquement la majorité des applications logicielles actuelles peuvent être classées dans des catégories (modèles) ou styles architecturaux tels que le

style *Client-Serveur*. Il serait très intéressant de pouvoir définir un modèle de déploiement par style architectural. Commencer à réfléchir à ce problème en amont et au niveau architectural, permettrait de pallier aux problèmes de déploiement d'une application et alléger la complexité de ce processus.

La formalisation d'un processus de déploiement au niveau architectural nécessite l'utilisation d'un formalisme mathématique pour décrire les différentes activités de ce processus ainsi que les informations de déploiements associées telles que les contraintes et les dépendances. Ce formalisme doit être capable de définir les particularités de ce processus concret, mais en même temps assurer un niveau d'abstraction élevé c-à-d, au niveau architectural. L'introduction d'un formalisme mathématique apportera une sémantique aux données et aux actions de ce processus et facilitera la détection des erreurs, la vérification et la validation. Ce qui représente une garantie de la sûreté et de la cohérence du déploiement.
À partir d'un modèle de déploiement architectural, différentes instances peuvent être produites. Ces instances servent pour simuler toutes les possibilités de déploiement puis choisir celle qui conviendrait le mieux selon des critères prédéfinis.

1.7 Conclusion

La génération des plans de déploiement pour une application à base de composants nécessite de préciser les entrées suivantes : l'application à base de composants en cours de déploiement, l'environnement distribué dans lequel l'application sera déployée et des contraintes relatives à ce déploiement défini par l'utilisateur.
L'adoption d'un processus de déploiement basé modèle semble la solution la plus adéquate pour deux raisons. Premièrement pour faire face à la complexité de ce processus qui ne cesse d'accroitre. Deuxièmement, du fait que la majorité des logiciels actuels peuvent être classés dans des styles architecturaux connus. Ceci permet de capitaliser l'effort en définissant un modèle de déploiement par style architectural.

La spécification d'un modèle de déploiement architectural permettra de se positionner à un niveau d'abstraction élevé et par conséquent pouvoir raisonner sur la cohérence du résultat de ce processus et rectifier les objectifs visés très tôt dans le cycle de développement d'un logiciel.

L'intervention d'un formalisme mathématique pour décrire le modèle de

déploiement architectural apportera une sémantique aux actions et aux données manipulées et aidera à la vérification et la validation des résultats.

2 Langage de Description d'Architectures AADL

Sommaire

Anybody who comes to you and says he has a perfect language is either naive or a salesman.

B. Stroustrup, *créateur du langage C++.*

2.1 Introduction

AADL [1] (Architecture Analysis and Design Language) (SAE 2008) normalisé par SAE(Society of Automotive Engineer), est un langage de description d'architectures logicielles. C'est le successeur du langage METAH (Vestal 2000), premier ADL particulièrement dédié aux systèmes avioniques. Il fut développé par Honeywell depuis 1991 pour le DARPA (Defense Advanced Research Projects Agency) et AMCOM (Aviation and Missile COMmand of US Army). C'est pour cette raison que la première appellation de AADL fut (Avionique Architecture Description Language).

AADL est un langage destiné à faciliter la conception et l'analyse de systèmes complexes, critiques et temps réel, comme ceux destinés à l'aéronautique, l'automobile et le spatial. La première version du standard date de 2004, et le langage continue d'évoluer régulièrement depuis ce temps.

Dans ce chapitre, nous présentons la syntaxe de ce langage en mettant l'accent sur ses particularités qui font de lui un langage concret.

2.2 Éléments Architecturaux

Une spécification en langage AADL peut être exprimée avec différentes formes (représentation), soit en texte brut, en XML, ou avec une représentation graphique.
La syntaxe XML est introduite dans le standard dans le but de faciliter l'interopérabilité entre les différents outils et permettre l'exploitation des spécifications AADL à travers des langages de programmation tel que JAVA. La représentation graphique venue pour pallier à la complexité du texte et malgré son exhaustivité, reste très peu expressive (Vergnaud 2006).

Cette multiplicité de représentations favorise l'utilisation d'AADL par de nombreux outils différents, graphiques ou non. Le développement de profils UML tel que MARTRE (Modeling and Analysis of Real-Time and Embedded systems) (Turki et al. 2010) permet également d'envisager l'intégration d'AADL au sein d'outils de modélisation UML.

Une architecture AADL est décrite comme un ensemble de composants logiciels (processus, thread, groupe de threads, données et sous-programmes)

1. www.aadl.info

qui s'exécute sur une plateforme d'exécution décrite par des composants matériels (processeurs, mémoires, périphériques et bus). Les interfaces des composants sont modélisées par des ports. Les connexions entre les différents composants ainsi que l'allocation des composants de la plateforme pour les composants logiciels sont ensuite spécifiés.

Sur le plan vocabulaire, AADL est très riche et a de grandes capacités d'expression. Une description AADL consiste en un ensemble de déclarations de composants pouvant être instanciées pour former un modèle d'architecture.

Ainsi, une déclaration abstraite d'un composant AADL est scindée en deux parties ; ***type*** et ***implementation***. La déclaration *type* d'un composant peut contenir des clauses définissant ses interfaces (*features*), ses flux (*flows*) possibles, etc. Par contre, une déclaration *implementation* spécifie la structure interne d'un composant en termes de sous-composants (*subcomponents*), connexions (*connections*) entre ces sous-composants, ou les modes (*modes*) pour représenter leurs états opérationnels alternatifs. Cette description peut être enrichie à travers des propriétés qui permettent de spécifier les aspects fonctionnels et non fonctionnels du système logiciel (Feiler et al. 2006).

La notion de composant en AADL est hiérarchique. Une implémentation d'un composant peut contenir des sous-composants. Un sous-composant est une *instance* d'une implémentation qui hérite les caractéristiques définies dans sa partie *type*.
Une déclaration *type* d'un composant peut être étendue par une autre déclaration *type* (figure 2.1) ou simplement implémentée par une ou plusieurs *inplémentations* qui peuvent à leur tour être étendues par d'autres *inplémentations* (Feiler et Gluch 2012, Feiler et al. 2006).

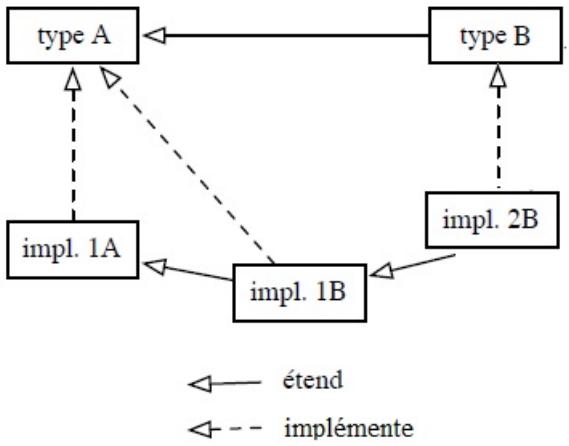

FIGURE 2.1 – *Relations entre Type et Implementation en AADL*

Toutes ces spécificités d'AADL permettent à l'architecte logiciel d'ex-

primer des exigences utilisateur et les paramètres de déploiement et de reconfiguration d'un système. De nombreux outils autour du langage sont disponibles, tels que : OSATE (AADL Team 2004), TOPCASED (Farail et al. 2006), Cheddar (Singhoff et al. 2008) et Ocarina (Lasnier et al. 2009). Ils permettent d'éditer des modèles AADL, de générer l'application logicielle du système à partir d'une description AADL ou de conduire diverses analyses. En particulier, OSATE, éditeur AADL sous Eclipse, offre la possibilité d'analyser par simulation le comportement du système opérationnel (i.e. la spécification AADL instanciée). Les éléments architecturaux de base de ce langage sont illustrés à travers l'exemple dans la section (2.2.5) édité et validé par l'outil *Osate 1.5*.
Une fois la correction de la spécification est vérifiée, le fichier XML correspondant est alors généré. Ce fichier XML peut être exploité pour des fins diverses telles que la représentation graphique, l'analyse et la vérification (figure 2.6).

Catégories de Composants AADL Un composant représente une entité matérielle ou logicielle qui fait partie d'un système modélisé en AADL. Un élément AADL a un *type*, qui définit son interface fonctionnelle. Le *type* de composant spécifie l'interface externe du composant que son implémentation doit satisfaire et avec qui d'autres composants peuvent interagir. Il contient des déclarations qui représentent les interfaces du composant, ses flux et ses propriétés.

Les interfaces d'un composant peuvent être des ports (*Ports*), des groupes de ports, des données, des composants (sous-programmes) contenus dans le composant qui sont accessibles de l'extérieur. Les ports et les sous-programmes d'un composant peuvent être connectés aux ports compatibles ou sous-programmes d'autres composants via des connexions *connections*, ceci représente un échange de données entre ces composants.

Dans AADL, nous pouvons rencontrer plusieurs types de composants, répartis en trois grandes catégories :

- les composants logiciels définissent les éléments applicatifs de l'architecture ;
- les composants de la plate-forme d'exécution modélisent les éléments matériels ;
- les composants systèmes regroupent différents composants en entités logiques pour structurer l'architecture.

2.2.1 Composants Matériels

Cette catégorie est dotée d'un ensemble de composants hardware essentiels et suffisants pour spécifier correctement et exhaustivement une plateforme d'exécution. On y trouve quatre classes de composants : Processeur (*Processor*), Mémoire (*Memory*), Bus de communication (*Bus*) et Périphérique (*Device*). Chacun de ces composants possède une notation graphique qui permet de le distinguer visuellement dans une configuration AADL (figure 2.2).

• Un composant de type *Processor* ou processeur (figure 2.2.a) est une abstraction du matériel ou logiciel responsable de l'ordonnancement et de l'exécution des threads. L'implémentation d'un composant de type processeur peut contenir des déclarations de sous-composants tel qu'une mémoire. Une propriété du processeur spécifiant le protocole d'ordonnancement est la suivante :

Allowed_Dispatch_Protocol : list of Supported_Dispatch_Protocols ;

• Un composant de type *Memory* ou mémoire (figure 2.2.b) représente un élément de la plate-forme d'exécution qui stocke des images binaires. Un *type* de composant mémoire peut contenir des déclarations concernant des accès au bus et leurs propriétés. Une implémentation de la mémoire peut contenir une clause *modes* et également d'autres propriétés telle que.

Memory_Protocol : **enumeration** (*read_only*, *write_only*, *read_write*)=> *read_write* ;

• Un composant de type *Bus* (figure 2.2.c) est une composante de la plateforme d'exécution qui permet d'échanger des commandes et des données entre mémoires, processeurs et périphériques. Ce composant de la plateforme représente un canal de communication, défini généralement par un matériel et des protocoles de communication.
Parmi les propriétés que l'on peut spécifier pour un bus, on cite :

Allowed_Connection_Protocol : **list of enumeration** (*Data_Connection*, *Event_Connection*) ;

• Un composant de type *Device* ou périphérique (figure 2.2.d) ou simplement périphérique, représente une composante matérielle de la plateforme d'exécution qui réalise une interface avec l'environnement extérieur. Les dispositifs peuvent être équipés d'un processeur, d'une mémoire et des logiciels qui ne sont pas explicitement modélisés. Si le dispositif est associé à des

logiciels tels que les pilotes de périphériques qui doivent résider dans une mémoire et s'exécuter sur un processeur externe à l'appareil, ceci peut être spécifié par la valeur des propriétés rattachées au périphérique telle que :

Device_Dispatch_Protocol : **Supported_Dispatch_Protocols** => *Aperiodic* ;

FIGURE 2.2 – *Représentation Graphique des Composants Matériels AADL.*

2.2.2 Composants Logiciels

Cette catégorie comptent six classes de composants logiciels, On y trouve les processus (*Process*), fil d'exécution Thread (*Thread*), groupe de threads (*Thread group*), les sous-programmes ou appels des sous-programmes (*Subprogram* et *Subprogram Call*), les donnés (*Data*) et le service d'exécution prédéfini (Predeclared Runtime Service) (Farail et al. 2006) [2].

• Le type de composant *Data* (figure 2.3.a) représente un type de données dans le texte source. Sa structure interne, qui peut être les variables d'instance d'une classe ou les champs d'un enregistrement, est représentée par des sous-composants de données dans une implémentation du composant même. Ce composant est utilisé pour faire passer des messages à travers des ports de type *in/out data* ou *in/out event data* ou comme paramètres pour les appels des sous-programmes. Plusieurs types de données ont été spécifiés pour déclarer la nature d'un composant *data* tels que *aadlboolean*, *aadlstring*, *aadlinteger*, and *aadlreal*.
Exemple de propriétés d'un composant *Data* :

Source_Data_Size : taille en bits de la donnée dans le texte source.

• Un composant de type *subprogram* (2.3.b) constitue un point d'entrée d'exécution dans le texte source. Un sous-programme ne peut avoir aucun état interne (données statiques). Tous les paramètres et les points d'accès aux données externes doivent être explicitement déclarés dans la déclaration de *type* de sous-programme.
Comme propriétés pour ce type de composant, nous pouvons citer :

Compute_Execution_Time : intervalle de temps.

Source_Language : langage utilisé pour implanter le sous-programme.

2. Voir manuel de l'outil Osate

• Un fil d'exécution (*thread*) (2.3.c) représente un flux séquentiel de commandes qui exécute des instructions contenues dans une image binaire produite à partir du texte source. Un thread est considéré l'élément actif d'une spécification AADL et s'exécute toujours au sein de l'espace d'adressage virtuel d'un processus, c'est à dire, les images binaires qui constituent l'espace d'adressage virtuel doivent être chargées avant que tout thread puisse s'exécuter.
Une des propriétés spécifiques aux threads est la suivante :

Dispatch_Protocol : Aperiodic/Periodic/Sporadic.

Cette propriété permet de définir le protocole d'ordonnancement/activation d'un thread. Ils existent trois valeurs pour cette propriété, chacune détermine comment le thread va être exécuté. Un thread périodique est activé à intervalle de temps régulier spécifié par sa période. Les threads apériodiques sont activés lorsqu'ils reçoivent un événement sur un de leurs ports ou qu'une requête d'exécution d'un de leurs sous programme arrive. Les threads sporadiques ont le même comportement que les threads apériodiques, mais un temps minimum entre deux activations doit être respecté.

• Un groupe de threads (*Threads Group*) (2.3.d) est un type de composant organisationnel qui sert à regrouper logiquement les threads contenus dans un processus. Le *type*, d'un composant groupe de threads, spécifie les caractéristiques et les accès requis à travers lesquels les threads contenus dans un groupe de threads interagissent avec les composants externes.
Un groupe de threads ne représente ni un espace d'adressage virtuel ni une unité d'exécution. Par conséquent, un groupe de threads doit être contenu dans un processus.

Synchronized_Component : inherit aadlboolean => **true**.

• Le type de composant processus (*Process*) (2.3.e) représente un espace d'adressage virtuel. Cet espace d'adressage contient le code et les données associées aux threads et aux sous programmes du processus. La propriété processus de *Runtime_Protection* indique si cet espace d'adressage virtuel est protégé en exécution, c'est à dire qu'elle représente une unité de séparation de l'espace dont les restrictions sont appliquées à l'exécution. Une implémentation complète d'un processus doit contenir au moins un thread ou un groupe threads afin que celui-ci soit considéré opérationnel.
Une des propriétés de ce type de composant est :

Scheduling_Protocol : liste protocoles d'ordonnancement.

• En fin, les services d'exécution prédéfinis sont des annexes spécifiques à ce standard. Ils permettent de définir des sous-programmes pré-déclarés qui sont inclus avec chaque thread, et la mise en œuvre du processus. Les noms utilisés pour les composants explicitement déclarés, les caractéristiques, les connexions et les comportements doivent être différents des noms de tous les composants pré-déclarés du standard.

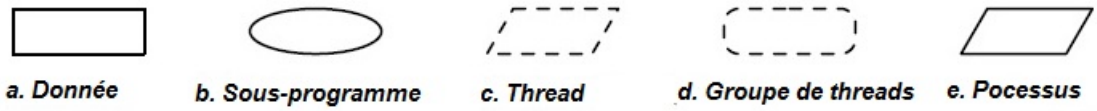

FIGURE 2.3 – *Représentation Graphique des Composants Logiciels AADL.*

2.2.3 Composant Système

Un composant Système (*System*) en AADL est un composant composite qui peut contenir d'autres composants. Il permet de spécifier tous les modules qui composent une application.

FIGURE 2.4 – *Représentation Graphique d'un Composant Système*

Le composant système joue le rôle de conteneur global, il peut intégrer tous les composants censés décrire une architecture logicielle ainsi que les composants matériels correspondants (figure 2.4).

Catégorie	**Type**	**Implémentation**
system	**Features :** -server subprogram -port -port group -provides data access -provides bus access -requires data access -requires bus access **Flow specifications** : oui **Properties** : oui	**Subcomponents :** -data -process -processor -memory -bus -device -system **Subprogram calls** : non **Connections** : oui **Flows** : oui **Modes** : oui **Properties** : oui

TABLE 2.1 – *Structure d'un Système AADL*

La table 2.1 présente la structure du composant *système* et les composants qui peuvent y être spécifiés directement. Notons que le composant thread ne peut être placé directement dans un composant système mais peut être introduit via un processus.

2.2.4 Propriétés et Annexes

Propriétés AADL : elles permettent de spécifier les caractéristiques de quasiment tous les éléments architecturaux AADL (composant, sous-composant, interfaces, etc.).
Par exemple, on peut spécifier le protocole d'ordonnancement d'un processeur, la période d'un thread, ou la bande passante d'un bus, etc. Ainsi, il est possible de décrire les contraintes applicables à une architecture en affectant des valeurs spécifiques aux composants décrits.

Annexes AADL : dans les modèles AADL, les annexes permettent d'ajouter des informations supplémentaires dans le modèle d'une application, exprimées dans une syntaxe indépendante et différente d'AADL (ex : OCL). Généralement, il s'agit d'informations concernant le comportement d'un composant.

2.2.5 Exemple d'une Architecture AADL

Pour illustrer les concepts et notations AADL, considérons dans cette section une spécification AADL générique modélisant une application classique : Émetteur/Récepteur (table 2.2). Cette spécification est constituée d'un composant de type système (*system*) noté *The_system*, de sous-composants logiciels (tel que : *The_process*) et matériels (tels que : *The_processor*, *The_memory* et *The_bus*), c'est le minimum requis pour pouvoir exécuter le processus. Une relation liant chaque élément logiciel à un élément matériel est prévue à l'aide des propriétés de la forme "*Actual...Binding*" (voir clause *properties* de la table 2.2).

L'implémentation *The_system.impl* dans table 2.2 constitue une implémentation possible du composant *The_system*.

La table 2.3 présente la déclaration des deux parties (*type* et *implementaion*) pour chaque composant impliqué dans l'exemple. En AADL, l'interface d'un composant peut être définie sous forme d'un port (*port*), d'un groupe de ports (*port group*), d'un accès à un bus de communication (*bus access*), ou encore l'accès à une donnée partagée (*data access*). Chaque composant peut avoir, une

```
system The_system
end The_system;
system implementation The_system.impl
subcomponents
   the_processor : processor Processor1.impl;
   the_process : process Process1.impl;
   the_memory : memory Memory1.impl;
   the_bus : bus Bus1.impl;
properties
Actual_Memory_Binding => reference the_memory applies to the_process;
Actual_Processor_Binding => reference the_processor applies to the_process.thr_S;
Actual_Processor_Binding => reference the_processor applies to the_process.thr_R;
Actual_Connection_Binding => reference the_bus applies to the_process.cnx;
end The_system.impl;
```

TABLE 2.2 – *Exemple de spécification AADL d'un système.*

définition récursive de ses sous-composants. Les spécifications AADL (table 2.3) permettent de présenter en particulier, les détails d'implémentation des composants matériels. Les propriétés déclarées donnent plus de précisions sur l'aspect opérationnel de ces composants. Enfin, une configuration représente un graphe de composants et de connecteurs.

Le composant logiciel *the_process* est une instance de type *Process1*, constitué de deux autres sous-composants de type thread, *thr_S* (émetteur) et *thr_R* (récepteur) (table 2.4). Ces threads interagissent via une connexion *cnx* (table 2.3), définie entre les ports d'événements (*event port*) *inPort* et *outPort*. D'autres détails d'implémentation des threads sont donnés par des propriétés, telles que la période et le temps d'exécution (voir table 2.4).

2.3 Aspects Dynamiques

Le langage AADL ne se limite pas à la description statique d'architectures. Il permet également de spécifier un certain dynamisme dans les spécifications AADL à travers le concept de *mode* et *transition entre modes*.

2.3.1 Modes

Les *modes* en AADL représentent des états de fonctionnement alternatifs et prédéfinis d'un composant ou de tout un système (Feiler et al. (2006)).

process Process1 **end** Process1 ; **process implementation** Process1.impl **subcomponents** thr_S : **thread** ThreadSender.impl ; thr_R : **thread** ThreadReceiver.impl ; **connections** cnx : **event port** thr_S.outPort -> thr_R.inPort ; **end** Process1.impl ;	**processor** Processor1 **features** busAcc : **requires bus access** Bus1.impl ; **end** Processor1 ; **processor implementation** Processor1.impl **subcomponents** mem : **memory** Memory1.impl ; **properties** Scheduling_Protocol => (RMS) ; **end** Processor1.impl ;
memory Memory1 **end** Memory1 **memory implementation** Memory1.impl **end** Memory1.impl ;	**bus** Bus1 **end** Bus1 **bus implementation** Bus1.impl **properties** Propagation_Delay => 5ms .. 5ms ; **end** Bus1.impl ;

TABLE 2.3 – *Spécification AADL des sous-composants de "The_system".*

L'utilisation de modes permet de décrire un système comme un ensemble fini de configurations qui représenteront des états successifs de l'architecture dans le temps. Ceci permet de modéliser des modifications dynamiques, mais prédéfinies, de l'architecture de l'application (Rolland 2008).

À chaque composant/connecteur quelque soit son niveau dans la hiérarchie de la spécification AADL, plusieurs modes de fonctionnement peuvent être définis traduisant par exemple, l'activation /désactivation d'un thread, le changement des valeurs des propriétés, le remaniement dans la topologie des connexions, etc. Cela représente la dynamique d'un point de vue logiciel. D'un point de vue plateforme matérielle, un mode peut traduire le remplacement d'un processeur par un autre, le basculement entre deux bus de communication en fonction du débit de la communication, etc.
AADL permet de décrire les conditions de changement de mode, afin de définir une machine à états pour l'implémentation du composant (Vergnaud 2006). Les modes permettent de modéliser la *reconfiguration* du système en fonction d'événements définis au niveau des composants. Seuls les événements *event ports* peuvent déclencher un changement de mode. Un mode peut

thread ThreadSender **features** outPort : **out event** port ; **end** ThreadSender ;	**thread implementation** ThreadSender.impl **properties** Period => 120ms ; Compute_Execution_Time => 30ms..40ms ; Dispatch_Protocol => (Periodic) ; **end** ThreadSender.impl ;
thread ThreadReceiver **features** inPort : **in event port** ; **end** ThreadReceiver ;	**thread implementation** ThreadReceiver.impl **properties** Period => 120ms ; Compute_Execution_Time => 30ms..40ms ; Dispatch_Protocol => (Aperiodic) ; **end** ThreadReceiver.impl ;

TABLE 2.4 – *Spécification AADL des threads : "ThreadSender" et "ThreadReceiver".*

être déclaré pour des données, thread, groupe de threads, processus, système, processeur, bus, mémoire, et les implémentations des périphériques.

La clause *Modes* dans une spécification sert à déclarer des modes qui peuvent y appartenir en utilisant la syntaxe suivante :

mode_identifier :[**initial**] mode [*mode_property_assocation**] ;

Dans une déclaration utilisant les modes, au moins un des modes doit être déclaré comme étant *Initial*.

La table 2.5 contient la déclaration des modes définis pour le composant *Process1* de l'exemple précédent dans l'implémentation *Process1.impl*. Deux modes (*mode1* et *mode2*) sont définis dont *mode1* est *Initial*. Le mot clé *in modes* est utilisé pour préciser que cette caractéristique n'est valable que dans ce mode.

2.3.2 Transitions de Modes

La clause *Modes* sert également à déclarer la transition d'un mode à un autre. Pour qu'une transition soit déclenchée, il faut la lier à l'arrivée d'un événement depuis un port de type *Event-port*. Une transition est en faite le passage d'un état opérationnel du système vers un autre ou d'une configuration vers une autre.

Dans l'exemple de la table 2.5, l'événement qui peut parvenir depuis le port *P_ev* va déclencher une transition de modes depuis le *mode1* vers le *mode2*. Ce passage entre modes permet au processus actif de basculer d'un

```
process Process1
features
        P_ev : in event port;
end Process1;
```

```
process implementation Process1.impl
subcomponents
        thr_S : thread ThreadSender.impl;
        thr_R1 : thread ThreadReceiver.impl1 in modes (mode1);
        thr_R2 : thread ThreadReceiver.impl2 in modes (mode2);
connections
        cnx1 : event port thr_S.outport->thr_R1.inport in modes (mode1);
        cnx2 : event port thr_S.outport->thr_R2.inport in modes (mode2);
modes
        mode1 : initial mode;
        mode2 : mode;
        mode1 -[P_ev]-> mode2;
        mode2 -[P_ev]-> mode1;
end Process1.impl;
```

```
thread implementation ThreadReceiver.impl1
 properties
        Compute_Execution_Time => 5 Ms .. 5 Ms;
        Period => 10 Ms;
end ThreadReceiver.impl1;
```

```
thread implementation ThreadReceiver.impl2
properties
        Compute_Execution_Time => 8 Ms .. 8 Ms;
        Period => 20 Ms;
end ThreadReceiver.impl2;
```

TABLE 2.5 – *Utilisation des modes dans une spécification AADL.*

mode d'exécution à un autre (soit de *thr_R1* vers *thr_R2*).

Notons que l'effet d'une transition peut se répercuter sur les autres composants de la spécification reliés par l'une des relations contenant-conteneur, conteneur-contenant ou tout simplement par une connexion. L'effet est alors observé comme une cascade de transitions permettant à tout le système de changer complètement de configuration.

La syntaxe d'une transition est comme suit :

ModeSrc-[**Event**]-> *ModeDst* ;

Où *ModeSrc* représente le mode source et *ModeDst* est le mode destination. La transition est déclenchée par l'avènement de l'événement *Event*.

2.4 Aspects de Déploiement

Une spécification AADL est la combinaison de l'aspect logiciel et matériel d'un système. Dans une spécification, une liaison explicite doit être définie entre un composant logiciel et celui de type matériel supposant le supporter dans la plateforme d'exécution. Ces relations sont exprimées par des propriétés de type *Binding*.
Dans le vocabulaire d'AADL, on trouve trois classes de propriétés de type *Binding*, chacune est prévue pour une utilisation différente.

- **Availble_xxxx_Binding** : spécifie que l'ensemble des sous-composants du composant *xxxx* sont mis à la disposition des composants logiciels d'une liaison à l'extérieur du système. L'ensemble est spécifié par une liste de noms d'éléments du système.
- **Allowed_xxxx_Binding** : force un composant logiciel à s'exécuter sur un composant matériel spécifique.
- **Actual_xxxx_Binding** : indique le composant matériel à utiliser pour l'exécution. Cette propriété perd son effet si elle est précédée ou suivie par un propriété *Allowed_xxxx_Binding* concernant le même composant.

N.B. *xxxx* : désigne un composant matériel tel qu'un processeur, mémoire ou bus.

Exemple : Dans l'exemple précédent de la table 2.2, nous pouvons constater l'utilisation de la propriété **Actual_xxxx_Binding** pour spécifier les relations liant chaque composant logiciel du système avec le composant matériel correspondant. Par exemple, nous avons la propriété :

Actual_Processor_Binding => **reference** the_processor **applies to** the_process.thr_S ;

Cette propriété indique que le sous-composant *thr_S* du composant *the_process* s'exécutera sur le processeur *the_processor* avec les caractéristiques définies pour ce processeur. Notons ici que le point "." qui sépare *the_process* et *thr_S* indique l'imbrication dans la syntaxe AADL. Ainsi, un sous-composant imbriqué dans un composant, peut hériter les propriétés et leur valeurs définies pour ce composant y compris les propriétés de déploiement.

2.5 Outils Pratiques

Plusieurs problèmes ont été soulevés par l'usage du langage AADL tels que :

- comment passer à l'implantation d'un système décrit par une spécification AADL (transcrire fidèlement le modèle vers un langage de programmation) ?
- Quelle est la sémantique opérationnelle, dénotationnelle et axiomatique associées aux structures de ce langage ?
- Comment définir les propriétés fonctionnelles et non fonctionnelles d'une application ?

À cet effet, plusieurs outils ont été développés autour du langage AADL, afin de pouvoir tirer le maximum de ce langage de modélisation et exploiter ses capacités. De ce fait, AADL dispose actuellement de plusieurs annexes qui exploitent les trois représentations des modèles AADL (texte, XML et graphique).

La figure 2.5 montre les trois représentations de modèles et leurs rôles par rapport à la représentation AADL textuelle et graphique ainsi que les outils qui traitent et analysent les modèles AADL (AADL Team 2004).

Dans ce qui suit, nous présentons quelques outils qui ont prouvés leur utilité et efficacité dans l'analyse et la manipulation des architectures AADL.

2.5.1 Osate

OSATE (AADL Team 2004) est un outil AADL de référence sous Eclipse (Moore et al. 2004). C'est un outil libre permettant, à travers un éditeur, l'expression textuelle et la vérification syntaxiques et sémantique générale des architectures AADL. Il fournit également un support complet pour la

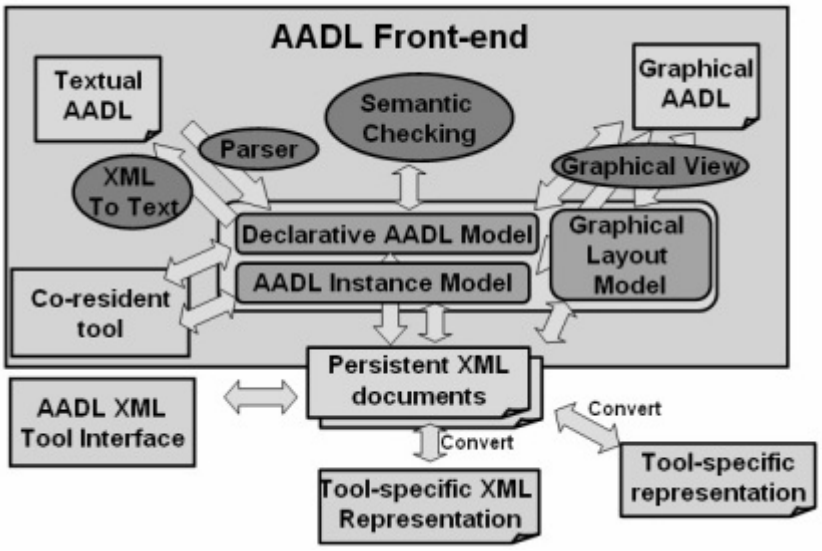

FIGURE 2.5 – *Modèles et Outils pour le Langage AADL*

spécification des méta-modèles AADL sur une plateforme Eclipse.

Pour valider l'exemple de la section 2.2.5 (spécification AADL), nous avons utilisé l'outil *Osate 1.5*. Il permet d'analyser syntaxiquement une spécification AADL et ne la valide que si elle est cohérente et exhaustive du point de vue architectural. Une fois la spécification est validée, le fichier XML correspondant est crée. Ce dernier peut être exploité par d'autres plugins pour des fins diverses telles que la représentation graphique, l'analyse et la vérification. La table 2.6 représente une portion de code XML relative à la spécification AADL généré par l'outil Osate . Elle décrit la structure et les propriétés de la déclaration d'implémentation du composant *Processor1*.
Depuis 2011, Osate2 [3] est disponible pour la version 2 d'AADL.

La figure 2.6 montre la manière dont l'outil graphique d'Osate schématise les composants et les sous-composants dans une spécification AADL. Notons ici la représentation plate (non hiérarchique) de la structure d'un système.

2.5.2 Cheddar

Initialement écrit dans un but pédagogique, Cheddar (Singhoff et al. 2008) est un outil gratuit [4] de simulation d'ordonnancement en temps réel. Cheddar est conçu pour vérifier des contraintes temporelles des travaux d'un système temps réel spécifié par le langage AADL.

3. https ://wiki.sei.cmu.edu/aadl/
4. Disponible sur http ://beru.univ-brest.fr/ singhoff/cheddar/src/

```
< processorImplname = "Processor1.impl"
compType = "/aadlSpec[name = Send_Rreceive]/processorType[@name=Processor1]" >
<properties>
<propertyAssociation propertyDefinition[@name=Scheduling_Protocol]" >
< propertyValue = RMS..."/ >
</propertyAssociation>
</properties>
<subcomponents>
< memorySubcomponentname = "mem"...classifier = "[@name=Memory1.impl]"/ >
</subcomponents>
</processorImpl>
```

TABLE 2.6 – *Spécification AADL du Composant "Processor1.impl", exprimée en XML, génerée par Osate.*

Cheddar peut être employé comme plugin avec STOOD (Dissaux et Singhoff 2008) ou avec TOPCASED (Farail et al. 2006). Il est principalement constitué de deux composantes logicielles :

- Un **éditeur** où l'on peut décrire l'application à analyser et sur lequel, les résultats de simulation seront affichés (figure 2.7).
- Une **bibliothèque** comportant les principaux résultats de la théorie d'ordonnancement temps réel ainsi que quelques outils de files d'attente.

La figure 2.7 représente une capture d'écran de l'exécution de l'outil Cheddar où l'on peut observer les résultats de simulation de l'exécution d'une application AADL.

2.5.3 Ocarina

Ocarina (Lasnier et al. 2009) est un compilateur AADL libre [5]. C'est une suite d'outils écrit en Ada permettant la génération automatique et massive de code pour des applications critiques, temps réel réparties embarquées, à partir d'un modèle AADL. Ocarina est développé à l'ENST (École Télécom ParisTech). Il est considéré comme un noyau central pouvant être intégré dans différentes applications.

Plusieurs modules composent Ocarina (voir figure 2.8) jouant chacun un rôle différent :

- **Ocarina Librairies** : les bibliothèques Ocarina constituent le noyau central qui fournit des utilitaires pour manipuler un modèle AADL.

5. Disponible sur http ://aadl.enst.fr/ocarina

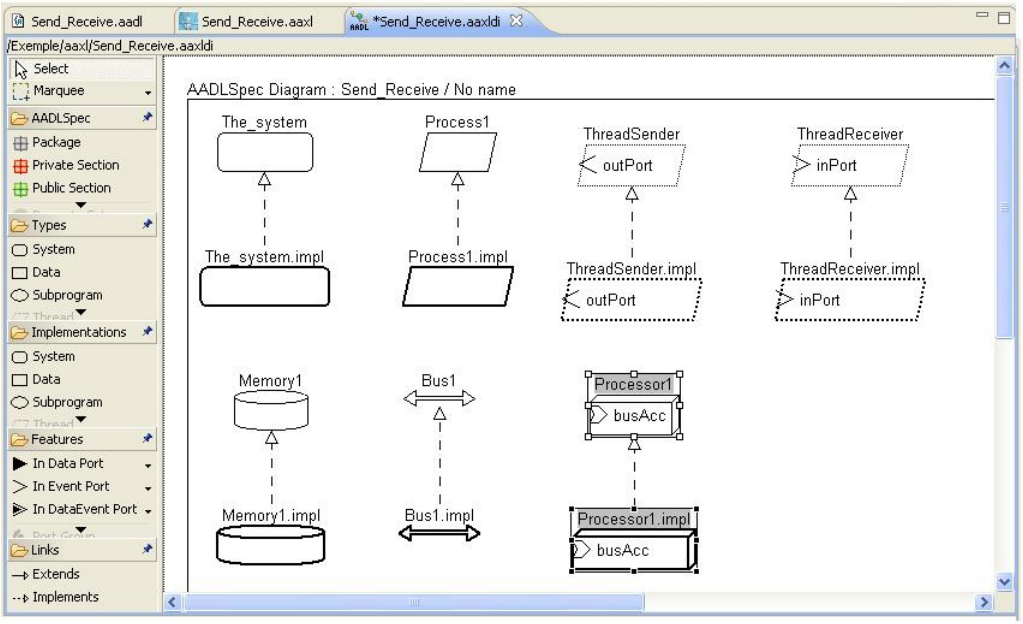

FIGURE 2.6 – *Spécification AADL présentée par l'outil graphique d'Osate.*

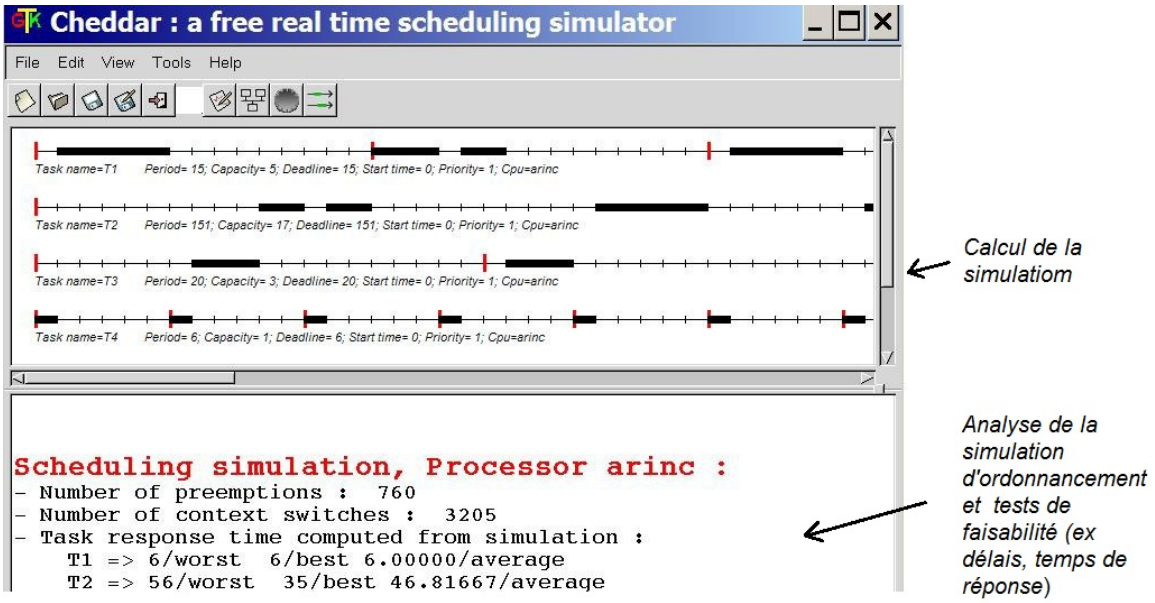

FIGURE 2.7 – *Capture d'écran de l'interface de l'outil Cheddar*

- **AADL parser** : un analyseur AADL a été développé pour la validation et le reformatage des modèles AADL. Il peut lire une description AADL réparties sur plusieurs fichiers.
- **Code generator** : les descriptions AADL peuvent être traitées pour générer une configuration entière. La version actuelle d'Ocarina permet à l'utilisateur de générer du code à partir d'une description de l'architecture AADL vers une application *Ada* fonctionnant au-dessus de l'intergiciel *PolyORB* et *PolyORB-HI* (Vergnaud et al. 2004).
- **Model Transformation** : Ocarina peut être utilisé pour l'analyse d'ordonnançabilité des modèles AADL en intégrant l'outil *Cheddar*.

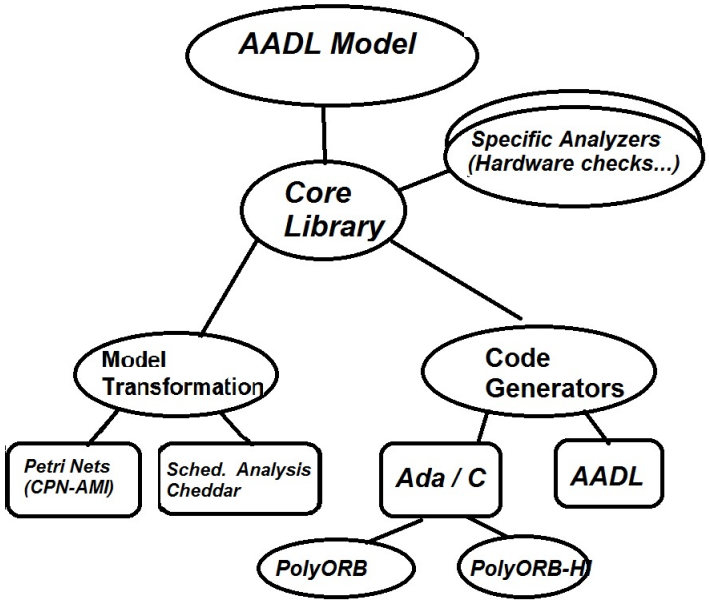

FIGURE 2.8 – *Modules composant Ocarina*

2.5.4 STOOD

STOOD (Dissaux 2003) est un outil de modélisation et de conception de logiciels embarqués développé par la société *Ellidiss*. C'est un générateur de code pour des applications monolithiques, il génère du code ADA et C à partir des spécifications AADL. Il est basé sur la méthode HOOD.

HOOD (Carmichael 1992) étant une méthode de conception architecturale, aide un concepteur logiciel dans le partitionnement du logiciel en modules avec des interfaces bien définies pouvant être directement mises en œuvre ou encore divisées en modules plus petits. La méthode supporte les approches fonctionnelles ainsi que celles basées sur l'objet et orientées conception. Elle intègre à la fois la programmation modulaire, centrée sur le

modèle client-serveur, ainsi que les relations de composition et d'héritage.

STOOD a été développé pour des conditions spécifiques afin de soutenir les développements logiciels aéronautiques. Il est également un outil graphique qui supporte trois notations graphiques, HOOD, UML2 et AADL. C'est un outil commercial [6]

Enfin, signalons que plusieurs tentatives de couplage de ces outils afin de pouvoir réaliser des analyses complètes d'une spécification AADL ont été entreprises. Une concernant l'intégration des outils *Cheddar* et *STOOD* (Dissaux et Singhoff 2008) et l'autre essayant de profiter des deux outils *OCARINA* et *Cheddar* (Hugues et Singhoff 2009).
Ces tentatives de couplage des outils de modélisation et d'analyse nécessitent que les sorties de l'un et les entrées de l'autre soient conformes et aient strictement la même définition sémantique des modèles échangés. Ceci est particulièrement important pour les systèmes temps réel et les architectures logicielles. Une telle garantie peut être apportée par l'utilisation commune d'AADL tout au long de la chaîne d'outils.

2.6 Conclusion

Dans ce chapitre, nous avons présenté les éléments architecturaux du langage AADL. Nous avons montré la motivation de ce langage, la structure d'un modèle AADL, sa syntaxe textuelle et graphique. Le langage permet de décrire la structure d'une application par une collection de composants logiciels s'exécutant sur des composants matériels, les différents composants logiciels et matériels peuvent être regroupés de façon logique au sein d'un composant système.

AADL se concentre sur les aspects architecturaux d'un système. Il permet la description des composants et leur connexions, mais ne traite pas directement leur comportement, ni la sémantique associée aux données manipulées. Cependant, l'aspect comportemental de la description peut être ajouté au moyen d'une annexe dédiée à cet effet. De la même façon, les différentes contraintes s'appliquant au déploiement des applications sur les topologies matérielles peuvent être exprimées au moyen de propriétés de type "Binding".

Nous avons essayé à travers un exemple simple 'Émetteur-Récepteur'

6. www.Ellidiss.com

d'expliquer la syntaxe et la structure d'une architecture AADL. Particulièrement, nous avons exposé la manière avec la quelle AADL assure l'agencement des composants logiciels sur les composants matériels. Dans les chapitres suivants, nous montrerons comment exploiter un formalisme mathématique pour définir une sémantique claire à ce langage.

Systèmes Réactifs Bigraphiques 3

Sommaire

Vous pouvez trouver autant de définitions d'un concept qu'il y a de lecteurs d'un article.

S. Thomason

3.1 Introduction

En mathématiques, les graphes représentent une aptitude au raisonnement et à l'abstraction. En informatique, se sont une structure de données et un moyen pour la modélisation et la résolution des problèmes. Ils sont très utilisés pour leur facilité de compréhension et leur pouvoir de description des aspects statique et dynamique des applications informatiques. D'ailleurs, plusieurs formalismes et langages utilisent les graphes pour modéliser une spécification ou un concept tels que : les réseaux de Petri, le langage UML, etc.

Le monde est de plus en plus peuplé par des agents interactifs distribués dans l'espace, ils peuvent être réels ou abstraits. Ces agents peuvent être artificiels, comme dans les systèmes informatiques qui gèrent et surveillent le trafic ou la santé, ou bien ils peuvent être naturels, par exemple, communication entre les humains, ou des cellules biologiques. Il est important d'être capable de modéliser ces réseaux d'agents afin de comprendre et d'optimiser leur comportement Milner (2005).

Milner (2009) décrit dans son livre un tel modèle, en présentant une théorie unifiée et rigoureuse, structurelle, basée sur les graphes, pour les systèmes d'agents en interaction. Cette théorie est un pont entre les théories existantes de processus concurrents et les aspirations des systèmes ubiquitaires, dont la taille énorme remet en question la compréhension de tels systèmes (Milner 2009).

Les Systèmes Réactifs Bigraphiques (BRS : Bigraphical Reactive Systems) sont donc un nouveau cadre graphique proposé par Milner et co-auteurs (Jensen et Milner 2004) comme une théorie unificatrice des modèles de processus pour les systèmes distribués, concurrents et mobiles. Un système réactif bigraphique est constitué d'un type de graphes (habituellement généré à partir d'une signature) et communément appelés 'Bigraphes' et un ensemble de règles de réaction (Milner 2006). Les configurations possibles du système, ainsi que les règles de réaction précisent comment ces configurations peuvent évoluer. L'avantage d'utiliser les systèmes réactifs bigraphiques est qu'ils fournissent des résultats génériques.

Plusieurs travaux dans le monde se sont intéressés à l'usage de ce formalisme pour modéliser différents concepts et systèmes. Nous présentons dans ce qui suit les plus connus tels que :

Dans Grohmann et Miculan (2007), Grohmann (2008), les auteurs se sont intéressés aux BRS et particulièrement à la construction de systèmes de transitions étiquetés (*LTS*) à partir de systèmes réactifs définis sur les BRS *orientés*. Ces derniers permettent de représenter les systèmes et les calculs qui ne sont pas encore couverts par les LTS ordinaires. Les auteurs ont montré que les bigraphes orientés fournissent un cadre général de fonctionnement où les systèmes réactifs peuvent être représentés et étudiés et à partir desquels les LTS avec leur bisimulation peuvent être systématiquement dérivés.

Par ailleurs et dans un autre contexte, les bigraphes ont été utilisés pour modéliser des langages de programmation (Birkedal et al. 2007, Bacci et al. 2009, Glenstrup et al. 2010). Les auteurs ont exploité l'aspect formel des bigraphes pour mettre en œuvre des langages de programmation dédiés aux systèmes réactifs et mobiles. Les auteurs pensent que le fait d'utiliser un tel formalisme faciliterait la conception et la vérification de tels systèmes. Ainsi ils ont développé une logique dédié aux bigraphes et à ce contexte.

Chang et al. (2007) sont les premiers à voir dans les BRS une puissance de l'expression des styles architecturaux et leurs instances. Dans leur travaux, les auteurs ont essayé d'étendre le formalisme original pour qu'il puisse prendre en charge la modélisation et le test de conformité des instances architecturales vis-à-vis de leur style de base.

L'objectif principal de ce chapitre est d'introduire les systèmes réactifs bigraphiques et leurs concepts généraux. Dans un premier temps, nous présentons quelques définitions telles que fournies par les fondateurs de ce concept, ensuite nous exposons les principales opérations que l'on peut effectuer sur les bigraphes agrémentées par des exemples illustratifs. Une liste d'outils développés autour des bigraphes est enfin exposée.

3.2 Présentation des Bigraphes

Les Systèmes Réactifs Bigraphiques (*BRS* pour Bigraphical Reactive Systems) ou communément 'Bigraphes', définis dans Jensen et Milner (2004), sont un modèle mathématique et graphique intégrant les dimensions *interaction* et *distribution spatiale* des applications distribuées à code mobile.

Ce nouveau type de graphes fut introduit dans le but de prendre en charge les caractéristiques et les spécificités des systèmes réactifs.

3.2.1 Définition d'un Système Réactif

Un système logiciel peut être classé dans l'une de ces catégories : transformationnel, réactif, ou interactif. Un système est dit transformationnel s'il s'exécute sur des données pour fournir au bout d'un certain temps un résultat. Un compilateur est un exemple typique de tels systèmes. A l'inverse, un système est dit réactif lorsque il réagit constamment à son environnement à la vitesse imposée par celui-ci. Comme exemple de systèmes réactifs, nous pouvons citer tout système qui fonctionne en temps réel et interagissant avec son environnement d'une façon instantanée tels qu'un système de pilotage automatique des avions, un robot ou un jeu vidéo. Un tel système doit donc réagir à son environnement au fur et à mesure que ce dernier évolue. La dernière classe est celle des systèmes interactifs. Un tel système réagit constamment avec son environnement mais avec sa propre vitesse, comme le font par exemple les systèmes d'exploitation et les interfaces homme-machine (Acosta Bermejo 2001).

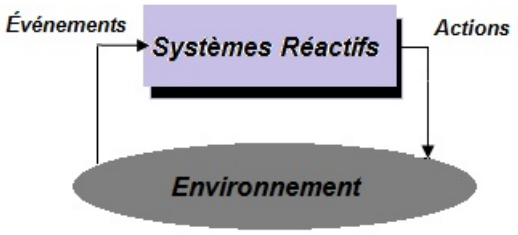

FIGURE 3.1 – *Systèmes Réactifs*

La figure 3.1 illustre le mécanisme d'interaction d'un système réactif avec son environnement. La réaction du système aux événements est traduite par des actions effectuées sur l'environnement.

3.2.2 Définition d'un Bigraphe

Un bigraphe est en fait la combinaison de deux structures indépendantes : le graphe des places et le graphe des liens, d'où le nom de 'Bigraphe'. L'intersection entre ces deux graphes est un ensemble commun de nœuds, correspondant aux entités physiques ou virtuelles d'une application. Le graphe des places (*Places graph*) a la structure d'une forêt, présentant la distribution géographique de l'application. Le graphe des liens (*Link graph*) est un hyper-graphe montrant le schéma de connectivité des différents nœuds Jensen et Milner (2004).

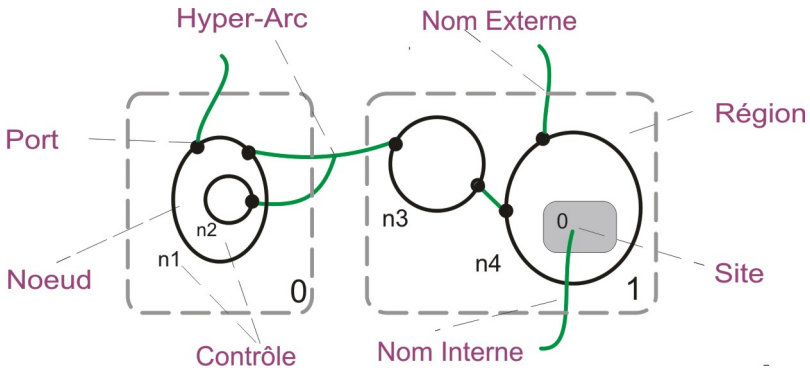

FIGURE 3.2 – *Éléments d'un Bigraphe*

Alors qu'un arc dans le graphe des places montre la relation d'imbrication entre les éléments de l'application, un hyper-arc dans le graphe des liens établit une connexion entre les ports de ces éléments. Chaque arbre dans le graphe des places représente une *région* qui peut contenir des *sites*, correspondant aux feuilles de l'arbre, où d'autres bigraphes peuvent être insérés ou hébergés par composition.
Les constituants de base d'un bigraphe sont : régions (*root*), nœuds (*nodes*), sites (*site* ou *place*), arcs (fermés ou ouverts), ports, noms externes ou sortants (*outer-names*) et internes ou entrants (*inner-names*) (voir figure 3.2).

Dans ce qui suit nous détaillerons la définition d'un bigraphe telle qu'elle a été introduite par ses fondateurs.

Définition 1 : *Un bigraphe* $G = (V, E, Ctrl, G^P, G^L) : I \rightarrow J$ *est défini par (Milner 2008) :*

- V *est un ensemble fini de nœuds qui peuvent être imbriqués,* E *est un ensemble fini d'hyper-arcs ;*
- $Ctrl : V \rightarrow K$ *est une transformation qui associe à chaque nœud* v_i *de* V *un contrôleur k de* K *tel que* $Ctrl(v_i)=(np, a, d)$ *; où* np *indique le nombre de ports que possède le nœud,* a *indique si le nœud est atomique ou composite,* d *renseigne sur l'aspect dynamique ou statique du nœud ;*
- $G^P = (V, E, Ctrl, prnt) : m \rightarrow n$ *est le graphe des places associé à* G*, où* $prnt : m \cup V \rightarrow n \cup V$ *est une fonction de parenté associant à chaque nœud son parent hiérarchique.*
- $G^L = (V, E, Ctrl, link) : X \rightarrow Y$ *est le graphe des liens de* G*, où* $link : X \cup P \rightarrow Y \cup P$ *est une transformation montrant le flux de données des noms internes* X *ou les ports* P *vers les noms externes* Y *ou les arcs* E *;*
- $I = \langle m, X \rangle$ *et* $J = \langle n, Y \rangle$ *sont respectivement les interfaces internes et externes du bigraphe* G*, avec* m *le nombre de sites, i.e. emplacements dans*

le graphe susceptibles d'abriter de nouveaux éléments, X *est l'ensemble des noms internes,* n *est le nombre de régions, i.e. éléments du bigraphe pouvant s'intégrer dans d'autres bigraphes,* Y *est l'ensemble des noms externes. Une interface égale à* $\langle 0, \emptyset \rangle$ *est notée* $I = \varepsilon$.

Exemple 1 : Le bigraphe G (figure 4.3) inspiré de Conforti et al. (2005), modélise un ensemble d'utilisateurs et de machines PCs répartis entre deux salles R1 et R2. L'ensemble des nœuds de ce bigraphe est défini par $V = \{U1, U2, U3, Pc1, Pc2, Pc3, R1, R2\}$. L'ensemble $E = \{L0, L1, L2, L3, L4\}$ d'hyper-arcs ou liens, désigne le réseau d'interconnexion qui relie les PCs, ou bien entre les utilisateurs et les PCs. La transformation *Ctrl* (simplifiée) et la fonction *prnt* sont définies dans ce cas par :

$$Ctrl = \{(U1 : 2), (U2 : 2), (U3 : 2), (Pc1 : 2), (Pc2 : 2), Pc3 : 2), (R2 : 1), (R2 : 4)\}$$

$$prnt = \{(U1 : R1), (U2 : R2), (U3 : R2), (Pc1 : R1), (Pc2 : 2), (Pc3 : 2), (R1 : \emptyset), (R2 : \emptyset)\}$$

Le contrôleur k_{U1} du nœud *U1* est 2 (car *U1* a deux ports, l'atomicité et le dynamisme ne sont pas présentés). Le nombre de sites est m = 0, i.e., pas d'emplacements ou de sites libres. Le bigraphe contient deux régions numérotées 0 et 1 et représentées par des rectangles en pointillés. L'arc ouvert x est une interface externe du bigraphe, cette interface permet au bigraphe d'interagir avec d'autres bigraphes. Dans ce cas, elle représente la disponibilité/besoin de l'utilisateur k_{U1} de communiquer avec un autre utilisateur.

Chacun des graphes de place et de liens possèdent ses interfaces externes et internes (voir figure 4.3). Pour le premier (graphe des places) ses interfaces externes et internes sont respectivement le nombre de régions (racines) et le nombre de sites tel que les régions sont les nœuds sans parents, et les sites sont les emplacements vides où l'on peut insérer d'autres bigraphes.
Pour le deuxième (graphe des liens), l'interface interne est définie par les noms internes (liés aux sites) qui expriment la possibilité de réceptionner un bigraphe. L'interface externe, définie par les noms externes (liés aux régions ou aux nœuds), exprime la possibilité de déplacer ce nœud à un autre emplacement. L'action de réception et de déplacement vers un autre emplacement illustre l'aspect de mobilité dans les bigraphes.

Comme le bigraphe G est le résultat de la fusion des deux graphes, ses interfaces sont donc eux aussi le résultat de la fusion de leurs interfaces. Si on symbolise le graphe des places par $G_P : m \to n$ et le graphe des liens par

$G_L : X \to Y$ alors le bigraphe sera décrit comme suit : $G : \langle m, X \rangle \to \langle n, Y \rangle$.
Si on applique cela sur le bigraphe G (de la figure 4.3), on trouve que le graphe des places est décrit comme suit : $G_P : 0 \to 2$; 2 est l'interface externe indiquant le nombre de régions ; et o est l'interface interne montrant l'inexistence de sites. Le graphe des liens sera décrit par $G_L : \phi \to \{x\}$ qui signifie que l'interface externe est {x} ; et qu'il n'y a pas de noms internes. C'est pour cela que l'interface interne est notée ϕ.
Le bigraphe G peut être alors défini à travers ses interfaces internes et externes comme suit :

$$G = \langle 0, \phi \rangle \to \langle 2, x \rangle$$

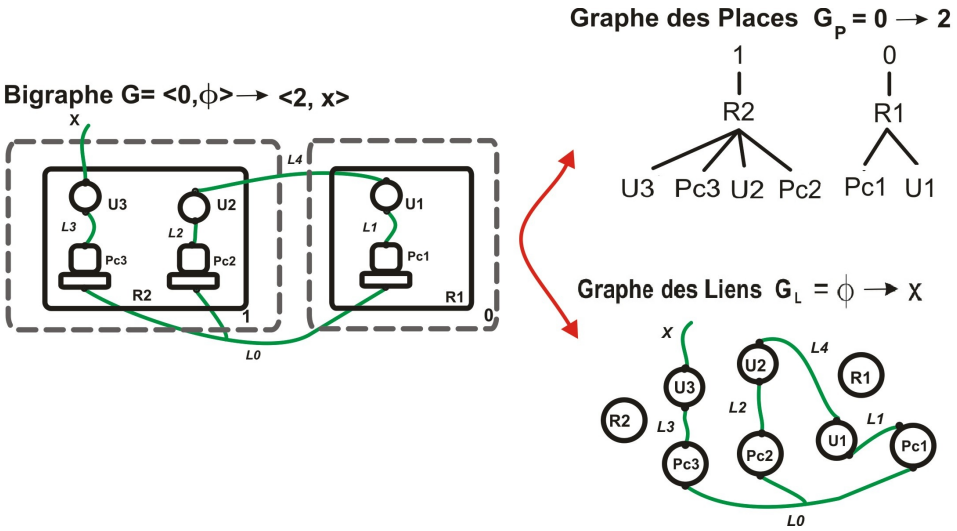

FIGURE 3.3 – *Exemple de Bigraphe.*

Remarque : Un système réactif bigraphique ou BRS est en fait un bigraphe muni d'un ensemble de règles de réactions décrivant sa dynamique.

3.3 Opérations sur les Bigraphes

Les bigraphes comme tout type de graphes peuvent subir des modifications et des manipulations. Plusieurs types d'opérations de manipulation sont possibles sur les bigraphes. Celles qui permettent de créer de nouveaux bigraphes en combinant d'autres bigraphes sont appelées opérations de *composition* et celles qui permettent de faire subir à un bigraphe des transformations à l'aide des règles de transformations sont appelées *règles de réaction*. Dans la section suivante, nous présentons quelques opérations sur les bigraphes à travers des exemples.

3.3.1 Composition Verticale

C'est une opération qui permet d'obtenir un nouveau bigraphe en combinant deux ou plusieurs bigraphes. Particulièrement, la composition de deux bigraphes F et H est une opération d'hébergement des régions du bigraphe F dans les sites du bigraphe hôte ou contextuel H. Cette opération ne peut être possible que s'il y a au moins autant de sites dans H que de régions dans F (une région par site). La composition des deux bigraphes se fera via un appariement des interfaces externes de F et les interfaces internes de H. La définition suivante dicte les contraintes à satisfaire pour réussir une opération de composition.

Définition 2 : *Soit* F *un bigraphe défini par ses interfaces internes et externes tel que* $F : I \rightarrow J$*, où* I= *domaine*(F) *est* J=*codomaine*(F)*. La composition* $G \circ F$ *de deux bigraphes* F *et* G *doit satisfaire les conditions suivantes :*

- *(c1) :* $G \circ F$ *est définie ssi* codomaine(F) = domaine(G)*;*
- *(c2) :* $H \circ (G \circ F) = (H \circ G) \circ F$ *il suffit que l'un des termes soit défini pour que l'autre le soit ;*
- *(c3) :* $id \circ F = F$ *et* $F = F \circ id$*, avec* id *est fonction Identité.*

Exemple 2 : Soient un bigraphe F (figure 3.4) constitué d'une région ayant x et y comme noms externes, et un bigraphe contextuel H possédant un site et deux noms internes x et y. La composition des bigraphes F et H est un nouveau bigraphe $G = H\ o\ F$ sans sites ni noms externes. Notons que les éléments du bigraphe F se sont intégrés aux éléments du bigraphe H à l'emplacement indiqué par le site o et les noms internes x et y de celui-ci.

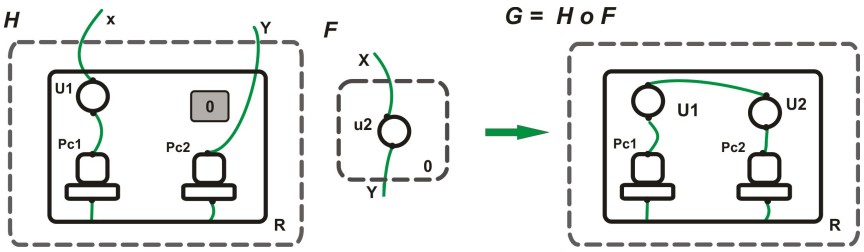

FIGURE 3.4 – *Composition de deux Bigraphes.*

3.3.2 Produit Tensoriel (ou Composition Horizontale)

Le produit tensoriel ou composition horizontale, est une opération plus simple que la précédente, elle consiste à faire coller deux bigraphes ou plus,

précisément c'est mettre un bigraphe à côté de l'autre et en joignant les interfaces communes. C'est juste une opération de juxtaposition de bigraphes pour obtenir un bigraphe plus large. Cette opération est notée par $\otimes$.

Définition 3 : *Étant donné deux bigraphes* $F_0 : I_0 \otimes J_0$ *et* $F_1 : I_1 \otimes J_1$ *où* $I_i = (m_i, X_i)(i = 0,1)$ *et* X_0, X_1 *sont disjoints ;* $J_i = (n_i, Y_i)(i = 0,1)$ *et* Y_0, Y_1 *sont disjoints.*
Alors le produit tensoriel $F_0 \otimes F_1 : I_0 \otimes I_1 \rightarrow J_0 \otimes J_1$ *où* $I_0 \otimes I_1 = (m_0 + m_1, X_0 \cup X_1)$ *;* $J_0 \otimes J_1 = (n_0 + n_1, Y_0 \cup Y_1)$.

Ainsi, le produit tensoriel $F_0 \otimes F_1$ de F_0 et F_1 est simplement obtenu en posant les bigraphes côte à côte et en joignant les arcs ouverts correspondants.

Exemple 3 : Soient deux bigraphes F_0 et F_1 (figure 3.5), le produit tensoriel est $F_0 \otimes F_1 = F$. Du fait que les deux bigraphes F_0 et F_1 possèdent respectivement dans leurs interfaces les arcs ouverts (noms externes) $\{x, w\}$ et $\{y, w\}$, le produit tensoriel est possible en faisant joindre les arcs ouverts ayant le même nom (w). Nous obtenons ainsi un seul bigraphe avec un ensemble de noms externes : $Y_0 \cup Y_1$= {x, y, w}.

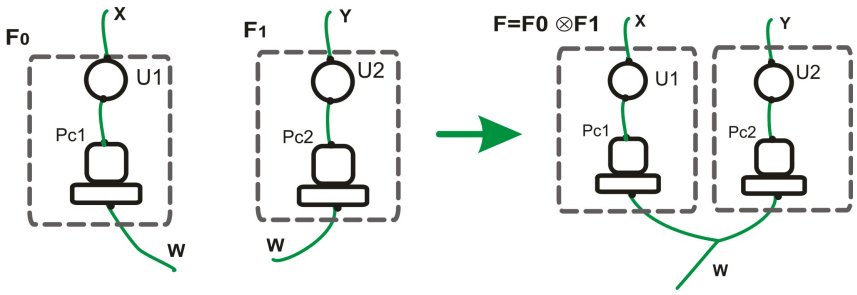

FIGURE 3.5 – *Application d'un Produit Tensoriel entre deux Bigraphes*

3.3.3 Transformation

La dynamique d'un bigraphe peut être spécifiée par des règles de transformation préalablement établies, dites *règles de réaction*. Deux types de transformation sont possibles sur un bigraphe. La transformation sur les *places* qui exprime l'opération de mobilité dans le système. Elle représente l'arrivée ou le départ d'une entité (représentée par un nœud). De son côté, la transformation sur les *liens* exprime la connexion ou déconnexion d'un nœud du bigraphe à travers l'une de ses interfaces. Une règle de réaction est un couple de bigraphes (*Redex* et *Reactum*) tel que :

– Le Redex spécifie le bigraphe à transformer ;

– Le Reactum spécifie le bigraphe après la transformation.

L'application d'une règle consiste à identifier dans le bigraphe contextuel une image du bigraphe *Redex* puis le remplacer par le bigraphe *Reactum*.

Définition 4 : *Formellement une règle de réaction, définie par Chang et al. (2007), a la forme suivante : $(R : m \to J, R' : m' \to J, \eta)$ où* R *est le bigraphe Redex et* R' *est le bigraphe Reactum. Les deux bigraphes n'ont pas de noms internes et peuvent avoir un nombre de sites différents (*m *et* m'*). Néanmoins, ils ont les mêmes interfaces externes (*J*). $\eta : m' \to m$ est une application qui établit une correspondance entre le nombre de sites de deux bigraphes Redex et Reactum.*
Notons ici que cette opération peut modifier les interfaces internes d'un bigraphe (nombre de sites).

Exemple 4 : La figure 3.6.a représente une règle de réaction dont la forme est (*R, R'*), tel que les interfaces internes de *R* et *R'* sont ε et les interfaces externes sont ε . Notons aussi l'absence de sites, ce qui donne la forme suivante à cette règle de réaction $(R, R') = (R : \varepsilon \to \langle 1, \emptyset \rangle$, $R' : \varepsilon \to \langle 1, \emptyset \rangle$ et $\eta : 0 \to 0)$. L'application de cette règle de réaction sur le bigraphe *F* aura comme effet la connexion de l'utilisateur *U*2 au *Pc*2.

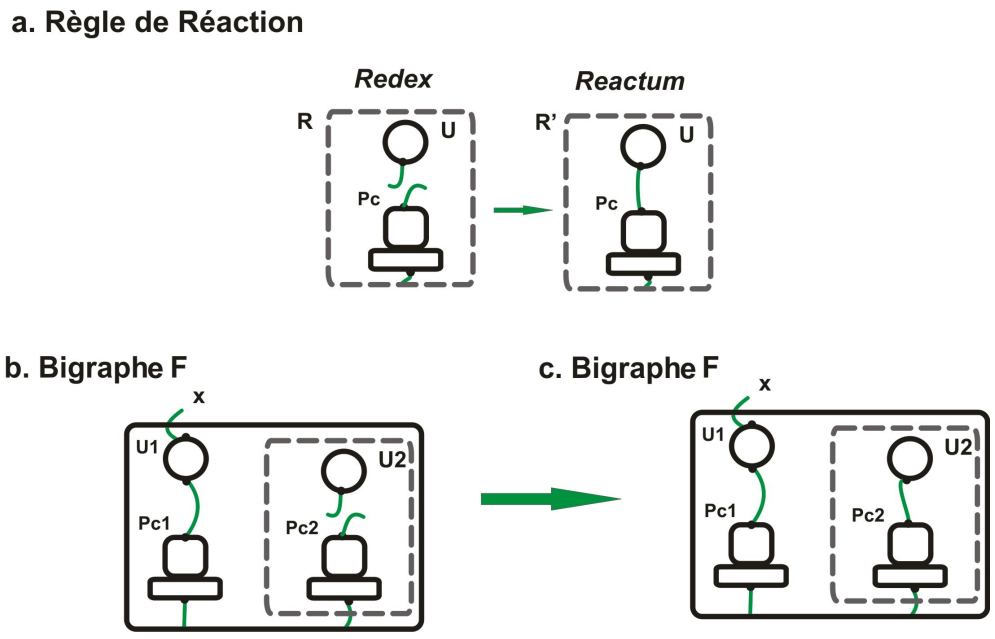

FIGURE 3.6 – *Application d'une règle de réaction sur un bigraphe F.*

3.4 Outils Pratiques autour des Bigraphes

Les bigraphes constituent un formalisme mathématique très expressif et puissant pour l'expression des systèmes distribués et mobiles qui ont en plus besoin d'être validés. Néanmoins, la nouveauté du formalisme fait qu'il manque encore d'outils et d'environnements d'édition et de validation. Cela dit, certains outils, dédiés à ce formalisme, ont vu le jour particulièrement à l'université *IT* de *Copenhague*, tels que :

3.4.1 Bptool

Dans le groupe "Programming, Logics, and Semantics" de l'université *IT* de *Copenhague, Danemark*, les chercheurs ont pour objectif de concevoir des langages de programmation basés sur la théorie des bigraphes. Ces langages de programmation seront pratiquement bien adaptés aux applications ubiquitaires qui se prêtent aisément au raisonnement formel.

L'outil BpTool (Glenstrup et al. 2010) rencontre un défi majeur dans la mise en œuvre de la dynamique des systèmes réactifs bigraphiques. Il faudra tout d'abord définir un mécanisme pour résoudre le problème d'appariement ou de correspondance entre bigraphes (figure 3.7), c'est-à dire déterminer, pour un bigraphe donné, quelle règle de réaction appliquer et comment cette règle de réaction peut être appliquée pour la réécriture du bigraphe. Le problème de l'appariement dans ses détails, est un problème assez complexe (problème NP-complet). Par conséquent, les auteurs ont décidé de considérer l'aspect formel du problème afin d'assurer un fondement pour l'implémentation d'un prototype sur une spécification prouvable correcte (Damgaard et al. 2013).

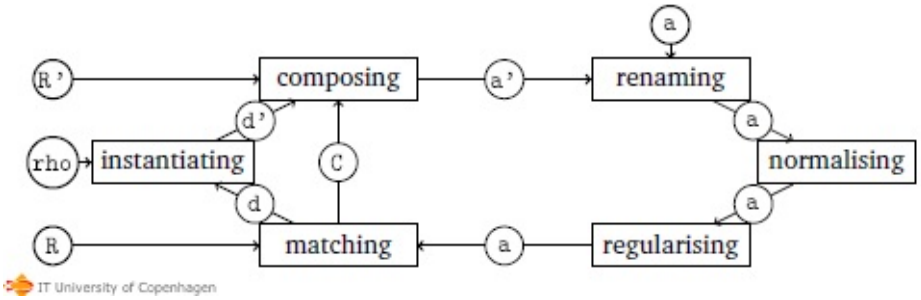

Figure 3.7 – *Ingénierie de BpTool (Glenstrup et al. 2010)*

Dans ce contexte l'équipe a implémenté l'outil *BpTool* [1] qui est la première implémentatiom des systèmes réactifs bigraphiques liés (*Binding BRS*) avec le langage *Standard ML* (SML). L'outil BpTool fournit la manipulation, la simulation et la visualisation des systèmes réactifs bigraphiques, et peut être

1. Disponible sur http ://www.itu.dk/research/pls/wiki/index.php/BPL_Tool

utilisé soit à travers le Web (non disponible actuellement) ou en utilisant une bibliothèque de programmation. Il ne dispose pas d'interface graphique et conçu pour le système *Linux*.

3.4.2 Dbtk

L'outil Dbtk (Bacci et al. 2009) est développé au sein de la même équipe de l'université IT de Copenhague. DBtk [2] supporte un langage textuel (Directed Bigraphical Language (DBL) (Grohmann 2008)) pour les bigraphes *orientés*. Il permet de représenter les bigraphes dans le format *SVG* basé sur *XML*, un standard de *W3C* pour les graphiques vectoriels. L'outil est composé de certains modules (voir figure 3.8) tels que :

- API pour la gestion des structures bigraphiques orientées, c'est-à-dire l'algèbre associée et les opérations.
- Un compilateur, un dé-compilateur et un afficheur en SVG (est un plugin permettant d'apporter le support du format SVG (images vectorielles) dans les navigateurs web).
- API pour le calcul d'une correspondance entre deux bigraphes Redex et Reactum.

Par conséquent, cet outil fournit les principales fonctions nécessaires à la mise en œuvre des simulateurs et des outils de vérification pour les bigraphes *orientés*.

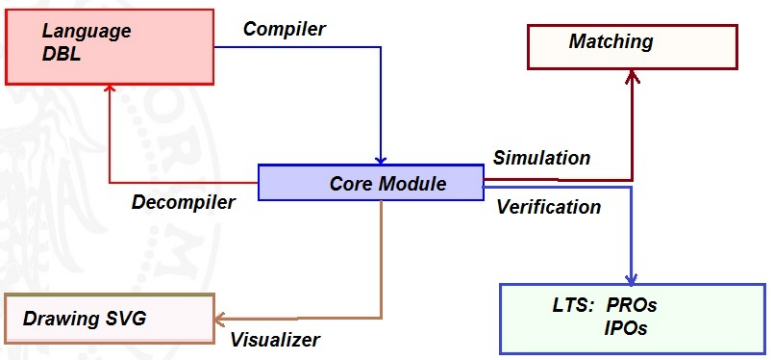

FIGURE 3.8 – *Architecture de l'outil Dbtk (Bacci et al. 2009)*

3.4.3 BigMC

BigMC [3] (Bigraphical Model Checker) (Perrone et al. 2012) est un modèle checker conçu pour fonctionner sur des modèles basés sur des systèmes ré-

2. Disponible sur http ://sole.dimi.uniud.it/̃davide.grohmann/dbtk/
3. Disponible sur http ://bigraph.org/bigmc/

actifs bigraphiques. Par la vérification du modèle, les auteurs sous entendent précisément le fait de vérifier si une propriété d'une spécification est vraie dans un modèle bigraphique particulier. Ce résultat est obtenu grâce à une recherche exhaustive de tous les états possibles du système. L'objectif principal de la vérification du modèle est de fournir un contre-exemple dans le cas où la spécification ne vérifie pas une propriété.

Pour les modèles arbitraires, ce genre de vérification est mathématiquement insurmontable puisque l'espace d'état est tout simplement trop énorme (et même infini dans de nombreux cas). Le défi de ce type de tâche est de limiter les types de modèles à vérifier à un sous-ensemble traitable, et de réduire le nombre d'états réels que nous devons vérifier directement afin de fournir des garanties d'exactitude concrètes.

La grammaire complète BigMC pour définir les termes d'un modèle bigraphique est donnée par la table 3.1.

M : := E ; M | E ;	T : := K :T | T | T | | T| $n | K | nil
E : := % passive k : arity	K : := k [names] | k
E : := % active k : arity	names : := n ; names | n
E : := % rule n $T \rightarrow T$	n : := [a-z A-Z][a-z A-Z 0-9]* | -
E : := % property n P	P : := matches(T) | terminal () | !P
E : := $T \rightarrow T$ | T	

TABLE 3.1 – *Termes du langage BigMc*

M représente un modèle bigraphique selon BigMc, qui peut être composé à partir d'autres modèles ou/et expressions *E*. Une expression *E* peut être une déclaration de nœud (dynamique et ayant une arité et un contrôle donnés), une règle de réaction, un terme *T*, ou une propriété *P*. Un terme *T* peut représenter un nœud simple, un site ou une région. Mais aussi, il peut être une combinaison de tous ces éléments. La propriété *P* est une définition de l'état à vérifier avec cet outil. Cet état est censé être atteignable en appliquant une suite de règles de réaction définies dans le modèle. Dans ce cas, la propriété est considérée *vraie*. La vérification est implémentée par l'algorithme décrit dans Perrone et al. (2012).

La table 3.2 représente la définition d'un modèle bigraphique avec la grammaire définie pour l'outil BigMc[4]. L'exemple est composé de trois

4. Pour l'exemple consulter http ://bigraph.org/bigmc/manual/Term-language.html

nœuds *a*, *b* et *c*. Il n'y a pas de noms mais trois règles de réaction sont définies pour ce modèle bigraphique.

Définitions des contrôles *%active a : o ;* # o désigne que a est sans ports
%active b : o ;
%active c : o ;
Définition du modèle
a.b ;
Définition des règles de réaction
b -> a.b ;
a.a.b -> a ;
a.$o -> c.$o ;
$o désigne un site # Définition de la propriété à vérifier
%property growth size() >= $pred →size() ;

TABLE 3.2 – *Exemple d'un Modèle Bigraphique sous BigMC*

La propriété exprimée dans cet exemple (table 3.2) concerne la taille du modèle. Elle stipule que le modèle croit en appliquant les règles de réactions définies dans l'exemple. Pour tester et valider cette propriété, il faut exécuter la commande suivante :

$ *bigmc diverge_prop.bgm*

Ce qui donne la trace suivante :

**** Found violation of property : growth*
$***growth : size >= \$pred -> size()$*
$0 - a.nil < - *** VIOLATION **$
$>> a.a.b.nil -> a.nil$
$1 - a.a.b.nil$
$>> b.nil -> a.b.nil$
$2 - a.b.nil$
$>> (root)$
$mc :: step() : Counter - example found.$

Les états de la trace sont affichés du dernier état obtenu au premier. Ils sont étiquetés par les règles de réaction qui ayant été appliquées pour atteindre ces états depuis le premier jusqu'au dernier. L'interprétation du résultat montre un état qui peut servir de contre-exemple de violation de la propriété et qui signifie que le modèle a une taille qui ne croît pas. Donc la propriété *growth* n'est pas vérifiée.

3.5 Architecture Logicielle et Bigraphes

Les BRS furent conçus pour modéliser le calcul de processus pour la concurrence tels que le CCS, π-calcul, λ-calcul, etc (Milner 2008). Actuellement, ils sont de plus en plus utilisés pour spécifier et modéliser les systèmes et les applications de différents domaines.

D'autre part, plusieurs approches dans la littérature adoptent le concept des graphes et leurs transformations pour modéliser les styles architecturaux et leurs instances (Metayer 1998, Bruni et al. 2007). Les bigraphes furent utilisés pour la première fois pour spécifier les architectures logicielles dans les travaux de Chang et al. (2007; 2008). Dans leurs travaux, Chang et al. (2007) ont montré l'aptitude des BRS à représenter efficacement les architectures logicielles ainsi que les styles architecturaux et leur capacité à préserver les caractéristiques d'un style même après des opérations de reconfiguration.

Pour cela Chang et al. (2007), ont présenté une approche basée BRS pour vérifier la conformité d'une instance par rapport à son style architectural. Ils ont étendu les bigraphes et les BRS par "Σ-*Sorted bigraph*" en introduisant la notion de *type* pour les nœuds d'un bigraphe afin de permettre la description d'une instance architecturale et le style architectural correspondant. Ils ont fournit une approche supportant le test de la conformité. Inspirés par cette idée, nous utilisons les BRS pour décrire une architecture logicielle à base de composants mais également sa plateforme d'exécution cible. Nous nous servirons des concepts bigraphiques pour définir les relations qui existent entre un composant logiciel et le composant matériel correspondant.

3.6 Conclusion

Ce chapitre constitue une brève introduction aux BRS ou Bigraphes sans pour autant négliger les points importants dans la définition d'un bigraphe. Nous avons essayé à travers des exemples de décrire les principales opérations que l'on peut effectuer sur les bigraphes.
Certains outils censés manipuler les bigraphes ont été présentés. *BigMC* semble le plus adapté à traiter et tester des bigraphes sans se limiter à un domaine particulier ou à un type particulier de bigraphes.

Nous présenterons dans ce qui suit comment nous pouvons utiliser les bigraphes pour représenter concrètement, à la fois, les architectures logicielles et matérielles.

4 Formalisation d'un Système AADL à base de Bigraphes

Sommaire

Pour gagner du bien, le savoir-faire vaut mieux que le savoir.

Beaumarchais.

4.1 Introduction

AADL (SAE 2008) est un langage de description d'architecture assez complet et exhaustif. Il est dédié à la description et la spécification des systèmes embarqués temps réel. Dans ce type de systèmes, les composantes logicielles et matérielles sont fortement couplées. Dans ce contexte, AADL permet en une seule spécification de préciser les caractéristiques des entités logicielles qui permettent de définir une application logicielle tels que les données, threads et les processus et celles (de type matériel) constituants la plateforme d'exécution tels que le processeur, mémoire et bus de communication. Ces caractéristiques font d'AADL un langage concret.
En plus, une spécification *type* d'un système AADL est considérée comme un modèle de système dont l'instanciation peut être exprimée par une clause *implementation*. Cette caractéristique place ce langage à un niveau d'abstraction élevé . Entre ces deux niveaux bien distincts, AADL reste un langage qui a besoin d'un formalisme pour exprimer sa sémantique (Benammar et al. 2009).

Plusieurs travaux dans la littérature se sont intéressés à la formalisation des concepts AADL. Dans Benammar (2011), l'auteur a comparé et résumé les tentatives de formalisation des concepts AADL (voir table.4.1). Cette étude a dégagé plusieurs classes de modèles formels utilisés autour de AADL. Nous y trouvons les modèles à base des systèmes de transition et de réseaux de Petri temporisés (*TPN*), les machines à états abstraits (*TASM*) et d'autres qui utilisent les algèbres de processus telle que *ACSR*. Auxquels est venue s'ajouter la logique de réécriture révisée.
Cette classification a mis en évidence les similitudes et les différences entre les différentes approches de formalisation. Chacune d'elles procède à transformer la description architecturale AADL en d'autres langages ayant une sémantique formelle et permettant d'effectuer des simulations et des vérifications.

Dans ses travaux, Vergnaud (2006) s'est intéressé aux propriétés structurelles définies dans une architecture AADL et pour cela il a utilisé des Réseaux de Petri colorés générés à partir d'une description AADL pour étudier des propriétés structurelles comme le blocage des architectures.

Dans Chkouri et al. (2008), les auteurs ont étudié une méthodologie générale et mis en œuvre un outil associé pour traduire une spécification AADL. Ils ont ainsi, défini une annexe comportementale associée dans le langage BIP (Behavior Interaction Priority). Ceci permet la simulation des systèmes

décrits en AADL et l'application à ces systèmes les techniques de vérification formelle développées pour *BIP*, telle que la détection de blocage sur ces systèmes.

Concernant les travaux d'Abdoul et al. (2008), les auteurs se sont intéressés à la transformation de modèles AADL vers un modèle formel *LTS* (Système de Transition Étiqueté) afin de le vérifier. Cette transformation portait sur trois aspects : la structure, le comportement et la description sémantique de l'exécution. Les règles de transformation doivent tenir compte de ces aspects. La plateforme de méta modélisation proposée (*Kermeta*) est utilisée pour mettre en œuvre ces règles ayant été appliquées à une étude de cas.

Les modèles à base d'algèbre de processus *ACSR*, présentés par Sokolsky et al. (2009) sont en effet un ensemble d'outils pour la vérification comportementale et la validation de modèles d'architecture des systèmes embarqués exprimés dans le langage AADL. Ces outils servent à la simulation et l'analyse temporelle des modèles AADL en mettant en œuvre la sémantique d'AADL par l'algèbre des processus en temps réel. L'analyseur d'ordonnancement permet de déterminer si le système possède suffisamment de ressources pour satisfaire les contraintes de la synchronisation. En parallèle, et presque dans le même contexte, Berthomieu et al. (2009) ont utilisé le langage intermédiaire *FIACRE* pour représenter le comportement et les aspects de synchronisation des systèmes exprimés dans le langage AADL.

Yang et al. (2009) se sont intéressés à l'extension de l'annexe comportementale par un mécanisme de répartition du modèle d'exécution AADL. Ils proposent une sémantique formelle pour l'annexe en utilisant une machine à états abstraits temporisés (*TASM*). *TASM* étend le formalisme ASM pour permettre l'expression explicite du temps, des ressources, de la communication, de la composition, et du parallélisme. Dans un premier temps, la sémantique du modèle d'exécution AADL est donnée, puis une définition formelle de la sémantique des aspects de l'annexe comportementale est spécifiée. Une transformation du modèle AADL en utilisant le langage de transformation ATL (Atlas Transformation Langage) était nécessaire pour réaliser une simulation et vérification du modèle.

Récemment, les auteurs dans (Benammar 2011) ont utilisé la logique de réécriture révisée comme formalisme pour décrire des propriétés d'exécution au sein d'une configuration AADL. Ils se sont intéressés aux propriétés du

composant *Thread*. Pour valider le modèle résultat, nommé ABaREL, ils ont utilisé le langage formel *MAUDE*.

Modèle Formel de Base	Éléments AADL Spécifiés	Langage Utilisé	Type d'Analyse
Système de Transition Temporisé (TTS) (Berthomieu et al. 2009)	X	FIACRE	Model-Checking
Système de Transition Étiqueté (Abdoul et al. 2008)	X	IF	Accessibilité, Inter-blocage
Système de Transition Étiqueté (Chkouri et al. 2008)	Thread, Processus, Processeurs	BIP	Inter-blocage
Modèles à base des Réseaux de Petri Temporisés (TPN Ordres Partiels) Vergnaud et al. (2004)	Interactions	VTS	Accessibilité, Inter-blocage
Machines à états Abstraits Temporisés (TASM) (Yang et al. 2009)	X	ATL	Consommation de ressources, Synchronisation
Modèles à base d'Algèbre de Processus (ACSR) (Sokolsky et al. 2009)	X	X	Ordonnancement, Synchronisation
ABAReL (Benammar 2011)	Threads	Maude	Propriétés d'exécution temporelles, Accessibilité, Inter-blocage et Model-cheking.

TABLE 4.1 – *Classification des Approches de Formalisation*

Ces travaux de formalisation, malgré leur nombre, variété et diversité, se sont focalisés sur des points particuliers des systèmes AADL. Ils ne se sont pas intéressés à la configuration générale d'une spécification AADL ni aux aspects matériels qu'elle peut contenir et non plus à la relation qui peut lier un composant logiciel à un composant matériel de la même spécification. Un autre aspect fut négligé, et que nous considérons important et pratique à un

certain niveau d'abstraction, il s'agit de l'absence de l'aspect graphique dans les formalismes utilisés.
Cela dit, il est envisageable d'utiliser l'un de ces formalismes pour formaliser et vérifier d'autres aspects de la configuration AADL tels que les propriétés d'exécution et les propriétés non fonctionnelles.

D'un autre côté, le graphe étant un formalisme mathématique, a toujours constitué un modèle formel et visuel pour définir une architecture logicielle. Les nœuds représentent les composants de l'architecture et les arcs définissent alors les connecteurs entre les différents composants. En plus, les opérations de reconfiguration structurelle d'une architecture peuvent être vues comme des opérations de transformation de graphes. D'autre part, l'introduction de la notion de typage dans les graphes permet non seulement de travailler sur une architecture logicielle donnée, mais sur tout un style architectural, et ce en définissant le vocabulaire des éléments conceptuels d'une architecture et l'ensemble de règles indiquant comment ces éléments peuvent être connectés.

Dans ce chapitre, nous proposons un modèle à base de bigraphes, permettant la modélisation d'une architecture logicielle et la plateforme d'exécution associée, exprimées dans un langage de description tel que AADL. Nous projetons ensuite cette modélisation sur les styles architecturaux où nous présentons la capacité du formalisme choisi (BRS) à les modéliser (Benlahrache et al. 2010).

4.2 Principe de Base

Le passage d'une architecture AADL vers un bigraphe n'est pas une étape directe, mais nécessite une étape intermédiaire, essentielle et importante qui consiste en la définition des éléments AADL en termes d'éléments bigraphiques. Les principaux éléments d'une instance architecturale dans AADL doivent trouver leur définition en termes des concepts d'un bigraphe tout en préservant la sémantique tels que les composants, les connecteurs, les ports, les rôles et les configurations. Une correspondance est alors définie entre tous ces éléments dans le langage AADL et ceux d'un bigraphe. Pour cela, nous nous sommes appuyés sur des travaux antérieures et spécialement sur ceux de Chang et al. (2007). La description suivante essaye de résumer ces correspondances :

– Une configuration AADL contenant plusieurs composants peut être représentée formellement par un bigraphe où les nœuds correspondent aux différents composants et connecteurs de cette configuration.

– L'aspect hiérarchique de la description AADL est pris en charge par le concept d'imbrication des nœuds dans un bigraphe. La structure arborescente du graphe des places permet de visualiser la hiérarchie des composants ou connecteurs composites.
– De plus, la notion de contrôleur d'un nœud (composant ou connecteur) sert à définir les interfaces d'un composant (ports) ou celles d'un connecteur (rôles) ainsi que les comportements associés. Chaque arc décrit une connexion entre les ports et les rôles.

La table suivante (table 4.2) montre la correspondance entre tous ces éléments (Benlahrache et al. 2011).

Elément Architectural AADL	**Sémantique en termes de Bigraphe**
Configuration (composant system)	Bigraphe / Région
Composant : Logiciel, Matériel, Connecteur	Nœud
Port /Rôle	Port / Nom Interne ou externe
Interaction port-rôle	Hyperarc
Hiérarchie	Imbrication des Nœuds / Sites
Mode	Bigraphe Redex/Reactum
Transition entre Modes	Règle de Réaction

TABLE 4.2 – *Formalisation des éléments architecturaux AADL.*

Partant du principe qu'une spécification AADL n'est considérée complète et exhaustive que si et seulement si la description de l'aspect matériel qui servira de plateforme d'exécution pour l'aspect logiciel de la même spécification AADL est clairement spécifié. Nous attribuons alors dans notre démarche de modélisation, à chaque configuration AADL d'un système un couple de bigraphes G_S et G_H (S pour software, H pour hardware) pour modéliser respectivement la partie logicielle et matérielle du système.

De point de vue AADL, les entités de la partie logicielle du système ont besoin d'être déployées sur des composants matériels. Cette relation est définie à travers les propriétés AADL de type *Binding* telle que la propriété "*Actual...Binding*" (voir section 2.2.5). Nous nous appuyons sur les occurrences de cette propriété afin de pouvoir déterminer quel composant matériel défini dans le bigraphe G_H sera lié à quel composant logiciel également défini dans le bigraphe G_S. La syntaxe de cette propriété étant de la forme suivante :

$$Actual...Binding => \textbf{\textit{reference}} < composant\_matriel > \textbf{\textit{applies to}} < composant\_logiciel >$$

Les informations fournies par cette propriété sont utilisées d'une part, pour construire les deux bigraphes, G_H et G_S, associés à la déclaration d'une configuration AADL et d'autre part pour établir la relation qui peut les lier en termes de *sites*, régions et noms internes/externes. Le nombre d'occurrences de cette propriété indique globalement le nombre de sites et de régions dans le bigraphe. Le terme <*composant_matériel*> désigne un site ayant un numéro et présentant un *nom interne* suivi du signe '+' dans notre notation.
De même, le terme <*composant_logiciel*> désigne une région ayant un numéro et un *nom externe* que nous ferons suivre par le signe '-'. En particulier, nous notons les deux éléments précédents, s'ils sont reliés par la même propriété, avec un même libellé et un même numéro mais avec des signes opposés. Évidemment, si deux éléments logiciels ou plus, de même catégorie, font référence au même élément matériel alors ils seront contenus dans la même région et émettront via le même hyper-arc. Les signes - et + permettent de distinguer entre nom interne et externe.

Les étapes nécessaires pour passer d'une spécification AADL vers un modèle bigraphique sont résumées dans l'algorithme 4.3.

Un composant est défini par un *Contrôleur* dans le langage bigraphique. Le contrôleur d'un composant contient plusieurs information sur celui-ci tels que le nom, le nombre de ports, son type (atomique ou non) ainsi que sa dynamique.

Les noms internes et externes, dans notre modèle sont porteurs d'information sur les dépendances de déploiement. Ces informations sont des facteurs importants lors de l'opération d'installation pour assurer son bon déroulement. Il est important de s'assurer que les composants de la plateforme d'exécution sont capables de satisfaire les exigences des composants logiciels telles que l'espace mémoire disponible, la période et la date limite des threads, la vitesse et la politique d'ordonnancement du processeur, et les contraintes de partage. Pour compléter la modélisation et pour représenter les informations portées par les noms internes et externes, nous empruntons la syntaxe XML utilisée par l'outil Osate. La table 4.4 résume la disposition des informations, en format XML, concernant ces propriétés.

4.3 Aspects Logiciels

Le bigraphe G_S, désignant la partie logicielle d'une configuration architecturale AADL, est donné par la définition suivante :

Algorithme : transformer une spécification AADL en Bigraphes

Entrées : spécification AADL complète et valide

Sorties : 2 bigraphes G_S et G_H (au moins) ;

Début

A. **Pour** tout composant C_s de type *system* **faire** créer deux bigraphes G_S et G_H ;

B. **Pour créer** G_S **faire**

1. **Pour** tout composant-software c_i **faire**
   - Ajouter un nœud c_i à G_S
   - contrôleur(c_i).nom = nom du composant c_i ;
     // '.'désigne l'imbrication
   - contrôleur(c_i).port←nombre d'interfaces de c_i ;
2. **Si** nombre (sous-composants + nombre de connexions) de $c_i \neq 0$ **alors**
   contrôleur(c_i).type ← non-atomique
   sinon contrôleur(c_i).type ← *atomique* ;
3. **Si** *modes* de c_i=vrai **alors**
   contrôleur(c_i).dyn ← *Dynamique* ;
4. **Pour** toute connexion cnx_i de c_i de la forme $cnx_i : a \rightarrow b$ **faire**
   ajouter un arc de a vers b ;
   // a et b des ports de c_i
5. **Pour** toute propriété *Actual* ... *Binding* **faire**
   ajouter un *nom-externe* au c_i ;
6. **Si** contrôleur(c_i).type = *non-atomique* **alors**
   Pour tout sous-composant de c_i **faire**
   aller à 1.

C. **Pour** créer G_H (ses nœuds sont les sous composants matériels) **faire**
Pour tout composant matériel c_j **faire**

1. Répéter les étapes de B. (de 1 à 6) ;
2. **Pour** toute propriété *Actual* ... *Binding* de c_j **faire**
   - ajouter un site au nœud c_j
   - ajouter un *nom-interne=nom-externe* de c_i correspondant.

Fin.

TABLE 4.3 – *Algorithme de Génération des Bigraphes*

```
<outername name="">
    <region, id-region=""/>
    <soft-component name="component-name">
        <property name="" value=""/>
            ⋮
        <property name="" value=""/>
    </soft-component>
</outernanme>
```

```
<innername name="">
    <site, id-site="" />
    <hard-component name="component-name " >
        <property name="" value=""/>
            ⋮
        <property name="" value=""/>
    </hard-component >
</innernanme>
```

TABLE 4.4 – *Spécification des interfaces internes et externes en XML.*

Définition 1 : *Le bigraphe associé à la description logicielle est défini par :* $G_S = (V_S, E_S, Ctrl_S, G_S^P, G_S^L) : I_S \rightarrow J_S$ *tels que (Benlahrache et al. 2011) :*

- V_S *est un ensemble fini de composants et sous composants de la partie logicielle AADL ainsi que les connexions définies entre les différentes interfaces de ces composants ;*
- E_S *est un ensemble d'hyper-arcs représentant les interactions entre composants logiciels AADL ;*
- $Ctrl_S : V_S \rightarrow K_S$ *est une transformation qui associe à chaque composant AADL* v_i *de* V_S *son contrôleur* $k_i \in K_S$ *indiquant en plus du nombre de ports d'interconnexion de ce composant, l'atomicité et la dynamique de celui-ci ;*
- $G_S^P = (V_S, E_S, Ctrl_S, prnt_S) : m_S \rightarrow n_S$ *est le graphe des places associé à* G_S*, où* $prnt_S = m_S \cup V_S \rightarrow n_S \cup V_S$ *est la fonction de parenté indiquant l'imbrication des sous-composants logiciels dans les composants conteneurs tel que un* thread *dans un composant* process*. Ce graphe assure une vision structurelle et hiérarchique des différents composants contenus dans une spécification AADL.*
- $G_S^L = (V_S, E_S, Ctrl_S, link_S) : X_S \rightarrow Y_S$ *est le graphe des liens de* G_S*, où* $link_S = X_S \cup P_S \rightarrow Y_S \cup E_S$ *est une transformation montrant le flux de données/événements des ports* P_S *vers les arcs* E_S*. Ce flux décrit d'une façon claire et visuelle les interactions entre les composants au sein d'un même système.*

Les noms internes X_S et les noms externes Y_S permettent de designer les interactions de ce système avec son environnement contextuel.

– *$I_S = \langle m_S, X_S \rangle$ et $J_S = \langle n_S, Y_S \rangle$ sont respectivement les interfaces internes et externes du graphe G_S, avec m_S est le nombre de sites, X_S est l'ensemble des noms internes. Un site prêt à recevoir une région doit présenter, lui ou son conteneur, un nom interne. n_S est le nombre de régions, éléments du bigraphe pouvant représenter des composants logiciels ayant besoin de s'intégrer dans d'autres composants logiciels ou matériels. Y_S est l'ensemble des noms externes.*

Exemple 1 : Pour illustrer la définition ci-dessus, nous l'appliquerons à l'exemple AADL présenté dans le chapitre 2 section 2.6. L'exemple traite une communication entre deux threads Émetteur et Receveur.

Le bigraphe G_S formalisant la partie logicielle du système de cet exemple, déduit en appliquant notre approche de formalisation est donné par :

– $V_S = \{the\_process, thr\_S, thr\_R, cnx\}$
– $E_S = \{e0, e1\}$
– $ctrl_S = \{(the\_process : 1), (thr\_S : 2), (thr\_R : 2), (cnx : 3)\}$
– $prnt_S = \{(the\_process : \emptyset), (thr\_S : the\_process), (thr\_R : the\_process), (cnx : the\_process)\}$.

La figure 4.1 schématise clairement cet exemple. Dans ce cas de figure :

– $m_S = 0$ et $X_S = \emptyset$. Ce qui fait que $I_S = \langle 0, \emptyset \rangle = \varepsilon$ pas de noms internes ni de sites disponibles ;
– $n_S = 1$ et Y_S contient trois noms externes $\{m^-, c^-, x^-\}$. Alors $J_S = \langle 1, \{m^-, c^-, x^-\} \rangle$; cette interface externe indique que trois éléments du bigraphe G_S nécessitent une opération d'installation et que ces trois éléments exigent des composants matériels différents (m, c, x).
– L'hyperarc x^- indique le partage du processeur par les threads *thr_S* et *thr_R*.

Finalement le bigraphe est défini par ses interfaces tel que $G_S = \varepsilon \rightarrow \langle 1, \{m^-, c^-, x^-\} \rangle$.

À partir du bigraphe construit à partir de la spécification AADL, nous pouvons extraire les deux structures (graphes des places et graphe des liens) et les exploiter pour analyser une architecture AADL. Le graphe des places (voir figure 4.2.a) représente clairement l'imbrication des composants. Nous pouvons remarquer que le composant *the_process* possède au moins un sous composant de type thread, ce qui est nécessaire pour que le processus soit

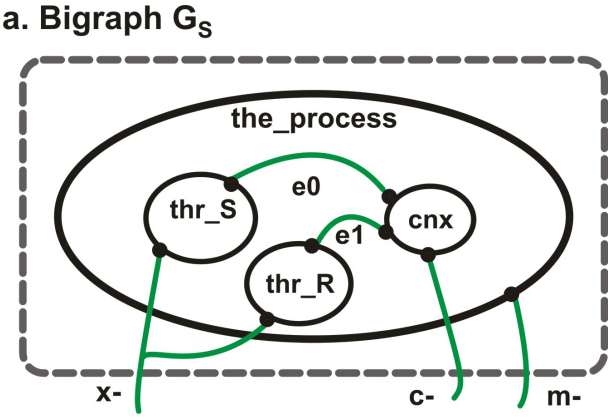

FIGURE 4.1 – *Bigraphe Logiciel G_S*

actif. Le graphe des liens (voir figure 4.2.b) permet de vérifier les connexions qui du point de vue AADL ne sont valides que si elles permettent de relier un thread à un autre quelque soit leur niveau d'imbrication sinon la connexion est considérée défaillante et toute communication entre composants est impossible.

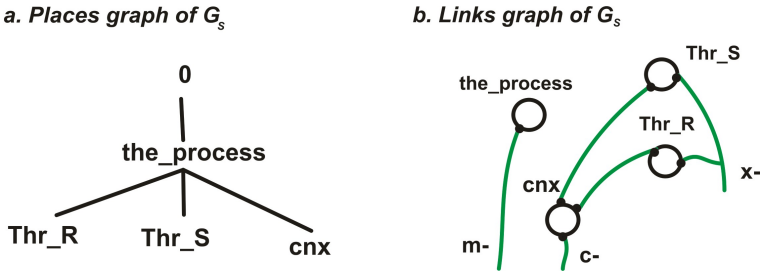

FIGURE 4.2 – *Graphes des Places et des Liens*

4.4 Aspects Matériels

La plateforme devant servir pour l'exécution de l'exemple cité dans la section 4.3, sera elle aussi représentée par un bigraphe. Ces éléments sont les composants AADL utilisés pour spécifier l'architecture. Ainsi, et de façon similaire, nous formalisons les éléments matériels du bigraphe G_H .

Définition 2 : *Le bigraphe définissant la plateforme d'exécution cible d'une déclaration AADL est donné par $G_H = (V_H, E_H, Ctrl_H, G_H^P, G_H^L) : I_H \rightarrow J_H$, tel que (Benlahrache et al. 2011) :*

- *V_H est un ensemble fini de composants matériels qui peuvent être imbriqués les uns dans les autres ; par exemple une mémoire dans un processeur ;*

- E_H *est un ensemble fini d'hyper-arcs, représentant les interactions entre ces composants matériels tels que entre un processeur et un bus ou une mémoire et un bus ;*
- $Ctrl_H : V_H \rightarrow K_H$ *associe à chaque composant matériel* $v_i de V_H$ *un contrôleur indiquant le nombre de ses interfaces ;*
- $G_H^P = (V_H, E_H, Ctrl_H, prnt_H) : m_H \rightarrow n_H$ *est le graphe des places associé à* G_H*, où* $prnt_H = m_H \cup V_H \rightarrow n_H \cup V_H$ *; généralement prnt peut s'avérer utile uniquement si le composant considéré est un processeur possédant une mémoire. L'imbrication au niveau des composants matériels n'est pas toujours possible.*
- $G_H^L = (V_H, E_H, Ctrl_H, link_H) : X_H \rightarrow Y_H$ *est le graphe des liens de* G_H*, où* $link_H = X_H \cup P_H \rightarrow Y_H \cup E_H$ *montre le flux de données des ports* P_H *vers les arcs* E_H *et celui des interactions des noms internes* X_H *et les noms externes* Y_H *avec l'extérieur ;*
- $I_H = \langle m_H, X_H \rangle$ *et* $J_H = \langle n_H, Y_H \rangle$ *sont respectivement les interfaces internes et externes du graphe* G_H*, avec* m_H *est le nombre de sites i.e. emplacements dans le graphe représentant la disponibilité des composants matériels nécessaires pour la réussite de l'opération d'installation des composants logiciels.* X_H *est l'ensemble des noms internes, représentant les interfaces des sites.* n_H *est le nombre de régions.* Y_H *est l'ensemble des noms externes.*

Exemple 2 : Reprenons la description AADL de l'exemple de la section précédente, le bigraphe de la figure 4.3 désigne la partie matérielle de cette spécification AADL.

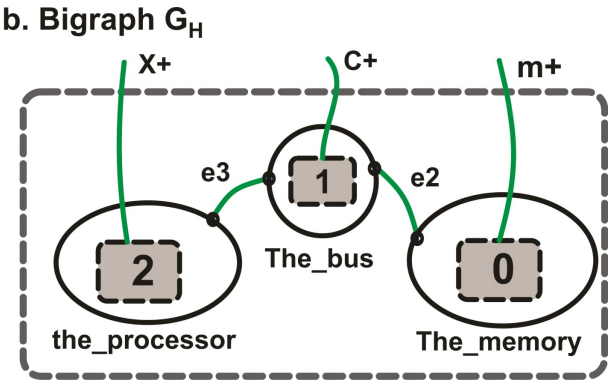

FIGURE 4.3 – *Bigraphe Matériel* G_H

Les propriétés d'exécution des composants de cet exemple et qui peuvent exprimer des dépendances entre les composants logiciels et matériels correspondants, peuvent être formulées en XML comme suit (table 4.5) :

< *processorImplname* = "**Processor1.impl**"
compType = "/*aadlSpec*[*name* = *Send*$_R$*eceive*]/*processorType*[**@name=Processor1**]" >
<**properties**>
<propertyAssociation *propertyDefinition*[**@name=Scheduling_Protocol**]" >
< **propertyValue** = **RMS**..."/ >
</propertyAssociation>
</**properties**>
<**subcomponents**>
< *memorySubcomponent***name** = "**mem**"...*classifier* = "[**@name=Memory1.impl**]"/ >
</**subcomponents**>
</**processorImpl**>

Table 4.5 – *Spécification AADL du Composant "Processor1.impl" exprimée en XML.*

4.5 Formalisation des Styles Architecturaux

En effet, plusieurs approches dans la littérature adoptent le concept des graphes et leurs transformations pour modéliser les styles architecturaux et leurs instances. Dans Metayer (1998), les auteurs proposent de lier les styles architecturaux aux grammaires de graphes pour pouvoir considérer par la suite les reconfigurations d'architectures comme étant des transformations de graphes. Cette même idée a été exploitée dans Bruni et al. (2007) et étendue par une approche hiérarchique plus générique (ADR) facilitant la représentation des règles de reconfiguration plus complexes ayant comme paramètres des architectures typées.
Autrement dit, les styles architecturaux sont spécifiés via des algèbres ayant des modèles d'interprétation à base de graphes. Les règles de construction d'une architecture (ou productions) sont vues comme étant des opérations de composition d'architectures élémentaires, selon leurs types, donnant des résultats bien définis. Ainsi, les reconfigurations d'architectures avec préservation de styles peuvent être exprimées par des règles de la réécriture de termes algébriques formés autour des informations sur ces architectures.

L'originalité de ces travaux réside dans le fait qu'ils utilisent un seul modèle unificateur, réécriture des graphes, pour représenter les architectures logicielles, leur comportement ainsi que leurs reconfigurations. D'autres travaux se référant aux mêmes objectifs sont ceux de Chang et al. (2007) qui partent du constat de la nécessité de définir des types de composants, de connecteurs et de ports/rôles pour former un style, sans négliger la spécification des sous structures récurrentes exigées, ainsi que leurs propriétés et

contraintes. Ils proposent d'utiliser les BRS pour modéliser un style architectural d'un système en tenant compte de toutes ses spécificités. Les types associés aux interfaces expriment le typage des ports/rôles et les attributs des contrôleurs dénotent les types des composants et des connecteurs. Dans leur contexte de travail, une architecture est définie donc par un bigraphe alors que la famille des bigraphes déterminés par un BRS forme son style. En outre, ils utilisent les règles de réaction du BRS pour représenter les reconfigurations de ces architectures et s'assurer qu'elles préservent leur style.

Toutes ces approches (Metayer 1998, Bruni et al. 2007, Chang et al. 2007) etc., adoptent certes le concept des graphes pour modéliser les styles architecturaux et leurs instances, mais sans pour autant se soucier de la modélisation des *plateformes d'exécution* qui vont supporter le déploiement de ces instances. En s'inspirant de l'idée de Chang et al. (2007; 2008) concernant la modélisation des styles architecturaux en termes de BRS, nous avons proposé dans (Benlahrache et al. 2010), une approche préliminaire pour la modélisation des architectures logicielles et les plateformes d'exécution associées.

La différence et l'intérêt de cette approche de modélisation d'un style d'architectures par rapport aux autres approches existantes, est la capacité du formalisme sous-jacent (BRS) à représenter naturellement des systèmes complexes réels, en particuliers les systèmes mobiles et sensibles au contexte (*context-aware*). En effet, leur modèle sémantique est basé sur la théorie des catégories. Cette base mathématique leur prodigue un moyen visuel assez simple permettant de fournir des informations sur les deux aspects statiques et dynamiques d'un système.

Nous reprenons ici les détails des points issus des travaux de Chang et al. (2007) et Chang et al. (2008), que nous jugeons importants pour notre approche de modélisation des styles :

- Les architectures peuvent être regroupées dans un style architectural qui spécifie uniquement les contraintes les plus importantes, au niveau par exemple de la structure, du comportement, de l'utilisation des ressources, des composants et des connecteurs dans un système. Du moment qu'un bigraphe peut représenter une architecture, alors tous les bigraphes déduits d'un BRS forment un style.
- Des conditions supplémentaires peuvent enrichir ce modèle pour décrire les contraintes d'un style donné. Il peut ainsi aider à la définition de la sémantique d'un système et son analyse.
- De plus, les instances d'architectures d'un style ont souvent besoin

d'être reconfigurées durant leurs exécutions. Ces changements possibles conformément à un style sont définis par les règles de transformation (réaction) d'un BRS.

Notre approche de modélisation est composée essentiellement de trois phases. Dans un premier temps, nous utilisons un BRS pour décrire la configuration d'une application comme une instance d'un style architectural donné (figure 4.4.1). En second lieu, un autre BRS va servir pour décrire la plate-forme d'exécution chargée d'accueillir les composants de l'application lors de son installation durant un processus de déploiement (figure 4.4.2).

La première phase nous permet de spécifier uniquement les contraintes les plus importantes de l'architecture, au niveau par exemple de la structure, du comportement, de l'utilisation des ressources, des instances de composants à créer et la configuration de leurs attributs ainsi que les liaisons de leurs ports. Pour cela, nous adoptons un méta modèle à base des bigraphes, proposé dans Chang et al. (2007) pour définir formellement un style architectural regroupant une famille d'architectures ayant un ensemble de propriétés communes.
Ce modèle est spécifié formellement par la définition suivante :

Définition 3 : *Un style architectural S d'une architecture AL est défini par un BRS* $Cat_{AL} = (B_{AL}, R_{AL})$. *La catégorie de bigraphes* Cat_{AL} *est composée d'un ensemble de bigraphes* B_{AL} *et d'un ensemble de règles de réactions* R_{AL} *sur ces bigraphes. Un bigraphe F de* $B_{AL} = (V_F, E_F, Ctrl_F, G_F^L, G_F^P) : I \longrightarrow J$ *où :*

- V_F *est l'ensemble des composants et connecteurs de l'architecture,*
- E_F *est l'ensemble des hyper-arcs représentants les différentes interactions entre ports et rôles des éléments de* V_F *,*
- $Ctrl_F$ *est le contrôleur associé à l'ensemble des nœuds de* V_F*,*
- G_F^L *et* G_F^P *représentent respectivement le graphe de Places et le graphe de Liens du bigraphe F.*
- $I_F = \langle m_F, X_F \rangle = \epsilon$*, l'instance de cette architecture est sans sites ni noms internes. Cela signifie que la structure architecturale de l'application correspondante est complète et n'a pas besoin d'être enrichie par d'autres composants.*
- $J_F = \langle n_F, Y_F \rangle$ *où* n_F *désigne la portée de la distribution spatiale de l'architecture logicielle considérée, et* Y_F *désigne l'ensemble des interfaces externes d'interaction avec l'environnement de déploiement. Ces interfaces expriment également les différentes dépendances que l'environnement doit satisfaire pour réussir un déploiement.*
- R_{AL} *: constitue l'ensemble des règles de réactions possibles sur ce bigraphe et*

qui représentent en termes d'architecture logicielle, un ensemble d'opérations de reconfiguration possible sur cette instance d'architecture.

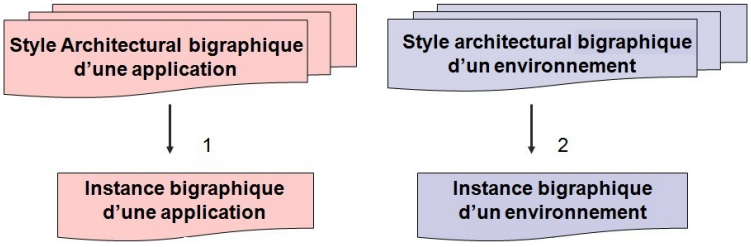

FIGURE 4.4 – *Formalisation d'un Style Architectural*

La deuxième phase consiste en la spécification d'un autre BRS noté $Cat_{ENV}(S) = (B_{ENV}, R_{ENV})$, utilisé pour décrire la plateforme d'exécution chargée d'accueillir les composants logiciels de l'application lors de son déploiement. R_{ENV} représente l'ensemble des règles de réaction sur les bigraphes B_{ENV} de Cat_{ENV}. Cette modélisation explicite des plateformes favorise la prise en compte de leurs caractéristiques et contraintes.

Définition 4 : *Soit H un bigraphe de B_{ENV} représentant l'environnement cible, $H = (V_H, E_H, Ctrl_H, G_H^P, G_H^L) : I_H \rightarrow J_H$ tels que :*

- *$V_H, E_H, Ctrl_H, G_H^P$ et G_H^L sont les principaux éléments du bigraphe,*
- *$I_H = \langle m_H, X_H \rangle$, avec $m_H \neq 0$ et $X_H \neq \Phi$, Cette double condition exprime que l'environnement doit être capable d'offrir des sites (m_H) pour héberger des applications logicielles et des interfaces internes (X_H) servant à l'interaction. Les interfaces internes peuvent alors désigner dans ce cas les dépendances de déploiement que peut satisfaire l'environnement vis-à-vis d'une application logicielle donnée.*
- *$J_H = \langle n_H, Y_H \rangle$, avec éventuellement $n_H \geq 0$ et $Y_H \neq \Phi$ en cas d'un environnement distribué dans l'espace.*
- *R_{ENV} définit l'ensemble des opérations de reconfiguration possibles sur l'environnement.*

L'opération de construction du graphe de places va permettre d'identifier les *sites* que peut contenir le bigraphe de l'environnement de déploiement H. Ces sites représentent des emplacements libres dans l'environnement et susceptibles d'accueillir des régions en provenance de l'application à déployer, on parle alors d'hébergement. Ces sites peuvent être : un espace mémoire disponible, un processeur libre, une bibliothèque prête à être exploitée etc. La même opération conduit à l'identification des régions dans F qui sont des

zones *relogeables* de l'application.

L'opération de construction du graphe de liens va permettre d'identifier les interfaces internes que peut contenir le bigraphe de l'environnement de déploiement H. Ces interfaces représentent des dépendances que l'environnement doit satisfaire et qui doivent correspondre aux interfaces externes de l'application.

La dernière phase est une conséquence directe d'une telle formalisation, elle est consacrée à la description formelle du processus de déploiement, en utilisant les opérations sur des bigraphes appliquée dans notre contexte aux bigraphes représentant respectivement l'architecture logicielle d'une application et celle de l'environnement matériel qui va la supporter (figure 4.5.(3)).

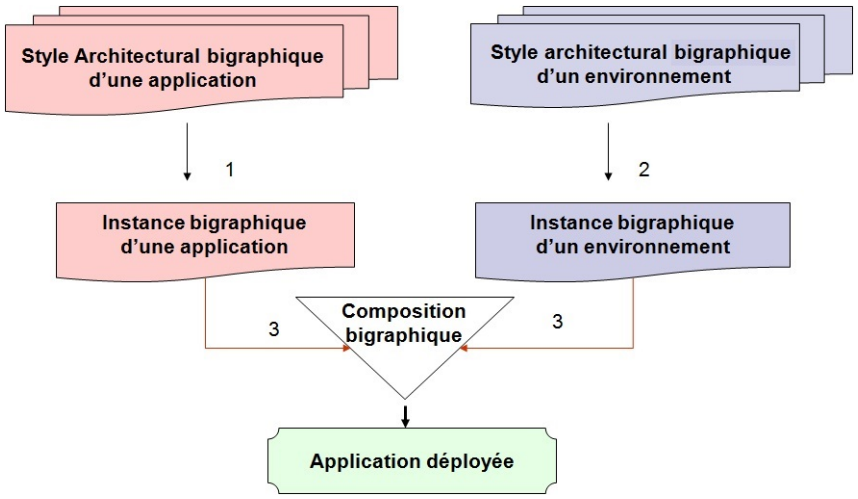

FIGURE 4.5 – *Déploiement Architectural*

4.6 Cas d'exemple : Patient Monitoring System

Afin d'illustrer notre approche, nous avons utilisé un exemple (Benlahrache et Belala 2013) qui a également servi pour représenter d'autres formalismes proposés dans Metayer (1998) et Hadj Kacem (2008).

Le système PMS (Patient Monitoring System) est un système logiciel de contrôle de patients à distance. Ce système permet aux infirmières d'un service hospitalier ou d'une clinique de contrôler leurs patients à distance. À chaque patient, un contrôleur est attaché à son lit permettant de prendre des mesures. Ces mesures sont généralement appelés les constantes physiologiques vitales d'un patient car elle ne doivent pas varier constamment ni

prendre des valeurs non tolérées. Nous citons comme mesure celle de la fréquence cardiaque ou pouls, la pression artérielle, saturation en Oxygène et la température.
Grâce à ce contrôleur, chaque infirmière peut vérifier l'état de son patient en demandant à distance les informations le concernant. Par contre, lorsque l'état d'un patient devient anormal, c'est à dire quand l'une de ces constantes vitales sort de sa limite, le contrôleur de lit envoie un signal d'alarme à l'infirmière responsable.

4.6.1 Présentation Architecturale du PMS

L'architecture de ce système est évolutive et composée essentiellement de ces différents composants (Metayer 1998) :

- le 'Contrôleur'(*Controller* : l'équipement directement branché au patient),
- Un composant 'Infirmière' (appelé ici *Nurse*) joue le rôle d'interface de communication entre la personne infirmière et le composant gestionnaire des événements,
- un composant central 'Service d'événements' (*ServiceEvent*) dont le rôle est d'assurer la communication et la gestion des événements entre les deux composants : *Controller* et *Nurse*.

Le composant *ServiceEvent* a particulièrement la mission de décrypter l'information parvenue du contrôleur de patient et d'envoyer si nécessaire un signal d'alarme à l'infirmière. Ces composants communiquent via un bus de communication dont ils partagent l'usage. La figure 4.6 présente l'aspect générale du système globale du PMS.
Suite à cette brève description informelle du système, une description dans le langage AADL fera l'objet de la section suivante.

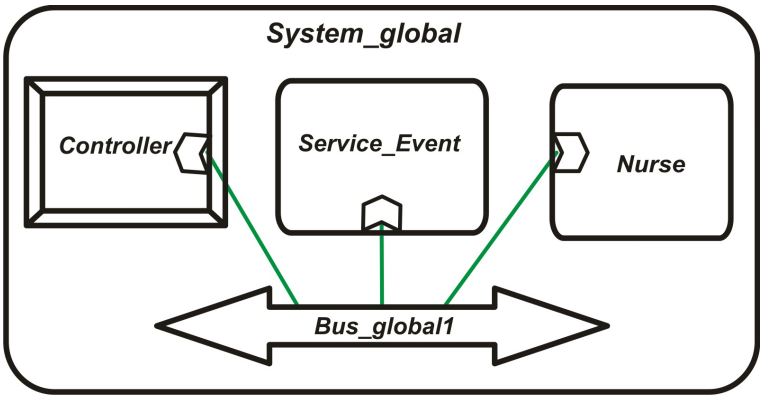

FIGURE 4.6 – *Architecture du Système PMS*

4.6.2 Spécification AADL

Dans ce qui suit, nous présentons les éléments essentiels de la description AADL du Système PMS. Le choix des parties présentées ici, est fait de manière à mettre en évidence les concepts abordés dans cette thèse. Pour plus de détails, une annexe contenant toute la spécification AADL est ajoutée à la fin de ce document. La figure 4.7 présente une portion du code AADL ayant servi à la spécification du module global décrivant le système PMS en AADL.

```
system global
end global;

system implementation global.impl
subcomponents
   Service_Event : system Service_Sys.impl;
   Nurse  : system Nurse_Sys.impl;
   Controller: device control.impl;
   Bus_global1: bus bus_gl.impl;
connections
   cnx0: event port Service_Event.out_alarm -> Nurse.in_alarm;
   cnx1: event port Nurse.out_answer -> Service_Event.in_answer;
   cnx2: event data port Controller.out_pressure -> Service_Event.in_pressure;
   cnx3: event data port Controller.out_temp -> Service_Event.in_temp;
properties
   Actual_Connection_Binding =>  reference Bus_global1 applies to cnx0;
   Actual_Connection_Binding =>  reference Bus_global1 applies to cnx1;
   Actual_Connection_Binding =>  reference Bus_global1 applies to cnx2;
   Actual_Connection_Binding =>  reference Bus_global1 applies to cnx3;
   Actual_Connection_Binding =>  reference Bus_global1 applies to Service_Event.Ser_pro;
   Actual_Connection_Binding =>  reference Bus_global1 applies to Nurse.Nur_pro;
end global.impl;
```

FIGURE 4.7 – *Spécification AADL du PMS*

La figure 4.8 est également une portion du code dans laquelle nous pouvons découvrir les principales clauses décrivant les composants constituant le sous système *Service_Event*. Nous pouvons distinguer les principales parties de cette déclaration : *Service_Sys* et *Service_Sys.impl* qui représentent respectivement les parties *Type* et *Implementataion* d'un module AADL. La partie *Service_Sys.impl* décrit les sous composants de ce sous système, les connexions définies entre ses composants ainsi que les principales propriétés de type ""Actual....Binging". La figure 4.9 est un schéma illustratif d'une telle spécification.

Pour plus de détails, la table 4.6 décrit les sous-composants logiciels du sous système *Service_Event*. Ces composants sont essentiellement le processus *Service_process* et ses différentes connexions.

La table 4.7 présente la déclaration des composants matériels constituant la plateforme d'exécution associée à *Service_event*.

Dans la table 4.8, les détails de l'implémentation du composant *Ser_Thread* sont présentés.

process Service_process **features** out_alarm : out event port ; in_answer : in event port ; in_pressure : in event data port pressure ; in_temp : in event data port temp ; **end** Service_process ;	**process implementation** Service_process.impl **subcomponents** Ser_thr : thread Service_thread.impl ; **connections** cnx14 : event port Ser_thr.out_alarm− >out_alarm ; cnx15 : event port in_answer− > Ser_thr.in_answer ; cnx16 : event data port in_pressure− > Ser_thr.in_pressure ; cnx17 : event data port in_temp− >Ser_thr.in_temp ; **properties** Actual_Connection_Binding=>reference Ser_bus applies to cnx14 ; Actual_Connection_Binding=>reference Ser_bus applies to cnx15 ; Actual_Connection_Binding=>reference Ser_bus applies to cnx16 ; Actual_Connection_Binding=>reference Ser_bus applies to cnx17 ; **end** Service_process.impl ;

TABLE 4.6 – *Description AADL des sous-composants de "Service_event"*

processor PMS_processor **features** busAcc1 : requires bus access bus_gl ; **end** PMS_processor ;	**processor implementation** PMS_processor.impl **subcomponents** mem : memory PMS_memory.impl ; **end** PMS_processor.impl ;
memory PMS_memory **features** bus1 : requires bus access **end** PMS_memory ;	**memory implementation** PMS_memory.impl – without particularities **end** PMS_memory.impl ;
bus bus_gl **end** bus_gl ;	**bus implementation** bus_gl.impl properties Propagation_Delay => 5 Ms..5 Ms ; Transmission_Time => 6 Ms..8 Ms ; **end** bus_gl.impl ;

TABLE 4.7 – *Description AADL des composants matériels du PMS*

```
system Service_Sys
features
 out_alarm : in out event port ;
 in_answer : in out event port ;
 in_pressure: in out event data port pressure;
 in_temp: in out event data port temp;
 end Service_Sys;

system implementation Service_Sys.impl
subcomponents
  Ser_pro: process Service_process.impl;
  Ser_mem: memory PMS_memory.impl;
  Ser_proc: processor PMS_processor.impl;
  Ser_bus: bus bus_gl.impl;
connections
   cnx10: event port Ser_pro.out_alarm -> out_alarm;
   cnx11: event port in_answer -> Ser_pro.in_answer;
   cnx12: event data port in_pressure ->Ser_pro.in_pressure;
   cnx13: event data port in_temp -> Ser_pro.in_temp;
properties
  Actual_Memory_Binding =>  reference Ser_mem
  applies to Ser pro;
  Actual_Processor_Binding =>  reference Ser_proc
  applies to Ser_pro;
  Actual_Connection_Binding => reference Ser_bus
  applies to Ser_pro;
 end Service_Sys.impl;
```

FIGURE 4.8 – *Spécification AADL du Service_Event*

Dans la section suivante, nous présentons l'application de notre approche de modélisation sur l'exemple *PMS*. La figure 4.10 donne l'aspect bigraphique de l'ensemble des entités logicielles qui composent le système *PMS*. Nous pouvons constater que le sous système *Service_Event* est au centre de ce système et interagit directement avec les autres composants : *Controller* et *Nurse*.

thread Service_thread	**thread implementation** Service_thread.impl
features	**properties**
out_alarm : out event port ;	Dispatch_Protocol=>aperiodic ;
in_answer : in event port ;	Compute_Execution_Time=>30 Ms .. 40 Ms ;
in_pressure : in event data port pressure ;	Period=>1 sec ;
in_temp : in event data port temp ;	**end** Service_thread.impl ;
end Service_thread ;	

TABLE 4.8 – *Description AADL du Service_thread*

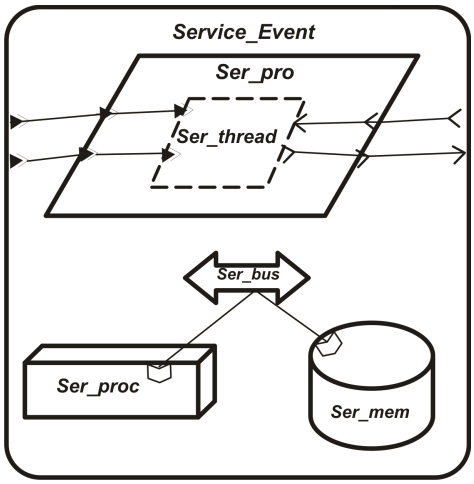

FIGURE 4.9 – *Schéma de la spécification AADL de Service_Event*

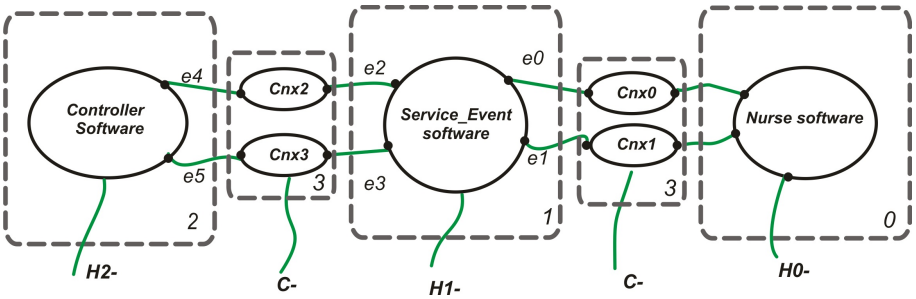

FIGURE 4.10 – *Vue bigraphique des entités logicielles du PMS*

4.6.3 Bigraphe pour la Description Logicielle

Dans cette section, nous présentons l'application des correspondances définies dans la table 4.2 afin de modéliser l'aspect logiciel par un bigraphe nommé G_S (S pour *S*oftware).

Pour cet exemple de système logiciel *PMS*, nous nous sommes intéressés au le sous-système *Service_Event*. La spécification AADL de ce sous-système est exhaustive et représente le schéma d'un système AADL complet (voir figure 4.8, table 4.6 et table 4.8).

Ainsi, nous identifions chaque élément du modèle bigraphique G_S comme suit :

- $V_S = \{Ser\_pro, Ser\_thr, cnx10, cnx11, ..., cnx17\}$;
- $E_S = \{e_0, e_1, ..., e_{15}\}$;
- $m_S = 0$ et $X_S = \emptyset$, il ne contient pas de noms internes, donc $I_S = \langle 0, \emptyset \rangle$. Ce bigraphe ne pourra pas héberger d'autres bigraphes. cette application est complète, pas besoin d'ajouter d'autres composants.
- $n_S = 2, Y_S = \{x^-, m^-, c^-\}$, $J_S = \langle 2, \{x^-, m^-, c^-\} \rangle$. Avec ces noms ex-

ternes, le bigraphe G_S exprime sa disponibilité à être hébergé sur trois sites différents.

Finalement, G_S exprimé par ses interfaces : $\langle 0, \emptyset \rangle \rightarrow \langle 2, \{x^-, m^-, c^-\} \rangle$.

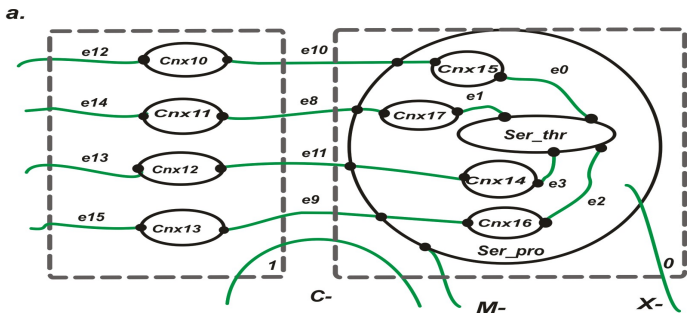

FIGURE 4.11 – *Vue bigraphique de la partie software du "Service_Event"*

4.6.4 Bigraphe pour la Description Matérielle

D'une façon similaire, nous attribuons à tous les composants matériels de la spécification AADL du système *PMS* un autre bigraphe G_H. Ce bigraphe est en relation avec le bigraphe G_S parce qu'ils sont issus de la même spécification AADL.

La figure 4.12 représente le bigraphe schématisant l'aspect matériel du sous système *Service_Event* du système *PMS*.

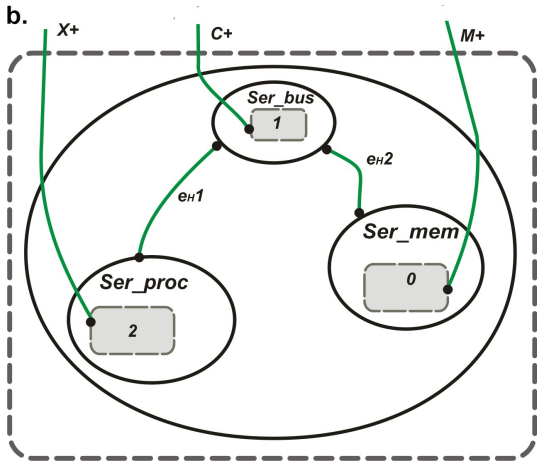

FIGURE 4.12 – *Vue bigraphique de la partie hardware du "Service_Event"*

L'application de la définition de la section 4.4 permet d'extraire les informations nécessaires à la définition du bigraphe G_H et cela a donné les éléments suivants :

- $V_H = \{Ser\_proc, Ser\_mem, Ser\_bus\}$;
- $E_H = \{e_{H1}, e_{H2}, \}$;
- $m_H = 3$ et $X_H = \{x^+, m^+, c^+\}$ ce qui donne $I_H = \langle 3, \{x^+, m^+, c^+\}\rangle$;
- $n_H = 1, Y_H = \emptyset$, il ne contient pas de noms externes. Ainsi, $J_H = \langle 1, \emptyset\rangle$.

En résumé, $G_H = \langle 3, \{x^+, m^+, c^+\}\rangle \rightarrow \langle 1, \emptyset\rangle$.

Il faut noter que dans le langage AADL, seule la mémoire (le composant matériel) peut être imbriquée dans un autre composant matériel qui est le processeur.

4.7 Conclusion

Dans ce chapitre, nous avons mis en évidence la capacité des BRS dans la modélisation des deux types de systèmes AADL, application et plateforme d'exécution, contenus dans la déclaration complète d'une configuration AADL. Nous leur avons associés deux bigraphes distincts G_S et G_H. Les bigraphes obtenues offrent une représentation claire de la structure de la spécification AADL, sa hiérarchie ainsi que les liens qui peuvent lier un composant logiciel à autre matériel. L'expression des dépendances entre les composants logiciels et matériels peut être spécifiée via les noms internes et externes des bigraphes dans des structures XML .

Les BRS offre également la possibilité de prise en charge de la spécification d'un style architectural que nous avons défini par une catégorie représentant un méta-modèle BRS : Cat_{AL} définissant le style architectural de l'application. De la même façon, nous avons défini le style de l'environnement d'exécution d'une instance par une autre instance du méta modèle BRS : Cat_{ENV}.

La formalisation bigraphique proposée ici donne une sémantique claire et sans ambiguïté à la spécification AADL et sa dynamique. L'aspect graphique du bigraphe permet de représenter clairement la structure hiérarchique de toute spécification AADL. En outre, les opérations inhérentes permettent de spécifier formellement des concepts architecturaux plus complexes, en particulier l'installation et la reconfiguration des composants des systèmes.

Le chapitre suivant sera dédié à la présentation du processus de déploiement qui n'est autre qu'une opération de composition de deux bigraphes issus des catégories Cat_{AL} et Cat_{ENV}.

5 Un Modèle Formel pour le Déploiement

Sommaire

Informatique : alliance d'une science inexacte et d'une activité humaine faillible.

Luc Fayard.

5.1 Introduction

Le déploiement d'un logiciel est un processus complexe et il a fait objet de plusieurs études et travaux durant les dernières décennies (voir Chapitre 2) dans le but de pouvoir estimer l'impact des choix et des techniques utilisés et déterminer le coût d'un tel processus souvent abordé d'une façon manuelle et ad-hoc. Le processus de déploiement architectural suit les principales étapes du processus de déploiement d'un logiciel mais il opère à un niveau d'abstraction élevé. Il offre plus de liberté et facilite la réflexion et le raisonnement sur les résultats obtenus. Il est souvent décrit comme un déploiement basé-modèle. Sauf que ces modèles souffrent soit de leur dépendances d'une technologie particulière et deviennent trop spécifiques, soit de l'absence d'une base formelle leur attribuant une sémantique claire.

AADL est un langage de description d'architectures doté de concepts lui permettant de décrire clairement les dépendances entre les éléments logiciels et matériels d'une spécification architecturale. Le déploiement est décrit à travers des propriétés liant chaque composant logiciel avec le composant matériel correspondant.

R. Milner et ses collaborateurs (Milner 2008) se sont appuyés sur la notion de mobilité pour motiver la séparation entre les places et les liens dans un bigraphe. Ils ont alors distinguer entre l'évolution structurelle et l'évolution comportementale d'un bigraphe. Nous exploitons ces aspects pour décrire l'évolution d'une architecture logicielle. Nous considérons aussi les aspects structurels et comportementaux d'une opération d'installation d'une application dans un environnement cible. La notion de place peut être traduite par la disponibilité des ressources requises par l'application, et la notion de liens (à travers les noms externes et internes) peut se traduire par les dépendances qu'il faut satisfaire pour réussir le déploiement.

En partant de la modélisation présentée dans le chapitre précédent, dans la quelle nous établissons une description formelle fondée sur les BRS d'une application *AADL* et celle de son environnement d'exécution (Benlahrache et al. 2011), nous présentons dans ce chapitre, un cadre formel, toujours à base des bigraphes, permettant la modélisation de l'opération de déploiement d'une application dans un environnement cible. Le modèle proposé est un modèle générique, formel et **visuel**. Il se prête facilement à la vérification et la validation.

Nous nous intéressons en premier lieu à l'opération d'installation d'une

entité logicielle sur une plateforme d'exécution que nous projetons ensuite sur la spécification *AADL* du cas *PMS*. Ensuite, nous allons utiliser le même formalisme pour décrire la reconfiguration de ce système (Benlahrache et Belala 2013).

5.2 Déploiement Architectural Bigraphique

Le processus de déploiement est constitué d'un certain nombre de tâches (voir chapitre 2) dont celle de l'installation qui occupe une place primordiale et dont le succès ou l'échec influe directement sur l'exécution ultérieure du système.

La formalisation bigraphique proposée offre une sémantique bien définie à la spécification *AADL* et sa dynamique. L'aspect graphique des bigraphes permet de représenter clairement la structure hiérarchique de toute la spécification *AADL*. En outre, les opérations inhérentes permettent de spécifier formellement des concepts architecturaux *AADL* plus complexes, en particulier d'installation et de reconfiguration des composantes d'un système.

La figure 5.1 montre l'analogie entre le processus de déploiement tel que décrit et montré dans la littérature et celui modélisé par les BRS :

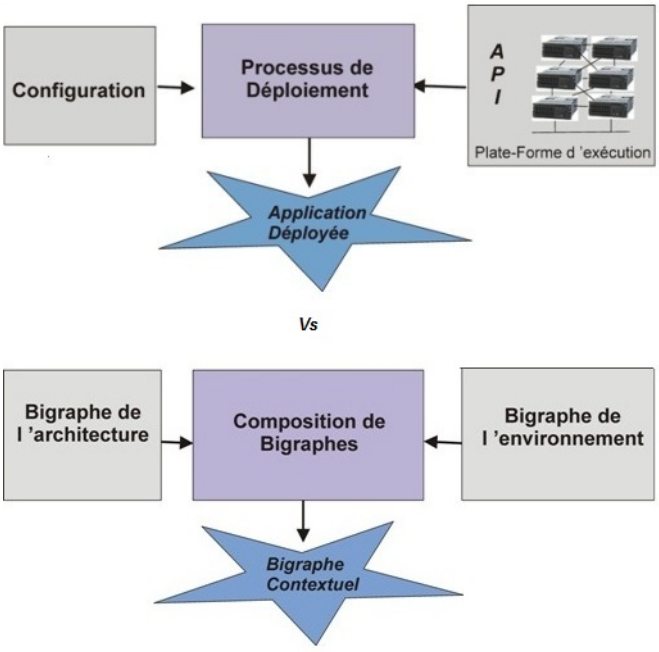

Figure 5.1 – *Processus de Déploiement Vs Bigraphes*

Le déploiement de logiciels est un ensemble d'activités qui font qu'un

système logiciel soit opérationnel. Le déploiement de n'importe quel logiciel implique la copie de ses constituants depuis un site producteur (source) vers un site d'exécution (cible). L'installation, la configuration, l'activation, la reconfiguration (mises à jour) et, finalement la désactivation et la désinstallation sont les principales activités connues du processus de déploiement (Carzaniga et al. 1998).

Dans ce qui suit, nous nous intéressons à ce processus d'un point de vue architectural en donnant une spécification formelle des deux activités : l'installation puis la reconfiguration.

5.3 Formalisation de l'Activité d'Installation

La tâche d'installation est l'activité de déploiement la plus complexe. Elle doit assurer l'assemblage correct de toutes les ressources nécessaires pour l'exécution des applications. Ces ressources peuvent être matérielles ou logicielles. Généralement, c'est à ce niveau que la vérification des dépendances est faite. Une de ces dépendances logicielles, par exemple, peut concerner l'existence d'un fichier, ou simplement une version particulière de celui-ci. Une dépendance matérielle s'intéresse, par exemple, à la taille d'une mémoire, la vitesse d'un processeur ou le délai de propagation dans un support de communication. La sémantique de cette activité d'installation n'est pas encore bien définie. Elle est souvent décrite comme un ensemble de mesures spécifiques dictées par le producteur de logiciel pour installer une application donnée sur une plateforme particulière utilisant une technologie spécifique (Talwar et al. 2005, Dearle 2007). Nous contribuerons dans ce travail de thèse, à définir cette tâche importante de déploiement au niveau architectural en utilisant un cadre basé bigraphes.

5.3.1 Principe de Base

Comme nous l'avons déjà montré, une application est désormais définie par un bigraphe G_S ayant un ensemble de régions. La plateforme d'exécution est également considérée comme un bigraphe G_H contenant une collection de sites capables d'accueillir ces régions. La sémantique de l'opération d'installation est alors définie par l'opération de composition de ces deux bigraphes G_S et G_H.

Afin d'assurer une installation fiable de l'application sur son environnement d'exécution, la vérification de certaines conditions est alors néces-

saire. Cette vérification se dégage naturellement du formalisme proposé, telles que : La correspondance entre l'interface d'entrée de l'environnement $I_H = \langle m_H, X_H \rangle$ et l'interface externe de l'application $J_S = \langle n_S, Y_S \rangle$, c-à -d., la correspondance des noms externes de l'application avec les noms internes de l'environnement et le nombre de régions de l'application avec le nombre de sites de l'environnement. Il faut pouvoir s'assurer qu'il y a suffisamment de sites libres dans G_H pour pouvoir héberger G_S. Formellement, la formalisation de cette opération est réalisée comme suit :

- Soit un bigraphe G_H modélisant la plateforme de déploiement,
- Soit un bigraphe G_S modélisant l'architecture à déployer,
- L'opération d'installation de l'instance G_S dans l'environnement G_H est spécifiée par un bigraphe $G = G_S \circ G_H$ résultant de la composition des bigraphes G_S et G_H.

Formellement, cette installation est définie comme suit :

Définition : *Étant donnés deux bigraphes G_S et G_H associés respectivement à une instance d'architecture et un environnement de déploiement, le bigraphe G obtenu par composition des deux bigraphes est défini par $G = \{V_G, E_G, Ctrl_G, G_G^P, G_G^L\} : I \rightarrow J$ tels que :*

- $V_G = V_S \cup V_H$,
- $E_G = E_S \cup E_H$,
- $Ctrl_G = Ctrl_S \cup Ctrl_H$ *: ensemble des contrôleurs associés aux nœuds de G.*
- $I_G = \langle m_G, X_G \rangle$ *et* $J_G = \langle n_G, Y_G \rangle$*, constituent ses interfaces avec :* $m_G = m_H - n_S$, $X_G = X_H - Y_S$, $n_G = n_H$, *et* $Y_G = Y_H$, *sachant que* m_G *représente le nombre de sites encore disponibles de G et qui sont ceux de* G_H *moins ceux occupés par les régions de* G_S*. Les régions de G sont celles de* G_H.

5.3.2 Cas des Composants *AADL* Composites

L'installation d'une architecture d'un système peut être considérée à plusieurs niveaux selon la complexité de celle-ci en termes d'imbrication des composants. Pour une application distribuée, le premier souci est la détermination du nombre et la distribution spatiale des sites ainsi que la nature des canaux de communication qui les relient. Une fois cette étape surmontée, vient le problème d'installation de chacun des modules sur la plateforme associée. En AADL, l'installation peut être spécifiée pour chaque composant.

Exemple : Pour installer l'application *PMS* introduite dans le chapitre précédent (Section 4.6) sur une plateforme d'exécution donnée, nous devrions

avoir d'abord défini les modèles bigraphiques G_S et G_H des systèmes correspondants et puis établir une correspondance formelle entre eux en termes de concepts bigraphiques. La partie du logiciel de ce système est composée de trois modules distribués dans l'espace (*Service_Event*, *Nurse* et *Controller*) qui vont donc être représentés par trois régions numérotées 0, 1, et 2 (figure 5.2), il leurs faudra trois sites pour réussir une opération d'installation. Dans ce cas précis, chacun des sites représente une plateforme à part.

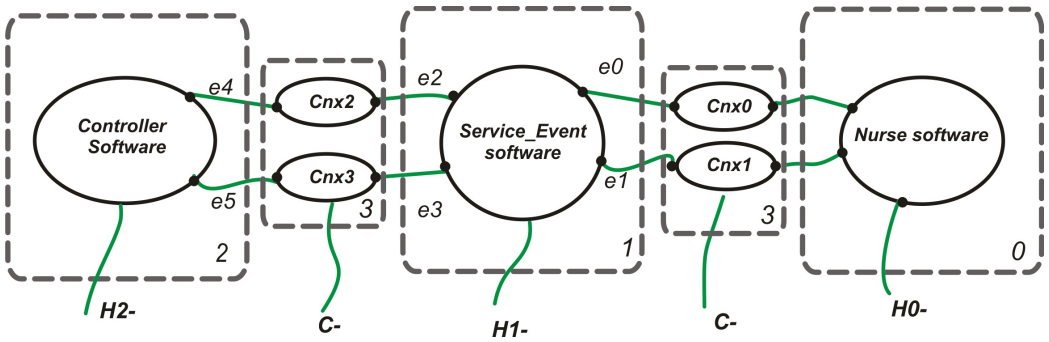

FIGURE 5.2 – *Vue Bigraphique Global du PMS*

Les arcs ouverts avec les noms externes ($H0^-, H1^-$ et $H2^-$) (figure 5.2) indiquent le nombre de sites nécessaires pour l'installation de l'application *PMS*. Les connexions cnx_0 à cnx_3 appartiennent à la même région (3). Les arcs ouverts (C^-) révèlent que toutes les connexions de l'application peuvent partager le même support de communication (communication via un réseau).

Il s'agit ici de l'application générale de la définition 5.3.1, pour des composants non atomiques ou composites d'un logiciel ou un matériel donné exprimé en *AADL*. Pour des spécifications *AADL* plus complexes (ayant plusieurs niveaux d'hiérarchie) et dans le but de formaliser l'opération d'installation des composants logiciels (définis par G_S) sur les composants matériels d'une plateforme d'exécution (définis par G_H), indépendamment de la hiérarchie des composants décrite initialement dans *AADL*, nous devons d'abord aplatir les modèles G_S et G_H pour atteindre librement l'installation de sous-composants, sans tenir compte de leur composants parents (conteneur). Ensuite, nous appliquons une opération de composition judicieuse des bigraphes résultats, guidée par le nombre et les numéros des régions et des sites, ainsi que par les noms internes et externes.

Le mécanisme d'aplatissement n'est en faite qu'une opération de décomposition des bigraphes mais menée dans un sens où l'on fait extraire les nœuds internes (sous composants *AADL*) en dehors du bigraphe conte-

neur. Cette opération doit répondre à certaines exigences. La décomposition d'un bigraphe G dans un contexte C est donnée par la formule suivante : $G = C \circ S \circ d$ où d représente un certain nombre de paramètres.
L'algorithme 5.1 décrit les principales étapes nécessaires pour obtenir des bigraphes aplatis à partir d'une spécification AADL en vue de réaliser l'opération de composition.
Cet algorithme ressemble en grande partie à celui exposé dans la section 2 du chapitre précédent mais il diffère de ce dernier dans la phase de détermination des régions. Un bigraphe aplati doit avoir plus de régions car les sous-composants internes doivent être exposés et non dissimulés dans leur composants conteneurs. Bien sûr, cette mesure concerne uniquement les sous-composants AADL impliqués dans une propriété de type *Actual...Binding*.

5.3.3 Cas des Composants *AADL* Atomiques

L'installation des composants atomiques nécessitent que les bigraphes obtenus soient aplatis. Nous explicitons notre propos à travers la formalisation de l'opération de l'installation appliquée à l'exemple du *PMS* et particulièrement au sous-système *Service_Event*. De ce fait, l'installation formalisée par une opération de composition des deux bigraphes G_S et G_H (voir Section 4.6), nécessite que le bigraphe G_S, et contrairement à G_H, soit aplati afin de faciliter l'installation individuelle des sous-composants définis dans G_S.
Pour cela, nous décomposons le bigraphe G_S en deux bigraphes plus simples, $G_S = C_S o Sc_S$, où Sc_S est le bigraphe des sous-composants (figure 5.3.a) et C_S le bigraphe contextuel (figure 5.3.b).

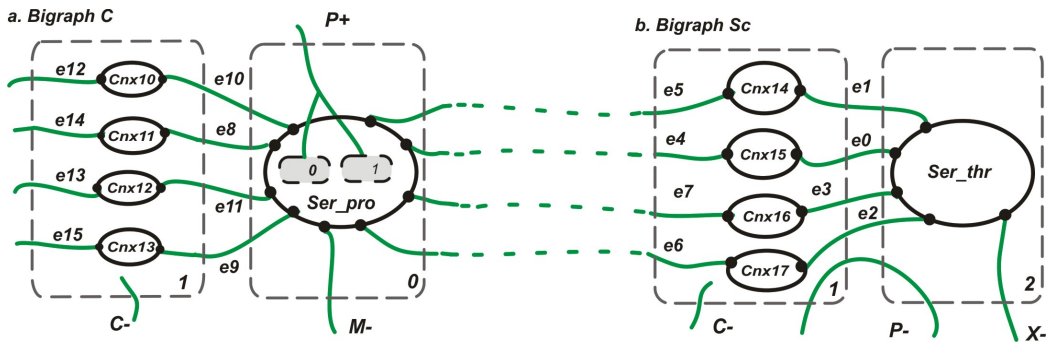

FIGURE 5.3 – *Représentation des bigraphes contextuel et sous-composants du Service_Event*

Les nœuds et les arcs du bigraphe *Sc* sont déduits des clauses *subcomponents* et *connections* de chaque composant conteneur dans la spécification *AADL*. Les interfaces ajoutées (***p- et p+***) préservent le lien de parenté entre le composant conteneur (*Ser_pro*) et ses sous-composants (*Ser_thr*, cnx_{14-17}). Les

Algorithme : transformation d'une spécification AADL en bigraphes *aplatis*.

Objectif : obtenir des bigraphes non composites d'une spécification logicielle afin de faciliter l'opération d'installation.

Entrées : spécification AADL complète et valide.
Sorties : bigraphes G_S et G_H aplatis (régions).

Début

1. **Pour** tout composant C_s de type *system* **faire** créer deux bigraphes G_S et G_H ;
2. **Pour** G_S **faire** créer une région avec un numéro ;
3. **Pour** tout composant-software c_i **faire**
   a. contrôleur(c_i).nom = nom(composant) ;
   b. contrôleur(c_i).port ←nombre d'interfaces ;
   c. **Si** nombre (sous-composants + nombre de connexions) $\neq$ 0
      alors contrôleur(c_i).type ← non-atomique
      sinon contrôleur(c_i).type ← atomique ;
   d. **Si** modes(c_i) =vrai **alors** contrôle(c_i).dyn ← dynamique ;
   e. **Pour** toute connexion *cnx* de c_i de la forme *cnx* : a⇒b
      faire ajouter un arc de a vers b ; (a et b des ports) ;
   f. **Pour** toute propriété *Actual* . . . *Binding* **faire**
      ajouter un nom externe au composant c_i ;
   g. **Pour** tout sous-composant ou connexion ayant un nom externe **faire**
      i. Vérifier **si** une région existante possède le même nom-externe.
         Sinon alors ajouter une nouvelle région
         avec un nouveau numéro ;
      ii. Placer le nœud correspondant au sous-composant ou à la connexion
         dans cette région ;
      iii. Ajouter un nom externe (p) à ce nœud ;
      iv. Ajouter un site au niveau du nœud du composant conteneur
         avec un nom-interne (p).
   h. **Si** contrôleur(c_i).type = non-atomique **alors**
      Pour tout sous-composant de c_i **faire**
         aller à 3.
4. **Pour** créer G_H (ses nœuds sont les sous composants matériels)
   a. Répéter les étapes de 3. (de a. à e.) ;
   b. **Pour** tout composant matériel c_j d'une propriété *Actual* . . . *Binding* **faire**
      ajouter un site au nœud associé
      ajouter un nom-interne=nom-externe du composant c_i
         correspondant de la même propriété ;
   c. **Si** contrôleur(c_j).type = *non-atomique* **alors**
      Pour tout sous-composant de c_i
      faire aller à 4.a.
      // Possible uniquement dans le cas de mémoire imbriquée
      // dans un processeur.

Fin.

TABLE 5.1 – *Algorithme de Génération des Bigraphes Aplatis*

connexions (cnx_{10-13} et cnx_{14-17}) sont regroupées dans les mêmes régions car elles sont concernées par la même propriété de type "*Actual...Binding*" et qui fait référence au même composant matériel "*Ser_bus*".

Étant donné le bigraphe G_S, décomposé en deux bigraphes C_S et Sc_S, sa composition avec le bigraphe G_H pour formaliser l'opération d'installation déclarée informellement dans une instance *AADL* au moyen de la propriété "*Actual...Binding*", sera réalisée en deux étapes. La première étape consiste à composer les bigraphes G_H et C_S : $(G_H o C_S)$. Le résultat obtenu servira comme bigraphe contextuel pour effectuer la deuxième étape qui consiste à le composer avec le bigraphe Sc :

$$G = G_H o G_S = G_H o (C_S o Sc_S) = (G_H o C_S) o Sc_S$$

La composition des bigraphes est une opération associative (définition 2 du chapitre 3).

La figure 5.4 schématise la manière avec laquelle les composants et sous composants logiciels sont déployés sur les composants matériels correspondants.

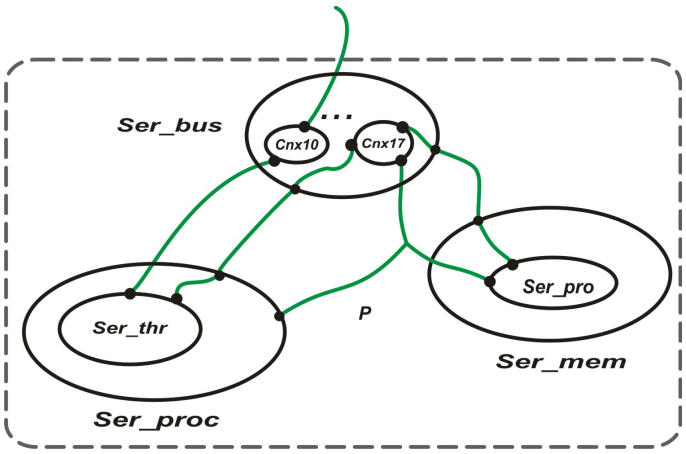

FIGURE 5.4 – *Sémantique de l'Installation par la Composition de Bigraphes*

Par ailleurs, le bigraphe obtenu offre un modèle mathématique permettant le raisonnement sur l'opération d'installation et la possibilité de vérifier sa correction en termes de satisfaction des dépendances entre les différents éléments impliqués. Ce modèle graphique visualise clairement les correspondances entre les composants logiciels et matériels nécessaires au bon fonctionnement d'une application. On peut constater qu'une exécution correcte d'un thread dépend essentiellement du composant processeur et le bon déroulement d'une connexion est lié à la nature du bus et ses caractéristiques.

Le bigraphe résultat (figure 5.4) dans le cas de notre exemple est défini comme suit :

- $G = (V, E, Ctrl, G^P, G^L) : I \rightarrow J$;
- $V = \{Ser\_pro, Ser\_thr, Ser\_proc, Ser\_mem, Ser\_bus, cnx10, cnx11, ..., cnx17\}$;
- $E = \{e10, e11, ...e17, p\}$;
- $I = \langle 0, \emptyset \rangle$ et $J = \langle 1, p \rangle$.

Notons ici que les noms internes et externes disparaissent une fois ils sont satisfaits (lorsqu'un nom externe d'une région correspond à un nom interne du site ayant le même numéro, on dit que la dépendance est satisfaite). Ceci exprime déjà une certaine complétude au niveau des dépendances et permet une certaine garantie de la cohérence de l'opération de l'installation.

L'extraction des graphes de places et de liens du bigraphe résultat (voir figure 5.5) permet de mettre en évidence les relations logicielles/matérielles. Ceci peut s'avérer très utile pour valider les opérations de reconfiguration sachant que celles-ci peuvent concerner les deux types de composants à la fois.

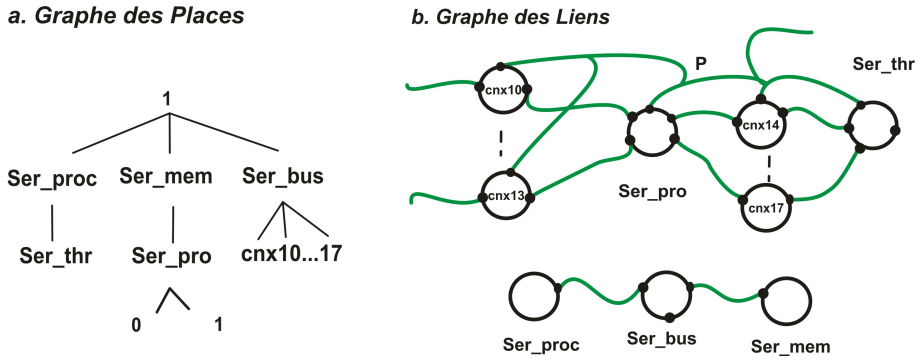

FIGURE 5.5 – *Graphes des places et des liens du bigraphe G.*

Le graphe des places du bigraphe **G** (figure 5.5.a) montre les relations entre les composants logiciels et ceux de type matériel. Cette imbrication implicite des composants devrait guider les choix de reconfiguration. Le graphe de liens (figure 5.5.b) exprime le flux de données entre les composants logiciels et matériels du système.

5.4 Formalisation de l'Activité de Reconfiguration

La reconfiguration logicielle est l'une des activités importantes du processus de déploiement, mais contrairement aux autres activités de ce processus, elle a fortement suscité l'intérêt des chercheurs vue son rôle très important.

Elle permet aux applications installées de réagir aux évènements de l'utilisateur ou de l'environnement et d'effectuer des changements d'une façon autonome pouvant aboutir à une qualité de service meilleure (Allen et al. 1998).

La reconfiguration d'une architecture à base de composants consiste à proposer au niveau architectural des transformations pour préserver ses propriétés malgré certains changements. Cette reconfiguration peut être définie par les opérations suivantes : l'ajout d'un composant, la suppression d'un composant existant ou le raffinement des composants par remplacement et ajout, la suppression ou le raffinement de liens d'interaction.

La dynamique dans une spécification *AADL* est traduite par le concept des "*modes*" et transitions entre "*modes*". Une spécification *AADL* peut se reconfigurer en basculant d'un mode de fonctionnement vers un autre. Dans un mode, on peut raffiner les caractéristiques des composant logiciels/ matériels, donner de nouvelles valeurs aux propriétés, connecter ou déconnecter des interfaces, etc.
La transition entre deux modes peut être déclenchée par un événement qui doit parvenir d'un port de type *in_event* ou *in_out_event*. Le dernier type de port est utilisé pour faire propager l'effet de l'événement vers un composant conteneur (sens de l'intérieur à l'extérieur) ou vers un sous-composant (de l'extérieur vers l'intérieur) afin d'assurer une reconfiguration en cascade si le cas le nécessite.

Dans Chang et al. (2007), les auteurs ont montré la capacité des BRS, en général, pour modéliser la reconfiguration appliquée à des styles architecturaux. Dans ce qui suit, nous allons montrer la capacité des BRS pour modéliser la reconfiguration architecturale définie dans une spécification *AADL*.

Une opération de reconfiguration en *AADL* est définie par un couple de modes dont le premier désigne la configuration initiale et le second désigne la configuration finale. L'arrivée d'un évènement déclencheur permet de passer d'un mode à un autre. D'un point de vue *AADL*, la composition interne d'une configuration changera selon le mode activé.

En *AADL*, nous notons que toute la configuration du système est défini par une paire *(S, M)* exprimant la dynamique d'un système *S* dans un mode *M*. La reconfiguration est spécifiée alors par des transitions entre les configu-

rations grâce à des changements de mode. Pour parvenir à sa formalisation, nous exploitons le comportement dynamique des bigraphes. Les règles de réaction exprimant divers changements entre bigraphes peuvent être utilisées dans ce contexte.

Définition *Soit une transition de modes :* $[(C, M) \xrightarrow{event} (C, M')]$ *associée à une spécification AADL d'un composant C, sa sémantique formelle est définie par une règle de réaction composée de deux bigraphes (Redex, Reactum), noté r :* (R, R') *où :*

- *R est un modèle bigraphique de C en mode M ;*
- *R′ est un modèle bigraphique de C dans un nouveau mode M′.*

5.4.1 Reconfiguration Logicielle

Nous reprenons dans cet exemple la spécification *AADL* du système PMS, et on considère que le sous-composant *Nurse* peut avoir un comportement particulier lorsque l'infirmière *Nurse_Sys* système passe du mode filaire au mode sans fil (figure 5.6). Nous obtenons la spécification *AADL* donnée (voir figure 5.7) qui exprime deux états possibles de fonctionnement du système global.

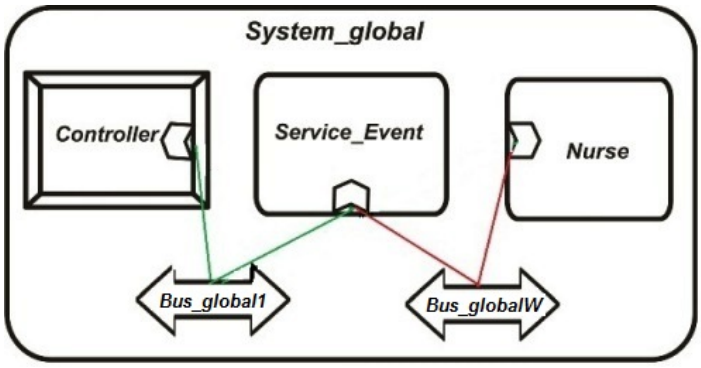

FIGURE 5.6 – *Architecture du PMS en mode sans Fil*

Ces états sont définis par l'introduction de deux modes *AADL* : "*Wifi_on*" et "*Wifi_off*" ainsi que leurs transitions correspondantes.
Comme le montre la figure 5.7, chaque transition de mode est déclenchée par un événement local "*Nurse.Wifi_event_out*". Ceci exprime une auto-reconfiguration du système.
Par conséquent, cet événement permet de faire passer l'ensemble du système en mode sans fil. Les interactions entre le composant *Service_Event* et le composant *Nurse* sont les plus concernées par ces changements. Elles permettent de connecter le nouveau composant *Ser_thr1* à l'ensemble du système.
Ces nouvelles connexions peuvent utiliser un nouveau support de communication spécifique *Bus_globalW* (voir figure 5.6). Nous rappelons ici que l'effet

de l'événement déclenché par le système *Nurse* se propage via des connexions (étiquetées *cnxW*) pour l'instance du processus *Ser_pro* via le port *Wifi_event*.

La figure 5.8, par exemple, montre la spécification *AADL* de l'implémentation du processus de type *Service_process* avec la déclaration des modes. Nous notons que'en plus des connexions spécifiées dans le mode "*wifi_off*", d'autres connexions ont été spécifiées pour le mode "*wifi_on*".

Ce composant particulier *Service_process* peut avoir deux états d'observables (modes). Ainsi, deux bigraphes distincts sont définis : R et R'(voir figure 5.9). La règle de réaction ayant R comme bigraphe *Redex* et R' comme bigraphe *Reactum*, spécifie formellement la transition entre deux modes du composant *Service_process* à partir de l'état filaire vers l'état sans fil :
$[(Ser\_pro, wifi\_off) \quad \xrightarrow{wifi\_event} \quad (Ser\_pro, wifi\_on)]$.

La règle de réaction est notée comme suit :

$$r= (R, R') = (R : 0 \to \langle 1, \emptyset \rangle, R' : 0 \to \langle 1, \emptyset \rangle, \eta : 0 \to 0).$$

La figure 5.8 représente une transformation que peut subir le composant *Service_process*. Ce dernier, pour basculer du mode *Wifi_Off* au mode *Wifi_On*, doit exécuter des threads différents (*Ser_thr*, *Ser_thr1*). Le passage d'un mode à un autre est rythmé par l'arrivée d'un événement *Wifi_Event*.

5.4.2 Reconfiguration Matérielle

De façon similaire, la modélisation d'une reconfiguration matérielle en AADL peut être définie par des modes définis pour les composants matériels de la spécification. La reconfiguration matérielle peut être traduite par le remplacement d'un processeur, mémoire, bus ou un périphérique par un autre. Elle est préconisée pour faire face aux pannes matérielles et assurer une qualité de service. D'un point de vue bigraphique, ces transformations matérielles sont aussi représentées par des règles de réaction.

Un exemple de reconfiguration matérielle est le remplacement du canal filaire par deux canaux, l'un filaire et l'autre sans fil. La figure 5.10 montre le passage du système *service_Event* d'un mode vers un autre. Au départ, le système utilisait un canal de communication filaire (Ser_Bus), mais pour des raisons d'adaptation, le système va utiliser désormais deux canaux de communication. Le premier pour assurer les communications entre le périphé-

```
system global
end global;

system implementation global.impl
subcomponents
    ......................
  Bus_global: bus bus_gl.impl;
  Bus_globalW:bus  bus_gl.implw;
connections
    ......................
  cnx0w: event port Service_Event.out_alarmw -> Nurse.in_alarmw;
  cnx1w: event port Nurse.out_answerw -> Service_Event.in_answerw;
  cnxw: event port Nurse.wifi_event_out -> Service_Event.wifi_event;
modes
   wifi_off: initial mode;
   wifi_on: mode;
   wifi_off-[Nurse.wifi_event_out]->wifi_on;
   wifi_on-[Nurse.wifi_event_out]->wifi_off;
properties
  -- wire mode
   Actual_Connection_Binding =>  reference Bus_global applies to cnx0
   in modes (wifi_off);
   Actual_Connection_Binding =>  reference Bus_global applies to cnx1
   in modes (wifi_off);
   Actual_Connection_Binding =>  reference Bus_global applies to cnx2
   in modes (wifi_off);
    ..... – Toutes les autres connexions en mode 'Wifi_off'
   --wifi mode
   Actual_Connection_Binding =>  reference Bus_globalW applies to
   cnx0w in modes (wifi_on);
   Actual_Connection_Binding =>  reference Bus_globalW applies to
   cnx1w in modes (wifi_on);
   Actual_Connection_Binding =>  reference Bus_globalW applies to
   cnxw in modes (wifi_on);
   Actual_Connection_Binding =>  reference Bus_globalW applies to
   Service_Event.Ser_pro in modes (wifi_on);
   Actual_Connection_Binding =>  reference Bus_globalW applies to
   Nurse.Nur_pro in modes (wifi_on);
end global.impl;
```

FIGURE 5.7 – *Spécification AADL de la reconfiguration du système global*

```
process implementation Service_process.impl
  subcomponents
    Ser_thr: thread Service_thread.impl in modes (wifi_off);
    Ser_thr1: thread Service_thread.impl1 in modes (wifi_on);
  connections
    cnx14: event port Ser_thr.out_alarm -> out_alarm in modes (wifi_off);
    ...................
    cnxw: event port wifi_event -> Ser_thr.wifi_event in modes (wifi_off);
    cnxw1: event port wifi_event -> Ser_thr1.wifi_event in modes (wifi_on);
    cnx14w: event port Ser_thr1.out_alarm -> out_alarm in modes (wifi_on);
    cnx15w: event port in_answer -> Ser_thr1.in_answer in modes (wifi_on);
    cnx16w: event data port in_pressure -> Ser_thr1.in_pressure
    in modes (wifi_on);
    cnx17w: event data port in_temp -> Ser_thr1.in_temp in modes (wifi_on);
  modes
    wifi_off: initial mode ;
    wifi_on: mode ;
    wifi_off -[ wifi_event ]-> wifi_on;
    wifi_on -[ wifi_event ]-> wifi_off;
end Service_process.impl;
```

FIGURE 5.8 – *Spécification AADL de la Reconfiguration du "Service_process"*

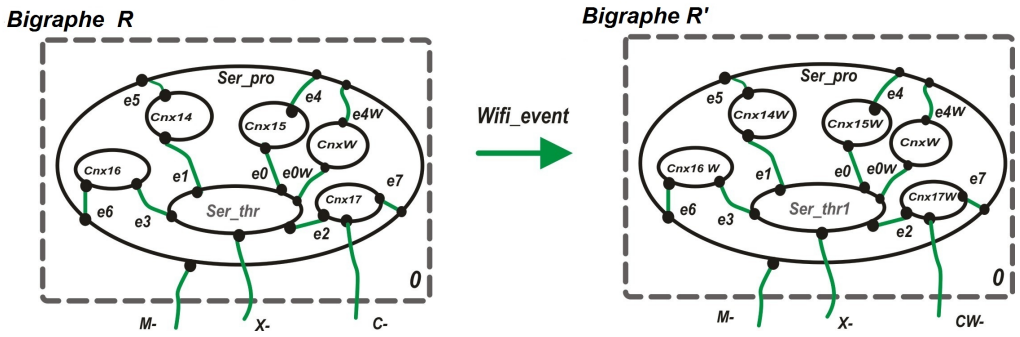

FIGURE 5.9 – *Règle de réaction décrivant la reconfiguration de "Ser_pro"*

rique *Controller* et le *Service_Event*, il s'agit du bus filaire initial. Par contre, pour garder le contact avec l'infirmière à travers le sous-système *Nurse*, le sous-système *Service_Event* utilise un deuxième canal de communication sans fil (Ser_BusW). La figure 5.10 représente la règle de réaction censée décrire cette reconfiguration.

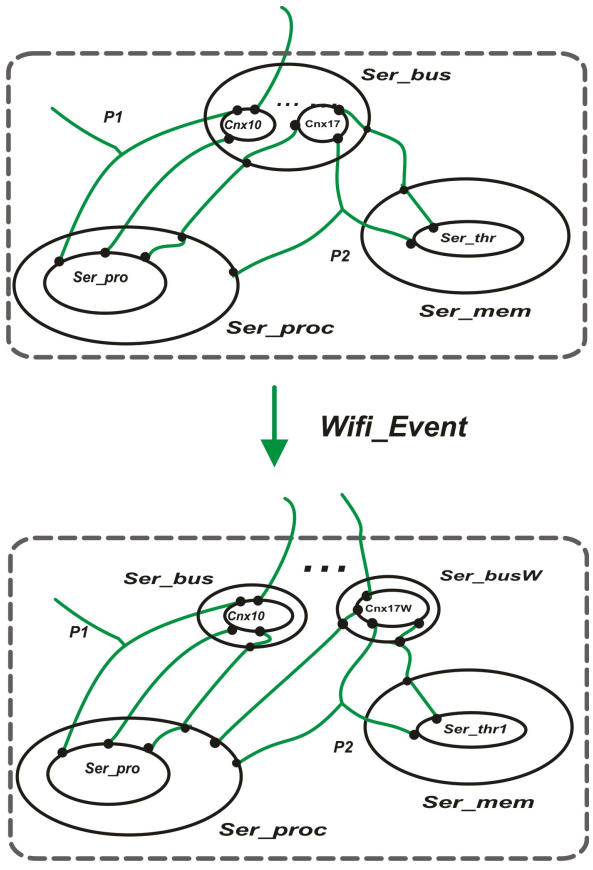

FIGURE 5.10 – *Exemple de reconfiguration Matérielle.*

5.5 Conclusion

Dans ce chapitre, nous avons proposé un modèle mathématique, à base de bigraphes, permettant la modélisation de l'opération de déploiement d'une application sur un environnement cible.
Cet environnement est souvent représenté par un ensemble de composants matériels (processeurs, mémoires, bus de communication et divers dispositifs) et logiciels (Systèmes d'exploitation, Intergiciel...) nécessaires pour le déploiement et le lancement de cette application logicielle. Le déploiement doit être cohérent afin d'assurer d'une part, le bon fonctionnement de l'application et d'autre part, l'intégrité d'un tel environnement.

Ainsi, nous avons défini formellement l'activité d'installation par une opération de composition $G = G_H \circ G_S$ des deux bigraphes G_H et G_S.
L'opération de reconfiguration logicielle ou matérielle fut exprimée par une opération de transformation de bigraphe en utilisant les règles de réaction associées.

6 Vers une Plateforme basée Maude pour la Manipulation des Bigraphes

Sommaire

Il y a deux façons de faire la conception d'un logiciel. Une façon est de le rendre si simple qu'il n'y a, selon toute apparence, aucun défaut. Et l'autre est de le faire si compliqué qu'il n'y a pas de défaut apparent.

Tony Hoare.

6.1 Introduction

Les bigraphes représentent un moyen de modélisation pour divers types de systèmes. Afin de profiter des modèles bigraphiques obtenus à partir des spécifications architecturales décrivant des applications logicielles spécifiées en AADL, il est judicieux de pouvoir les visualiser, exploiter et vérifier.

Les outils récemment développés pour les bigraphes, dont l'outil *BigMC*, permettent l'édition, la manipulation et la vérification des modèles bigraphiques. Mais ils ont été conçus et réalisés dans un spectre étroit et précis, ce qui ne permet pas leur réutilisation dans d'autres domaines ni leur intégration dans d'autres environnements.

À cet effet, nous proposons une plateforme dédiée aux bigraphes définissant les architectures logicielles et matérielles. L'objectif global est d'avoir une plateforme capable de recevoir en entrée l'architecture d'un système, ensuite une transformation vers les bigraphes est réalisée afin de mettre en évidence les deux structures d'un système, logicielle et matérielle. Sur ces deux structures bigraphiques, une opération de composition est effectuée simulant l'activité d'installation du processus de déploiement. La reconfiguration est représentée par des opérations de transformation des bigraphes.
Par ailleurs, le bigraphe est un modèle graphique, ceci nécessite la réalisation d'un éditeur graphique pour pouvoir le dessiner ou le visualiser.

Dans ce chapitre, nous présentons une plateforme dédiée aux modèles bigraphiques. Nous commençons par la présentation de l'architecture de cette plateforme. Ensuite, nous exposons ses différents modules développés. Comme conclusion, nous présentons le résultat de l'intégration des ces modules dans un seul outil.

6.2 Environnement pour les Spécifications AADL Bigraphiques

La figure 6.1 montre les fonctionnalités de la plateforme que nous proposons dans ce chapitre ainsi que son mécanisme de fonctionnement. Elle reçoit en entrée une architecture *AADL*, élabore son modèle bigraphique puis le soumet à des opérations de composition et de transformation. Évidement, dans cette plateforme nous pouvons dessiner des bigraphes, enregistrer, récupérer et modifier des bigraphes existants. Nous pouvons également, vérifier

et valider ces bigraphes. On y trouve les principaux modules.

- Le module *BigraphEditor* est une interface graphique, offrant des outils visuels pour dessiner un bigraphe, obtenu à partir d'une spécification donnée.
- Le module *AADL2Bigraph* permet de générer une structure bigraphique à partir d'une spécification architecturale décrite en *AADL*. Cette description doit tout d'abord être validée par le système *Osate*. Le module *AADL2Bigraph* exploite ce fichier afin de générer graphiquement un bigraphe tout en indiquant les bigraphes des parties logicielle et matérielle du système.
- Le module *BigraphOperations* permet, soit d'exploiter un bigraphe généré par le module *AADL2Bigraph*, soit de créer un nouveau bigraphe, en lui ajoutant des règles de réaction. *BigraphOperations* effectue des opérations sur des bigraphes telles que : la composition entre deux bigraphes ou la transformation d'un bigraphe selon une règle de réaction donnée.
- Le quatrième module de cette plateforme est censé exploiter les bigraphes obtenus par les modules précédents, puis vérifier la validité des bigraphes et les opérations de certaines propriétés du système inhérentes au processus de déploiement.

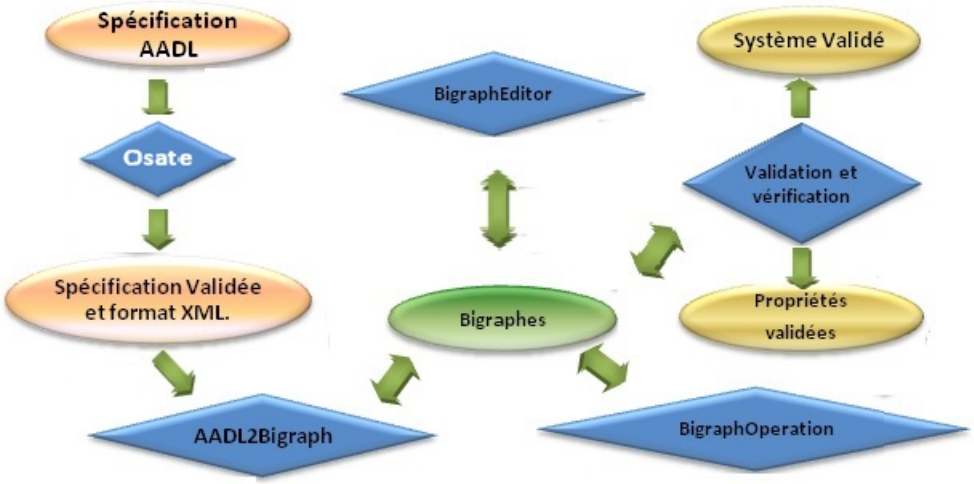

FIGURE 6.1 – *Architecture de la plateforme.*

Dans la section suivante nous présentons, de façon incrémentale, les modules de cette plateforme tout en mettant l'accent sur leurs fonctionnalités.

Pour des raisons de compatibilité et d'intégration, nous avons choisi de développer des modules indépendants sous forme de plugins *Eclipse*, construits

selon les modèles du Framework *GMF*. Sachant que les outils de développement *GMF* permettent de générer aisément des modèles graphiques que nous adaptons pour les bigraphes.

6.2.1 Présentation du Framework GMF

GMF (Graphical Modeling Framework) est un framework *Eclipse* (Moore et al. 2004), dédié à la création des éditeurs graphiques sous forme des plugins *Eclipse*. Il se base sur deux frameworks *EMF* (Eclipse Modeling Framework) (Budinsky et al. 2009) et *GEF* (Graphical Editing Framework) (voir figure 6.2).

Pour développer un outil avec GMF, trois grandes phases doivent être suivies : *modélisation*, *génération du code* et enfin l'*édition graphique* du modèle.

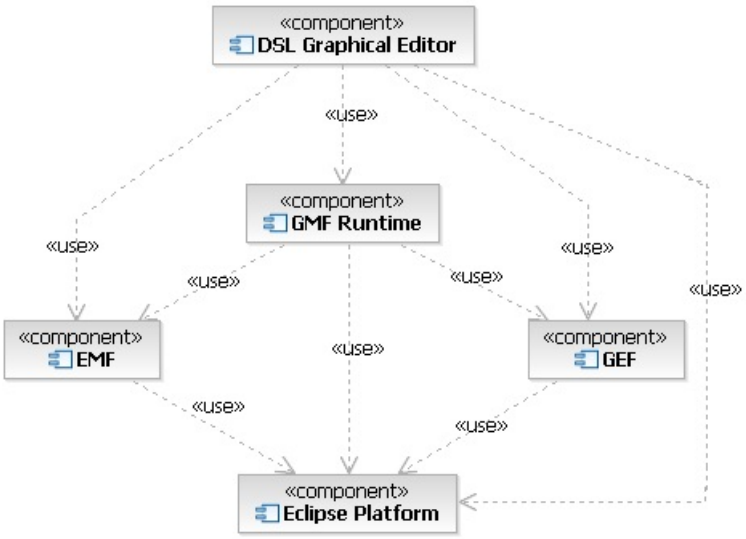

FIGURE 6.2 – *Architecture de l'Environnement GMF (www.eclipse.org).*

Modélisation

EMF est un framework de modélisation, il permet de créer un méta-modèle d'une application et générer son code correspondant. *EMF* permet de créer deux types de modèles, d'une part des modèles qui définissent des concepts souvent nommés méta-modèles, et d'autre part des modèles instanciant ces concepts. Un modèle EMF est une description des différentes entités du modèle ainsi que les relations qui peuvent les relier, il est appelé méta-modèle *Ecore* (*.ecore). Ce méta-modèle est représenté comme un ensemble de diagrammes de classes *UML* avec des propriétés additionnelles spécifiques à *EMF* qui supportent la génération automatique de méthodes pour la mani-

pulation des modèles *Objet* et pour le stockage persistant et la récupération d'un tel modèle dans un ou plusieurs documents *XML*.

Génération du Code

La génération de code est le processus de transformation qui convertit le méta-modèle (*.ecore*) en un code java. Le code *java* généré est une implémentation *élémentaire* (modèle) de l'application, représentant les interfaces et les classes *java* des éléments du modèle. La personnalisation du modèle se fait par la modification et l'adaptation du code Java généré. Cette phase est également assurée par le framework *EMF*.

Génération des Graphiques

GEF offre aux utilisateurs de la plateforme *Eclipse* tous les outils d'édition graphique nécessaires pour construire un éditeur graphique riche. *GEF* se sert des données et des modèles construits lors de la phase de modélisation pour créer un modèle graphique en se basant sur un processus de correspondance entre entité du méta-modèle *.ecore* et un graphique choisi par le développeur. Le framework offre la possibilité de créer des palettes de composants visuels que le développeur peut s'en servir pour dessiner un graphique en utilisant la technique de "drag-and-drop" (*glisser-déposer*).

6.2.2 BigraphEditor : Editeur graphique pour les bigraphes

L'outil *BigraphEditor* (Cherfia et al. 2011), que nous avons développé, permet d'éditer graphiquement des bigraphes. Il permet de créer un bigraphe à partir, soit d'une palette de composants visuels, soit de sa description en format XML.

Pour développer le plugin *BigraphEditor* et afin de définir, spécifiquement les éléments de la palette, un modèle des éléments du bigraphe est défini. Ce modèle est le méta-modèle (figure 6.3) décrivant la structure des bigraphes, c'est-à-dire, une représentation *Objet* des éléments d'un bigraphe. Ce modèle contient huit classes décrivant chacune un des éléments du bigraphe tels que : *Bigraph*, *Region*, *Site*, *Node*, *OpenedEdge*...etc. La présence de classe *LogicalSpace* est due aux exigences de l'implémentation. Les associations entre les classes décrivent les différentes relations qui relient ces éléments entre eux. L'association de type composition *ContainTwoBigraph* lie une règle de réaction (*ReactionRules*) avec la classe *Bigraph* dont la cardinalité est deux, car

une règle de réaction est l'ensemble de deux bigraphes *Reactum* et *Redex*.

Ce méta-modèle bigraphique est développé en utilisant la notation *Ecore* de l'environnement de modélisation EMF d'*Eclipse*. *GEF* s'en sert afin de générer les graphiques correspondants. Ce méta-modèle *Ecore* est utilisé comme un modèle de domaine pour le plugin *BigraphEditor* et éventuellement pour d'autres plugins.

La définition d'un méta modèle pour les bigraphes leur offre la capacité de réutilisation et de généricité. D'autre part, toute extension des bigraphes peut être faite aisément à partir de ce méta modèle dont les raffinements vont offrir des modèles distincts décrivant des systèmes différents.

La palette d'éléments graphiques de *BigraphEditor* présente les outils disponibles dans l'éditeur graphique. L'outil, le plus important, est celui de création, il permet d'instancier un élément du méta-modèle, qu'il soit représenté par un nœud ou un lien (une connexion).

Sur la figure 6.4, nous présentons une capture d'écran de l'éditeur *BigraphEditor*. Nous constatons que les éléments de la palette des composants visuels qui servent pour représenter les bigraphes sont : *Noeud*, *Site*, *Region*, *Hyperarc*, *Nom Interne*, *Nom Externe*. Les liens entre ces nœuds, sont représentés par : *Nom Interne/Hyperarc, Nom Externe/Hyperarc, Port/Hyperarc*. Nous retrouvons sur cette image, comme exemple, l'édition des bigraphes G_S et G_H de l'exemple *Émetteur-Receveur*, traité dans le chapitre 2.

6.2.3 AADL2Bigraph : Transformation d'une architecture AADL en Bigraphes

Afin d'automatiser la génération des bigraphes à partir des spécifications AADL, nous développons un outil qui implémente l'algorithme 1 présenté dans le chapitre 4.

Dans cette section, nous présentons l'outil *AADL2Bigraph* (Cherfia et al. 2011) (Fig.6.5). Cet outil est un plugin *Eclipse*, il utilise le résultat fourni par *Osate* pour générer un bigraphe et il se sert de l'outil *BigraphEditor* pour l'éditer.

La figure 6.5 présente le processus de fonctionnement de ce plugin. La première étape consiste à utiliser le plugin *Osate* qui admet en entrée une spécification AADL format texte (**.aadl*) et après la correction des erreurs

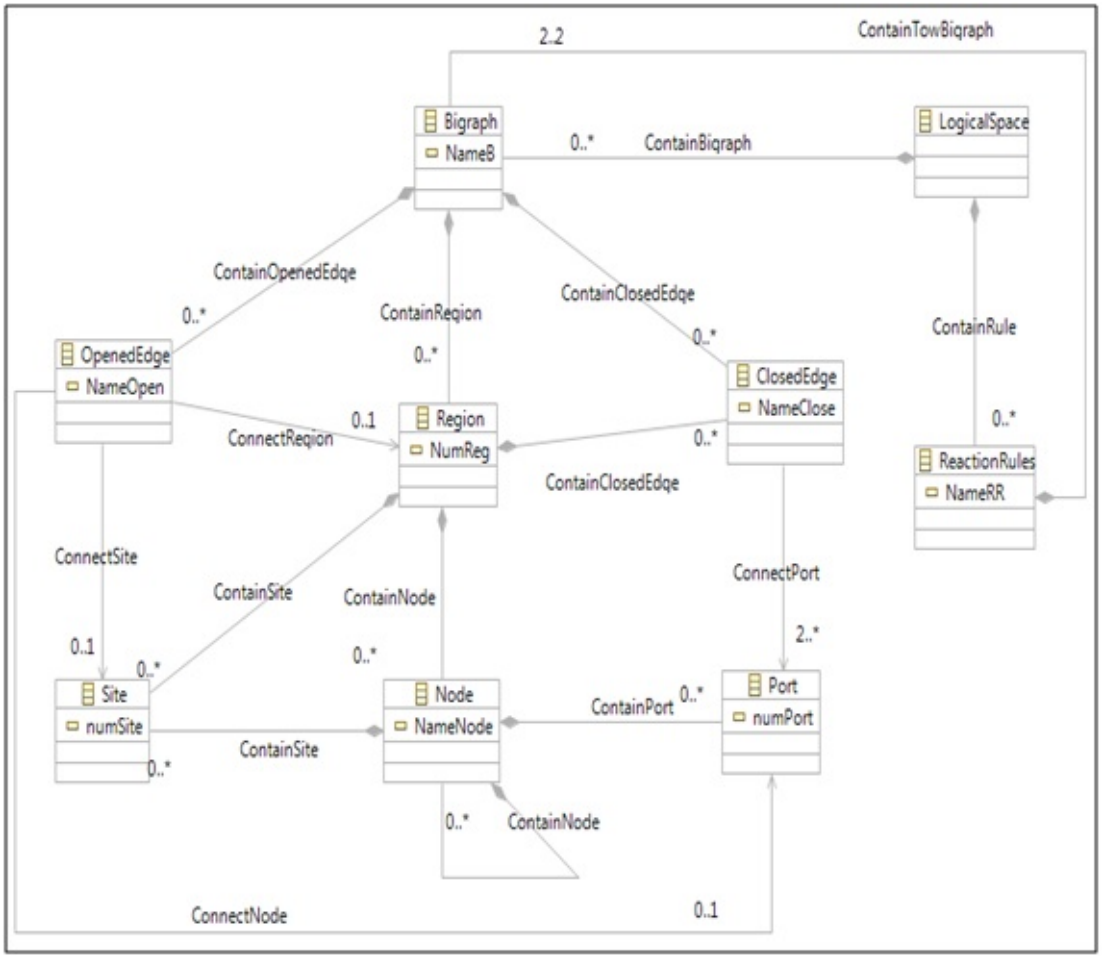

FIGURE 6.3 – *Méta-modèle Ecore d'un Bigraphe.*

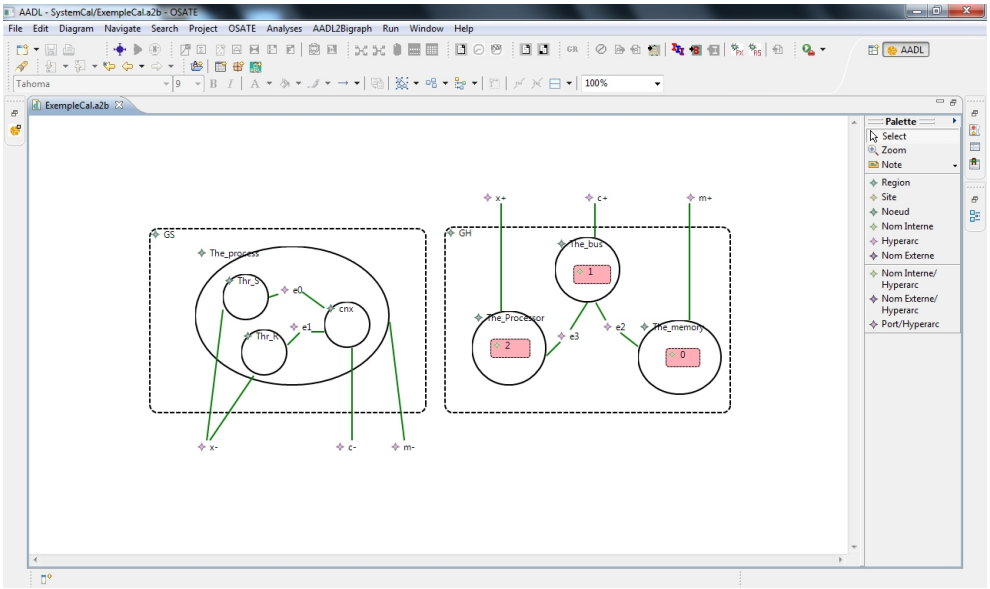

FIGURE 6.4 – *Interface de BigraphEditor.*

syntaxiques et validation de l'architecture, il génère automatiquement une image de cette spécification en format XML (*.*aaxl*) (étapes 1 et 2 de fig.6.5). Le rôle du plugin *AADL2Bigraph* est alors d'exploiter ce fichier fourni pour faire un parsing (étape 3 fig.6.5) et extraire les éléments nécessaires afin de générer les bases de construction des deux bigraphes (logiciel et matériel) et leurs dépendances. Le résultat fourni par *AADL2Bigraph* est enregistré dans un fichier d'extension (*.*a2b*) pour *AADL vers Bigraphe*. L'étape suivante dans ce processus nécessite la transformation du contenu d'un fichier de type *.*a2b* vers un modèle graphique et l'éditer via le plugin *BigraphEditor* (étape 4 de fig.6.5).

Cet outil permet de générer les éléments d'un bigraphe, tout en favorisant la séparation entre les deux structures d'une spécification AADL : logicielle et matérielle, chacune sous forme d'un bigraphe. Le fichier résultat peut être exploité pour transformer et manipuler les bigraphes obtenus.

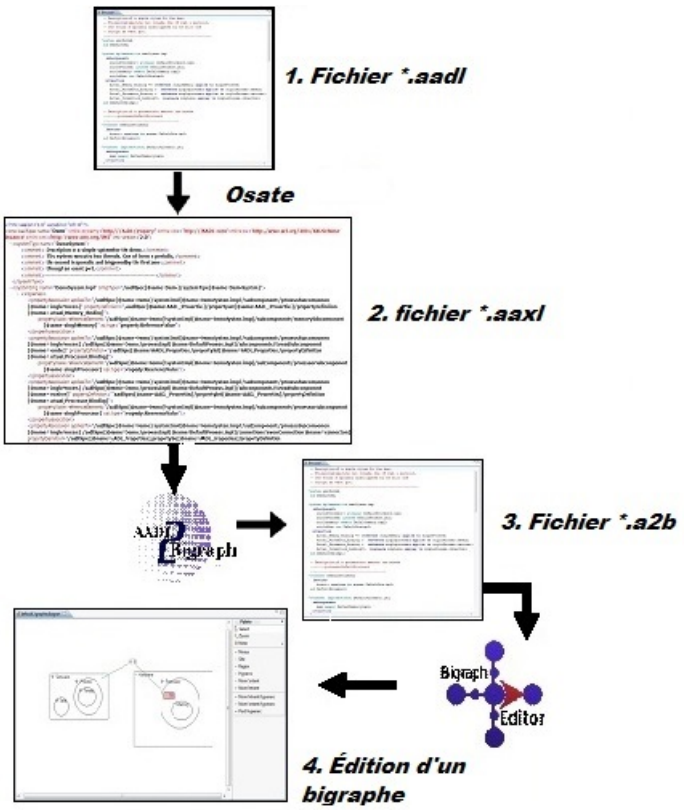

FIGURE 6.5 – *Transformation d'une spécification AADL en Bigraphe.*

6.2.4 BigraphOperations : Opérations sur les Bigraphes

Un plugin *BigraphOperations* (Boucebsi et al. 2012) est développé afin de pouvoir manipuler les modèles bigraphiques obtenus suite à des transformations des spécifications AADL par le plugin *AADL2Bigraph*. Ce plugin est

doté d'un éditeur graphique pour la création et l'édition des bigraphes ainsi que pour la spécification des *règles de réaction* associées. Il dispose également d'une palette de composants visuels adaptées à ses fonctionnalités (figure 6.6). Il permet d'effectuer des opérations sur les bigraphes telles que la composition, la transformation et le produit tensoriel.

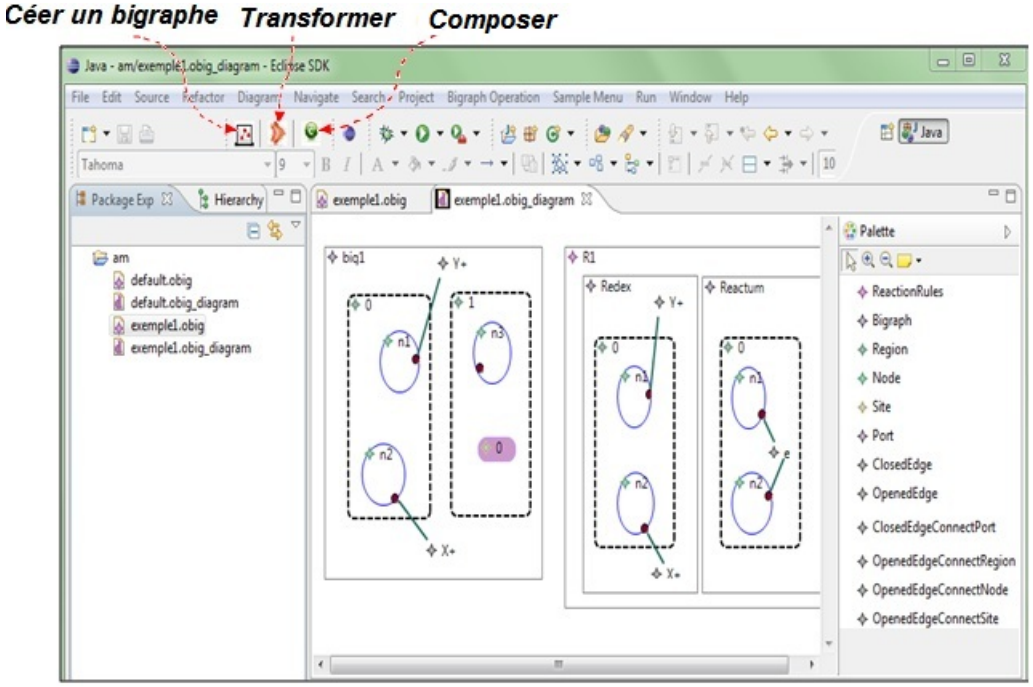

FIGURE 6.6 – *Interface de BigraphOperations*

À la différence du premier plugin *BigraphEditor*, présenté dans le paragraphe 1.2.2, ce plugin assure la définition et l'édition des *règles de réactions* associées à ce bigraphe (figure 6.6). Le principe de fonctionnement de ce plugin est résumé dans la figure 6.7.

Le plugin *BigraphOperations* permet d'appliquer des règles de réactions définies pour un bigraphe pour le transformer. Le nouveau bigraphe obtenu hérite ces règles de réactions et il peut subir, à son tour, d'autres transformations. Pour les besoins de la phase de modélisation de ce plugin, nous reprenons le méta-modèle défini dans la section 1.2.2 (figure 6.3).

L'intérêt, que présente cet outil vis à vis les autres plugins, est qu'il nous permet de voir l'effet de l'application d'une transformation sur un bigraphe et essentiellement la composition de deux bigraphes qui représente pour notre modèle une opération d'installation.

Ces trois plugins présentés dans cette section, participent à la la création, génération automatique, édition graphique et transformation des bigraphes représentant des architectures logicielles.

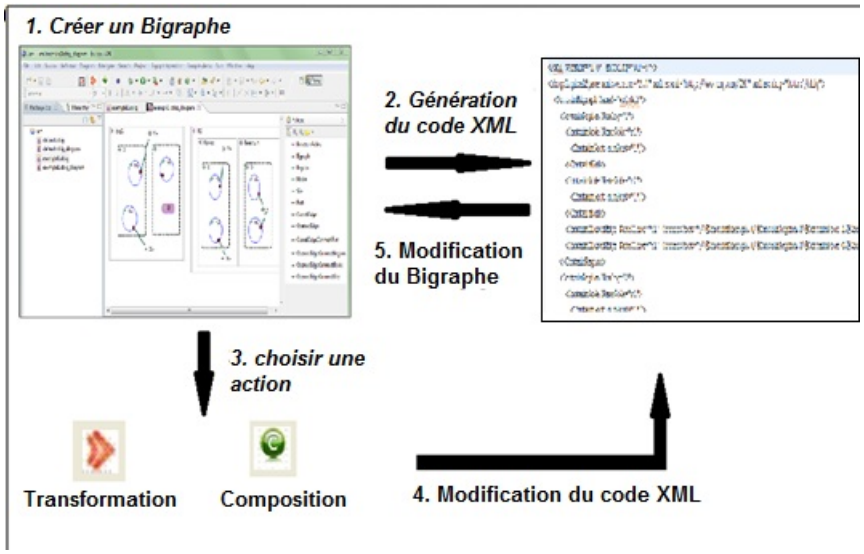

FIGURE 6.7 – *Principes de Fonctionnement de BigraphOperations*

6.3 Vérification et Validation des Modèles Bigraphiques

La modélisation sert à présenter d'une façon simplifier les aspects complexes d'un système. Les modèles obtenus offrent plusieurs caractéristiques telles que la présentation abstraite d'un système (une meilleure visibilité ou lisibilité), la vérification et la validation de ses propriétés sachant que toute propriété prouvée sur un modèle abstrait est conservée sur le modèle concret. Les modèles bigraphiques des architectures logicielles correspondants à une spécification AADL doivent satisfaire certaines propriétés afin de s'assurer de leur correction telles que la correction de l'installation et la cohérence de la reconfiguration. Ces propriétés sont considérées importantes et pertinentes pour le processus de déploiement.

À cet effet, nous nous sommes servis d'outils et de méthodes pour valider particulièrement ces opérations du processus de déploiement architectural d'une spécification AADL.

Parmi les méthodes de vérification et de validation, on y trouve le model-checking (Clarke et al. 2001). C'est une technique de vérification des propriétés sur des systèmes réactifs représentants des systèmes concurrents à états finis.
Cette technique de vérification a un certain nombre d'avantages par rapport aux approches traditionnelles qui sont basées sur la simulation, le test et le raisonnement déductif.

Schématiquement, un algorithme de model-checking prend en entrée une

abstraction ou un modèle du comportement d'un système réactif et une spécification d'un ensemble de propriétés que doit vérifier le système considéré. La vérification du modèle est automatique et généralement assez rapide. Comme réponse et si la conception contient une erreur, le model-checking va produire un contre-exemple qui peut être utilisé pour identifier la source de l'erreur.
La vérification via un model-checking d'un modèle bigraphique nécessite la définition d'un modèle et un ensemble de propriétés à vérifier.

Dans le contexte de cette thèse, qui est la modélisation de certaines tâches du processus de déploiement d'une architecture logicielle, nous avons jugé utile de vérifier deux propriétés : la correction de l'opération d'installation et la cohérence de la reconfiguration.

La première propriété exprime la satisfaction des dépendances logicielles-matérielles lors de l'installation. La propriété est satisfaite si et seulement si toutes les entités logicielles de l'architecture trouvent les composants matériels de la plateforme matérielle nécessaire à leur exécution. La deuxième propriété sert à la vérification de la cohérence d'une reconfiguration. Cette propriété exprime que le système peut évoluer d'un état vers un autre tout en restant cohérent. Ainsi, pour vérifier via un model-checking un modèle bigraphique, nous résumons les éléments à considérer :

1. le modèle est un *BRS* (bigraphe et un ensemble de règles de réaction) décrivant le système et son évolution,
2. les propriétés à vérifier sur le système modélisé par un BRS. Dans ce travail, nous nous intéressons à deux types de propriétés jugées importantes et inhérentes au processus de déploiement que nous avons modélisé : la correction de l'installation et la cohérence de la reconfiguration. Elles sont désignées respectivement par :
   - Propriété **Well-instal** : est la *vivacité* d'un système, exprimant le fait que : "quelque chose de bien arrivera finalement" au système. Dans notre cas, c'est une *installation correcte* de tous les composants logiciels sur les composants matériels adéquats.
   - Propriété **Well-reconf** : est l'*accessibilité* exprimant que : "le système finira par arriver à un état voulu à partir d'un état donné".

Initialement, la première idée qui peut surgir dans ce contexte est d'utiliser, pour valider nos modèles bigraphiques, les outils existants autour des bigraphes. Nous optons pour l'outil *BigMC* qui est le plus utilisé pour vérifier ce type de propriétés.

6.3.1 Model-Checker BigMC

L'outil *BigMC* (Perrone et al. 2012), un model-checker dédié aux bigraphes, semble être le seul outil qui permet de vérifier les propriétés d'un bigraphe tel que défini par ses fondateurs.
Le principe de cet outil est de définir un modèle bigraphique en utilisant sa *propre* grammaire ainsi qu'une suite de règles de réaction associées à ce modèle. Pour vérifier une propriété, *BigMC* applique sur le modèle les règles de réaction définies jusqu'à ce qu'il rencontre, dans le meilleur cas, un état vérifiant la propriété ou il retourne un contre-exemple permettant ainsi d'identifier un cas d'exécution non conforme.

Dans ce qui suit, nous exposons les principales étapes de cette vérification à travers un exemple d'illustration (Benlahrache et al. 2012).

Définition du Modèle

Dans le modèle décrit par la table 6.1, nous pouvons constater l'application des notations utilisées par *BigMC* pour le bigraphe du modèle *Émetteur-Receveur* décrit dans le chapitre 2 et utilisé dans les chapitres 3, 4 et 5.
Le modèle est déclaré comme un ensemble de nœuds, où chacun est défini par son identificateur et son arité (nombre de ports), un ensemble de liens et noms. Nous remarquons que dans cette déclaration, il n'y a pas de distinction entre liens fermés ou ouverts et noms internes ou externes.

L'aspect dynamique associé au modèle dans ce cas est défini par la clause "# *Réactions Rules*". En respectant la grammaire *BigMC*, nous devons prévoir toutes les règles de réactions qui peuvent être utiles dans la description du comportement du système.

Vérification de la Correction de l'Installation

La spécification de la propriété **Well-instal** se fait par le mot clé *property* (voir table 6.1). Elle exprime l'installation correcte des entités logicielles (tels que les nœuds thr_r, cnx...) sur les entités matérielles correspondantes (the_processor, the_bus...).

La table 6.2 montre le résultat de la vérification de la propriété de *Well-instal*. Nous constatons que le *matching* est atteint (*Complete !*) et la propriété est vérifiée après un nombre de transformations égal à *10*. Donc, nous confir-

# Déclarations des Nœuds	# Déclarations des Liens et noms
% active the_processor : 2 ;	% name x ;
% passive thr_S : 3 ;	% name m ;
% passive thr_R : 3 ;	% name p ;
% active the_bus : 3 ;	% name c ;
% passive cnx : 4 ;	% name e0 ;
% active the_memory : 2 ;	% name e1 ;
% active the_process : 2 ;	% name e2 ;
	% name e3 ;

Sender/Receiver model Définition du modèle

the_processor[x,e3] | the_bus[c,e2,e3] | the_memory[m,e2] | the_process[m,-].(thr_R[x,e1,-] | thr_S[x,e0,-] | cnx[c,e0,e1,-]) ;

Reaction rules

the_process[m,-].(thr_R[x,e1,-] | thr_S[x,e0,-] | cnx[c,e0,e1,-])-> the_process[m,p] || (thr_R[x,e1,p]| thr_S[x,e0,p]) || cnx[c,e0,e1,p] ;
the_memory[m,e2].$0 || the_process[m,p] -> the_memory[-,e2].the_process[-,p].($1 | $2) || nil ;
the_processor[x,e3].$2 || (thr_R[x,e1,p] | thr_S[x,e0,p]) -> the_processor[-,e3].(thr_R[-,e1,p] | thr_S[-,e0,p]) || nil ;
the_bus[c,e2,e3].$1 || cnx[c,e0,e1,p]-> the_bus[-,e2,e3].cnx[-,e0,e1,p] || nil ;

% property *Well-instal* (the_processor[-,e3].(thr_R[-,e0,p]| thr_S[-, e1, p]) | the_bus[-, e2, e3].cnx[-,e1,e0,p]| the_memory[-, e2].the_process[-, p].($1 | $2)) ;

TABLE 6.1 – *Vérification de la Correction de l'Installation par BigMC.*

mons que l'installation a réussi sans perte d'information et que toutes les dépendances ont été satisfaites en vérifiant que tous les noms internes et externes du bigraphe (m, c et x) ont disparus, ce qui explique une satisfaction des dépendances.

Vérification de Cohérence de la Reconfiguration

Pour spécifier une nouvelle propriété *Well-reconf*, nous sommes contraints de redéfinir le modèle afin qu'il se prête mieux à la deuxième vérification en introduisant d'autres règles réaction. La table (6.3) décrit ce modèle avec la propriété *Well-reconf*. Nous supposons que le processus *the_process* arrive à se reconfigurer en activant un thread à la place d'un autre. Dans ce cas de figure, le thread *th_R1* défini initialement dans le modèle va être remplacé par le thread *th_R2* (au bout de quelques transformations).

La vérification de la propriété de reconfiguration de ce modèle par l'outil *BigMC* donne un résultat expliqué par les dernières lignes de la table 6.4. Ainsi le processus de vérification est arrivé au bout (complete !) après 14 transformations subies par le modèle initial. Le fait que la vérification n'a pas abouti sur un contre-exemple, cela signifie que le modèle peut atteindre un autre état représentant une nouvelle configuration.

Dans un premier temps, les orientations d'implémentation et de vérification pour ce projet de thèse étaient vers l'utilisation d'un logiciel de vérification de bigraphes existant à savoir *BigMC*. Sauf que, et après une manipulation de cet outil, plusieurs limites sont apparues :

- *BigMC* utilise un ensemble restreint de la syntaxe des bigraphes. Particulièrement, il ne permet pas de spécifier les ports et ne différencie pas entre nom-interne et nom-externe d'un nœud.
- il ne possède aucune base formelle spécifiée, il est codé en Java avec une utilisation massive de la structure de données *Pile*,
- il ne génère pas du code exécutable du système vérifié,
- il utilise un seul module pour décrire le modèle et ses propriétés,
- son processus vérification n'est pas générique : le modèle doit s'adapter à la vérification d'une propriété donnée.
- en plus, il n'est pas évident de pouvoir l'intégrer dans un environnement de développement, tel que *Eclipse*, en vue d'un couplage avec d'autres outils.

Pour toutes ces raisons, nous nous sommes retournés, pour vérifier nos

Welcome to BigMC !
$>$ *C* : *Program* $\sim$ 1*BigMC/bin/BigMC* $-$ *m*1000 $-$ *r*50 $-$ *pC* : \...\ BigMC_AADLmodel.bgm

1 : (the_processor[x,e3].nil | the_bus[c,e2,e3].nil | the_memory[m,e2].nil | the_process[m,-].(thr_R[x,e1,-].nil | thr_S[x,e0,-].nil | cnx[c,e0,e1,-].nil))
2 : (the_processor[x,e3].nil | the_bus[c,e2,e3].nil | the_memory[m,e2].nil | the_process[m,p].nil || (thr_R[x,e1,p].nil | thr_S[x,e0,p].nil) || cnx[c,e0,e1,p].nil)
3 : (nil || (thr_R[x,e1,p].nil | thr_S[x,e0,p].nil) || cnx[c,e0,e1,p].nil | the_processor[x,e3].nil | the_bus[c,e2,e3].nil | the_memory[- ,e2].the_process[-,p])
4 : (the_processor[-,e3].(thr_R[-,e1,p].nil | thr_S[- ,e0,p].nil) | the_bus[c,e2,e3].nil | the_memory[m,e2].nil | the_process[m,p].nil || nil || cnx[c,e0,e1,p].nil)
5 : (the_processor[x,e3].nil | the_bus[-,e2,e3].cnx[- ,e0,e1,p].nil | the_memory[m,e2].nil | the_process[m,p].nil || (thr_R[x,e1,p].nil | thr_S[x,e0,p].nil) || nil)
6 : (nil || nil || cnx[c,e0,e1,p].nil | the_processor[- ,e3].(thr_R[-,e1,p].nil | thr_S[-,e0,p].nil) | the_bus[c,e2,e3].nil | the_memory[-,e2].the_process[-,p])
7 : (nil || (thr_R[x,e1,p].nil | thr_S[x,e0,p].nil) || nil | the_processor[x,e3].nil | the_bus[-,e2,e3].cnx[- ,e0,e1,p].nil | the_memory[- ,e2].the_process[-,p])
8 : (the_processor[-,e3].(thr_S[-,e0,p].nil | thr_R[- ,e1,p].nil) | the_bus[- ,e2,e3].cnx[-,e0,e1,p].nil | the_process[m,p].nil || nil || nil | the_memory[m,e2].nil)
9 : (the_memory[-,e2].the_process[-,p].() | the_bus[- ,e2,e3].cnx[-,e0,e1,p].nil | the_processor[-,e3].(thr_S[- ,e0,p].nil | thr_R[-,e1,p].nil) | nil || nil || nil)

[*mc* :: *step*] *Complete !*
[*mc* :: *report*] [q : 0 / g : 9] @ *10*

TABLE 6.2 – *Résultat de la vérification de "Well-instal".*

# Déclarations des Nœuds	# Déclarations des Liens et noms
% active the_processor : 2 ;	% name x ;
% passive thr_S : 3 ;	% name m ;
% passive thr_R1 : 3 ;	% name p ;
% passive thr_R2 : 3 ;	% name c ;
% active the_bus : 3 ;	% name e0 ;
% passive cnx : 4 ;	% name e1 ;
% active the_memory : 2 ;	% name e2 ;
% active the_process : 2 ;	% name e3 ;

Sender/Receiver model Définition du modèle

the_processor[x,e3] | the_bus[c,e2,e3] | the_memory[m,e2] | the_process[m,-].(thr_R1[x,e1,-] | thr_S[x,e0,-] | cnx[c,e0,e1,-]) ;

Reaction rules

the_process[m,-].(thr_R1[x,e1,-] | thr_S[x,e0,-] | cnx[c,e0,e1,-])-> the_process[m,p] || (thr_R1[x,e1,p]| thr_S[x,e0,p]) || cnx[c,e0,e1,p] ;
the_memory[m,e2].$0 || the_process[m,p] -> the_memory[-,e2].the_process[-,p].($1 | $2) || nil ;
the_processor[x,e3].$2 || (thr_R1[x,e1,p] | thr_S[x,e0,p]) -> the_processor[-,e3].(thr_R1[-,e1,p] | thr_S[-,e0,p]) || nil ;
the_bus[c,e2,e3].$1 || cnx[c,e0,e1,p]-> the_bus[-,e2,e3].cnx[-,e0,e1,p] || nil ;
the_processor[-,e3].(thr_R1[-,e1,p].nil | thr_S[-,e0,p].nil) -> the_processor[-,e3].(thr_R2[-,e1,p].nil | thr_S[-,e0,p].nil)

% property *Well-reconf* (the_processor[-,e3].(thr_R1[-,e1,p].nil | thr_S[-,e0,p].nil) | the_bus[-, e2, e3].cnx[-,e0,e1,p].nil | the_memory[-,e2].the_process[-,p].())

TABLE 6.3 – *Vérification de la Reconfiguration par BigMC*

Welcome to BigMC !
> *C* : *Progra* ∼ 1*BigMC*/*bin*/*BigMC* − *m*1000 − *r*50 − *pC* : \...\ BigMC_AADLmodel.bgm
1 : (the_processor[x,e3].nil | the_bus[c,e2,e3].nil | the_memory[m,e2].nil | the_process[m,-].(thr_R1[x,e1,-].nil | thr_S[x,eo,-].nil | cnx[c,eo,e1,-].nil))
2 : (the_processor[x,e3].nil | the_bus[c,e2,e3].nil | the_memory[m,e2].nil | the_process[m,p].nil || (thr_R1[x,e1,p].nil | thr_S[x,eo,p].nil) || cnx[c,eo,e1,p].nil)
3 : (nil || (thr_R1[x,e1,p].nil | thr_S[x,eo,p].nil) || cnx[c,eo,e1,p].nil | the_processor[x,e3].nil | the_bus[c,e2,e3].nil | the_memory[-,e2].the_process[-,p].())
4 : (the_processor[x,e3].nil | the_bus[-,e2,e3].cnx[-,eo,e1,p].nil | the_memory[m,e2].nil | the_process[m,p].nil || (thr_R1[x,e1,p].nil | thr_S[x,eo,p].nil) || nil)
5 : (the_processor[-,e3].(thr_R1[-,e1,p].nil | thr_S[-,eo,p].nil) | the_bus[c,e2,e3].nil | the_memory[m,e2].nil | the_process[m,p].nil || nil || cnx[c,eo,e1,p].nil)
6 : (nil || (thr_R1[x,e1,p].nil | thr_S[x,eo,p].nil) || nil | the_processor[x,e3].nil | the_bus[-,e2,e3].cnx[-,eo,e1,p].nil | the_memory[-,e2].the_process[-,p].())
7 : (nil || nil || cnx[c,eo,e1,p].nil | the_processor[-,e3].(thr_R1[- ,e1,p].nil | thr_S[- ,eo,p].nil) | the_bus[c,e2,e3].nil | the_memory[-,e2].the_process[-,p].())
8 : (the_processor[-,e3].(thr_S[-,eo,p].nil | thr_R1[-,e1,p].nil) | the_process[m,p].nil || nil || nil | the_memory[m,e2].nil | the_bus[-,e2,e3].cnx[-,eo,e1,p].nil)
9 : (the_bus[c,e2,e3].nil | the_processor[-,e3].(thr_S[-,eo,p].nil | thr_R2[-,e1,p].nil) | the_process[m,p].nil || nil || cnx[c,eo,e1,p].nil | the_memory[m,e2].nil)
10 : (the_processor[-,e3].(thr_S[-,eo,p].nil | thr_R2[-,e1,p].nil) | the_memory[- ,e2].the_process[-,p].() | the_bus[-,e2,e3].cnx[- ,eo,e1,p].nil | nil || nil || nil)
11 : (the_processor[-,e3].(thr_S[-,eo,p].nil | thr_R2[-,e1,p].nil) | nil || nil || cnx[c,eo,e1,p].nil | the_memory[-,e2].the_process[- ,p].() | the_bus[c,e2,e3].nil)
12 : (the_processor[-,e3].(thr_S[-,eo,p].nil | thr_R2[-,e1,p].nil) | the_process[m,p].nil || nil || nil | the_memory[m,e2].nil | the_bus[-,e2,e3].cnx[-,eo,e1,p].nil)
13 : (the_memory[-,e2].the_process[-,p].() | the_bus[- ,e2,e3].cnx[-,eo,e1,p].nil | the_processor[-,e3].(thr_S[- ,eo,p].nil | thr_R2[-,e1,p].nil) | nil || nil || nil)
[*mc* :: *step*] *Complete !*
[*mc* :: *report*] [q : 0 / g : 13] @ *14*

TABLE 6.4 – *Résultat de Vérification de Well-reconf par BigMC*

modèles bigraphiques, vers un autre outil dont les fondements formels sont bien connus, il s'agit du système *Maude* et son *model-checker LTL*.

6.3.2 Système Maude et Model-Checker LTL

Étant conscients des atouts et les points forts du système *Maude* et de son model-Cheker *LTL* (Eker et al. 2003), nous avons décidé de l'utiliser alternativement pour valider les propriétés de notre modèle bigraphique issu d'une spécification AADL. Nous avons considéré, essentiellement, les mêmes propriétés : la correction de l'installation (*Well-instal*) et la cohérence de la reconfiguration (*Well-reconf*). *Maude* est un langage de spécification et d'implémentation. Il exécute des programmes qui correspondent à des théories de la *logique de réécriture*.
La logique de *réécriture* offre un cadre formel nécessaire pour la spécification et l'étude du comportement des systèmes concurrents. Elle a été introduite par Meseguer (1997) comme un résultat des travaux sur les logiques générales. Depuis, cette logique a été largement utilisée pour spécifier et analyser des systèmes et langages dans différents domaines d'applications (Benammar 2011). Un autre aspect important de la logique de réécriture est qu'elle représente un cadre logique et sémantique général dans lequel des langages et des modèles de calcul de nature largement différentes peuvent être représentés. *Maude* est le langage le plus connu qui implémente cette logique.

Il possède une syntaxe simple expressive et performante. Il offre peu de constructions syntaxiques mais une sémantique claire. *Maude* est doté d'un ensemble d'outils qui permettent l'exécution et la vérification des modèles. Il offre un outil de validation très puissant : le model-checker *LTL* (Logique Temporelle Linéaire). Cet outil permet de vérifier les propriétés d'un système qui doivent être exprimées par des formules *LTL*.

On distingue trois types de modules dans *Maude* (Clavel et al. 2007c;b) :
- module fonctionnel : permet de représenter une théorie fonctionnelle,
- module système : permet d'exprimer une théorie de réécriture,
- module orienté objet : permet l'expression des concepts *Objet*.

Les deux premiers types de modules peuvent être définis dans *CORE MAUDE* (Clavel et al. 2007a), par contre le dernier doit être défini dans *FULL MAUDE* (Clavel et al. 2007c) ainsi que les vues et les modules paramétrés.

Les atouts du système *Maude* qui nous ont encouragés à l'exploiter sont :

- *Maude* favorise l'utilisation d'une syntaxe *user-defined*, ce qui permet de spécifier les types et les opérations avec des noms significatifs,
- il permet de modéliser tout type de système,
- il génère un système exécutable,
- un modèle décrit par *Maude*, est générique.
- le système *Maude* est intégrable dans des environnements de développements.
- la vérification par le model-checker de *Maude* est fondée sur une base formelle qui est la logique de *réécriture*.

Définition du Modèle

Une transformation des modèles bigraphiques vers *Maude* est nécessaire pour générer un module *Maude* exécutable qui constituera une des entrées du model-checker. Une correspondance entre les éléments bigraphiques (nœud, site, région, etc.) et les éléments d'un module *Maude* est établie (voir table 6.5).

Aspect	**Élément Bigraphique**	**Concept dans Maude**
Structurel	- Bigraphe	Terme de Sorte ***Bigraph***
	- Région R	Terme de sorte ***Region***
	- Noeud N	Terme de sorte ***Node***
	- Site S	Terme de sous-sorte ***Site*** de sorte ***Node***
	- Port P	Terme de sorte ***Port***
	- Nom Interne/Externe	Terme de sorte ***HyperEdge***
Dynamique	- Instance de bigraphe	Terme de sorte ***Configuration***
	- Règle de Réaction	Règle de réécriture ***R*** : ***Configuration*** $\rightarrow$ ***Configuration***

TABLE 6.5 – *Implémentation d'un Bigraphe en Maude*

Le plugin *Bigraph2Maude* (Alliouche et al. 2013) vient pour implémenter l'approche de transformation des bigraphes en *Maude*. le processus de manipulation des bigraphes. *Bigraph2Maude* se sert des description des éléments bigraphiques décrites dans les figures 6.8 et 6.9 pour réaliser ses translations. De la même manière que les plugins précédents, il se sert des outils *GMF* sous *Eclipse* pour réaliser une transformation automatique des bigraphes vers des modules *Maude*. Ceci favorise son intégration avec les autres plugins dans l'environnement *Eclipse*.

La figure 6.8 représente les déclarations nécessaires dans un module fonctionnel de *Maude* pour pouvoir définir un bigraphe. Les types (Sort) décrivent

les éléments d'un bigraphe. On y trouve les sortes *Bigraph*, *Region*, *Node*, *Site* et un ensemble de sortes nécessaires à la définition des constituants d'un bigraphe.
Dans le même module, nous décrivons les opérations nécessaires à la construction d'un terme représentant un bigraphe. L'opération "##" représente une concaténation de régions construisant un bigraphe. L'opération "*Nest*" exprime la relation d'imbrication (père-fils) entre deux nœuds du bigraphe.

BIGRAPH.maude

```
fmod BIGRAPH is
sorts Bigraph Region ListRegion ListElement  IdRegion Node  Site IdNode
      IdSite  Port ListPort HyperEdgeC  HyperEdgeO  idEdgeC idEdgeO
      ListHyperEdge Nest ListNest  PlaceGraph.

subsorts IdSite IdNode Nest  ListNest < PlaceGraph .
subsort Nest  <  ListNest  .
subsort Site Node < ListElement .
subsort Region < ListRegion .
subsort ListRegion < Bigraph .
subsort  Port < ListPort .
subsorts HyperEdgeC HyperEdgeO < ListHyperEdge .
  ⋮
              **** REGION
op _##_ : ListRegion ListRegion -> ListRegion [ ctor assoc comm id: nilReg prec 25 ] .
op `[_:_#_#_`] : IdRegion ListElement PlaceGraph ListHyperEdge -> Region [ctor prec 24 ] .

              *** PlaceGraph
op _`,_ :  ListNest ListNest -> ListNest [ctor assoc comm id: nilLnk  prec 22 ]  .
op Nest`(_`,_`) : IdNode IdNode -> Nest  [ctor prec 21 ]  .
op Nest`(_`,_`) : IdNode IdSite -> Nest [ctor prec 21 ]  .
  ..............

              ***  NODES  & SITES
op __ : ListElement ListElement -> ListElement [ ctor assoc comm prec 23 id: nilElem ] .
op <_> : IdSite -> Site [ctor prec 22 ] .
op <_`,_> : IdNode ListPort  -> Node [ctor prec 22 ] .
```

FIGURE 6.8 – *Déclaration des éléments bigraphiques en Maude*

Les règles de réaction décrivant la dynamique d'un bigraphe sont traduites par des règles de réécriture en *Maude*. Le système *Maude* va exécuter le modèle en utilisant ces règles de réécritures. Nous avons alors défini un module de type *Système*, dans lequel nous avons défini les règles de réécritures qui seront appliquées à un bigraphe structurel dans un état donné (*Configuration*). Le changement ou transition d'un état du bigraphe vers un autre est représenté par l'application et l'exécution d'une règle de réécriture ou plusieurs en même temps(concurrence).

Dans la figure 6.9, les règles de réécriture *compose* et *reconf* décrivent de façon générique les mécanismes nécessaires pour réaliser une composition et une transformation de bigraphes.

Plusieurs autres règles peuvent être définies selon les préoccupations du spécifieur.

BIGRAPH-BEH.fm

```
mod BIGRAPH-BEH is  *** Module système pour la définition des règles de réecriture
including BIGRAPH BOOL .

sort Configuration .
subsort ListRegion < Configuration .

op _&_ : Configuration Configuration -> Configuration .
  ..................
  ..................

*** La composition

rl [compose] : [r0 : < n0 , x pL0 , ( s0 , eL) > # cL] & [r1 : < n1 ,  x pL1 ,
               fL > # oL ]  =>  [r0 : < n0 , pL0 , ( < n1 ,  pL1 ,  fL > , eL )>
               ## remp(x,oL) # remp(x,cL) if corr(s0,r1) == true .

*** La reconfiguration d'une région

rl [reconf] : [r0 : < n0 , pL0 ,  eL > # cL] & [r1 : < n1 ,  pL1 ,  fL > # eL ]
        =>  [r1 : < change(n0, n1) ,  change(pL0, pL1) ,  change(eL, fL) >
            # change(cL, eL) ] .
```

FIGURE 6.9 – *Déclaration des Règles de Réécriture*

Exemple : Pour l'exemple considéré dans ce chapitre, nous avons utilisé les spécifications décrites dans les figures 6.8 et 6.9 pour décrire les bigraphes du système 'Émetteur-Receveur'. La figure 6.10 présente le module *Maude* spécifiant notre exemple. On peut distinguer les bigraphes *big1* et *big2* représentant respectivement les bigraphes G_S et G_H de l'exemple considéré. *big3* est le bigraphe G_S après une opération de décomposition.

L'effet de l'application de la règle de réécriture **compose**, de la figure 6.9, sur les bigraphes *big2* et *big3* est représenté dans la figure 6.11, où l'on peut observer le résultat de la composition des bigraphes *big2* et *big3*.

Dans ce qui suit, nous présentons les principales manipulations nécessaires à la réalisation des vérifications via le model-checker de *Maude*.

Vérification des Propriétés

Pour vérifier les propriétés *Well-instal* et *Well-reconf* sur le module *BIG-EXAMPLES* de la figure 6.10, nous avons ajouté deux modules fonctionnels *Maude*. Un est dédié à la spécification de ces deux propriétés et le deuxième pour exprimer l'analyse proprement dite (voir figures 6.12, 6.13). Les propriétés *Well-instal* et *Well-reconf* sont déclarées comme des propositions LTL,

Bigraph-Examples.maude

```
fmod BIGRAPH-EXAMPLES is
including BIGRAPH .

ops big1 big2 big3 big4 big5 dep: -> Bigraph [ctor] .
-------------------------------------------
**** Bigraphe Gs
eq big1 = { [ the-soft : < the-process , m-p , (( < thr-s , x-s e0-s , nilElem > ,
     < cnx , e0-cn e1-cn c-cn , nilElem > ), < thr-r , x-r  e1-r , nilElem > ) >]  ##
     e0 < e0-s e0-cn > # e1 < e1-r e1-cn > # x < x-s x-r > # m < m-p > # c < c-cn > } .

**** Bigraphe Gh
eq big2 = { [ the-hard : ((< the-processor, x-p e2-p, ( < s0 > , nilElem) > ,
         < the-bus,  e2-b e3-m c-b, ( < s1 > , nilElem )>) , < the-memory, e3-m  m-me ,
          ( < s2 > , nilElem ) > )]  ## e2 < e2-p e2-b > # e3 < e3-m e3-b > # x < x-p >
          # m < m-me > # c < c-b > } .

**** Bigraphe Gs décomposé
eq big3 = { [ r0 : < the-process , m-p p-p, (( < ss0 > , < ss1 >) , nilElem) > ] #
          [ r1 : < cnx , e0-cn e1-cn c-cn p-c , nilElem > ] # [ r2 :
          (< thr-s , x-s e0-s p-s , nilElem > , < thr-r , x-r  e1-r p-r, nilElem >) ]
           ## e0 < e0-s e0-cn > # e1 < e1-r e1-cn > # p <p-r p-s p-c p-p > # x < x-s x-r>
            # m < m-p > # c < c-cn >} .
-------------------------------------------------------------------------
```

FIGURE 6.10 – *Exemple Émetteur-Receveur sous Maude*

Maude Console

```
Elapsed time: 00:00:00.017

Maude> rewrite in BIG-EXAMPLES : big2 & big3 .
rewrites: 197 in 1628036047000ms cpu (6ms real) (0 rewrites/second)
result Region: [the-hard : < ss0 > < ss1 > < the-process,p-p > < thr-s,e0-s p-s
    > < thr-r,x-p e1-r p-r > < cnx,e0-cn e1-cn p-c > < the-bus,e2-b e3-b > <
    the-memory,e3-m > < the-processor,e2-p > # Nest(the-bus,cnx),Nest(
    the-memory,the-process),Nest(the-processor,thr-s) # e0 < e0-s e0-cn > ## e1
    < e1-r e1-cn > ## e2 < e2-p e2-b > ## e3 < e3-b e3-m > ##  p < p-p
    p-c p-s p-r >]
Maude>
```

FIGURE 6.11 – *Résultat d'une Opération d'Installation*

elles sont satisfaites (valeur 'true' est retournée) si et seulement si les configurations *Conf1* et *Conf2* sont atteintes respectivement. Ces configurations pour notre exemple sont indiquées au niveau des équations (1 et 2) du module de la figure 6.12.

Bigraph-check.maude

```
mod BIGRAPH-Properties is
protecting BIG-EXAMPLES .
including SATISFACTION .
subsort Configuration < State .

ops Conf1 Conf2 : -> Configuration .
op Well-instal : -> Prop [ctor] .
op Well-reconf : -> Prop [ctor] .

eq Conf1 = big4 .     *** Equation 1
eq Conf2 = big5 .     *** Equation 2

eq Conf1 |= Well-instal = true .
eq Conf2 |= Well-reconf = true .

endm
```

FIGURE 6.12 – *Spécification des propriétés*

Le résultat d'une vérification est une valeur booléenne indiquant vraie dans le cas où la propriété est vérifiée ou un contre exemple est retourné dans la cas contraire. Les commandes qui invoquent cette vérification sont données par :

Maude> reduce in modelCheck(Init1, Well-instal) .
Maude> reduce in modelCheck(Init2, Well-reconf) .

Bigraph-check.maude

```
mod BIGRAPH-Analisys is
including BIGRAPH-Properties .
including MODEL-CHECKER .
including LTL-SIMPLIFIER .

ops Init1 Init2 : -> Configuration [ctor] .

eq Init1 = big2 & big3 .

eq Init2 = reconf ( big4 , the-processor2 ) .

endm
```

FIGURE 6.13 – *Analyse via le Model-Checker LTL*

6.4 CONCLUSION

Dans ce chapitre, nous avons présenté plusieurs modules (plugins), chacun est dédié à une phase de la transformation d'une architecture AADL vers

un modèles bigraphique éditable et vérifiable.

Après l'intégration des différents plugins obtenus dans l'environnement *Eclipse*. Nous pouvons bénéficier d'un outil global permettant la génération et la manipulation des bigraphes qui modélisent des architectures logicielles et leur évolution. La figure 6.14 montre l'architecture générale de la plateforme *BigraphPlatform*. Elle se base sur plusieurs langages et relient différents systèmes. Ceci reflète exactement les efforts consentis pour concrétiser nos résultats théoriques cités dans les premiers chapitres de ce manuscrit.

La figure 6.15 est une capture d'écran d'une partie du menu de cette plateforme après l'intégration des différents plugins.

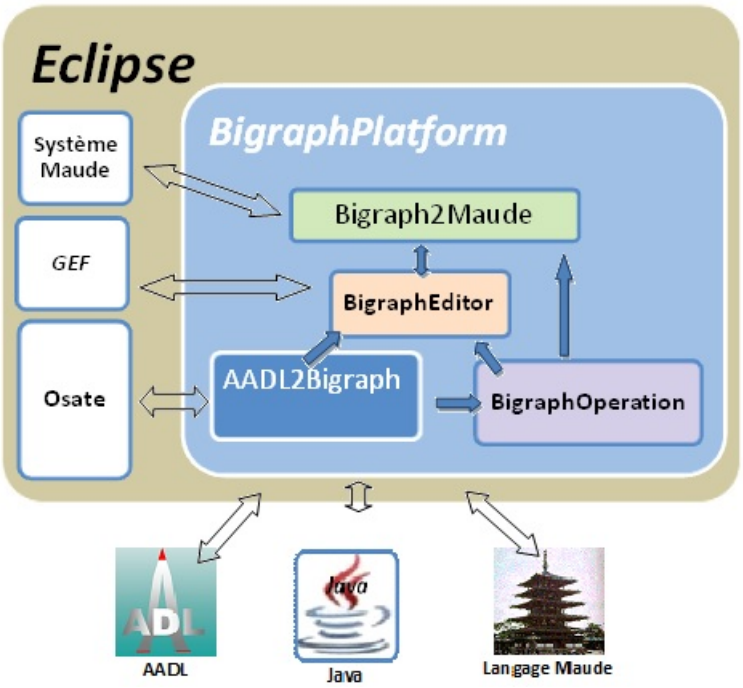

FIGURE 6.14 – *Architecture de BigraphPlatform*

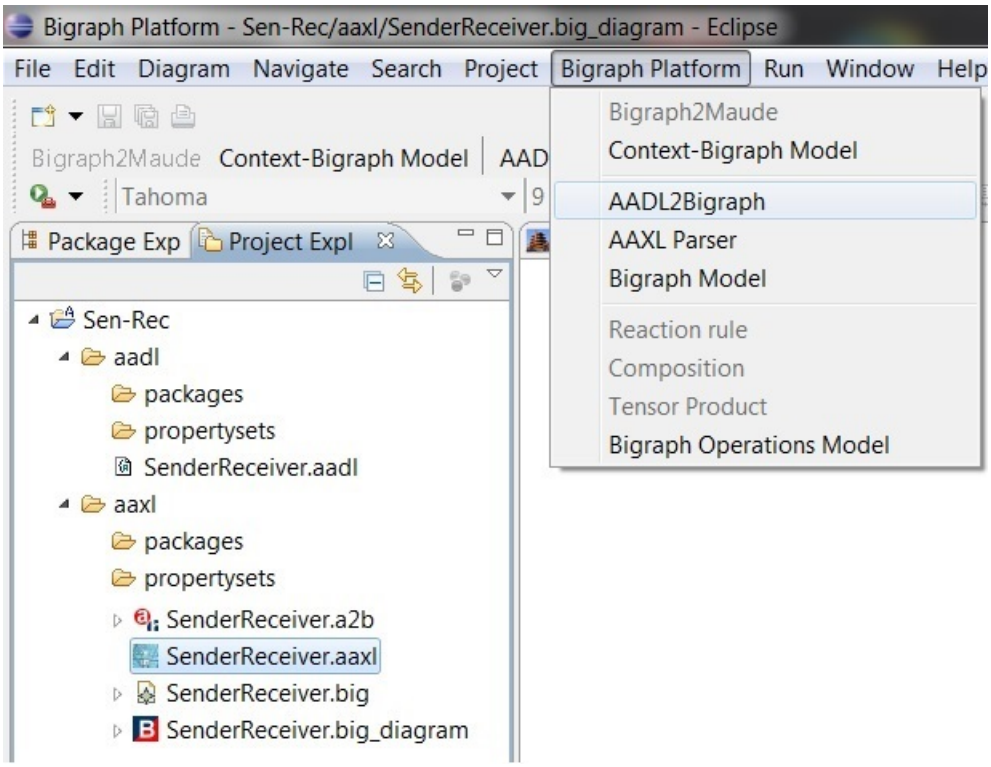

FIGURE 6.15 – *Capture d'Ecran de la Plateforme Résultat*

Conclusion Générale

L'ordinateur le plus important est celui qui bouillonne dans nos crânes.

Alan J. Perlis.

Dans ce document, nous avons présenté l'ensemble des travaux réalisés dans le cadre de cette thèse. Nous avons passé en revue l'état de l'art et nous avons étudié la littérature concernant les domaines de recherche abordés. Nous avons introduit les concepts relatifs aux architectures logicielles et leur processus de déploiement en se basant sur les standards et les travaux de recherche. Nous avons présenté les langages de description des architectures les plus utilisés et les plus publiés. Nous avons mis l'accent sur les mécanismes de spécification du processus de déploiement dans ces langages.

L'objectif de cette thèse étant de proposer un modèle formel de déploiement d'architectures logicielles qui vise la vérification de l'installation des architectures logicielles en amont et à maîtriser leurs évolutions. Nous avons pour cela proposé une approche autour du langage AADL et les bigraphes dont l'objectif est, à la fois de maîtriser la complexité liée au déploiement d'une architecture logicielle et de favoriser le raisonnement et la vérification de celle-ci. Cette approche se fait fort de respecter les grands principes du génie logiciel à savoir l'abstraction, la cohérence et l'anticipation du changement.

Contributions

Dans cette thèse, nous avons proposé un modèle formel à base de BRS, permettant la modélisation de processus de déploiement d'une application architecturale dans un environnement cible. Nous avons :

- proposé un modèle générique de description d'une architecture logicielle et son environnement d'exécution par deux bigraphes indépendants,

- puis, nous avons décrit formellement le processus de déploiement en exploitant les opérations de manipulation des bigraphes,
- le bigraphe résultat a servi pour raisonner sur les dépendances logicielles/matérielles et particulièrement celles liées à l'opération de reconfiguration,
- ensuite, le langage AADL a été utilisé pour concrétiser le résultat théorique obtenu. En effet, un mapping entre les éléments architecturaux AADL et ceux des bigraphes a été défini tout en proposant des règles de transformation génériques (méta règles),
- nous avons ainsi modélisé les deux structures AADL (application et plateforme d'exécution) contenues dans la déclaration complète d'une configuration AADL, par deux bigraphes distincts. Ces derniers sont alors composés pour produire un nouveau bigraphe représentant le schéma de l'installation de l'application sur la plate-forme d'exécution. Nous avons également montré la capacité des *BRS* à modéliser une reconfiguration dans une architecture AADL exprimée à travers des *modes*.
- Pour illustrer notre approche sur un cas d'étude de grandeur naturelle, nous l'avons projetée sur la spécification d'une architecture AADL représentant un système distribué critique et temps-réel *PMS*,
- La possibilité de générer automatiquement des structures bigraphiques à partir d'une spécification AADL a été démontrée à travers l'outil *AADL2Bigraph*.
- *BigraphOperations* est un plugin qui sert à la réalisation des opérations de composition et de transformation des bigraphes,
- *Bigraph2Maude* est un autre plugin qui sert à transformer des modèles bigraphiques aux spécifications *Maude* afin de valider et vérifier le processus de déploiement.

Perspectives

Dans le futur et sur le plan théorique, nous projetons de :

- Étendre l'usage des bigraphes pour la formalisation du processus de déploiement des architectures d'applications sensibles au contexte.
- Formaliser avec les BRS, les autres activités du processus de déploiement.
- Prendre en charge le model-checking d'autres types de propriétés, surtout les propriétés non fonctionnelles telle que la QoS.
- Modéliser des applications complexes émergentes telles que la biotech-

nologie et *cloud computing* par des bigraphes et tester la capacité de ce formalisme dans d'autres domaines.
– Étudier la possibilité de formalisation d'autres ADLs.

Sur le plan pratique, nous comptons de :
– compléter le développement d'autres plugins pour la plateforme *BigraphPlatform*,
– introduire les stratégies *Maude* pour spécifier et les vérifier automatiquement des modèles bigraphiques,
– enrichir les informations des plugins par une base de données contenant les caractéristiques de plateformes existantes pour pouvoir déterminer pour une architecture donnée la plateforme la plus adéquate pour son déploiement (moins d'incompatibilités possibles).

Bibliographie

SEI. AADL Team. Osate : An extensible source aadl tool environment. Rapport technique, Site : ww.aadl.info/aadl/currentsite/.../osate-down.html, 2004. (Cité pages 35 et 46.)

T. Abdoul, J. Champeau, P. Dhaussy, P. Pillain, et J. Roger. AADL execution semantics transformation for formal verification. Dans *13th International Conference on Engineering of Complex Computer Systems*, pages 263–268, 2008. (Cité pages 70 et 71.)

ACCORD. Etat de l'art sur les langages de description d'architecture (ADLs). Rapport technique, CNAM, EDF RD, ENST, ENSTBretagne, France Telecom RD, INRIA, LIFL, Softeam et RNTL, 2002. (Cité pages 11 et 12.)

R. Acosta Bermejo. Reactive operating system reactive java objects. *Electronic Journal on Networks and Distributed Processing*, 3(11) :337–354, 2001. ISSN 1262-3261. (Cité page 56.)

R. Allen. *A Formal Approach to Software Architecture*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PE, USA, May 1997. (Cité page 22.)

R. Allen, R. Douence, et D. Garlan. Specifying and analyzing dynamic software architectures. Dans *in Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, pages 21–37, Lisbon, Portugal, 1998. (Cité pages 2 et 102.)

R. Allen et D. Garlan. The wright architectural specification language. Rapport Technique MU-CS-96-TBD, Carnegie Mellon University, School of Computer Science, Pittsburg, PA, USA, September 1996. (Cité page 17.)

M. Alliouche, A. Laggoune, et F. Belala. Un outil de translation des bigraphes en maude. Master's thesis, Université Constantine 2, Algérie, 2013. (Cité page 126.)

G. Bacci, D. Grohmann, et M. Miculan. Dbtk : A toolkit for directed bigraphs. Dans *CALCO*, pages 413–422, 2009. (Cité pages vii, 55 et 64.)

O. Barais. *Construire et Maîtriser l'evolution d'une architecture logicielle à base de composants*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Lille, France, Nov 2005. (Cité page 19.)

L. Bass, P. Clements, et R. Kazman. *Software Architecture in Practice*. Addison Wesley, 2003. ISBN 0-321-15495-9. (Cité pages 9, 10 et 13.)

L. Bass, P. Clements, et R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Pearson Education, 2012. ISBN 9780321815736. (Cité pages 10, 13 et 15.)

M. Benammar. *Une Approche Basée Architecture pour la Spécification Formelle des Systèmes Embarqués*. Thèse de doctorat, Université Mentouri de Constantine, 2011. (Cité pages 4, 69, 70, 71 et 125.)

M. Benammar, F. Belala, K. Barkaoui, et N. Benlahrache. Extension d'ABA-ReL par les propriétés d'exécution. Dans *Revue de la Nouvelle Technologie de l'Information RNTI-L-4, 3ième Conférence Francophone Sur les Architectures Logicielles, CFP-CAL'09, éditions Cépaduès*, pages 45–58, Nancy, France, Mars 2009. (Cité page 69.)

N. Benlahrache et F. Belala. Towards formalising installation and reconfiguration tasks of AADL architecture. *Int. J. of Communication Networks and Distributed Systems (IJCNDS), Inderscience Publishers*, 11(4) :431–452, 2013. (Cité pages 84 et 94.)

N. Benlahrache, F. Belala, et K. Barkaoui. Description formelle du déploiement d'architectures AADL basée sur les systèmes réactifs bigraphiques (BRS). Dans *Proceedings CAL'11 (Conférence Francophone sur les Architectures Logicielles), Lille, France*, pages 65–74, 2011. (Cité pages 73, 76, 78 et 93.)

N. Benlahrache, F. Belala, C. Bouanaka, et M. Benammar. Vers un modèle de déploiement à base de bigraphes. Dans *Conférence Francophone sur les Architectures Logicielles, RNTI L-5, Edittion Cépadues*, pages 141–153, 2010. ISBN 978.2.85428.930.5. (Cité pages 72 et 81.)

N. Benlahrache, F. Belala, et T. A. Cherfia. Model checking brs based AADL specification. *International Journal of Computer Applications*, 52(21) :43–52, August 2012. (Cité page 119.)

B. Berthomieu, J. P. Bodeveix, C. Chaudet, S. Dal-Zilio, M. Filali, et F. Vernadat. Formal verification of aadl specifications in the topcased environment. Dans *Ada-Europe*, pages 207–221, 2009. (Cité pages 70 et 71.)

L. Birkedal, T. C. Damgaard, A. J. Glenstrup, et R. Milner. Matching of bigraphs. *Electronic Notes in Theoretical Computer Science*, 175(4) :3–19, 2007. (Cité page 55.)

R. Boucebsi, A. Dembri, I. Zegachou, et N. Benlahrache. Conception et réalisation d'un outil de manipulation des bigraphes. Master's thesis, Université Mentouri, Constantine, 2012. (Cité page 115.)

E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, et J. B. Stefani. The FRACTAL component model and its support in java. *Software : Practice and Experience. Special Issue : Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12) :1113 ?1380, September-October 2006. (Cité page 18.)

R. Bruni, A. Lluch Lafuente, U. Montanari, et E. Tuosto. Style-based architectural reconfigurations. *Bulletin of the Euro. Assoc. Theo. Comp. Science.*, 94 : 161–180, 2007. (Cité pages 67, 80 et 81.)

F. Budinsky, D. Steinberg, et E. Merks. *EMF : Eclipse Modeling Framework*. Eclipse (Addison-Wesley). Addison Wesley Professional, 2009. ISBN 9780321331885. (Cité page 111.)

T. Bures, P. Hnetynka, et F. Plasil. Sofa 2.0 : Balancing advanced features in a hierarchical component model. Dans *SERA*, pages 40–48, 2006. (Cité page 24.)

A. R. Carmichael. Defining software architectures using the Hierarchical Object-Oriented Design method (HOOD). Dans *Proceedings of the conference on TRI-Ad'92*, TRI-Ada'92, pages 211–219, New York, NY, USA, 1992. ACM. ISBN 0-89791-529-1. (Cité page 50.)

A. Carzaniga, A. Fuggetta, R.S. Hall, D. Heimbigner, A. van der Hoek, et A. L. Wolf. A characterization framework for software deployment technologies. Technical report cu-cs-857, University of Colorado, 1998. (Cité pages 19, 20 et 95.)

Z. Chang, X. Mao, et Z. Qi. An approach based on bigraphical reactive systems to check architectural instance conforming to its style. Dans *Symposiun on TASE'07*, pages 57–66, 2007. (Cité pages 5, 55, 62, 67, 72, 80, 81, 82 et 102.)

Z. Chang, X. Mao, et Z. Qi. Towards a formal model for reconfigurable software architectures by bigraphs. Dans *Proceeding of the 7th Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pages 331–334. IEEE Computer Society, 2008. (Cité pages 5, 67 et 81.)

T.A. Cherfia, A. Bouchaib, H. Titer, et N. Benlahrache. Conception et réalisation d'un outil de transformation de modèles AADL en bigraphes. Master's thesis, Université Mentouri, Constantine, 2011. (Cité pages 112 et 113.)

M. Y. Chkouri, A. Robert, M. Bozga, et J. Sifakis. Translating AADL into BIP - application to the verification of real-time systems. Dans *MoDELS Workshops*, pages 5–19, 2008. (Cité pages 69 et 71.)

A. Choutri. *Un Cadre Formel Basé Logique des Tuiles pour les ADLs*. PhD thesis, Laboratoire Lire, Université Mentouri., Constantine, Algeria, 2011. (Cité page 18.)

E. M. Clarke, O. Grumberg, et D. A. Peled. *Model Checking*. University Press Group Limited, 2001. ISBN 978-0-262-03270-4. (Cité page 117.)

M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, et C. L. Talcott. Introduction. Dans *All About Maude*, pages 1–28, 2007a. (Cité page 125.)

M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, et C. L. Talcott. Syntax and basic parsing. Dans *All About Maude*, pages 39–59, 2007b. (Cité page 125.)

M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, et C. L. Talcott. Using maude. Dans *All About Maude*, pages 31–37, 2007c. (Cité page 125.)

G. Conforti, D. Macedonio, et V. Sassone. Bilog spatial logics for bigraphs. *Proceeding of the 32th ICALP'05, LNCS, Springer Verlag*, 3580 :766–778, 2005. (Cité page 58.)

T. C. Damgaard, A.J. Glenstrup, L. Birkedal, et R. Milner. An inductive characterization of matching in binding bigraphs. *Formal Asp. Comput.*, 25(2) : 257–288, 2013. (Cité page 63.)

A. Dearle. Software deployment, past, present and future. Dans *2007 Future of Software Engineering*, FOSE '07, pages 269–284, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. (Cité pages 11 et 95.)

P. Dissaux. Stood and DO-178B. Rapport technique, TNI-Europe, 2003. (Cité page 50.)

P. Dissaux et F. Singhoff. Stood and cheddar : Aadl as a pivot language for analysing performances of real time architectures. Dans *4th European Congress ERTSS Embedded Real Time Software and System*, Jan 2008. (Cité pages 48 et 51.)

J. Dubus, A. Flissi, N. Dolet, et P. Merle. Une démarche orientée modèle pour déployer des systèmes logiciels répartis. *L'OBJET*, 14(1-2) :35–59, 2008. (Cité pages 20 et 27.)

S. Eker, J. Meseguer, et A. Sridharanarayanan. The Maude LTL Model Checker and its implementation. Dans Thomas Ball et Sriram K. Rajamani, éditeurs, *SPIN*, volume 2648 de *Lecture Notes in Computer Science*, pages 230–234. Springer, 2003. ISBN 3-540-40117-2. (Cité page 125.)

P. Farail, P. Gaufillet, A. Canals, C. L. Camus, D. Sciamma, P. Michel, X. Crégut, et M. Pantel. *TOPCASED : An Open Source Development Environment for Embedded Systems*. ISTE Editor, From MDD Concepts to Experiments and Illustrations, 2006. (Cité pages 35, 37 et 48.)

P. H. Feiler et D. P. Gluch. *Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language*. SEI series in software engineering. Addison-Wesley, 2012. ISBN 978-0-321-88894-5. (Cité page 34.)

P. H. Feiler, D. P. Gluch, et J. J.Hudak. The architecture analysis & design language : AADL. Technical report, Université Carnegie Mellon, 2006. (Cité pages 18, 34 et 41.)

A. Flissi et P. Merle. Une démarche dirigée par les modèles pour construire les machines de déploiement des intergiciels à composants. *L'OBJET*, 11 (1-2) :79–94, 2005. (Cité page 27.)

D. Garlan. Formal modeling and analysis of software architecture : Components, connectors, and events. Dans *SFM*, pages 1–24, 2003. (Cité pages 2, 3, 8 et 13.)

D. Garlan, R. Monroe, et D. Wile. Acme : An architecture description interchange language. Dans *in Proceedings of CASCON ?97*, pages 169–183, 1997. (Cité page 17.)

D. Garlan et M.Shaw. An introduction to software architecture. Rapport Technique CMU-CS-94-166, School of Computer Science Carnegie Mellon University, Pittsburgh, PA 15213-3890, January 1994. (Cité page 16.)

A. J. Glenstrup, T. C. Damgaard, L. Birkedal, et E. Højsgaard. An implementation of bigraph matching. Rapport technique, IT University IT University of Copenhagen, Technical Report Series TR-2010-135, 2010. (Cité pages vii, 55 et 63.)

D. Grohmann. Security, cryptography and directed bigraphs. Dans *Proceedings of the 4th International Conference on Graph Transformations (ICGT'08)*,

volume 5214 de *Lecture Notes in Computer Science*, pages 487–489. Springer-Verlag, 2008. (Cité pages 55 et 64.)

D. Grohmann et M. Miculan. Directed bigraphs. Dans *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics (MFPS'07)*, volume 173 de *Electronic Notes in Theoretical Computer Science*, pages 121–137. Elsevier, 2007. (Cité page 55.)

M. Hadj Kacem. *Modélisation des applications distribuées à architecture dynamique : Conception et Validation*. Thèse de doctorat, Université de Toulouse et de Sfax, 2008. (Cité pages 12 et 84.)

A. Heydarnoori et W. Binder. A graph-based approach for deploying component-based applications into channel-based distributed environments. *JSW*, 6(8) :1381–1394, 2011. (Cité pages 23 et 28.)

J. Hugues et F. Singhoff. Développement de systèmes à l'aide d'AADL-Ocarina/Cheddar. Dans *Tutoriel présenté à l ?école d'été temps réel*, pages 25–34, Sep 2009. (Cité page 51.)

O.H. Jensen et R. Milner. Bigraphs and mobiles processes (revised). Technical report 580, University of Cambridge, 2004. ISSN : 1476-2986. (Cité pages 5, 54, 55 et 56.)

G. Lasnier, B. Zalila, L. Pautet, et J. Hugues. OCARINA : An environment for AADL models analysis and automatic code generation for high integrity applications. *Site : http ://ocarina.enst.fr*, 2009. (Cité pages 35 et 48.)

V. Lestideau. *Modèles et environnement pour configurer et déployer des systèmes logiciels*. These, Université de Savoie, Dec 2003. (Cité page 25.)

D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, et W. Mann. Specification and analysis of system architecture using rapide. *IEEE Trans. Software Eng.*, 21(4) :336–355, 1995. (Cité page 17.)

N. Medvidovic. Formal modeling of software architectures at multiple levels of abstraction. Dans *In Proceedings of the California Software Symposium, USA*, pages 28–40, 1996. (Cité page 16.)

N. Medvidovic et M. Sam. Software deployment architecture and quality-of-service in pervasive environments. Dans *International workshop on Engineering of software services for pervasive environments : in conjunction with the 6th ESEC/FSE joint meeting, ACM*, ESSPE'07, pages 47–51, New York, NY, USA, 2007. ISBN 978-1-59593-798-8. (Cité pages 4, 9 et 26.)

N. Medvidovic et R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transaction Software Eng.*, 26(1) :770–793, 2000. (Cité pages 3 et 18.)

N. Merle. Un méta-modèle pour l'automatisation du déploiement d'applications logicielles. *CoRR*, cs.NI/0411061, 2004. (Cité page 27.)

N. Merle. *Architecture pour les Systèmes de Déploiemnt Logiciel à grande échelle : Prise en compte des concepts d'entreprise et de stratégie*. PhD thesis, Université Joseph Fourier, Grenoble, France, 2005. (Cité page 26.)

J. Meseguer. Research directions in rewriting logic. Dans *Computational Logic*, volume 165 de *Lecture Notes in Computer Science*, pages 424–433, Marktoberdorf, Germany, 1997. NATO Advanced Study Institute, Springer-Verlag. (Cité pages 6 et 125.)

D. L. Metayer. Describing software architecture styles using graph grammars. *IEEE Trans. Software Eng.*, 24(7) :521–533, 1998. (Cité pages 67, 80, 81, 84 et 85.)

R. Milner. Axioms for bigraphical structure. *Mathematical Structures in Computer Science*, 15(6) :1005–1032, 2005. (Cité page 54.)

R. Milner. Pure bigraphs : Structure and dynamics. *Inf. Comput.*, 204(1) : 60–122, 2006. (Cité page 54.)

R. Milner. Bigraphs and their algebra. *Proceeding of the LIX Colloquium on Emerging Trends in Concurrency Theory, Electronic Notes in Theoretical Computer Science Elsevier,*, 209 :5–19, 2008. (Cité pages 57, 67 et 93.)

R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009. (Cité page 54.)

W. Moore, D. Dean, A. Gerber, G. Wagenknecht, et P. Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, 2004. (Cité pages 46 et 111.)

OMG. *Model Driven Architecture (MDA) Guide*, 2003. OMG doc. ab/2003-06-01. (Cité page 3.)

OMG. Unified Modeling Language : Superstructure version 2.0. formal/2005-07-04, August 2005. (Cité pages 3, 4, 11, 18 et 27.)

OMG. Deployment and configuration of component-based distributed application specification, version 4.0. Technical report, Object Management Group, 2006. (Cité page 24.)

A. Parrish, B. Dixon, et D. Cordes. A conceptual foundation for component-based software deployment. *Journal of Systems and Software.*, 57(3) :193–200, 2001. (Cité pages 4 et 23.)

G. Perrone, S. Debois, et T. Hildebrandt. A model checker for bigraphs. Dans *Proceedings of the 27th ACM Symposium in Applied Computing (Software Verification and Testing Track) ACM-SAC'12*, pages 1320–1325, 2012. (Cité pages 6, 64, 65 et 119.)

J. F. Rolland. *Développement et validation d'architectures dynamiques*. PhD thesis, l'Université Toulouse III - Paul Sabatier, France., 2008. (Cité page 42.)

International SAE. Architecture Analysis and Description Language Standard. Rapport technique, Site : http ://www.sae.org., 2008. version 2. (Cité pages 4, 18, 33 et 69.)

M. Shaw et D. Garlan. *Software architecture - perspectives on an emerging discipline*. Prentice Hall, 1996. ISBN 978-0-13-182957-2. (Cité pages 9 et 11.)

F. Singhoff, A. Plantec, et P. Dissaux. The cheddar project : a free real time scheduling analyzer. *Site : http ://beru.univ-brest.fr/ singhoff/cheddar*, 2008. (Cité pages 35 et 47.)

O. Sokolsky, I. Lee, et D. Clarke. Process-Algebraic interpretation of AADL models. Dans *14th International Conference on Reliable Software Technologies, Ada-Europe*, 2009. (Cité pages 70 et 71.)

C. Szyperski, D. Gruntz, et S. Murer. *Component Software : Beyond Object-Oriented Programming*. Component Software Series. Pearson Education, 2003. ISBN 9788131705230. (Cité page 11.)

V. Talwar, Q. Wu, C. Pu, W. Yan, G. Jung, et D. S. Milojicic. Comparison of approaches to service deployment. Dans *ICDCS*, pages 543–552. IEEE Computer Society, 2005. (Cité pages vii, 23, 29 et 95.)

S. Turki, E. Senn, et D. Blouin. Mapping the MARTE UML profile to AADL. Dans *MoDELS 2010 ACES-MB Workshop Proceedings*, pages 11–20, 2010. (Cité page 33.)

T. Vergnaud. *Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées*. Thèse de doctorat, École doctorale d'informatique, télécommunications et électronique de Paris, 2006. (Cité pages 4, 33, 42 et 69.)

T. Vergnaud, J. Hugues, L. Pautet, et F. Kordon. PolyORB : A schizophrenic middleware to build versatile reliable distributed applications. Dans *Ada-Europe*, pages 106–119, 2004. (Cité pages 50 et 71.)

S. Vestal. Metah. *ACM SIGSOFT Software Engineering Notes*, 25(1) :105–, jan 2000. ISSN 0163-5948. (Cité page 33.)

S. Xihong, Z. Wu, H. Liu, D. Zuo, et X. Yang. A comprehensive approach of SA based software deployment reliability estimation in neural networks. *International Conference on Pervasive Computing, Signal Porcessing and Applications*, pages 49–53, 2010. (Cité pages 2 et 4.)

Z. Yang, H. Kai, M. Dianfu, et P. Lei. Towards a formal semantics for the AADL behavior annex. Dans *DATE*, pages 1166–1171, Nice, France, 2009. IEEE. (Cité pages 70 et 71.)

P. Zhang, H. Muccini, et B. Li. A classification and comparaison of model checking software architecture techniques. *Journal of System Software, doi :10.1016/j.jss.2009.11.79*, 83(5) :723–744, 2010. (Cité pages 3 et 18.)

Annexes

A

Sommaire

A.1 Spécification AADL complète du Système PMS Statique

```
system global
end global;

system implementation global.impl
subcomponents
   Service_Event : system Service_Sys.impl;
   Nurse  : system Nurse_Sys.impl;
   Controller: device control.impl;
   Bus_global1: bus bus_gl.impl;
connections
   cnx0: event port Service_Event.out_alarm -> Nurse.in_alarm;
   cnx1: event port Nurse.out_answer -> Service_Event.in_answer;
   cnx2: event data port Controller.out_pressure ->
            Service_Event.in_pressure;
   cnx3: event data port Controller.out_temp ->
            Service_Event.in_temp;
properties
   Actual_Connection_Binding =>  reference Bus_global1
      applies to cnx0;
   Actual_Connection_Binding =>  reference Bus_global1
      applies to cnx1;
   Actual_Connection_Binding =>  reference Bus_global1
      applies to cnx2;
   Actual_Connection_Binding =>  reference Bus_global1
      applies to cnx3;
   Actual_Connection_Binding =>  reference Bus_global1
      applies to Service_Event.Ser_pro;
   Actual_Connection_Binding =>  reference Bus_global1
      applies to Nurse.Nur_pro;
end global.impl;

-- data
data temp
end temp;
data pressure
end pressure;
data message
end message;

system Service_Sys
features
 out_alarm : in out event port ;
 in_answer : in out event port ;
 in_pressure: in out event data port pressure;
 in_temp: in out event data port temp;
 end Service_Sys;

system implementation Service_Sys.impl
subcomponents
  Ser_pro: process Service_process.impl;
  Ser_mem: memory PMS_memory.impl;
  Ser_proc: processor PMS_processor.impl;
  Ser_bus: bus bus_gl.impl;
connections
   cnx10: event port Ser_pro.out_alarm -> out_alarm;
   cnx11: event port in_answer -> Ser_pro.in_answer;
```

```
57      cnx12: event data port in_pressure ->Ser_pro.in_pressure ;
58      cnx13: event data port in_temp -> Ser_pro.in_temp;
59    properties
60      Actual_Memory_Binding =>  reference Ser_mem
61          applies to Ser_pro;
62      Actual_Processor_Binding =>  reference Ser_proc
63          applies to Ser_pro;
64      Actual_Connection_Binding => reference Ser_bus
65          applies to Ser_pro;
66     end Service_Sys.impl;
67
68    process Service_process
69    features
70     out_alarm : in out event port;
71     in_answer: in out event port;
72     in_pressure: in out event data port pressure;
73     in_temp: in out event data port temp;
74    end Service_process;
75
76    process implementation Service_process.impl
77    subcomponents
78     Ser_thr: thread Service_thread.impl;
79    connections
80       cnx14: event port Ser_thr.out_alarm -> out_alarm;
81       cnx15: event port in_answer -> Ser_thr.in_answer;
82       cnx16: event data port in_pressure ->Ser_thr.in_pressure;
83       cnx17: event data port in_temp -> Ser_thr.in_temp;
84     end Service_process.impl;
85
86
87    thread Service_thread
88    features
89     out_alarm : out event port;
90     in_answer : in event port;
91     in_pressure: in event data port pressure;
92     in_temp: in event data port temp;
93    end Service_thread;
94
95    thread implementation Service_thread.impl
96    properties
97        Dispatch_Protocol => aperiodic;
98        Compute_Execution_Time => 30 Ms .. 40 Ms;
99        Period => 1 sec;
100   annex behavior_specification {**
101       states
102         s0: initial complete state;
103       transitions
104         s0 -[ ]-> s0 {press_event!(1);};
105     **};
106   end Service_thread.impl;
107
108   memory PMS_memory
109   features
110     bus1: requires bus access bus_gl.impl;
111   end  PMS_memory;
112
```

```
113   memory implementation PMS_memory.impl
114   end PMS_memory.impl;
115
116   processor  PMS_processor
117   features
118    busAcc1: requires bus access bus_gl;
119   end  PMS_processor;
120
121   processor implementation PMS_processor.impl
122   subcomponents
123   mem : memory PMS_memory.impl;
124   end PMS_processor.impl;
125
126   -- Nurse system
127   system Nurse_Sys
128   features
129     in_alarm : in out event port;
130     out_answer : in out event port;
131   end Nurse_Sys;
132
133   system implementation Nurse_Sys.impl
134   subcomponents
135     Nur_pro: process Nurse_process.impl;
136     Nur_mem: memory PMS_memory.impl;
137     Nur_proc: processor PMS_processor.impl;
138     Nur_bus: bus bus_gl;
139   connections
140      cnx20: event port in_alarm -> Nur_pro.in_alarm;
141      cnx21: event port Nur_pro.out_answer -> out_answer;
142   properties
143     Actual_Memory_Binding =>  reference Nur_mem
144         applies to Nur_pro;
145     Actual_Processor_Binding =>  reference Nur_proc
146         applies to Nur_pro;
147     Actual_Connection_Binding =>  reference Nur_bus
148         applies to Nur_pro;
149   end Nurse_Sys.impl;
150
151   process Nurse_process
152   features
153    in_alarm : in out event port;
154    out_answer : in out event port;
155   end Nurse_process;
156
157   process implementation Nurse_process.impl
158   subcomponents
159    Nur_thr: thread Nurse_thread.impl;
160   connections
161      cnx22: event port in_alarm -> Nur_thr.in_alarm;
162      cnx23: event port Nur_thr.out_answer-> out_answer;
163   end Nurse_process.impl;
164
165
166   thread Nurse_thread
167   features
168    in_alarm : in event port;
```

```
169    out_answer : out event port;
170   end Nurse_thread;
171
172   thread implementation Nurse_thread.impl
173   properties
174       Dispatch_Protocol => aperiodic;
175       Period => 1 sec;
176       Compute_Execution_Time => 30 Ms .. 40 Ms;
177   end Nurse_thread.impl;
178
179   -- Controller
180   device Control
181   features
182     d1: requires bus access bus_gl.impl;
183     out_pressure: out event data port pressure;
184     out_temp: out event data port temp;
185   end Control;
186   device implementation Control.impl
187   --properties
188   end Control.impl;
189
190   bus bus_gl
191   end bus_gl;
192
193   bus implementation bus_gl.impl
194   properties
195     Propagation_Delay => 5 Ms..5 Ms;
196     Transmission_Time => 6 Ms..6 Ms;
197   end bus_gl.impl;
```

A.2 Spécification AADL complète du Système PMS Dynamique

```
system global
end global;

system implementation global.impl
subcomponents
   Service_Event : system Service_Sys.impl;
   Nurse  : system Nurse_Sys.impl;
   Controller: device control.impl;
   Bus_global: bus bus_gl.impl;
   Bus_globalW:bus  bus_gl.implw;
connections
   cnx0: event port Service_Event.out_alarm -> Nurse.in_alarm;
   cnx1: event port Nurse.out_answer -> Service_Event.in_answer;
   cnx2: event data port Controller.out_pressure ->

Service_Event.in_pressure;
   cnx3: event data port Controller.out_temp ->
Service_Event.in_temp;
   cnx0w: event port Service_Event.out_alarmw -> Nurse.in_alarmw;
   cnx1w: event port Nurse.out_answerw -> Service_Event.in_answerw;
   cnxw: event port Nurse.wifi_event_out ->
Service_Event.wifi_event;
modes
   wifi_off: initial mode;
   wifi_on: mode;
   wifi_off-[Nurse.wifi_event_out]->wifi_on;
   wifi_on-[Nurse.wifi_event_out]->wifi_off;
properties
  -- wire mode
   Actual_Connection_Binding =>  reference Bus_global
   applies to cnx0  in modes (wifi_off);
   Actual_Connection_Binding =>  reference Bus_global
   applies to cnx1 in modes (wifi_off);
   Actual_Connection_Binding =>  reference Bus_global
   applies to cnx2 in modes (wifi_off);
   Actual_Connection_Binding => reference Bus_global
   applies to cnx3 in modes (wifi_off);
   Actual_Connection_Binding =>reference Bus_global
   applies to Service_Event.Ser_pro in modes (wifi_off);
   Actual_Connection_Binding =>reference Bus_global
   applies to Nurse.Nur_pro in modes (wifi_off);
   --wifi mode
   Actual_Connection_Binding =>reference Bus_globalW
   applies to cnx0w in modes (wifi_on);
   Actual_Connection_Binding =>reference Bus_globalW
   applies to cnx1w in modes (wifi_on);
   Actual_Connection_Binding =>reference Bus_globalW
   applies to cnxw in modes (wifi_on);
   Actual_Connection_Binding =>reference Bus_globalW
   applies to Service_Event.Ser_pro in modes (wifi_on);
   Actual_Connection_Binding =>reference Bus_globalW
   applies to Nurse.Nur_pro in modes (wifi_on);
end global.impl;

-- data
data temp
```

```
57   end temp;
58   data pressure
59   end pressure;
60   data message
61   end message;
62
63   -- All declartaions of Service Event system compoments
64   system Service_Sys
65   features
66    out_alarm : in out event port ;
67    in_answer : in out event port ;
68    out_alarmw : in out event port ;
69    in_answerw : in out event port ;
70    wifi_event: in event port;
71    in_pressure: in out event data port pressure;
72    in_temp: in out event data port temp;
73    end Service_Sys;
74
75   system implementation Service_Sys.impl
76   subcomponents
77     Ser_pro: process Service_process.impl;
78     Ser_mem: memory PMS_memory.impl;
79     Ser_proc: processor PMS_processor.impl;
80     Ser_bus: bus bus_gl.impl;
81   connections
82     cnx10: event port Ser_pro.out_alarm -> out_alarm
83     in modes (wifi_off);
84     cnx11: event port in_answer -> Ser_pro.in_answer
85     in modes (wifi_off);
86     cnx10w: event port Ser_pro.out_alarm -> out_alarmw
87     in modes (wifi_on);
88     cnx11w: event port in_answerw->Ser_pro.in_answer
89     in modes (wifi_on);
90     cnx12: event data port in_pressure->Ser_pro.in_pressure;
91     cnx13: event data port in_temp->Ser_pro.in_temp;
92   modes
93      wifi_off: initial mode;
94      wifi_on: mode;
95      wifi_off-[wifi_event]->wifi_on;
96      wifi_on-[wifi_event]->wifi_off;
97   properties
98     Actual_Memory_Binding =>  reference Ser_mem
99     applies to Ser_pro;
100    Actual_Processor_Binding =>  reference Ser_proc
101    applies to Ser_pro;
102    Actual_Connection_Binding => reference Ser_bus
103    applies to Ser_pro;
104  end Service_Sys.impl;
105
106  process Service_process
107  features
108   out_alarm : in out event port;
109   in_answer: in out event port;
110   in_pressure: in out event data port pressure;
111   in_temp: in out event data port temp;
112  end Service_process;
```

```
113
114  process implementation Service_process.impl
115  subcomponents
116   Ser_thr: thread Service_thread.impl;
117  connections
118     cnx14: event port Ser_thr.out_alarm -> out_alarm;
119     cnx15: event port in_answer -> Ser_thr.in_answer;
120     cnx16: event data port in_pressure->Ser_thr.in_pressure;
121     cnx17: event data port in_temp -> Ser_thr.in_temp;
122   end Service_process.impl;
123
124  thread Service_thread
125  features
126   out_alarm : out event port;
127   in_answer : in event port;
128   in_pressure: in event data port pressure;
129   in_temp: in event data port temp;
130  end Service_thread;
131
132  thread implementation Service_thread.impl
133  properties
134      Dispatch_Protocol => aperiodic;
135      Compute_Execution_Time => 30 Ms .. 40 Ms;
136      Period => 1 sec;
137  annex behavior_specification {**
138      states
139        s0: initial complete state;
140      transitions
141        s0 -[ ]-> s0 {press_event!(1);};
142    **};
143  end Service_thread.impl;
144
145  memory PMS_memory
146  features
147    bus1: requires bus access bus_gl.impl;
148  end  PMS_memory;
149
150  memory implementation PMS_memory.impl
151  end PMS_memory.impl;
152
153  processor  PMS_processor
154  features
155   busAcc1: requires bus access bus_gl.impl;
156  end  PMS_processor;
157
158  processor implementation PMS_processor.impl
159  subcomponents
160  mem : memory PMS_memory.impl;
161  modes
162  wifi_off: initial mode;
163  wifi_on: mode;
164  end PMS_processor.impl;
165
166  -- Nurse system
167  system Nurse_Sys
168  features
```

```
169     in_alarm : in out event port;
170     out_answer : in out event port;
171     in_alarmw : in out event port;
172     out_answerw : in out event port;
173     wifi_event_in : in  event port;
174     wifi_event_out: out event port;
175   end Nurse_Sys;
176
177   system implementation Nurse_Sys.impl
178   subcomponents
179     Nur_pro: process Nurse_process.impl;
180     Nur_mem: memory PMS_memory.impl;
181     Nur_proc: processor PMS_processor.impl;
182     Nur_bus: bus bus_gl.impl in modes (Nur_wifi_off);
183     Nur_bus_W: bus bus_gl.implW in modes (Nur_wifi_on);
184     Mode_W: device wifi_mode.impl ;
185   connections
186      cnx20: event port in_alarm -> Nur_pro.in_alarm
187         in modes (Nur_wifi_off);
188      cnx21: event port Nur_pro.out_answer -> out_answer
189         in modes (Nur_wifi_off);
190      cnx20w: event port in_alarmw -> Nur_pro.in_alarm
191         in modes (Nur_wifi_on);
192      cnx21w: event port Nur_pro.out_answer -> out_answerw
193         in modes (Nur_wifi_on);
194      cnx22w: event port Mode_W.wifi_on_off -> Nur_pro.wifi_event;
195   modes
196     Nur_wifi_off: initial mode;
197     Nur_wifi_on: mode;
198     Nur_wifi_off-[wifi_event_in]->Nur_wifi_on;
199     Nur_wifi_on-[wifi_event_in]->Nur_wifi_off;
200   properties
201     Actual_Memory_Binding =>  reference Nur_mem
202         applies to Nur_pro;
203     Actual_Processor_Binding =>  reference Nur_proc
204         applies to Nur_pro;
205     Actual_Connection_Binding =>  reference Nur_bus
206         applies to Nur_pro in modes (Nur_wifi_off);
207     Actual_Connection_Binding =>  reference Nur_bus_w
208         applies to Nur_pro in modes (Nur_wifi_on);
209   end Nurse_Sys.impl;
210
211   process Nurse_process
212   features
213    in_alarm : in out event port;
214    out_answer : in out event port;
215    wifi_event: in event port;
216   end Nurse_process;
217
218   process implementation Nurse_process.impl
219   subcomponents
220    Nur_thr: thread Nurse_thread.impl;
221   connections
222      cnx22: event port in_alarm -> Nur_thr.in_alarm;
223      cnx23: event port Nur_thr.out_answer-> out_answer;
224   end Nurse_process.impl;
```

```
225
226   thread Nurse_thread
227   features
228    in_alarm :   in event port;
229    out_answer : out event port;
230   end Nurse_thread;
231
232   thread implementation Nurse_thread.impl
233   properties
234       Dispatch_Protocol => aperiodic;
235       Period => 1 sec;
236       Compute_Execution_Time => 30 Ms .. 40 Ms;
237   end Nurse_thread.impl;
238
239   device wifi_mode
240   features
241   wifi_on_off: out event port;
242   end wifi_mode;
243
244   device implementation wifi_mode.impl
245   end wifi_mode.impl;
246
247   -- Controller
248   device Control
249   features
250     d1: requires bus access bus_gl.impl;
251     out_pressure: out event data port pressure;
252     out_temp: out event data port temp;
253   end Control;
254   device implementation Control.impl
255   --properties
256   end Control.impl;
257
258   bus bus_gl
259   end bus_gl;
260
261   bus implementation bus_gl.impl
262   properties
263     Propagation_Delay => 5 Ms..5 Ms;
264     Transmission_Time => 6 Ms..6 Ms;
265   end bus_gl.impl;
266
267   bus implementation bus_gl.implW
268   --  ce bus est censé représenter le réseau wifi ou un point d'accès
269   properties
270     Propagation_Delay => 5 Ms..5 Ms;
271     Transmission_Time => 6 Ms..6 Ms;
272   end bus_gl.implW;
273
```

Titre Définition d'un Modèle de Déploiement d'une Architecture Logicielle à Base de BRS

Résumé Le déploiement constitue une phase importante dans le cycle de vie d'un logiciel, souvent construit de façon ad-hoc. Les préoccupations des architectes actuels s'articulent autour de la définition d'un processus de déploiement générique et formel permettant d'assembler et de distribuer correctement des applications logicielles quelle que soit leur technologie d'implémentation et facilitant en même temps l'analyse et la vérification. Nous montrons dans ce travail la pertinence des systèmes réactifs bigraphiques (BRS) pour la description formelle du processus de déploiement d'une architecture logicielle dans un environnement cible. Le modèle formel proposé permet en particulier de définir d'une part, les deux structures d'une architecture à savoir la plateforme et le scénario de l'application et d'autre part, l'opération d'installation correspondante. Nous projetons cette formalisation sur une architecture spécifiée en langage AADL. Nous montrons également la capacité du modèle proposé, une fois enrichi par des règles de réaction associées à un bigraphe, à prendre en charge la dynamique et la reconfiguration d'une spécification architecturale, en particulier celles décrites en AADL. Ce travail est clôturé par la proposition d'un environnement sous Eclipse à base du langage Maude. Il prend en charge la spécification d'une architecture en AADL, puis la transforme en modèle bigraphique qui sera ensuite vérifié par le model-checker de Maude pour en déduire un système exécutable.

Mots-clés Architecture Logicielle, Déploiement, Bigraphes, BRS, AADL

Title Definition of a Model based BRS for Software Architecture Deployment

Abstract The deployment is an important phase in the lifecycle of software, often built in an ad-hoc way. The concerns of software architects are centered on the definition of a generic deployment process to correctly assemble and distribute software applications regardless of their implementation technology. We show in this work the relevance of bigraphical reactive systems (BRS) for the formal description of the deployment process of a software application in a target environment. The proposed formal model allows in particular to define on the first hand, the two structures specified in AADL architecture namely the platform and the application scenario and in the other hand, the corresponding installation operation, usually described in this language by a set of properties. We also show the ability of the proposed model, once enriched by reaction rules associated with a bigraph, to support dynamics and reconfiguration of AADL specifications. This work ended with the proposal of an environment based on Maude language. It supports an architecture specification in AADL, then transforms it into bigraphique model which will then be checked by the Maude model checker to derive an executable system.

Keywords Software Architecture, Deployement, Bigraphs, BRS, AADL

Ce document a été préparé à l'aide de l'éditeur de texte TexMaker et du logiciel de composition typographique LaTeX 2_ε.