Modélisation @Runtime des systèmes à base de composants

Wafa GABSI

École Nationale d'Ingénieurs de Sfax
Tunisie

6 août 2011

A mes parents,
A mon mari,
A ma petite fille,
A ma sœur et mes frères,
A mon beau père et ma belle mère,
A mes amies Ikbel et Sihem,
A tous ceux que je n'ai pas cités
et qui ne me sont pas moins chers.

Remerciements

C'est avec un grand plaisir que je réserve cette page en signe de gratitude et de reconnaissance à tous ceux qui ont assisté ce travail.

Je veux tout d'abord remercier mes encadreurs M. Mohamed JMAIEL, professeur à l'École Nationale d'Ingénieurs de Sfax (ENIS), M. Bechir ZALILA, assistant à l'ENIS, et M. Slim KALLEL, assistant à la Faculté des Sciences Économiques et de Gestion de Sfax (FSEGS), qui n'ont pas épargné le moindre effort dans leurs encadrements de ce projet. C'est grâce à leur aide précieux et leurs conseils que j'ai pu achever ce travail. Qu'ils trouvent dans ce mémoire le témoignage de ma profonde reconnaissance.

Je tiens à témoigner ma gratitude et mes vifs remerciements à M. Khalil DRIRA, directeur de recherche au *Laboratoire d'Analyse et d'Architecture des Systèmes* de Toulouse (LAAS-CNRS), pour avoir accepté de présider le comité d'examen. Mes remerciements sont aussi adressés à M. Ahmed HADJ KACEM, professeur à la FSEGS, pour avoir accepté d'être membre de la commission d'examen.

Je remercie également tous les membres de l'unité de Recherche en Développement et Contrôle d'Applications Distribuées pour l'ambiance amicale qu'ils m'ont réservée.

Table des matières

Table des figures

Liste des tableaux

Listings

Introduction Générale

De nos jours, nous vivons une évolution rapide dans le domaine du développement logiciel. Chacun de sa manière, essaie d'améliorer la qualité de production durant le cycle de développement logiciel en passant par toutes les étapes visant à atteindre les besoins et les objectifs des utilisateurs. Dans ce contexte, un système logiciel distribué embarqué à base de composants doit être capable de s'adapter aux modifications causées soit par l'environnement extérieur soit par l'utilisateur lui-même. Cette adaptation est accomplie dans le but de traduire des nouveaux besoins afin de garantir la fiabilité du système. Ces modifications peuvent être engendrées au niveau du modèle ou au niveau de l'exécution.

En effet, un modèle est une représentation abstraite du système qui doit être liée avec lui afin de décrire son état et son comportement instantanément. Le modèle doit offrir des informations exactes concernant le système afin de guider à la bonne décision d'adaptation. Pour un système temps réel embarqué à base de composants, on s'attend toujours à des changements imprévus non seulement au niveau de l'exécution mais aussi au niveau du modèle. Ces changements inattendus rendent nécessaire une correspondance bidirectionnelle entre les deux : si le système change, le modèle doit changer et vice versa. Suite à un manque de correspondance entre les deux, on peut détecter un mal fonctionnement ou une mauvaise gestion du système. Ceci est réellement du à l'absence de surveillance continue de l'application aux niveaux conceptuel et exécutif pour assurer une adaptation dynamique instantanée.

Pour un processus de développement classique, le concepteur commence par traduire les besoins de l'utilisateur dans un modèle pendant la phase de conception. Ensuite, il passe à la phase d'implantation dont la sortie est un code conforme au modèle. Ce code va être ensuite déployé et exécuté pour avoir finalement une application qui tourne sur une plateforme donnée. Une modification au niveau du modèle ne peut pas être projetée directement sur l'exécution. Elle entraîne la répétition de tout le reste du processus ce qui rend l'opération coûteuse en terme de temps et ne garantit pas la bonne correspondance à tout moment et surtout au cours de l'exécution. De l'autre côté, un utilisateur au niveau de l'exécution n'admet pas une vue abstraite des composants que comporte le système ce qui mène parfois à une gestion risquée de l'application et surtout dans le cas des systèmes fortement dynamiques. Dans ce cas, des instances de composants ou de connexions sont souvent ajoutées, modifiées ou supprimées au niveau de l'exécution mais pas au niveau du modèle. Ce qui nécessite l'adaptation dynamique dans les deux sens.

Pour assurer l'adaptation du système d'une manière manuelle ou automatique on doit également le surveiller. Cette surveillance doit être établie sur les deux niveaux conceptuel et exécutif permettant ainsi de suivre le système tout au long de

son évolution. La surveillance et l'adaptation dynamique des systèmes distribués à base de composants présentent aujourd'hui un défi pour le développement logiciel et spécialement l'adaptation au cours de l'exécution.

Une solution, envisagée afin d'assurer la surveillance et l'adaptation dynamique, est la modélisation en cours d'exécution (**Models@Runtime**) [BBF09]. Cette technique consiste à étendre les techniques de l'ingénierie dirigée par les modèles (*Model Driven Engineering*) en considérant une application en cours d'exécution. D'une part, elle permet de modéliser l'application avec un certain niveau d'abstraction. D'autre part, elle permet de la surveiller en cours de son exécution. Il s'agit d'assurer l'adaptation dynamique dans deux sens. Le premier sens, de l'exécution vers le modèle : toute nouvelle configuration atteinte par le système en cours d'exécution sera projetée sur le modèle. Le deuxième sens, du modèle vers l'exécution : tout changement subi par un composant du modèle sera traduit dans l'exécution du système. Dans ce cadre, certaines approches se concentrent sur l'adaptation dans le premier sens comme le traçage en ligne de l'exécution sous forme de graphe permettant de suivre le cycle de vie des éléments logiciels ou l'intégration de service au niveau du modèle permettant de prévoir si l'adaptation mène à un état risqué ou non de l'application. Il existe d'autres approches qui traitent l'adaptation dans les deux sens à savoir l'utilisation du modèle de conception pour l'adaptation en cours d'exécution ou l'adaptation des lignes de production dynamiques en utilisant un Framework orienté modèle et aspect.

La programmation orientée aspect (*Aspect Oriented Programming*) [KLM+97] est un paradigme qui permet de séparer le code fonctionnel d'une application (correspondant aux préoccupations fonctionnelles cela veut dire les exigences fonctionnelles fixées par l'utilisateur dans son cahier de charge) du code des aspects qui couvre les préoccupations transversales (besoins non fonctionnels dans le but d'amélioration de la qualité du code, de la QoS, de la sécurité...).

Les langages de description d'architecture (*Architecture Description Languages, ADLs*) [MT00], présentent un support pour la description de la topologie de l'application. Ils décrient l'ensemble des composants que comporte le système et les différentes liaisons et connections entre eux. Ils permettent ainsi à l'utilisateur de structurer les différents éléments que comporte le système. Et ceci dans le but de simplifier la gestion de ce système et d'offrir une présentation plus claire et facile à interpréter. Parmi ces langages, on cite *AADL* (*Architecture Analysis & Design Language*) [SAE04, MT00, FGH06]. Ce langage permet de décrire tout un système en intégrant les deux parties logicielle et matérielle. Ainsi il peut refléter l'état détaillé de l'exécution à un certain niveau d'abstraction.

Dans le cadre de ce projet de mastère, nous envisageons de combiner l'utilisation de la programmation orientée aspect et les langages de description d'architecture pour assurer la modélisation au cours de l'exécution des systèmes à base de composants. Nous visons à assurer la surveillance du système tout au long de son évolution dans les deux sens sans repasser ni par la phase d'implantation ni par la phase de déploiement. Tout changement au niveau du modèle sera traduit dans l'exécution et toute modification au niveau de l'exécution sera projetée dans le modèle. Nous cherchons à garantir une mise à jour dynamique instantanée des modifications sur les deux niveaux. Notre approche consiste à définir un processus complet assurant

l'adaptation dynamique des systèmes à base de composants en cours d'exécution.

Partant de la phase de conception, le concepteur décrit son modèle avec AADL comme langage de description d'architecture. Puis nous passons à la phase d'implantation en utilisant un générateur de code. Ainsi, nous avons le code du système à contrôler. Enfin, une adaptation peut être déclenchée au cours de la surveillance et donc effectuée dans l'un des deux sens.

Sur le plan pratique, nous développons un éditeur graphique que l'on appelle « *AADL Graphical Editor* » permettant une description architecturale avec AADL. Nous implantons aussi un protocole de communication permettant l'échange de messages au sein d'une application distribuée. Ces messages constituent en fait des traces décrivant les changements au niveau du modèle ainsi qu'au niveau de l'exécution. Afin de valider notre approche, nous considérons un cas d'étude d'un guichet automatique bancaire (GAB). Nous décrivons ce système avec AADL en utilisant notre éditeur. Puis nous mettons en œuvre le processus d'adaptation en l'appliquant sur ce système.

La suite de ce mémoire est organisée comme suit : le chapitre 1 est consacré à la présentation du contexte général de notre travail ainsi que la technique **Models@Runtime**. Le chapitre 2 présente les différentes approches de modélisation au cours de l'exécution existantes suivi d'une synthèse selon différents critères. Le chapitre 3 présente notre approche proposée. Le chapitre 4 montre la mise en œuvre de notre approche et le chapitre 5 met en évidence l'étude de cas utilisée pour la valider. Enfin, nous terminons ce rapport par une conclusion et quelques perspectives.

Chapitre 1

Contexte général

1.1 Introduction

En face de l'évolution importante connue par les systèmes informatiques, l'innovation d'une technologie en vue d'amélioration des propriétés et des performances d'un système logiciel devient un travail fondamental. Dans ce contexte, il y a eu l'apparition de la discipline de l'architecture logicielle (*Software Architecture*) [CCMC96, PW92] permettant de simplifier et d'aider à concevoir et développer des systèmes complexes. Ces architectures continuent toujours à progresser dans le but de répondre aux besoins croissants dans ce domaine. Le concept de développement orienté aspect au niveau architectural (*Aspect Oriented Software Development, AOSD*) [FECA05] est devenu lié à la notion d'architecture logicielle dans le but de faciliter la conception, de réduire les coûts de la maintenance et d'améliorer la qualité des produits logiciels. Les langages de description d'architecture mettent en faveur la pratique de l'architecture logicielle tout au long du cycle du développement logiciel. C'est à la base de ces langages qu'on a introduit le nouveau concept de **modélisation @Runtime** . Cette nouvelle technologie étend les techniques de l'ingénierie dirigée par les modèles (*MDE*) en considérant une application au cours de l'exécution. Notre but dans ce projet de mastère est d'utiliser la modélisation @Runtime pour établir un processus complet d'adaptation dynamique des systèmes à base de composants.

Dans ce chapitre, nous plaçons notre projet de mastère dans son cadre général. D'abord, nous commençons par définir l'architecture logicielle et présenter ses principaux défis. Ensuite nous présentons le concept des langages de description d'architecture en détaillant le langage AADL. Puis, nous passons à une description de l'architecture dirigée par les modèles. Par la suite, nous présentons le paradigme du développement orienté aspect des logiciels tout en définissant la programmation orientée aspect et en citant ses avantages. Après, nous introduisons le concept de la modélisation en cours d'exécution, nous étalons ses différents objectifs et nous citons ses principaux défis. Finalement, nous clôturons par une conclusion pour ce chapitre.

1.2 Architecture logicielle

1.2.1 Définition

L'architecture logicielle [CCMC96, PW92, KOS06] est une image globale reflétant la composition d'un système informatique en termes de composants et des différentes interactions entre eux. Elle décrit les comportements des composants que comporte le système en haut niveau d'abstraction sans prendre en compte les détails d'implantation. Elle décrit aussi l'interaction du système avec l'environnement extérieur à savoir l'environnement technologique. L'architecture logicielle doit donc spécifier comment un système informatique doit être organisé afin de répondre aux exigences et d'atteindre les objectifs. Cette discipline est mieux adaptée dans le cas des systèmes logiciels complexes permettant de les décomposer en un ensemble de composants élémentaires et de sous-composants et d'envisager les relations existantes entre eux. C'est au niveau conceptuel qu'elle intervient pour concevoir l'organisation du système. En outre, au niveau de développement, l'architecture logicielle peut concevoir l'implantation du système et les moyens d'adaptation aux changements. Ces changements consistent à la reconfiguration architecturale ou comportementale, la gestion d'erreurs de conception ou la traduction de nouveaux besoins.

Parmi les architectures logicielles, on cite par exemple, le calcul distribué, l'architecture trois tiers et l'architecture orientée service.

1.2.2 Défis de l'architecture logicielle

La description d'une architecture logicielle est une tâche assez complexe relativement à ses objectifs et ses défis. Elle vise à améliorer la qualité de production logicielle en partant d'une meilleure conception et en éliminant le taux d'erreurs de conception aussi bien que celles d'implantation. En effet, la description d'une architecture logicielle et la production du logiciel partagent un ensemble de défis :

- La modularité : la décomposition d'un système en un ensemble de composants et de connexions entre eux rend possible la répartition du développement d'une application sur plusieurs groupes de personnes. Ainsi, on peut séparer la mise en œuvre des composants et de leurs connexions associées.
- La réutilisation : le composant est une entité élémentaire sur laquelle est basée une description d'architecture logicielle. En effet, cette dernière regroupe un ensemble de composants interconnectés via leurs interfaces. Ces interfaces doivent être définies de façon à permettre l'utilisation de ce composant dans différentes applications.
- L'abstraction : l'architecture dirigée par les modèles (*MDA*) [MB08] se base sur l'utilisation des modèles tout au long du cycle du développement. Ces modèles sont définis avec un certain niveau d'abstraction afin de rester indépendants de toute plateforme et technologie.
- L'anticipation des changements : l'architecture des systèmes à base de composants peut fréquemment changer. Ce changement est du à des causes internes ou des raisons externes nécessitant l'adaptation de l'architecture suite aux nouveaux besoins. La description de l'architecture logicielle doit donc prévoir la possibilité d'évolution et préciser la variabilité du système.

– La construction incrémentale : la description d'une architecture logicielle doit fournir au concepteur la possibilité de construire son système d'une manière incrémentale en partant des préoccupations de base et en ajoutant au fur et à mesure des préoccupations techniques. Cependant, surmonter ce défi reste difficile à satisfaire par rapport aux langages de description d'architectures existants.

1.3 Langages de description d'architecture

L'architecture logicielle vise à décrire le système sous la forme d'un ensemble de composants interconnectés. Cette description est indépendante de la plateforme technique. Elle offre une représentation à un haut niveau d'abstraction. De plus, un composant peut contenir lui-même un assemblage de sous composants qui interagissent et qui allouent des ressources matérielles. Cette discipline rend la modélisation des systèmes logiciels plus complexe. Dés lors, la naissance du besoin d'un formalisme de description d'architectures possédant une syntaxe et une sémantique qui offre des abstractions et des mécanismes adaptés à la modélisation de l'architecture logicielle. Dans ce cadre, les langages de description d'architecture (*ADL : Architecture Description Language*) sont apparus pour couvrir ces besoins.

1.3.1 Définition

Les ADLs, proposées dans les années 90, forment une famille de langages dans le but de décrire une architecture logicielle. Un ADL admet ses propres caractéristiques, ses propres fonctions et ses objectifs distincts des autres. Parmi ces langages, on cite les plus connus : AADL et Fractal. Ces langages permettent de préciser l'architecture des différents composants d'un système et de décrire explicitement ses différentes relations à un certain niveau d'abstraction sans entrer dans les détails de la plateforme. Ils offrent également un support de modélisation mettant en faveur l'organisation des différentes entités par le concepteur. Les ADL sont généralement graphiques et textuelles disposant d'outils adjoints. Ils reposent sur un lexique approprié et commun pour tous les acteurs participant à la description architecturale comme les concepteurs, les développeurs et les testeurs. Ils ont l'avantage de faciliter la réutilisation des composants et le contrôle d'application et d'éliminer toute sorte d'ambigüité [MT00].

1.3.2 Concept d'ADL

Les composants, les connecteurs et les configurations représentent les concepts fondamentaux des ADLs. Dans la suite, nous décrivons chacun d'eux brièvement.

– `Composant` : une description architecturale est formée principalement d'un ensemble de composants. Un composant est une entité logicielle ou matérielle qui joue le rôle d'une unité de calcul ou d'un dépôt de données. Il possède un état et aussi un type qui relate son interface. Cette dernière, sert à son tour d'exprimer les liens d'un composant avec les autres. Un composant est soit simple soit composé. Dans ce cas, il est décrit comme un assemblage de sous

composants reliés les uns aux autres. Au niveau de l'implantation finale du système, un composant correspond à une unité de compilation [MT00].
– `Connecteur` : un connecteur est un élément architectural permettant explicitement de modéliser et d'illustrer la liaison d'un composant avec les autres tout en définissant les règles qui guident ses interactions.

Figure 1.1 – Composants et connecteurs [Zal08]

La figure 1.1 donne un exemple d'un connecteur qui relie deux composants. Un composant peut interagir avec le milieu externe uniquement à travers ses interfaces et des connecteurs.
L'interface du composant offre des services nécessaires pour son fonctionnement. Ces services se manifestent par des variables, des opérations ou des messages. Concernant la définition de l'interface, tout dépend de l'ADL choisi. Parmi les ADLs, il existe certains qui considèrent chaque point de l'interface comme un port, et d'autres qui estime un port en tant que toute l'interface. Le connecteur peut décrire différents types d'interactions. On peut les classer sous deux catégories. Les interactions simples telles que l'accès à une variable partagée ou l'appel de procédure et celles complexes à savoir l'échange de données sous forme de flux [MT00].
– `Configuration` : une configuration est constituée d'un ensemble de composants et de connecteurs qui interagissent. En effet, on peut dire qu'une configuration reflète un état d'une description architecturale du système. La définition de la configuration sert à vérifier si les différentes entités (composants et connecteurs) sont raisonnablement connectées. Elle permet de confirmer si le comportement du système souhaité est alors bien décrit [MT00].

1.3.3 Types d'ADLs

Il existe trois familles de langages de description d'architecture :

Les ADLs formels

Ce type d'ADLs, comme leur nom l'indique, permet de décrire formellement un système. Cette description théorique aide à l'analyse. Ces ADLs admettent les concepts de composants et de connecteurs d'une manière abstraite sans les lier à ses correspondants dans la pratique. Ces langages sont mal intégrés dans un processus de génération de code. Parmi les ADLs formels, nous citons ACME et Rapide.

Les ADLs concrets

Ce type d'ADLs reflète la nature mieux que les langages formels. Ils raffinent le concept de composant en établissant plusieurs séries correspondant chacune à

une variabilité logicielle ou matérielle du monde réel. Dés lors, Cette catégorie de langages est mieux adaptée pour la génération automatique de code à partir des modèles. AADL et UML[1] sont les ADLs concrets les plus connus.

Les ADLs restreints

Cette famille de langages permet une description de l'ensemble des composants logiciels sans mettre en faveur une sémantique opérationnelle forte. Par exemple, Fractal est un ADL restreint.

1.3.4 AADL

Initialement, AADL est un ADL développé pour répondre aux besoins des systèmes avioniques. Pour cette raison, l'acronyme signifiait « *Avionics Architecture Description Language* ». Puis il a été standardisé pour répondre aux besoins de tous systèmes temps-réel embarqués. Le nouvel acronyme est « *Architecture Analysis& Design Language* ». AADL est publié par la SAE (Society of Automotive Engineers) en deux versions. La première, AADL 1.0 est publiée en Octobre 2004 [SAE04] et la deuxième AADLV2 est publiée en Janvier 2009 [SAE09]. AADL est un langage de description d'architecture dédié principalement à l'analyse et la conception d'architecture logicielle et matérielle d'un système en se basant sur la notion de composants. Comme tout ADL, AADL permet de décrire ainsi un système comme un ensemble de composants interconnectés. C'est un langage très riche puisqu'il permet de décrire les éléments architecturaux ainsi que les éléments non architecturaux liés aux informations de performances critiques tels que les exigences temporelles.

De plus, AADL offre trois types de représentations. Il s'agit d'une représentation textuelle, graphique ou sous format XML (*eXtensible Markup Language*) comme l'indique la figure 1.2.

Un concepteur peut adapter l'une de ces représentations à son travail selon ses besoins. Nous nous intéressons dans notre travail à la représentation graphique aussi bien qu'à la description XML. Nous allons détailler l'utilisation de chacune d'elles dans le chapitre 3.

Afin de décrire une architecture logicielle et matérielle d'un système, AADL définit plusieurs catégories de composants, réparties en trois grandes familles : les composants logiciels, les composants matériels et les composants hybrides. Dans la suite de cette section, nous détaillons les catégories de composants AADL ainsi que les différents composants qu'ils contiennent.

Catégories de composants

◇ **Composants logiciels**

Ils définissent les éléments applicatifs de l'architecture. Ce sont les entités logicielles qui forment le système. Ils sont au nombre de cinq :

1. Unified Modeling Language, www.uml.org

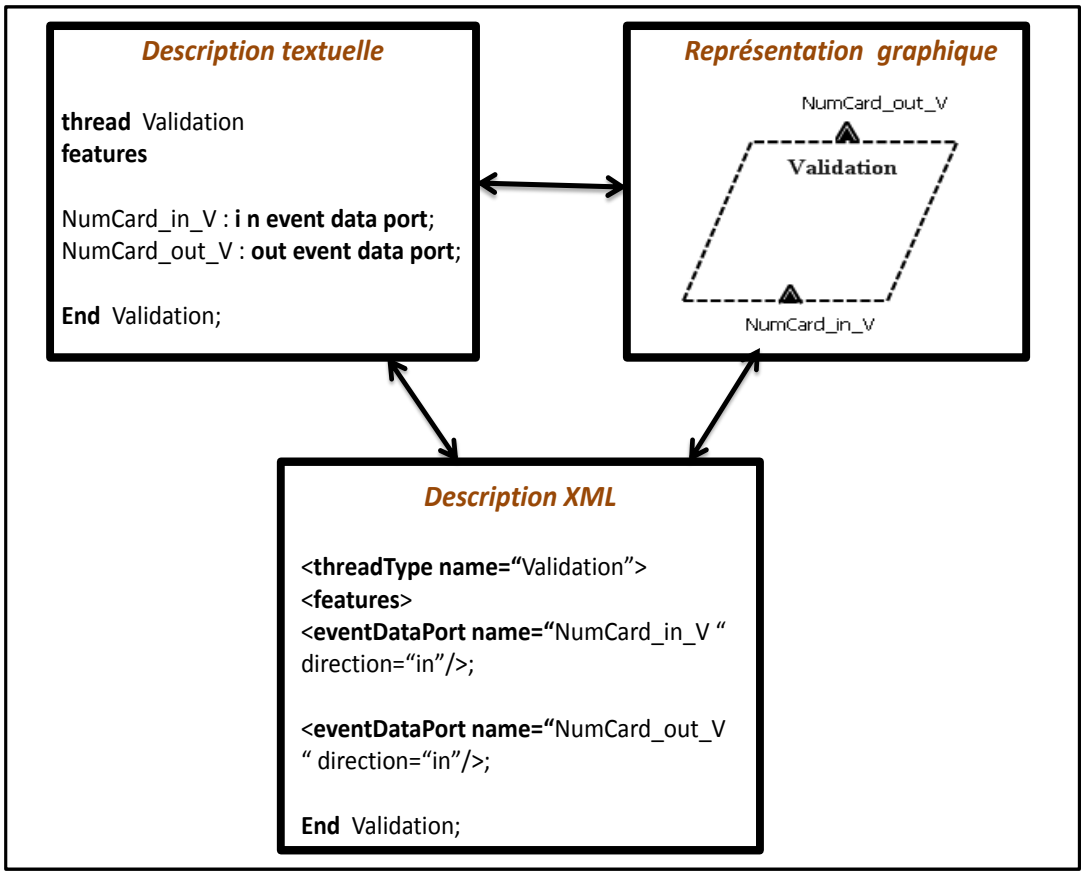

Figure 1.2 – Différentes représentations d'AADL

- `Process` : ce composant est utilisé pour modéliser les processus lourds de l'application. Un processus AADL est un espace mémoire dans lequel s'exécutent les processus légers (Threads) et qui sert à contenir les données (Data).
- `Thread` : les processus légers (tâches) sont modélisés grâce aux threads. Ces threads modélisent les fils d'exécution qui constituent la partie active de l'application.
- `Thread Group` : Dans le cas où de nombreux processus légers d'un système possèdent des caractéristiques proches, pour éviter la duplication de code, AADL introduit la notion de groupes de processus légers « thread group ». Ils décrivent des tâches partageant un nombre de propriétés.
- `Subprogram` : est utilisé pour modéliser les sous-programmes. Les sous-programmes représentent un fragment de code séquentiel exécutable, qui est appelé avec des paramètres.
- `Data` : les données représentent des types de données, lorsqu'elles sont déclarées sous la forme de composants. Elles représentent des variables partagées lorsqu'elles sont instanciées sous la forme de sous-composants. Ce composant permet de modéliser ces données.

◇ **Composants matériels**

Ils modélisent les éléments de la plate-forme d'exécution. Il existe quatre catégories de composants matériels :

- `Processor` : les processeurs sont décrits en utilisant le composant « processor ». Ce composant modélise un système d'exploitation. Les processeurs représentent des ensembles constitués d'un microprocesseur accompagné d'un ordonnanceur. Il exécute les fils d'exécution du système. L'unique catégorie de composant qu'il peut contenir est la catégorie des mémoires. De plus, il peut accéder à un bus via son interface.
- `Memory` : les mémoires sont modélisées avec le composant « memory ». Ce composant permet de représenter tous les dispositifs de stockage (disque dur, mémoire vive, etc). Les données et les programmes d'une application sont stockés dans des mémoires. Ces dernières sont accessibles par les processus légers (threads) en cours d'exécution.
- `Bus` : les bus sont modélisés par l'intermédiaire du composant « bus ». Ils doivent être liés aux processeurs « processors », mémoires« memories »et périphériques « devices »afin de transporter les informations entre ces composants.
- `Device` : les périphériques sont décrits à l'aide du composant « device ». Ils représentent l'environnement extérieur et modélisent une large variété de matériel (les capteurs, les appareils, etc). Un périphérique en AADL est considéré comme une boite noire puisque AADL ignore sa structure interne et ne permet pas de la décrire. Seulement l'interface et les caractéristiques externes du périphérique sont visibles par les autres composants.

◇ **Composants hybrides**

Ils permettent de regrouper différents composants en entités logiques pour structurer l'architecture. Un composant hybride est modélisé par l'intermédiaire du composant « `system` ». Les composants hybrides permettent de rassembler différents composants pour former des blocs logiques d'entités, ils facilitent ainsi la structuration de l'architecture. Contrairement aux autres catégories de composants, cette catégorie de composants ne représente pas une entité concrète. Le composant système représente alors un composant composite comme un ensemble de composants logiciels et matériels.

La figure 1.3 résume l'ensemble des composants d'AADL classés selon les différentes catégories.

Sous composants et appels

Un composant système constitue l'élément racine d'une description architecturale en AADL. Cette description est une arborescence d'instances de composants qui interagissent. La majorité de ses composants peut contenir à son tour un sous-composant qui représente une instance de la déclaration d'un composant. De cette façon, des composants contiennent d'autres sous-composants pour construire l'architecture de l'application. Par contre, il existe certains composants qui sont interdis de contenir des sous composants afin de respecter la logique de l'architecture de l'application. Le tableau 1.1, extrait du standard AADL 1.0 [SAE04] détaille les règles de contenance des sous-composants.

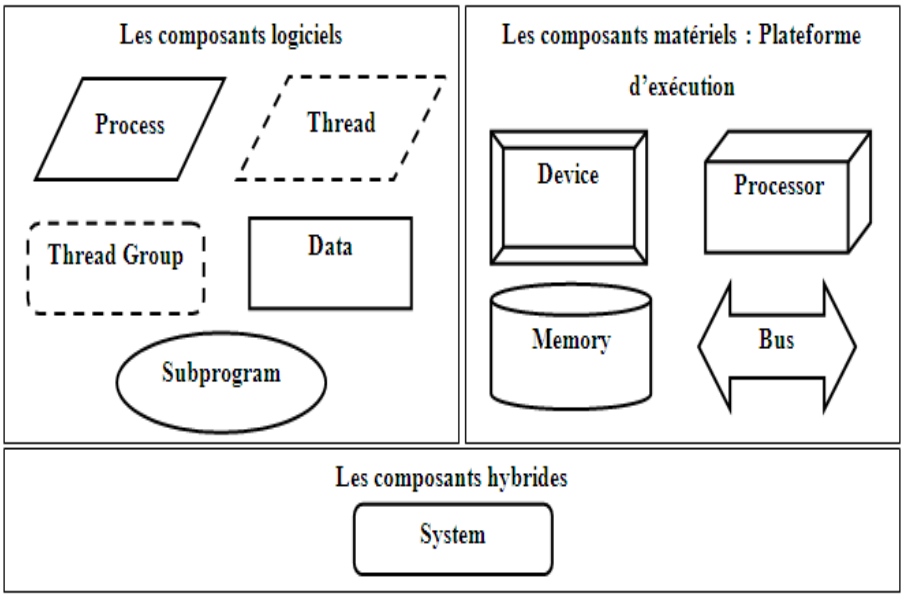

Figure 1.3 – Représentation graphique des composants AADL

Table 1.1 – Règles de contenance des sous-composants dans AADL 1.0 [SAE04]

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

L'implantation d'un processus léger ou d'un sous-programme peut contenir des séquences d'appels à d'autres sous-programmes décrivant ainsi un flot d'exécution.

Seules ces catégories de composants peuvent contenir de tels appels [Zal08]. Le listing 1.1illustre un exemple d'appel d'un sous-programme. Un thread `Task.impl1` appelle un sous programme `Hello_Spg_2`.

Listing 1.1 – Appel à un sous programme

```
1  subprogram Hello_Spg_2
2  end Hello_Spg_2;
3  thread Task
4  end Task;
5  thread implementation Task.impl_1
6  calls   {
7          P_Spg : subprogram Hello_Spg_1;
8          };
9  end Task.impl_1;
```

Interfaces et connexions

Les composants communiquent les uns avec les autres en connectant leurs éléments d'interface respectifs. Les connexions permettent de relier les interfaces des différents sous-composants à celles d'autres sous-composants ou aux interfaces du composant parent. La communication entre les différents composants est basée sur les éléments d'interfaces et les connexions entre eux. Réellement, les éléments d'interface permettent la description des flots de données et de contrôles entre composant. L'échange de données se fait à travers les ports de type donnée (`data port`) ou les paramètres des sous programmes. L'échange des signaux est effectué grâce aux ports de type événement (`event port`). L'échange des événements associés à des données est effectué par le biais des ports de type évènement donnée (`event data port`). La liaison entre deux ports établit une connexion. Ce type de connexion peut exister entre deux sous-composants ou entre un sous-composant et son composant parent.

Le listing 1.2 illustre une connexion entre un composant process `B.Impl` comportant un event data port d'entrée `In_Port` avec un de son composant fils `PingMe` de type thread comportant un event data port de sortie `Data_Sink`.

Listing 1.2 – Connexion entre un process et son sous-composant par le biais des ports

```
1  thread Q.Impl
2  features
3  Data_Sink: out event data port;
4  end Q.Ipml;
5  process implementation B.Impl
6  subcomponents
7  Ping_Me         : thread Q.Impl;
8  connections
9  event data port In_Port -> Ping_Me.Data_Sink;
10 end B.Impl;
```

Annexes et Propriétés

AADL est un langage extensible à travers les annexes et les propriétés permettant tous les deux d'enrichir une description architecturale avec des caractéristiques non

architecturales comme par exemple la nature d'un thread (périodique, sporadique et hybride). Dans la suite nous détaillons ces deux mécanismes.

◇ **Propriétés**

Il s'agit de caractéristiques associées à un composant donné. C'est un attribut permettant de spécifier des contraintes ou des caractéristiques s'appliquant aux entités constituant l'architecture. On peut par exemple fixer la période d'un processus léger ou une bande passante à un bus.

Le listing 1.3 illustre l'enrichissement d'un sous-programme Hello_Spg_1 par un ensemble de propriétés. Notons que ce sous-programme est implanté en C et que son implantation s'appelle user_Hello_Spg_1.

Listing 1.3 – Propriétés d'un sous-programme

```
1 subprogram Hello_Spg_1
2 properties
3 source_language => C;
4 source_name     => "user_Hello_Spg_1";
5 end Hello_Spg_1;
```

Ces propriétés sont standards et prédéfinies par le langage AADL qui supporte aussi la définition de propriétés spécifiques à une application donnée.

◇ **Annexes**

Les annexes permettent d'enrichir un modèle décrit avec le standard AADL avec des déclarations non architecturales exprimées avec un autre langage que AADL comme OCL ou AO4AADL.

Listing 1.4 – Utilisation d'annexes OCL dans AADL [SAE04]

```
1  data Sample
2  end Sample ;
3  thread Collect_Samples
4  features
5  Input_Sample : in data port Sample ;
6  Output_Average : out data port Sample ;
7  annex OCL {**
8             pre : 0 < Input_Sample < maxValue ;
9             post : 0 < Output_Sample < maxValue ;
10           **} ;
11 end Collect_Samples ;
```

Le listing 1.4 issu du standard AADL montre un processus léger qui contient une annexe pour le langage de contraintes OCL. L'annexe spécifie une pré-condition sur le port d'entrée du composant `Collect_Samples` et une post-condition sur la valeur de son port en sortie.

1.4 La démarche MDA

1.4.1 Définition

L'Architecture Dirigée par les Modèles (*Model Driven Architecture*), connue encore avec l'acronyme Ingénierie Dirigé par les Modèles *Model Driven Engineering, MDE*) [MB08], proposée par l'OMG [2], est une démarche qui se concentre sur la no-

2. Object Management Group, www.omg.org

tion de modèle. Heureusement, visant l'interopérabilité entre les différents systèmes, la réduction du coût de développement et l'augmentation de l'évolutivité, la MDA propose de séparer les spécifications fonctionnelles d'un système des spécifications liées à la plateforme d'implantation. La mise en œuvre de cette démarche est à base de modèle et de transformations entre différents modèles. Il s'agit de définir une architecture structurée à base de modèles indépendants de toutes plateformes, de système d'exploitation et même d'intergiciel. Puis d'effectuer des transformations successives jusqu'à arriver à une architecture spécifique à une plateforme donnée. Cette approche permet en effet d'assurer toutes les étapes de développement et de standardiser les transitions d'une étape à l'autre. Elle permet donc d'épargner l'effort effectué pendant les deux étapes d'analyse et de conception.

1.4.2 Mise en œuvre du MDA

L'approche MDA repose sur la définition de modèles et de transitions entre eux. On peut la décomposer en quatre étapes comme l'indique la figure 1.4 :

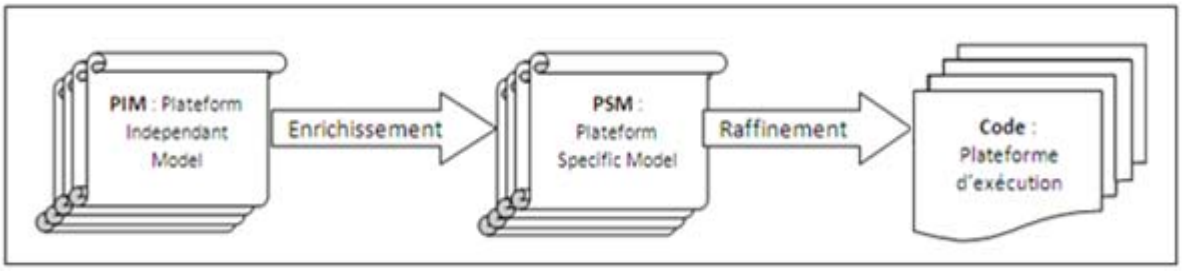

Figure 1.4 – Mise en œuvre de la démarche MDA

- Modèle indépendant de la plateforme : PIM
  La première étape réside dans la création d'un modèle indépendant de toute plateforme et technologie. Ce modèle, exprimé en UML, offre la description du logique métier et du comportement du système sans détailler son déploiement sur une plateforme. Il décrit le fonctionnement des différents composants. Il doit être clair, complet et correct dans le but de faciliter sa compréhension et sa validation par des experts du domaine. Il domine les deux phases d'analyse et de conception.
- Enrichissement
  La deuxième étape consiste à enrichir, détailler et filtrer le PIM par des informations non relatives à la plateforme. Donc le PIM doit être assez enrichi pour qu'on puisse le spécialiser vers une plateforme. On peut répéter cette étape un nombre indéterminé de fois afin d'améliorer le passage entre modèles.
- Modèle spécifique à la plateforme : PSM
  La troisième étape repose sur la sélection d'une plate-forme technique et la production du modèle spécifique correspondant PSM. Une fois que le PIM est suffisamment enrichi, on peut alors le transformer en PSM. Cette transformation est basée sur l'ajout de nouvelles informations liées à la plateforme. C'est

à ce niveau qu'on passe de la spécification indépendante à celle dépendante.
- Raffinement
  Il s'agit de raffiner le PSM jusqu'à obtenir une implantation exécutable. On cherche à générer le code adéquat spécifique à la plateforme choisie. Et ceci en ajoutant les informations propres à l'exécution et les données de configuration. Cette étape peut être répétée infiniment dans le but d'amélioration et de précision du PSM et donc du code généré.

C'est à la base de cette technique que le concept de modélisation @Runtime permet une modélisation avec différents niveaux d'abstraction.

1.5 Le développement orienté aspect de logiciel

La séparation des préoccupations et l'optimisation de la lisibilité de code deviennent des défis à surmonter auprès de la complexité des systèmes qui ne cesse à croître d'un jour à l'autre. Dans ce contexte, le développement orienté aspect des logiciels, (*Aspect Oriented Software Developement, AOSD*) [FECA05, NPMH02] est un paradigme qui assure la séparation des préoccupations tout au long du cycle de développement logiciel. En outre, ce paradigme permet aussi de faciliter la maintenance des logiciels.

1.5.1 Définition

L'AOSD est un paradigme qui cherche à rendre le processus d'évolution logicielle plus simple tout en séparant la programmation des différentes préoccupations. En effet, tout système admet des préoccupations fonctionnelles appelées aussi préoccupations métiers et d'autres non fonctionnels connues sous le nom préoccupations transversales ou encore techniques. La programmation orienté aspect(*POA*) [KIL+97] est un paradigme qui permet de séparer les deux préoccupations. Dans ce contexte la séparation des préoccupations vise à décrire le comportement d'une préoccupation transversale dans une unité modulaire autonome nommée « `Aspect` »en fournissant des techniques de séparation du code fonctionnel. Au début, l'AOSD était adapté uniquement pour la phase d'implantation. Mais après, il a été étendu pour couvrir toutes les phases du cycle de développement logiciel [FECA05].

1.5.2 Principe de la POA

Pour la programmation orientée aspect, on commence par différencier les préoccupations transversales (*crosscutting concern*) de celles métiers. Puis on procède à la programmation des préoccupations techniques en utilisant les aspects.

Un aspect est spécifié d'une façon autonome implantant un aspect technique particulier par exemple la génération de traces. Ensuite, un ensemble de points d'insertions (`Joinpoint`) sont définies pour établir la liaison entre le code de l'aspect et le code métier. L'ensemble de jointpoint définit un point de coupure (`Pointcut`). Par la suite, la préoccupation est mise en œuvre dans une portion de code appelé conseil (`Advice`). Le code de l'advice peut être exécuté avant, après ou autour d'un

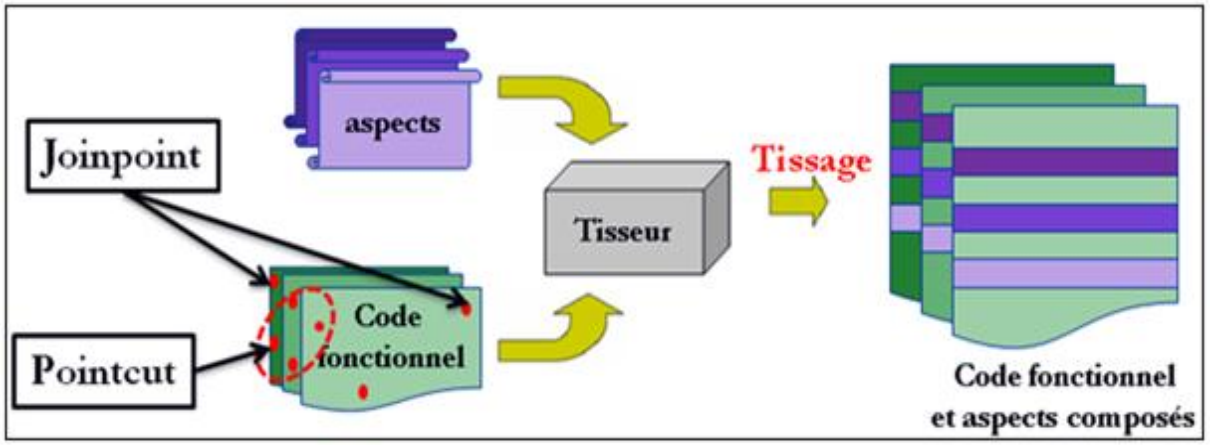

Figure 1.5 – Principe de la programmation orientée aspect

joinpoint. En fin, le code des aspects sera intégré finalement dans le code fonctionnel par l'intermédiaire de l'opération de tissage statique ou dynamique à travers le tisseur (`Weaver`).

1.5.3 Avantages

La programmation orientée aspect garantit une séparation entre les préoccupations fonctionnelles et les préoccupations transversales d'un système. C'est à la base de cette séparation que la POA offre plusieurs avantages :

- Un gain de productivité : un programmeur n'est chargé que du développement de l'aspect qui l'intéresse et donc son travail devient plus familier, commode et simple. Pour l'ensemble des programmeurs au sein d'une même équipe, cela les incite à en profiter pour bénéficier de la parallélisation de l'implantation.
- Meilleure réutilisation : puisque la POA met en faveur l'utilisation des modules implantant une seule préoccupation, donc elle rend la réutilisation d'un tel module plus facile et mieux adapté. Par exemple, un aspect édifiant la journalisation peut être intégré dans plusieurs applications sans modifier le code correspondant. De plus, ajouter une nouvelle préoccupation revient tout simplement à ajouter un nouvel aspect.
- Amélioration de la qualité du code : en séparant les préoccupations, on évite ainsi la duplication de code et donc on atteint une meilleure qualité du code qui est plus simple, lisible et compréhensible.
- Maintenance aisée : les aspects encapsulant des modules techniques peuvent être maintenus plus aisément puisqu'ils sont séparés du code métier.

1.6 Modélisation en cours d'exécution

Dans le cadre des avancements majeurs dans le domaine des systèmes logiciels, on cherche aujourd'hui à garantir une adaptation des modifications d'un système au cours de son exécution sans lui causer de risque de mal fonctionnement, de stabilité,

de fiabilité ou de sécurité. Cette adaptation peut être réalisée soit en ligne soit hors ligne. Il est déconseillé d'effectuer cette adaptation quand le système est hors ligne. A cet effet, il est né le besoin de mettre à jour le comportement du système en cours de son exécution sans ou avec une intervention humaine. Dans ce contexte, la **modélisation @Runtime** est introduite pour surmonter le défi d'adaptation en cours d'exécution.

1.6.1 Définition

La modélisation au cours de l'exécution(**Models@RunTime**) est une technique qui étend les techniques de l'ingénierie dirigée par les modèles tout en considérant une application en cours d'exécution. C'est un nouveau concept qui a été introduit pour la première fois en 2009 dans un article publié en IEEE Computer Society. Cet article définit ce concept comme suit :« a model@run.time is a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective » [BBF09].

D'après cette définition, un modèle@ Runtime peut jouer un double rôle. D'une part, ce modèle peut servir comme spécification du système qu'il représente. En effet, ces modèles permettent une modélisation des applications avec un certain niveau d'abstraction visant à suivre les détails du développement et de l'évolution du logiciel. D'autre part, ils supportent l'évolution dynamique. Ils ont la charge de surveiller, contrôler et gérer l'état et le comportement dynamique du système tout au long de son évolution sur les deux niveaux conceptuel et exécutif. A cet effet, cette technique permet de garantir le bon fonctionnement du système et de gérer tout type de reconfiguration architecturale ou comportementale.

1.6.2 Objectifs

Le nouveau concept de modélisation @Runtime est introduit dans le but de satisfaire certains besoins. Ces modèles permettent non seulement de modéliser l'application avec un certain niveau d'abstraction mais aussi de la surveiller en cours de son exécution. C'est pour cette raison, qu'on a pensé à utiliser cette technique dans le but d'intégrer sémantiquement des éléments hétérogènes dans un système en cours de son évolution et de participer à la génération automatique des ces objets qui seront insérés au niveau du système durant l'exécution ou à travers le système lui-même. Certains chercheurs souhaitent aussi fixer les erreurs de conception ou générer une nouvelle conception pour un système en cours d'exécution suite à la surveillance continue de l'application au cours de son exécution à travers la modélisation @Runtime. D'autres supposent d'autoriser l'adaptation d'un système par l'intervention humaine, à travers des agents embarqués dans le système lui-même, ou par la combinaison des deux en se basant sur cette technique de modélisation.

1.6.3 Défis

– La fiabilité : contrairement à la majorité des modèles utilisés dans l'analyse et la conception, ces modèles d'exécution sont généralement formels et clairs. Ce

couplage formel entre le modèle et le système qu'il représente est similaire à la relation entre les programmes décrits en haut niveau et son codage en langage machine. Par conséquent, ces modèles peuvent être très précis et les résultats de son analyse sont plus fiables que ceux d'un modèle logiciel.
- L'aide à la décision : le rôle de l'aide à la décision de la modélisation @Runtime rassemble à son rôle dans les deux phases d'analyse et de conception. Mais ces modèles peuvent être également interrogés à prédire les conséquences possibles des stratégies de réponse différemment à un évènement antérieur commis par l'un d'eux, grâce à l'efficacité des technologies modernes d'informatique.
- La gestion des situations inattendues : pour restreindre les configurations possibles qu'on peut atteindre et mettre en place des règles d'adaptation, on établie des méta-modèles qui définissent la structure et les comportements d'un système. Dans ce cas, on peut générer dynamiquement des nouvelles capacités pour répondre aux situations inattendues dans la conception de base en effectuant tout simplement des changements sur les méta-modèles.
- L'interprétation automatique : l'architecture des systèmes à base de composants peut fréquemment changer au cours de l'exécution suite à l'ajout de nouvelles instances de composants, l'ajout de connexions entre composants existants ou la suppression d'entités ou de connexions. Ces modifications doivent être étudiées et interprétées d'une façon automatique au cours de l'exécution pour répondre aux besoins d'adaptation dynamique. Afin de réaliser cette interprétation automatique, on doit mettre en place des méthodes et des standards de spécification sémantique convenables.
- Le passage à l'échelle (*scalability*) : les systèmes à grande échelle possèdent des composants hétérogènes et des éléments hautement distribués sur une grande échelle. Le concepteur d'un tel système est donc confronté à un ensemble de tâches difficiles. Un nouveau besoin repose sur le développement de technologies qui aident à gérer la complexité des données, la logique de prise de décision et le contrôle distribué des systèmes à grande échelle. Un autre besoin est d'être capable de gérer l'incertitude que tels systèmes peuvent poser.

1.7 Conclusion

Dans ce chapitre, nous avons commencé par introduire l'architecture logicielle. Ensuite nous avons définit les langages de description d'architecture et détaillé le langage AADL. Puis nous avons décrit la démarche MDA qui permet de définir une architecture structurée à base de modèles indépendamment de toute plateforme. Par la suite, nous avons présenté le paradigme AOSD. Ce concept permet la séparation des préoccupations et l'optimisation de la qualité de code. Ensuite, nous avons introduit un nouveau concept, c'est celui de la modélisation @Runtime. Cette technique est récemment introduite ayant confronté plusieurs défis visant à répondre aux nouveaux besoins croissants.

Dans le chapitre suivant, nous faisons une étude sur la technique modélisation @Runtime. Nous allons explorer la littérature pour pouvoir tirer profit des différentes approches qui travaillent autour de cette technique, les critiquer et proposer notre

nouvelle approche.

Chapitre 2

État de l'art

2.1 Introduction

Dans ce chapitre, nous étudions les travaux de recherche traitant la technique de modélisation @Runtime. Même si cette technique est récemment introduite, plusieurs chercheurs ont approfondis leurs travaux dans ce domaine. Cette technique est véritablement prometteuse pour la résolution des problèmes de reconfiguration dynamique au cours de l'exécution et des systèmes adaptatifs.

Dans ce chapitre nous nous intéressons à quelques travaux de recherche effectués dans ce domaine et qui sont les plus liés à notre approche. Nous commençons par introduire la technique de traçabilité, nous décrivons ses différentes étapes puis nous passons aux différentes représentations des traces. C'est à la base du mécanisme de trace que plusieurs approches détectent les changements des éléments objets d'adaptation. Parmi ces approches, on cite :

- L'utilisation des modèles à base de traces
- L'intégration d'un service au niveau du modèle
- L'utilisation d'un framework orienté modèle et orienté aspect pour l'adaptation des systèmes à base de composants
- L'utilisation du modèle de conception pour la modélisation @Runtime.

Nous détaillons chacune des ces approches puis nous clôturons par une synthèse générale et quelques objectifs.

2.2 Traçabilité

La traçabilité fournit la capacité de suivre le cycle de vie des éléments logiciels comme les objets et les threads. En effet, les traces sont capables d'apporter des informations sémantiques riches décrivant avec détails l'exécution du système. Et ceci dans le but de suivre l'état et le comportement du système tout au long de son évolution dans différents niveaux d'abstractions.

2.2.1 Définition

Même si, intuitivement, le concept de traçabilité est très clair, nous ne pouvons pas se contenter d'une définition tout à fait admise pour tous ceux qui sont profes-

sionnels dans ce domaine. Nous adoptons pour le reste de ce mémoire, la définition suivante : « La traçabilité est la capacité de décrire et de suivre le cycle de vie d'un objet et un moyen de modélisation des relations entre les entités logicielles d'une manière explicite » [WP09] . Cette définition est adaptée pour la trace d'exécution. Elle est appliquée dans le domaine de l'analyse des systèmes dynamiques.

2.2.2 Étapes de traçabilité

Pour mettre en place un mécanisme de trace dans un projet de développement logiciel, on doit suivre quatre étapes principales pour qu'on puisse se contenter du mécanisme de trace comme l'indique la figure 2.1.

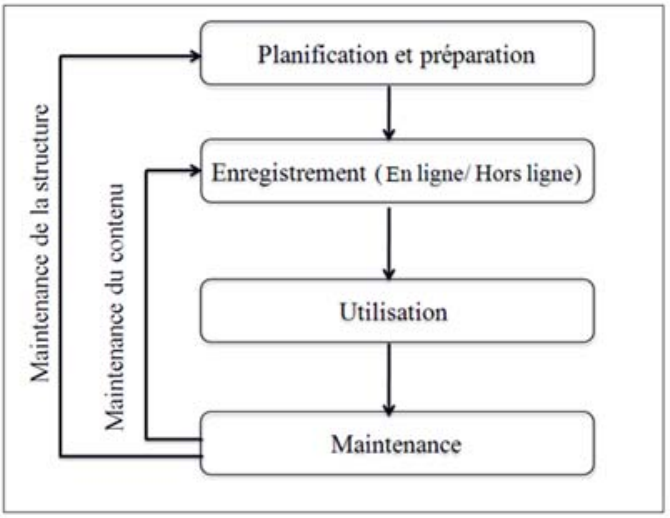

Figure 2.1 – Etapes de traçabilité [WP09]

Dans la suite, nous décrivons brièvement ces activités :

1. *Planification et préparation*
   Durant cette étape on choisit les outils et les méthodes de traçage. En effet, il existe une grande variété d'outils supportant différentes approches de traçage. A titre d'exemple, Echo est un outil utilisé avec le processus AGIL. SCEditor est aussi un outil permettant le traçage et possédant son propre éditeur, forme et interface.
2. *Enregistrement*
   Il s'agit de la sauvegarde des traces. Deux modes d'enregistrements sont envisagés :
   – Hors ligne : la sauvegarde des traces est effectuée après la terminaison de l'activité actuelle de développement d'une manière manuelle ou automatique.
   – En ligne : la sauvegarde est automatique et immédiate des traces.
3. *Utilisation*
   Les données que décrivent les traces sont accessibles afin de rédiger des rapports ou d'extraire des informations dans un but bien déterminé.

4. *Maintenance*
   C'est l'activité résultante d'un changement structurel du processus de développement ou une erreur ou oublie dans les données de traces. Deux cas se présentent :
   – Un changement ou ajustement au niveau des outils de traçage et donc on revient à la première étape.
   – Des erreurs qui doivent être corrigées au niveau de l'enregistrement.

2.2.3 Représentation des traces

Pour travailler avec les traces et pouvoir les utiliser, on doit les représenter correctement et lisiblement. Certaines approches utilisent typiquement le traçage au niveau code par des annotations. Il existe trois autres représentations :

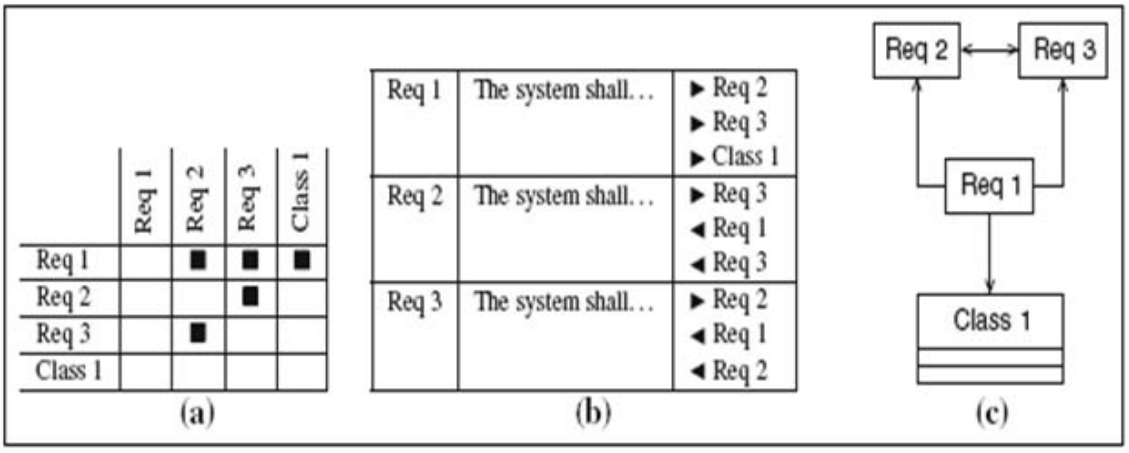

Figure 2.2 – Représentation des traces [WP09]

◇ **Matrice**

La première représentation des traces est sous la forme d'une matrice comme l'indique la figure 2.2(a). Les lignes et les colonnes représentent les objets sujets de traces (éléments de conception par exemple). La case remplie indique l'existence de lien entre l'objet à gauche et celui à droite. Au début, les matrices apportent une information simple (existence ou non d'un lien). Puis elles deviennent capables d'inclure d'autres informations enrichissant la trace par multiplication des dimensions. Dans le cas des projets réels énormes, ces matrices tendent vers des très grandes tailles ce qui les rend incompréhensibles et difficiles à interpréter.

◇ **Références croisées**

La deuxième représentation des traces est sous la forme de références croisées comme l'indique la figure 2.2(b). Les références croisées peuvent être intégrées et représentées en langage naturel.

L'avantage de cette représentation est qu'elle peut être interprétée par n'importe quel lecteur. Cependant, le regard des liens est très restreint puisqu'un utilisateur ne peut voir que les liens d'un seul élément à la fois.

◇ **Graphe**

La troisième représentation des traces est sous la forme de graphe comme l'indique la figure 2.2(c). C'est plus clair que les deux autres. En effet, l'interprétation des graphes est plus facile et apporte plus d'informations. Même pour les systèmes complexes, on s'assure que c'est toujours clair. Cette présentation des traces est la mieux adaptée dans le cas de l'ingénierie dirigée par les modèles.

Plusieurs approches sont proposées en vue de suivre un système en cours d'exécution et qui sont basées sur le mécanisme de trace à ce niveau. Nous nous servirons également de ce mécanisme pour la détection et l'enregistrement des modifications au niveau du modèle aussi bien qu'au niveau de l'exécution comme il est indiqué dans le chapitre 3. Nous présentons dans la suite quelques approches intéressantes et nous extrayons les apports et les limites.

2.3 Approches de modélisation au cours de l'exécution

La recherche dans le domaine de la modélisation @Runtime est prometteuse en face de l'évolution connue par l'adaptation des systèmes temps réel embarqués à base de composants. Dans ce contexte, certaines approches sont proposées pour surmonter le défi d'adaptation. Nous allons détailler quelques unes et les critiquer dans le but d'en extraire ses bénéfices et ses inconvénients.

2.3.1 L'utilisation des modèles à base de trace

L'ingénierie des systèmes logiciels utilise typiquement le niveau code pour la traçabilité dans le but de suivre et de contrôler un système en cours d'exécution. Une alternative consiste à générer et analyser des modèles à base de traces [Mao09]. L'utilisation d'un tel type de modèle, permet de suivre et de surveiller les modèles de conception des systèmes logiciels. Cette technique est basée sur la gestion des traces qui sont emmagasinées sous forme de modèle (graphe).

Description

L'utilisation d'un modèle basé sur les traces permet à l'utilisateur le contrôle et la supervision des modèles de conception du système. Pour générer des modèles à base de traces, on doit passer par la génération des traces à base de scénarii. Cela veut dire qu'on doit prévoir tous les scénarii possibles de configurations du système en utilisant des diagrammes comme les diagrammes de séquences, les diagrammes de classes ou les diagrammes d'états. Les métriques et les opérateurs des modèles basés sur les traces peuvent servir comme outil pour connaître d'avantage la structure et le comportement du système en cours d'exécution. Ceci est effectué à un certain niveau d'abstraction dans le but de définir les modèles de conception. Un modèle basé sur les traces peut être interprété comme un type spécial des modèles d'exécution. Ainsi, l'analyse de ces modèles supporte des exercices reliés aux tests, compréhension et évolution des modèles de conception. La même exécution concrète peut résulter en différentes traces, selon les outils utilisés pour le traçage. Donc, on a mis en place

un mécanisme unifié pour faire le mapping des éléments concrets de l'exécution vers des éléments dans le modèle. Il est possible de générer automatiquement le modèle à base de traces à partir de l'exécution du système. Tout dépend du type du modèle choisi et de l'environnement logiciel entourant.

On commence par générer des diagrammes de séquence en intégrant des propriétés de protection. Cette génération automatique se fait par le moyen de compilation des modèles avec un code d'aspect. Puis on utilise des outils appelés Traceurs « Tracer » pour les analyser en se basant sur les techniques de visualisation et d'exploration interactive. Le traceur permet l'analyse, la visualisation, le filtrage et la comparaison des traces à bases de scénarii. Cet outil présente les évènements occurrents dans une exécution et donc dans une trace. Il présente aussi la liste des diagrammes de séquence générés ordonnée par cas d'utilisation et les durées d'exécution d'un évènement.

Discussion

Une caractéristique importante de ces modèles est qu'ils ne sont pas adaptés uniquement des simples projections des informations concrètes. Plutôt ils manifestent des abstractions dans lesquels les entrées de la trace dépendent de l'historique, du contexte d'exécution et du modèle.

Ce niveau de réflexion fondé sur les modèles assure une visibilité unique du système en cours d'exécution et offre une analyse dynamique basée sur les modèles. Cette approche offre de plus la liberté totale de choix pour les langages d'implantation.

Cependant, elle n'offre la capacité de surveillance d'un système que dans un seul sens, de l'exécution vers le modèle. On part d'une application qui tourne sur une plateforme donnée, puis on commence à générer des diagrammes à base de traces décrivant avec détails l'exécution du système.

De plus, l'analyse de ses modèles, nécessite l'effort d'un expert du domaine pour choisir les outils convenables et interpréter les modèles générés. Ce qui peut alourdir la tâche de l'analyseur surtout qu'il ne s'agit plus d'un seul modèle d'exécution mais plutôt d'un modèle de conception et d'une série de diagrammes.

2.3.2 Intégration d'un service au niveau du modèle

Cette approche est introduite dans le cadre de la détection des erreurs reliées à l'adaptation et le contrôle de l'adaptation. Un système peut prendre décision d'accepter ou refuser une requête d'adaptation en surveillant l'historique des adaptations qui ont été déjà effectuées. Il s'agit d'intégrer un service au niveau du modèle d'exécution permettant de prévoir l'état du système après l'adaptation [OPDR08].

Description

Dans ce contexte, les modèles d'exécution ont la charge de surveiller l'application et de lancer l'adaptation quand elle est demandée. Le modèle est concrétisé en tant que service qui a la charge de prévoir si l'adaptation va lui amener à un état risquée ou non de l'application. La concrétisation du modèle en tant que service se fait en

deux étapes. En premier lieu, on enrichit le modèle d'exécution par des informations dépendantes de la plateforme. En deuxième lieu,au niveau du modèle, on vérifie si l'application est protégée après l'adaptation. Cette technique établit un protocole de service qui formalise la façon avec laquelle le service peut être utilisé et comment il peut communiquer avec la plateforme.

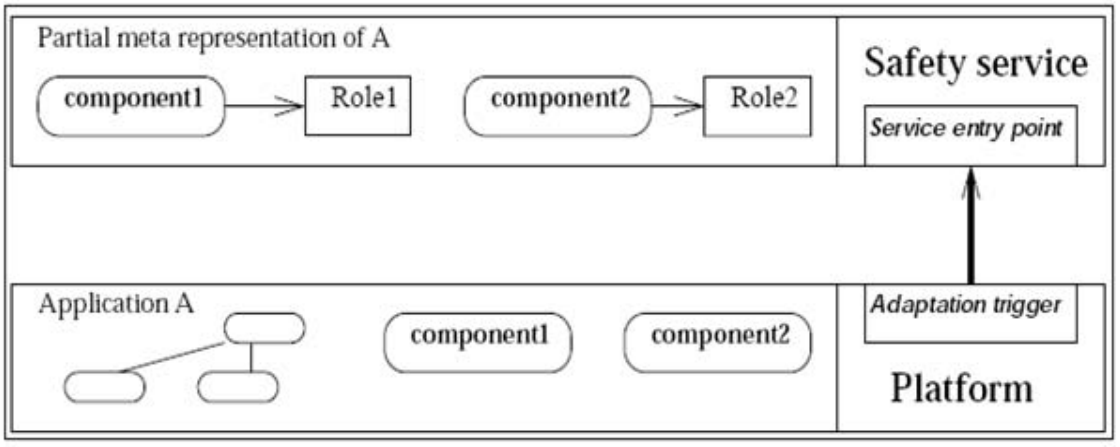

Figure 2.3 – Architecture du service de protection [OPDR08]

On part tout d'abord d'une application sûre et protégée. On identifie un ensemble de contraintes et de propriétés de protection et de surveillance qui couvre une marge d'erreurs locales par exemple les messages non compris et globales comme la divergence ou la synchronisation. Chaque propriété est garantie avec un groupe de contraintes OCL [1] [WK03] attachées au modèle d'exécution. Lors d'un déclenchement d'une adaptation, on vérifie les contraintes. Si cette adaptation mène alors à un état souhaité et protégé de l'application on l'applique et on garantit que l'application reste protégée. Si non, on ignore la demande d'adaptation.

Discussion

L'apport de cette approche est l'implantation d'un service qui rend possible l'usage du modèle avec différentes plateformes. Une plateforme peut utiliser le service sans connaître sa structure interne grâce à l'interface IDL (*CORBA* [2])comme point d'entrée. Cette intégration du service rend plus facile l'interaction du modèle d'exécution avec les applications réelles.

En revanche, Cette technique permet une adaptation dynamique d'une application en cours d'exécution dans un seul sens, de l'exécution vers le modèle. De plus, elle ne permet ni la génération automatique de code ni la réutilisation ni la modularité.

1. Object Constraint Language
2. www.corba.org

2.3.3 Utilisation d'un framework orienté aspect et orienté modèle pour l'adaptation

Cette approche, appelée Kermeta@Runtime [MBJ08, MFB+08], offre un framework orienté aspect et orienté modèle pour la supervision des systèmes à base de composants durant l'exécution. Il permet une adaptation dynamique des systèmes logiciels en cours d'exécution. Cette adaptation est faite dans deux sens : de l'exécution vers le modèle et du modèle vers le système en cours d'exécution.

Description

L'adaptation du système dans le premier sens décrivant le passage du système en cours d'exécution vers le modèle se fait en deux étapes. La première étape consiste à générer un modèle intermédiaire appelé « Modèle de référence » . Ce modèle est généralement décrit avec un langage autre que celui du modèle de conception et conforme au système à tout moment. Il est généré par un mécanisme d'introspection en répondant aux questions : Quels composants comporte le système actuel ? Et comment sont-ils liés ? En fait, on étend les opérations d'introspection pour découvrir les opérations exécutées et ces paramètres qui sont requis par les ports. Puis, on associe à chaque port une interface API (`java.lang.reflect`) qui fournit les classes et les interfaces afin d'obtenir des informations concernant les classes et les objets. La deuxième étape est la génération automatique du modèle à travers le modèle de référence en utilisant les langages de transformation de modèles comme Kermeta [3] [MSF+10].

Pour l'adaptation dans le deuxième sens, du modèle vers l'exécution, on commence par instancier le modèle de référence d'après un scratch en utilisant l'API d'introspection. Puis on effectue une comparaison entre les deux modèles en utilisant des outils de comparaison de modèles comme `EMFCompare`. Et ceci car le modèle de référence et celui de conception peuvent être de langages différents. Après l'analyse des différences et des similarités entre les deux modèles, on génère un ensemble de commandes responsables de l'ajout et/ou la suppression des composants et/ou des connecteurs. Ces commandes sont ordonnées selon ses priorités. Finalement, la séquence ordonnée de commandes est exécutée par la plateforme afin d'adapter le système en cours d'exécution.

Discussion

Dans ce cadre, les modèles résolvent le problème de complexité à travers les abstractions sur différents niveaux. Ils sont utilisés d'une part pour spécifier la variabilité dynamique au cours de la conception et d'autre part pour gérer l'adaptation au cours de l'exécution. Les techniques orientées aspect sont utilisés pour modéliser les préoccupations de l'adaptation séparément des autres préoccupations du système. En se basant sur la séparation avancée des préoccupations, l'adaptation devient plus facile à concevoir, comprendre et valider au cours de l'exécution dans les deux sens. Cette séparation offre également la modularité, une meilleure réutilisation et une maintenance aisée.

3. http ://kermeta.org/

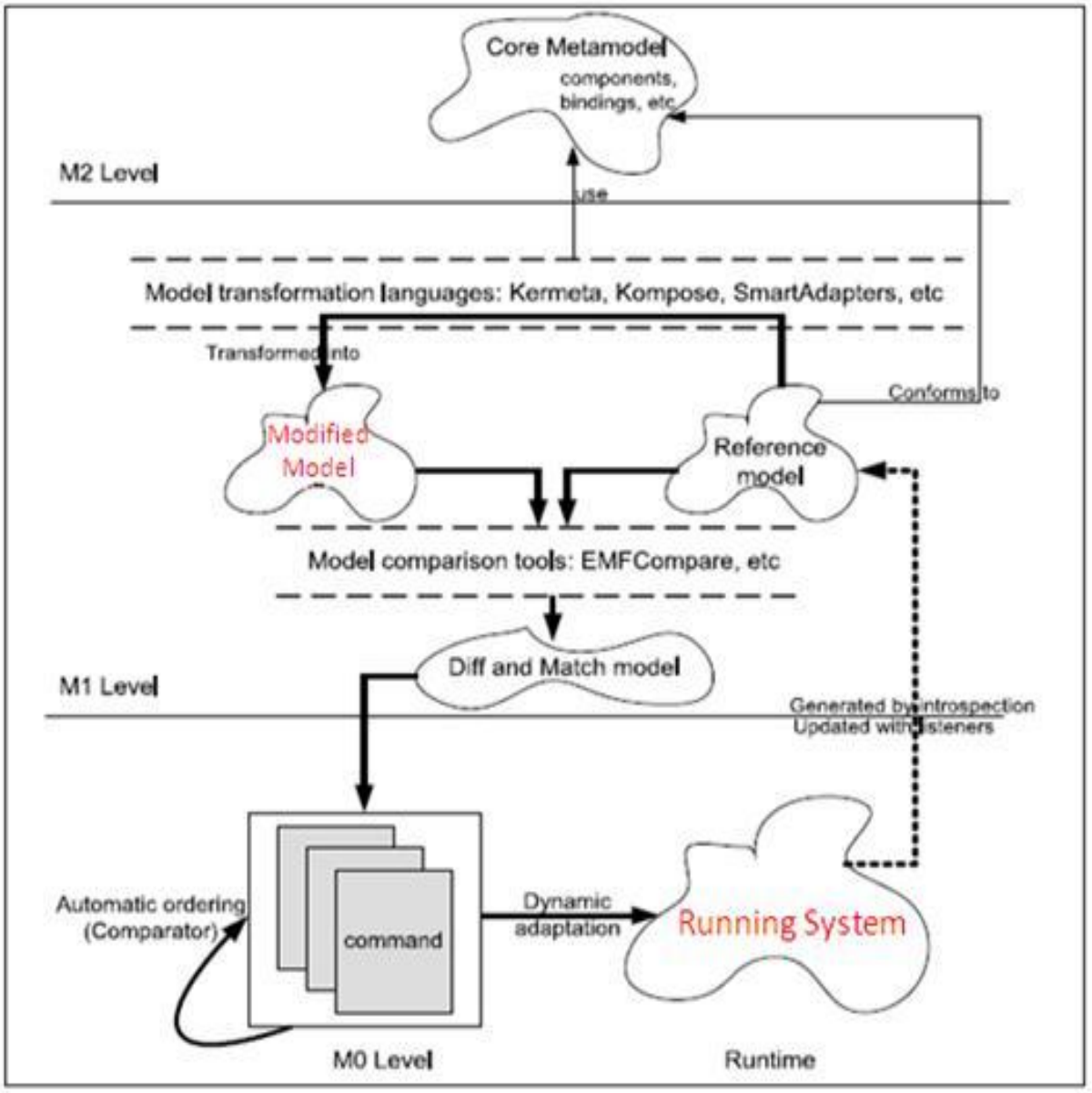

Figure 2.4 – Adaptation guidée par les aspects et les modèles en cours d'exécution [MFB+08]

Toutefois, l'utilisation des langages de transformation entre modèles peut engendrer l'augmentation du taux d'erreurs de conception dans les différents niveaux. En outre, cette approche parait un peu complexe par rapport aux autres. Elle met en évidence l'usage des outils de transformation et de comparaison de modèles dont un utilisateur doit les maîtriser.

2.3.4 Utilisation du modèle de conception pour la modélisation @Runtime

Cette approche consiste à modifier le modèle de conception au fur et à mesure des besoins d'adaptation [SSC09]. Dans ce contexte, les modèles d'exécution ont la charge de surveiller l'application et de lancer l'adaptation quand elle est demandée. Ils permettent de relier l'implantation, les modèles de conception et les règles d'adaptation.

Description

Pour appliquer les règles d'adaptation sur les modèles de conception, on doit suivre un processus automatique durant l'exécution. A travers ce processus, les modèles d'exécution seront alors générés automatiquement.

Pour générer les modèles d'exécution, on commence par définir un ensemble de règles d'adaptation qui vont être vérifiées par la suite dans le but de lancer ou non une adaptation.

- Les conditions contiennent les informations qui déclenchent l'adaptation.
- Les effets expliquent comment l'effectuer.

Ensuite, on passe à la génération du premier modèle d'exécution en suivant un algorithme qui comporte deux étapes.

- La première étape consiste à la sélection des différentes classes de spécification qui sont utilisées par les règles d'adaptation. Pour chaque condition active, l'ensemble des éléments à contrôler est identifié. Puis, un déclencheur (Trigger) est crée en utilisant les conditions actives de chaque règle d'adaptation. Enfin, l'ensemble des éléments est réduit pour éviter la redondance des éléments.
- La deuxième repose sur l'identification du point d'accès dans l'exécution. Les différents points d'accès peuvent être implantés avec différents langages. Actuellement, le modèle d'exécution est développé avec Kermeta. Pour chaque point d'accès une méthode Kermeta est crée avec le code intermédiaire Java.

Une fois le modèle d'exécution généré, il commence à surveiller le code l'application. Quand une adaptation est déclenchée, le modèle d'exécution accompagné des règles d'adaptation permettent la création d'un cliché (`snapshot`) de l'application contenant uniquement la partie cible de l'adaptation. Le but du cliché est de présenter une vue du logiciel et de lier les éléments de spécification à la plateforme. C'est une sorte de traçage sur les modèles et non sur l'exécution. Le processus de la création du cliché suit le même algorithme de génération des modèles d'exécution, mais en considérant uniquement les expressions d'adaptation. L'adaptation est par la suite accomplie au niveau du modèle de spécification obtenu par le cliché. Une nouvelle configuration du code est obtenue à partir de la nouvelle version du modèle de conception tout en utilisant les mêmes techniques de génération initiale du logiciel.

Discussion

Cette approche assure une adaptation dynamique du système en cours d'exécution dans les deux sens en se basant sur les règles d'adaptation et la

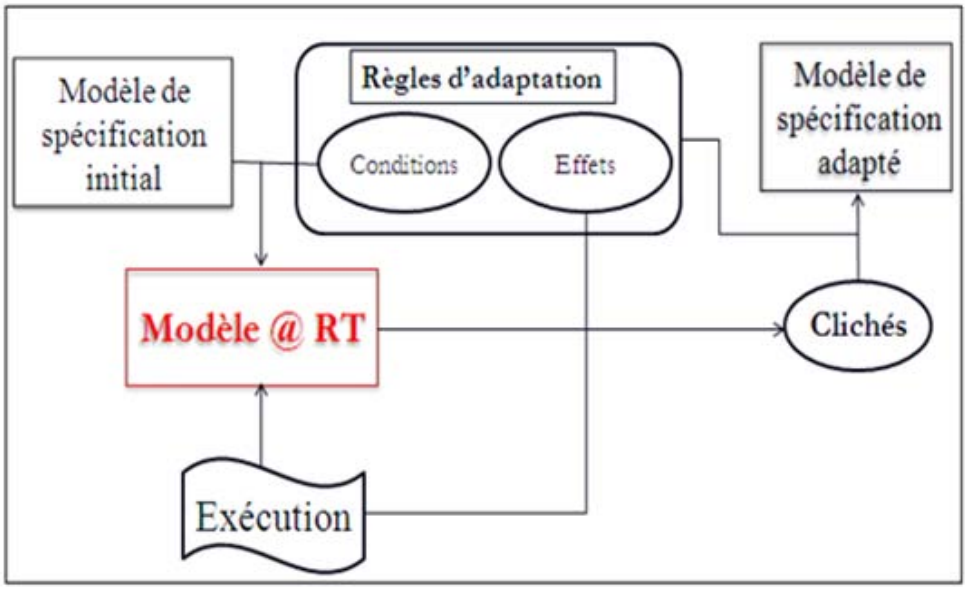

Figure 2.5 – Vue d'ensemble de l'approche [SSC09]

génération automatique de clichés. Elle est favorisée grâce à l'optimisation de l'efficacité des modèles d'exécution à l'égard de la surveillance partielle. En fait, le modèle d'exécution contrôle uniquement la partie cible d'adaptation et même le cliché contient seulement la partie du modèle objet d'adaptation. Ce cliché peut refléter la totalité du modèle dans le cas de besoin d'une vue d'ensemble du système. Cette technique offre la génération automatique de code à partir du modèle de conception ce qui peut faciliter énormément le travail du développeur.

Cependant, cette approche présente quelques inconvénients comme la différentiation du modèle d'exécution et celui de spécification. Dans ce cas le concepteur est censé manipuler différents langages de modélisation, celui de la conception et un autre pour le modèle d'exécution.

Dans le cas d'une erreur au niveau de l'adaptation en cours d'exécution, il doit comprendre les différents formalismes pour résoudre le problème. Aussi, cette approche est mieux adaptée dans le cas des systèmes contenant une grande partie statique de conception. Elle n'est pas prévue pour les systèmes fortement dynamiques par exemple. En outre, elle est basée sur les langages de transformation qui sont utilisés dans le cas d'un système centralisé. Par contre, les systèmes modernes sont souvent distribués.

2.4 Synthèse générale et objectifs

A l'issue de l'étude effectuée sur les techniques de modélisation @Runtime, nous pouvons conclure que :

– Il existe des techniques de modélisation @Runtime qui offrent une adaptation dynamique dans deux sens. Par ailleurs, d'autres traitent l'adaptation dans un seul sens sans penser à l'autre sens.

- On constate que certaines approches [MBJ08, MFB+08, SSC09] font la différentiation entre le modèle de conception et le modèle d'exécution. Elles utilisent des langages de modélisation distincts pour représenter ces deux modèles. Dans ce cas, nous avons besoin d'utiliser des outils et/ou des langages de transformation et/ou de comparaison de modèles. Le concepteur doit faire ainsi un effort supplémentaire afin d'apprendre des nouveaux langages. De plus, l'utilisation de différents formalismes augmente le risque d'erreurs.
- Parmi les techniques de modélisation @Runtime, il existe quelques unes qui permettent la génération automatique du code et d'autres qui ne le permettent pas. La génération automatique du code peut faciliter énormément le travail en mesure de produire le code adéquat de manière cohérente, rapide et sûre. Elle représente un besoin primordial pour les systèmes logiciels.
- Il n'y a pas un consensus sur les techniques de modélisation @Runtime des systèmes à base de composants distribués. Certaines approches permettent une adaptation dynamique des systèmes dans les deux sens en considérant l'application centralisée s'exécutant sur un seul processus.
- Certaines approches ont imposé le choix des langages de programmation et/ou de modélisation pour la mise en œuvre de leur approche. Par contre, d'autres techniques offrent plus de liberté de choix pour les deux langages. Ainsi, l'utilisateur ne sera pas obligé de déployer son application sur une plateforme bien déterminée.
- On remarque suite à cette étude l'existence des approches qui, basées sur le concept des aspects, permettent la séparation des préoccupations. Cette séparation donne plus de flexibilité à l'application grâce à la modularité et la réutilisation qu'elle offre. La séparation des préoccupations permet aussi d'améliorer la qualité et la lisibilité du code et de faciliter la maintenance des systèmes embarqués à base de composants.

Notre objectif est de définir un processus complet d'adaptation dynamique des systèmes embarqués à base de composants. Ce processus, basé sur le concept de modélisation @Runtime couvre tout le cycle de développement logiciel. Tout d'abord, nous commençons par traduire les besoins de l'utilisateur dans un modèle pendant la phase de modélisation. Nous avons choisi AADL comme langage de description d'architecture permettant la modélisation selon différents niveaux d'abstraction. Nous optons à offrir la possibilité d'étendre ce modèle décrit avec le standard AADL par des déclarations ou des propriétés exprimées avec AO4AADL [4] qui est une extension d'AADL par les concepts d'aspects. Nous avons pensé à utiliser AO4AADL dans le but d'assurer une séparation des préoccupations dés la phase de conception. Ensuite, nous passons à la phase d'implantation par l'intermédiaire d'une génération automatique du code vers une plateforme choisie. Une fois le code généré et déployé, nous obtenons notre système en cours d'exécution. Puis, nous envisageons une méthode de surveillance du système sur les deux niveaux conceptuel et exécutif.

Enfin, nous visons à accomplir une adaptation quand elle est demandée. Cette adaptation doit être effectuée automatiquement dans les deux sens sans repasser ni par la phase de déploiement ni par la phase d'implantation dans le temps estimé.

4. Aspect Oriented extension for AADL

En guise de conclusion, le tableau 2.1 permet de récapituler les approches étudiées. Dans ce tableau, nous évaluons ces approches selon différents critères.

- L'adaptation dans les deux sens ou dans un seul sens
- La séparation des préoccupations pour assurer la modularité et améliorer la réutilisabilité
- La génération automatique de code permettant de faciliter la tâche du développeur
- La prise en compte de l'adaptation des systèmes temps réels embarqués (RTES) et distribués (DS)
- Le choix de la plateforme est imposé à l'avance par l'approche ou c'est à l'utilisateur de choisir selon ses besoins
- L'utilisation d'un même modèle de conception et d'exécution. Le modèle de conception est établi au début puis mis à jour pour refléter l'exécution.

Table 2.1 – Synthèse des travaux étudiés

Approche	Adaptation dans deux sens	Séparation des préoccupations	Génération automatique de code	Adaptation des RTES	Adaptation des DS	Plateforme	Utilisation du même modèle
Maoz, 2009	Non	Oui	Non	Oui	Oui	Au choix	Non
Saudrais et al.,2009	Oui	Non	Oui	Oui	Non	Java	Non
Morin et al., 2010	Oui	Oui	Oui	Oui	Oui	Java	Non
Occello et al., 2008	Non	Non	Non	Non	Oui	Au choix	Oui
Notre approche	**Oui**	**Oui**	**Oui**	**Oui**	**Oui**	**Ada ou Java ou C**	**Oui**

2.5 Conclusion

Dans ce chapitre, nous avons étudié le mécanisme de traçabilité puisque la plupart des techniques d'adaptation sont à base de ce mécanisme. Puis nous avons analysé et examiné les différentes approches travaillant sur la modélisation @Runtime. Ensuite, nous avons synthétisé et résumé cette étude après avoir comparé les différentes approches qui sont les plus liées à la notre. Cette comparaison est basée sur certains critères qui nous paraissent primordiaux pour la modélisation @Runtime.

Sur la base des connaissances établies suite à notre étude bibliographique, nous avons essayé de tirer profit pour définir notre propre approche de modélisation @Runtime pour l'adaptation des systèmes embarqués à base de composants.

Nous proposons dans le chapitre suivant notre approche en détails.

Chapitre 3

Approche de modélisation @Runtime

3.1 Introduction

Dans le chapitre précédent, nous avons présenté une étude bibliographique des techniques de modélisation @Runtime existantes pour l'adaptation des systèmes embarqués. Comme déjà mentionné, nous nous intéressons dans ce travail de mastère à définir et valider une approche de modélisation @Runtime pour l'adaptation des systèmes embarqués à base de composants.

Notre contribution sera décrite en détails dans ce chapitre. Nous commençons, dans la première partie par présenter le principe général de notre approche. Ensuite nous passons à la détailler en suivant le cycle de développement adopté. Nous partons de la modélisation ainsi que l'implantation. Puis, dans le reste du chapitre, nous nous focalisons sur la présentation des deux procédures de surveillance et d'adaptation. Nous clôturons en fin par une conclusion.

3.2 Principe général de l'approche

Notre travail vise à définir une approche de modélisation au cours de l'exécution pour l'adaptation des systèmes à base de composants. Pour ce faire, il est indispensable de présenter un processus complet assurant cette adaptation.

Dans une première étape, nous commençons par concevoir notre modèle avec le langage AADL comme langage de description d'architecture. La description architecturale de notre système décrit avec le standard AADL est extensible soit par des propriétés soit par des annexes. Les propriétés peuvent enrichir une description architecturale par des caractéristiques non architecturales. Les annexes peuvent enrichir cette description par des déclarations exprimées avec un autre langage que AADL comme AO4AADL qui est une extension d'AADL pour les concepts d'aspects. Les propriétés et les annexes permettent ainsi d'enrichir un modèle AADL dans le but de séparer les différentes préoccupations fonctionnelles et transversales.

Dans une deuxième étape, nous passons à l'implantation du système par l'intermédiaire d'un outil de génération automatique de code. Nous utilisons « Oca-

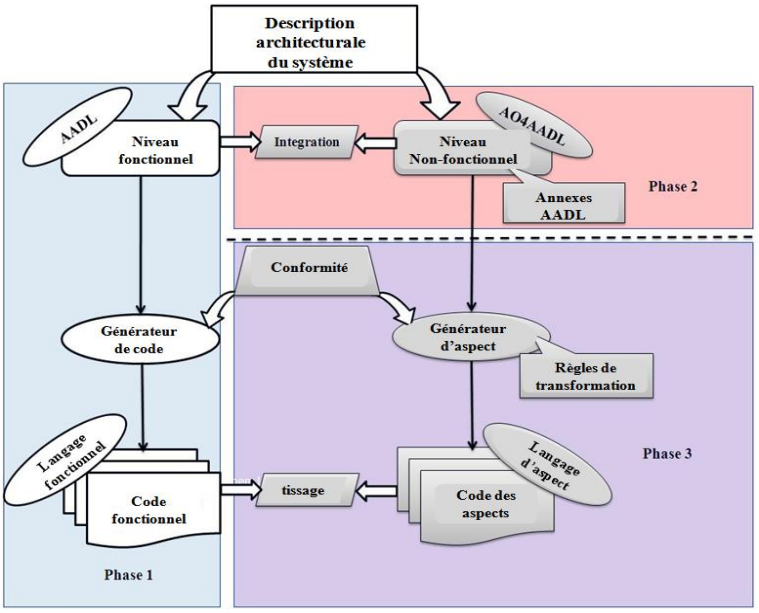

Figure 3.1 – Processus de développement [Lou10]

rina » [VZH06] qui est une série d'outils permettant de générer automatiquement le code fonctionnel d'un système correspondant à un modèle AADL donnée. Même le code des aspects correspondants à l'annexe AO4AADL peut être généré par Ocarina. Réellement, notre travail de mastère présente une continuité des travaux de mastère de Mme Sihem LOUKIL [Lou10, LKZJ10]. Sa contribution consistait à la définition d'un nouveau langage AO4AADL qui permet de spécifier les aspects architecturaux non fonctionnels au niveau du modèle AADL à travers une annexe AO4AADL. Pour mettre en œuvre son approche, elle a défini un processus de développement pour la génération de code à partir d'un modèle décrit avec AADL et AO4AADL.

La figure 3.1 illustre les différentes étapes de ce processus que nous avons adopté pour les deux phases de conception et d'implantation de notre approche.

L'étape suivante consiste à la surveillance du système obtenu. Il s'agit de mettre en place toute une procédure de surveillance de l'exécution et du modèle du système. Finalement, la surveillance du système doit déclencher une adaptation en cas de besoin. Cette adaptation dynamique doit être accomplie dans les deux sens d'une manière à répondre aux besoins externes et aussi internes dans des délais restreints.

3.3 Modélisation du système

Dans le but d'utiliser un même formalisme pour la représentation du modèle de conception et le modèle d'exécution, nous avons choisi AADL comme langage de description d'architecture concret. AADL permet de modéliser les préoccupations fonctionnelles de notre système sur différents niveaux d'abstraction. Pour couvrir les préoccupations transversales comme la sécurité ou la traçabilité, nous décidons d'étendre le modèle décrit avec le standard AADL par une annexe AO4AADL. Ainsi, nous garantissons une séparation des préoccupations au niveau architectu-

ral et donc nous assurons la modularité et une meilleure réutilisation. En outre, utilisant AADL, nous sommes capables d'uniformiser les deux modèles de conception et d'exécution. Ce qui garantit une réduction du taux d'erreurs de conception et d'adaptation puisque nous évitons l'usage des langages de transformation et de comparaison de modèles. Dans ce cadre, nous avons eu l'idée de définir une représentation graphique d'AO4AADL qui sera intégrée dans un modèle AADL par l'intermédiaire d'un éditeur spécifique qui le permet. Dans la section suivante, nous focalisons notre étude sur la recherche de l'éditeur adéquat.

« ***AADL Graphical Editor*** »

En explorant les différents éditeurs existants permettant une modélisation AADL, nous avons constaté que ces éditeurs sont peu nombreux, peu matures et ne répondent pas aux besoins spécifiques de notre approche de modélisation @RunTime. Réellement, nous avons testé ces éditeurs :

· **ADELE** : Il s'agit d'un sous projet open source du projet TOPCASED. Il est composé de six plugins développés avec Eclipse ayant chacun ses propres fonctionnalités. Mais cet éditeur reste instable ce qui provoque beaucoup de problèmes : blocage de l'éditeur, des erreurs inattendus, une brusque fermeture de l'éditeur, etc.

· **STOOD** : c'est un produit stable développé en C++. Il supporte plusieurs langages de modélisation comme AADL et UML2. Le problème de STOOD est qu'il est un produit payant. Seule une version de démonstration de 30 jours est disponible gratuitement.

· **OSATE-TOPCASED** : c'est un outil open source. Il est stable et performant. En revanche, il présente des limites en regard de la modélisation @RunTime, ce qui nous empêche de l'utiliser.

Ces trois outils existants sont très mal adaptés pour notre objectif principal puisqu'ils ne permettent pas une représentation graphique globale de l'application. Seules les représentations graphiques des déclarations de types et d'implantations des composants de l'application sont exposées dans la même interface. Mais pour ajouter et visualiser des sous-composants, nous devons cliquer sur l'implantation du composant en question et une nouvelle interface s'affiche. Cette dernière présente seulement l'implantation du composant et permet à l'utilisateur d'ajouter des sous composants et de les connecter. Donc, les trois outils cités ne permettent pas d'avoir une vision globale de l'application qui montre tous ses constituants. Or, pour assurer la modélisation @RunTime nous devons exposer toute la hiérarchie de l'architecture de l'application dans une même interface pour voir la dynamique du modèle en suivant l'exécution. Pour cela, nous avons pensé à développer notre propre éditeur qui répond à nos besoins. En effet, nous avons besoin d'un outil stable et robuste permettant au concepteur de représenter le modèle de l'application avec AADL avec un certain niveau d'abstraction. En outre, cet outil doit permettre une représentation graphique globale de l'architecture de l'application afin de répondre aux exigences de la modélisation @RunTime. Comme AADL a défini des règles de contenance des sous-composants, en plus des règles définissant les éléments d'interface possibles pour chaque catégorie de composant, l'outil à concevoir doit les respecter : il doit interdire à l'utilisateur toute tentative de violation de ces règles. De plus, nous avons besoin d'une trace sous format XMI (*XML Metadata Interchange*) présentant la traduction de la représentation graphique du modèle dans le but de faciliter la mise à jour du

modèle d'exécution au cours de l'adaptation. Car il est plus facile de gérer tel type de fichiers que des représentations graphiques et surtout dans le cas de traitement automatique. Nous cherchons également un éditeur flexible afin de pouvoir intégrer les concepts d'aspects offerts par AO4AADL dans un modèle généré à partir de cet éditeur.

Dans ce contexte, nous avons décidé de développer notre propre éditeur sous la forme d'un plugin Eclipse permettant de concevoir un modèle avec le standard AADL tout en intégrant des aspects AO4AADL. Nous avons choisi de développer cet éditeur sous la forme d'un plugin Eclipse car si non nous serons obligé de le développer de zéro. Cette solution ne répond pas à la contrainte de temps d'un projet de mastère. Il va falloir écouler des années de développement pour arriver finalement à complèter son développement. Par contre, avec Eclipse, nous pouvons réaliser un tel éditeur dans quelques mois en s'assurant qu'il répond véritablement à nos besoins.

Cet éditeur, que nous appelons « **AADL Graphical Editor** » est implanté dans le cadre d'un projet de fin d'étude au sein de notre unité de recherche ReDCAD.

3.4 Implantation du système

Suivant le cycle de développement logiciel, après avoir conçu le modèle correspondant à notre système, nous passons à la phase d'implantation. Ce passage d'une phase à l'autre est effectué soit manuellement soit automatiquement. Partant d'un modèle AADL intégrant les aspects AO4AADL, nous adoptons un processus de génération automatique de code. Comme nous avons déjà mentionné, nous allons suivre le processus proposé par [LKZJ10] qui est basé sur deux phases pour la génération de code comme l'indique la figure 3.1.

3.4.1 Implantation du code fonctionnel

La première phase consiste à implanter les exigences fonctionnelles de notre système sans prendre en considération les préoccupations techniques. Une fois le modèle du système déjà conçu, le concepteur procède à la génération automatique du code fonctionnel correspondant par l'intermédiaire d'un générateur de code. Particulièrement, pour la génération du code correspondant à un modèle décrit avec le standard AADL, il existe des compilateurs permettant la traduction en différents langages à savoir Ada [Ada83], C ou RTSJ (*Real Time Specification for Java*) [Aut09] qui sont offerts par la suite d'outils Ocarina. Cet outil est à base d'intergiciel POLYORB_HI [HZ07]. Ces générateurs permettent d'obtenir du code en langage de programmation impératif à partir d'une description AADL.

- **Générateur Ada**
  C'est avec Ada qu'a été développée la première version de l'intergiciel POLYORB-HI. A chaque entité de l'application est générée une hiérarchie de paquetages qui comporte toutes les déclarations de types de données utilisés ainsi qu'un ensemble de fonctions et de procédures assurant la traduction des sous-programmes nécessaires pour chaque nœud.
  Une description plus détaillée est disponible dans [Zal08].

– **Générateur C**
La version en langage C de POLYORB_HI offre une génération des composants similaire à celle du langage Ada. En fait, dans cette version les sous programmes sont traduits en fonctions et les mêmes types de données que le langage Ada sont traités. Toutefois, ces deux générateurs ne sont pas totalement similaires. Le langage C est procédural, ainsi la traduction en C d'un modèle AADL ne fait pas intervenir des instructions orientées objet comme Ada. Aussi, le langage C n'introduit pas la notion de paquetage. Les sous-programmes AADL sont traduits en C par des fonctions dont le type de retour est `void`.
Les accès aux composants de donnée ainsi qu'aux sous-programmes se font grâce aux pointeurs.
– **Générateur RTSJ**
Un troisième générateur de code offert par Ocarina introduit le concept de la programmation orientée objet en utilisant le langage Java Temps réel nommé RTSJ (*Real Time Specification for Java*).
Ce générateur est à base d'un ensemble de règles de transformations des composants AADL en RTSJ.

3.4.2 Implantation du code des aspects

Après avoir décrit les propriétés techniques assurant le bon fonctionnement de l'application et l'amélioration de la qualité du code à travers les aspects AO4AADL, le concepteur génère le code correspondant à ces propriétés. Vu que le langage AO4AADL est générique, ses aspects peuvent être traduits en plusieurs langages de programmation orientée aspect comme AspectJ [KHH+01] pour le langage Java, AspectC++ pour le langage C/C++, et AspectAda pour le langage Ada.

Une telle génération demande un générateur d'aspect et un ensemble de règles de transformation d'AO4AADL au langage d'aspect choisi. Ces règles de transformation doivent prendre en compte les règles de transformation déjà existantes pour la génération du code fonctionnel à partir de la description architecturale en AADL.

La conformité des générateurs est nécessaire pour garantir la cohérence entre le code fonctionnel et celui non fonctionnel. Car les aspects générés seront automatiquement intégrés dans le code fonctionnel par l'intermédiaire de l'opération de tissage. Cette conformité permet d'avoir à la fin un système complet prêt à être exécuté. Actuellement, un seul générateur de code des aspects AO4AADL est disponible. AO4AADL offre un prototype de générateur AspectJ basé sur un ensemble de règles de transformations. AspectJ est le seul générateur existant actuellement.

Pour des raisons de conformité, nous avons adopté le générateur RTSJ pour la génération des préoccupations fonctionnelles.

La figure3.2 résume le processus d'implantation que propose notre approche. Une fois le code fonctionnel du système généré aussi bien que celui des aspects, nous obtenons le système final prêt à être déployé et exécuté. Après l'exécution, nous nous intéressons à la surveillance.

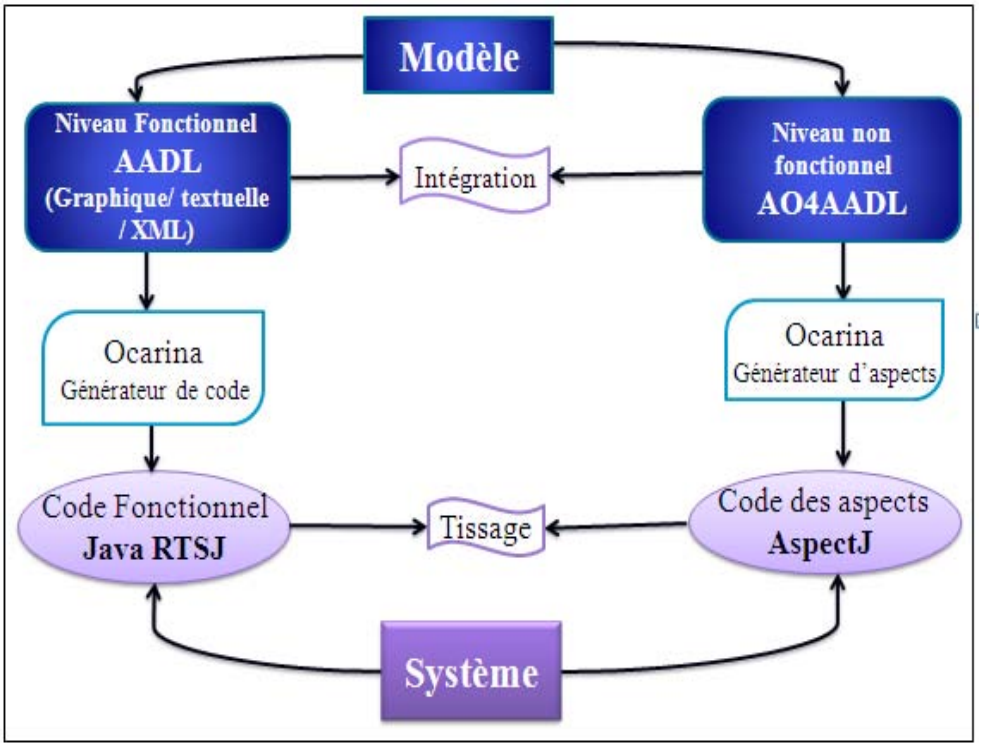

Figure 3.2 – Processus d'implantation proposé

3.5 Surveillance du système

Dans le but d'assurer une adaptation dynamique du système au cours de l'exécution, nous proposons une procédure de contrôle et de surveillance du système. Il s'agit de capturer les changements sur les deux niveaux conceptuel et exécutif pour agir en passant vers les nouvelles configurations souhaitées.

Dans certains cas, le concepteur a besoin de surveiller tous les composants constituant son application. On parle dans ce cas de surveillance totale. Par exemple, pour un système décrivant une chaîne de production, le chef des ouvriers doit surveiller toutes les opérations de fabrications nécessaires à la réalisation du produit. Dans d'autres cas, la surveillance unique d'un sous ensemble de composants répond aux besoins. Prenons l'exemple d'un système de détection d'incendie. La surveillance peut concerner un ensemble de compartiments d'un bâtiment à l'exception des locaux ne présentant pas de charge calorifique ou des locaux protégés par des extinctions automatiques d'incendie. C'est pour cette raison que nous avons introduit la surveillance partielle du système. Dans la suite de cette section nous allons détailler la surveillance totale et aussi partielle.

3.5.1 Surveillance totale

Dans ce cas, nous devons intercepter toute les entités constituant l'architecture de notre système. Nous nous intéressons aux changements subis par toutes les entités et les connexions existantes au sein de notre système. Dans ce contexte, un

changement consiste à l'ajout ou la suppression d'un thread, d'une connexion ou d'un sous programme. Même l'opération de modification des propriétés est traduite en deux opérations de suppression et d'ajout avec conservation d'état.

Réellement, nous ne pouvons pas ajouter ni supprimer des nouvelles implantations de composants au cours de l'exécution car ceci nécessite d'arrêter l'exécution, de recompiler et d'exécuter de nouveau notre système. Effectivement, il s'agit de l'activation ou la désactivation de composants ou de connexions déjà existants. En conséquence, la surveillance de notre système revient à l'interception des actions d'activation ou de désactivation de toutes les entités qui le constituent.

Par exemple, si notre système comporte trois threads (`Th1, Th2` et `Th3`), pour le surveiller nous devons intercepter les changements de tous ces trois threads (`Th1, Th2` et `Th3`) et de toutes les connexions qui les relient.

3.5.2 Surveillance partielle

Dans le contexte d'une surveillance partielle, il est indispensable d'écarter le contrôle des composants non concernées. Dans ce cas on est censé de surveiller uniquement la partie cible d'adaptation. Par exemple, si les deux threads `Th1` et `Th3` de l'application sont statiques, on intercepte uniquement les changements subis par le thread Th2. Pour que nous puissions sélectionner la partie cible d'interception, nous avons pensé à utiliser le langage AO4AADL. Comme déjà indiqué, ce langage permet d'intégrer des aspects au niveau architectural dans un modèle décrit avec le standard AADL. Ainsi, il nous fournit la possibilité de choisir certaines entités au niveau du modèle depuis notre propre éditeur « AADL Graphical Editor ». Ces aspects permettent de fixer les entités qu'on va surveiller au niveau du modèle.

En contre partie, pour la surveillance de l'exécution, nous partons des aspects AO4AADL associés au modèle et nous générons le code AspectJ correspondant qui sera intégré au niveau du code fonctionnel à travers un simple tissage. Ainsi, la partie cible d'interception est définie aussi au niveau du système.

3.5.3 Mécanisme de surveillance

Dans le cas de la surveillance totale, nous cherchons à intercepter toutes les entités (threads, sous programme et connexions) de l'application. Nous associons des aspects génériques au code fonctionnel permettant de suivre le cycle de vie de tous les composants et les connexions. Quant à la surveillance partielle, nous sélectionnons la partie sujette d'interception avec le langage AO4AADL au niveau du modèle et les aspects spécifiques correspondants seront générés automatiquement en AspectJ.

Dans les deux cas de surveillance totale ou partielle, les aspects permettent de capturer le changement au niveau du système ou au niveau du modèle. Après la détection, nous passons à la traçabilité en ligne des changements capturés sur les deux niveaux. Il s'agit de l'enregistrement immédiat et automatique des traces.

Afin de faciliter l'échange d'information traduisant la trace, nous avons choisi de décrire cette trace sous la forme d'un message XML. Ce choix est effectué pour simplifier le transfert de messages au sein de l'application répartie. De plus, la syntaxe

du langage XML est claire et facile à interpréter. En outre, la disponibilité d'outils et d'apis pour traiter du XML rend possible d'accéder à un Document XML à travers différents langages en se basant sur des interfaces de programmation standardisées. Par exemple, l'API DOM [1] permet de construire un objet en mémoire de la totalité d'un document XML. Cette API permet l'accès direct à tous les nœuds de l'arbre (éléments, texte, attributs), pour les lire ou les modifier.

Pour ce faire, nous avons implanté un schéma XML pour mettre en évidence les structures de traces possibles tout au long du processus d'adaptation. Comme nous l'avons déjà introduit, nous nous intéressons aux changements (ajout ou suppression) subis par les threads, les connexions et les sous-programmes.

Le listing 3.1 est un extrait de ce schéma représentant la structure globale de trace : une balise racine décrivant l'action d'ajout ou de suppression et une sous balise décrivant l'élément modifié.

Listing 3.1 – Extrait du schéma XML décrivant la structure globale de la trace

```
1  <xs:element name="add">
2      <xs:complexType>
3          <xs:choice>
4              <xs:element name="thread" type="commonElementType"/>
5              <xs:element name="spg" type="spgElementType"/>
6              <xs:element name="connexion" type="connexionElementType"/>
7          </xs:choice>
8      </xs:complexType>
9  </xs:element>
10 <xs:element name="delete">
11     <xs:complexType>
12         <xs:choice>
13             <xs:element name="thread" type="commonElementType"/>
14             <xs:element name="spg" type="spgElementType"/>
15             <xs:element name="connexion" type="connexionElementType"/>
16         </xs:choice>
17     </xs:complexType>
18 </xs:element>
```

Pour les éléments objets de modifications, nous avons fixé pour chacun d'eux un ensemble de paramètres qui seront nécessaires pour accomplir l'adaptation.

· **Thread**

Le thread partage les mêmes attributs communs avec les autres entités. Il est déclaré, comme l'indique le listing 3.2 en tant qu'élément commun possédant deux attributs : le nom de l'instance et le classifieur correspondant décrivant son type dans la représentation graphique.

Listing 3.2 – Extrait du schéma XML décrivant un changement d'un thread

```
1
2  <xs:complexType name="commonElementType">
3      <xs:attribute name="name" type="componentNameType" use="required"/>
4      <xs:attribute name="classifier" type="componentNameType" use="optional"/>
5  </xs:complexType>
```

Pour spécifier les paramètres d'une connexion ou d'un sous programme, nous partons des attributs communs déjà déclarés et nous ajoutons les attributs spécifiques à chaque type.

· **Sous programme**

1. Document Object Model, www.w3.org/DOM/

Il s'agit de l'ajout ou la suppression d'une instance d'un sous programme ou d'un appel à un sous programme. Puisqu'un sous programme est appelé soit par un thread soit par un autre sous programme, nous avons ajouté à la base des attributs les paramètres suivants : le type de l'action `type` (`instance` ou `call`), s'il s'agit d'un appel, nous ajoutons les attributs `callerType` décrivant le type de l'appelant (Thread ou sous programme), le nom de l'appelant `callerName` et le nom de l'instance `instance`. Le listing 3.3 présente un extrait du schéma XML décrivant la structure du message correspondant à un changement d'un sous programme.

Listing 3.3 – Extrait du schéma XML décrivant un changement d'un sous programme

```
<xs:complexType name="spgElementType">
    <xs:complexContent>
        <xs:extension base="commonElementType">
            <xs:attribute name="callerType" type="componentCallType" use="optional"/>
            <xs:attribute name="callerName" type="componentNameType" use="optional"/>
            <xs:attribute name="instanceName" type="componentNameType" use="optional"/>
            <xs:attribute name="type" use="required">
                <xs:simpleType >
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="instance"/>
                        <xs:enumeration value="call"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

· **Connexion**

Pour décrire une connexion, il suffit de décrire le port source `sourcePort`, le port destination `destPort`, le composant source `compSource` et le composant destination `compDest`. Le listing 3.4 présente un extrait du schéma XML décrivant la structure du message correspondant à un changement d'une connexion.

Listing 3.4 – Extrait du schéma XML décrivant une modification d'une connexion

```
<xs:complexType name="connexionElementType">
   <xs:complexContent>
      <xs:extension base="commonElementType">
         <xs:attribute name="sourcePort" type="portNameType" use="required"/>
         <xs:attribute name="destPort" type="portNameType" use="required"/>
         <xs:attribute name="compSource" type="componentNameType" use="required"/>
         <xs:attribute name="compDest" type="componentNameType" use="required"/>
      <xs:extension>
   </xs:complexContent>
</xs:complexType>
```

Afin de contrôler notre système tout au long de son évolution dans les deux sens, nous intégrons ce mécanisme de surveillance dans notre approche. Après avoir généré correctement les traces décrivant cette évolution, nous devons déclencher une adaptation quand elle est demandée et l'accomplir sur les deux niveaux conceptuel et exécutif.

Dans la section suivante, nous allons détailler les différentes étapes d'adaptation dynamique dans les deux sens.

3.6 Adaptation dynamique du système

Suite à une surveillance continue du système, nous pouvons déclencher une adaptation quand elle est nécessaire. Cette adaptation doit être accomplie dynamiquement dans le sens concerné.

3.6.1 Sens1 : Adaptation du modèle à travers l'exécution du système

L'adaptation dans ce sens est basée sur le traçage en ligne en utilisant le mécanisme des aspects. Comme nous avons déjà mentionné, si nous allons intercepter toute l'application, nous associons des aspects génériques au code fonctionnel. Si non, après avoir sélectionné la partie cible d'interception avec AO4AADL au niveau du modèle, nous générons les aspects spécifiques correspondants en utilisant le générateur d'Ocarina.

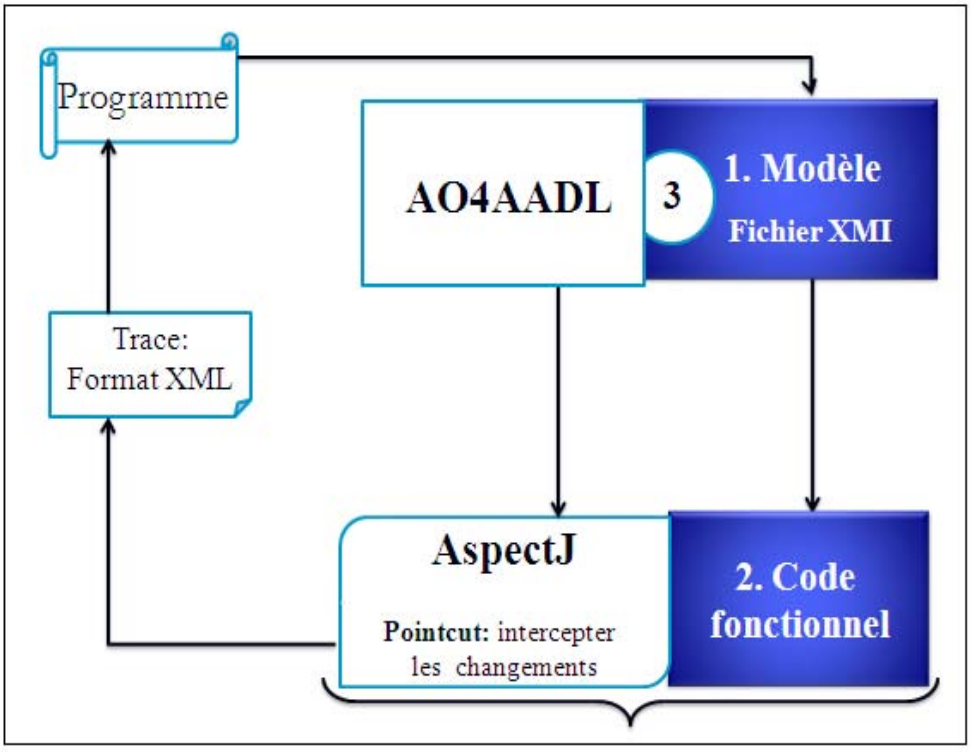

Figure 3.3 – Adaptation du modèle à partir de l'exécution (Sens1)

Ces aspects, qui sont soit génériques soit spécifiques, ont la charge de détecter les changements au niveau de l'exécution et d'agir sur le modèle. Le pointcut permet d'intercepter les changements des entités en question (Thread, sous programme et connexion) et d'en extraire les paramètres correspondants. L'advice permet tout d'abord de construire la trace décrivant cette modification sous la forme d'un message XML conformément au schéma XML prédéfini dans la section précédente. Ensuite, il l'envoie vers la machine concernée.

Pour le faire, nous avons défini un protocole qui assure la création, l'envoi et la réception des messages au sein de l'application distribuée. Au niveau du modèle, le protocole doit assurer une réception fiable et correcte ce qui peut être réalisé par l'intermédiaire d'un programme. Ce dernier reçoit le message, l'analyse en utilisant des parseurs et en extrayant les paramètres décrivant le changement puis agit sur le modèle en tant que fichier XMI. Nous avons choisi d'agir sur le modèle comme fichier XMI car c'est plus facile de traiter un tel type de fichier que de manipuler des représentations graphiques.

C'est pour cette raison que nous avons imposé la traduction du modèle AADL en fichier XMI lors du développement de notre propre éditeur « AADL Graphical Editor ». Réellement, il s'agit d'ajouter ou de supprimer des balises dans ce fichier puis le modèle graphique sera rafraîchi automatiquement.

La figure 3.3 donne une vue d'ensemble de l'adaptation dans ce sens.

3.6.2 Sens2 : Adaptation de l'exécution à travers le modèle

Concernant l'adaptation dans le deuxième sens, nous partons des changements effectués manuellement au niveau du modèle en ajoutant ou en supprimant des nouveaux composants ou des connexions. Notre modèle est décrit avec AADL tout en intégrant des aspects AO4AADL permettant d'une part de décrire les préoccupations transversales et d'autres part de choisir la partie du système qu'on va surveiller. Notre propre éditeur est conçu afin de permettre cette intégration.

En outre, il a été étendu afin de supporter une telle détection automatique des modifications réalisées sur le modèle. Nous avons étendu cet éditeur dans le but de lui greffer une bibliothèque de fonctions permettant de capturer les changements subis par le modèle. Il s'agit de l'appel des méthodes « `hooks` » lors de la modification manuelle du modèle.

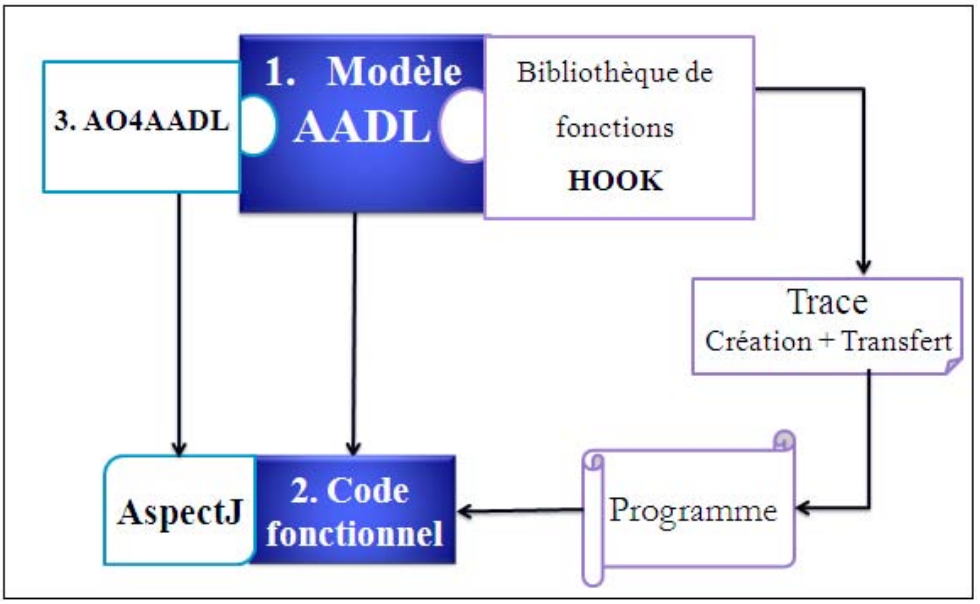

Figure 3.4 – Adaptation de l'exécution à partir du modèle (Sens2)

Un `hook` (appelé encore « `crochet` ») permet à l'utilisateur de personnaliser le fonctionnement d'un logiciel en lui faisant réaliser des actions supplémentaires à des moments bien déterminés. Le concepteur doit prévoir les `hooks` durant le fonctionnement de son logiciel afin de laisser des points d'entrées vers des listes d'actions. Par défaut, un `hook` est généré vide. Toutefois, le concepteur dans le but de personnaliser son logiciel doit greffer des morceaux de code au niveau des points d'entrées concernés.

Ces méthodes `hooks` doivent dans notre cas créer la trace traduisant un changement sous la forme d'un message XML toujours conformément au schéma XML prédéfini et puis de l'envoyer vers l'exécution via notre protocole. A ce niveau, un programme et chargé de recevoir le message, de l'analyser et d'agir sur l'exécution.

La figure 3.4 donne une vue d'ensemble de l'adaptation dans ce sens.

3.6.3 Protocole de communication

L'idée de ce protocole est née suite à l'apparition de quelques nouveaux besoins. Nous cherchons à offrir une bibliothèque de fonctionnalités nécessaires à la gestion d'une part de l'exécution d'une application et d'autre part de son modèle. Cette bibliothèque sert à tracer l'évolution de l'exécution en utilisant le mécanisme des aspects. Elle permet aussi d'appliquer les mises à jour nécessaires suite à une modification du modèle architectural. Afin d'assurer la correspondance bidirectionnelle entre le modèle et l'exécution d'un système réparti, nous avons besoin de mettre en place un mécanisme permettant la communication et l'échange de messages XML entre les différentes entités.

Il est à noter que le protocole à développer suppose que le système soit distribué. Nous proposons que le système s'exécute soit sur une seule machine soit d'une manière répartie sur plusieurs nœuds communicants dont chacun effectue une tâche particulière. Pour le modèle AADL, il est supposé présent sur une seule machine distante. Dés lors, notre protocole doit assurer :

1. La création des messages XML décrivant les changements au niveau de l'exécution aussi bien qu'au niveau du modèle en se référant au schéma XML défini pour décrire la structure détaillée du message.
2. L'établissement de connexion entre les différentes entités communicantes du système distribué et le transfert de ces messages.
3. La réception fiable et correcte des messages. Pour le faire, nous devons vérifier la structure du message construit avant l'envoi et du message transféré après la réception.
4. L'accomplissement de l'adaptation à travers des programmes assurant l'application des règles de transformation du modèle ou d'exécution sur les deux niveaux conceptuel et exécutif.

Ainsi, notre protocole permet de faciliter et d'ordonner l'adaptation du système en cours d'exécution dans les deux sens.

3.7 Conclusion

Dans ce chapitre, nous avons défini un processus complet permettant l'adaptation dynamique des systèmes à base de composants en se basant sur la modélisation @Runtime.

Nous avons détaillé toutes les phases du processus. Nous avons commencé tout d'abord par concevoir notre modèle avec AADL en intégrant des aspects AO4AADL en utilisant notre propre éditeur. Ensuite, nous avons adopté un prototype de génération automatique du code pour avoir une application qui tourne sur une plateforme donnée. Finalement, nous avons mis en place un mécanisme de surveillance et d'adaptation du système en cours d'exécution dan les deux sens.

Nous présentons, dans le chapitre suivant, une mise en œuvre de notre approche.

Chapitre 4

Mise en oeuvre de notre approche

4.1 Introduction

Dans le chapitre précédent, nous avons détaillé notre approche. Dans ce chapitre, afin de rendre possible l'utilisation de notre approche, nous passons à sa mise en œuvre.

Nous présentons dans un premier temps les outils et les langages de programmation utilisés pour la mise en œuvre de notre approche. Ensuite nous passons à la réalisation de notre propre éditeur. Puis nous nous intéressons à la mise œuvre du protocole. Finalement, nous clôturons par une conclusion pour ce chapitre.

4.2 Outils et langages

Pour la génération automatique du code, nous avons utilisé la suite d'outils « Ocarina ». Cet outil permet de générer non seulement le code fonctionnel correspondant au modèle AADL, mais aussi le code des aspects AO4AADL. Il est à la base d'un intergiciel minimal appelé POLYORB_HI.

Dans cette section, nous présentons l'outil Ocarina et l'intergiciel POLYORB_HI. Puis nous passons à la description des différents langages utilisés pour la mise en œuvre de notre approche à savoir AO4AADL, RTSJ, AspectJ et XMI.

4.2.1 Ocarina

Ocarina [Zal08, LZPH09] est une suite d'outils libres développée et maintenue par l'équipe S3 de TELECOM ParisTech depuis 2003. Elle permet la manipulation, l'analyse syntaxique et sémantique, la vérification et la génération des modèles AADL.

Ocarina, formée de trois bibliothèques principales, admet une architecture très similaire à celle d'un compilateur moderne comme le montre la figure 4.1.

– Une bibliothèque centrale qui permet d'élaborer et de manipuler les arbres syntaxiques à travers des routines de construction génériques et spécifiques. Précisément, il s'agit d'abstractions de bas niveau pour gérer les chaînes de caractères, les fichiers et les arbres syntaxiques.

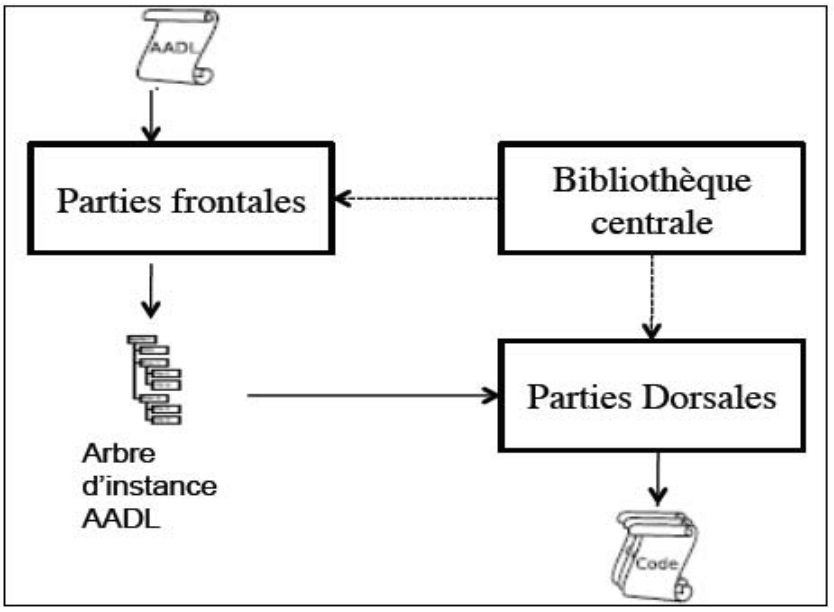

Figure 4.1 – Architecture globale d'Ocarina [Lou10]

– Un ensemble de parties frontales qui permettent une analyse syntaxique et sémantique des fichiers sources décrivant les modèles AADL ainsi que leurs annexes en se basant sur les routines de la bibliothèque centrale.
– Un ensemble de parties dorsales qui permettent d'une part de vérifier les modèles et d'autre part de générer le code en se basant sur les arbres produits par les parties frontales.

4.2.2 POLYORB-HI

POLYORB-HI [HZ07] est un intergiciel minimal dont l'architecture est fortement influencée par celle de l'intergiciel schizophrène POLYORB [1]. Il supporte toutes les constructions AADL nécessaires pour la production d'une application temps réel embarquée à base de composants. Il possède une faible empreinte mémoire. A partir du modèle AADL, il est configuré automatiquement . Toutes les ressources sont calculées et allouées d'une manière statique sans avoir besoin d'intervention humaine.

PolyORB-HI couplé à Ocarina, la suite d'outils décrite précédemment, fournissent un ensemble de constructions de base d'un AADL : Ils offrent des bibliothèques (de gestion des tables de routage, des messages, buffers nécessaires pour stocker les messages, instances des tâches, etc) disponibles pour les différents langages supportés par Ocarina comme C et Ada et java.

Nous avons eu recours pour le développement du protocole aux fonctionnalités offertes par PolyORB_HI_JAVA : une version de PolyORB_HI supportant la spécification java temps réel.

1. http ://polyorb.ow2.org/

4.2.3 Langages de programmation

Pour concevoir notre modèle, nous avons choisi d'utiliser AADL comme langage de description d'architecture comme nous l'avons décrit dans le chapitre 1. Ce modèle peut être enrichi par le langage AO4AADL : extension d'AADL par les concepts d'aspects.

Suivant le processus de développement proposé dans le chapitre précédent, le code fonctionnel correspondant au modèle AADL sera généré en RTSJ et le code des aspects AO4AADL sera généré en AspectJ. Puis, la manipulation du modèle pendant la surveillance et l'adaptation se base sur sa représentation XMI. Dans cette section, nous allons décrire ces différents langages utilisés.

- **AO4AADL**
  Ce langage a été proposé dans le cadre des travaux de mastère de Sihem LOUKIL [LKZJ10] au sein de l'unité de recherche ReDCAD. C'est un langage orienté aspect proposé dans le but d'étendre le langage AADL en utilisant les annexes. Cette contribution considère qu'un aspect peut être spécifié avec un langage autre que AADL et par la suite intégré dans des modèles AADL via les annexes. La définition d'un tel langage de description d'architecture orienté aspect est destinée à fournir un support linguistique à l'établissement de l'architecture des systèmes orientés aspects. Dans ce cas, la description architecturale orientée aspect d'un système en utilisant AADL et AO4AADL peut être traduite en des langages de programmation impératifs puisque AO4AADL est un langage générique. Ainsi, on obtient une plateforme d'exécution permettant de vérifier si le comportement du système est celui prévu ou non. Les aspects décrits en AO4AADL peuvent être traduits en différents langages d'aspects tels que AspectJ pour le langage Java, AspectAda pour le langage Ada, AspectC++ pour le langage C/C++, etc.
- **RTSJ (Real Time Specification for Java)**
  RTSJ [Aut09] est un langage qui étend le langage Java pour supporter les systèmes temps réel matériels et logiciels. Il permet une programmation orientée objet en introduisant des nouvelles caractéristiques appropriées à la gestion des mémoires et des threads. Il gère tous les types de threads (Sporadique, périodique et hybride) avec vérification de respect des échéances. Il permet également d'attribuer des priorités à ces threads en se basant sur un système d'exploitation temps réel. En outre, RTSJ offre un mécanisme d'allocation dynamique de mémoire. Ainsi, le programmeur peut gérer l'allocation et la collection d'objets sans faire appel au ramasse-miettes (*garbage collector*). En effet, RTSJ propose un autre type de thread appelé NoHeapRealtimeThread qui permet d'allouer les objets dans des régions de mémoire appelées ScopedMemory. Lors de la destruction de la région, tous les objets alloués par un thread sont alors libérés.
- **AspectJ**
  AspectJ [KHH+01] est un langage qui étend le langage Java pour supporter la programmation orientée aspect. La communauté d'aspects le considère comme le langage d'aspect le plus connu et le plus utilisé. Ce langage permet un tissage statique. Il définit l'aspect sous la forme d'une classe Java. Lors de la

compilation, le compilateur AspectJ génère du bytecode Java standard résultat de l'opération de tissage du code fonctionnel et du code d'aspect. Ainsi nous pouvons manipuler tous les concepts de la programmation orientée aspect grâce aux nouveaux concepts offerts par AspectJ.

– **XML et API JDOM**
Nous avons choisi XML [2] comme format d'échange et de création des messages manipulés par le protocole développé. Ce choix revient de la simplicité de ce langage. XML est un standard d'échange de données. Il offre une syntaxe claire et facile à interpréter. Ceci est grâce à la disponibilité d'outils et d'API permettant le traitement du XML pour différents langages comme le standard JDOM (*Java Document Object Model*) pour le langage Java. Cette API fournit une représentation mémoire d'un document XML sous la forme d'un arbre d'objets pour aider à sa manipulation selon des interfaces de programmation standardisées Java.

4.3 Editeur graphique pour la description AADL

Afin de répondre à nos besoins liés au modèle AADL intégrant des aspects AO4AADL, nous avons décidé de développer notre propre éditeur pour ce langage sous la forme d'un plugin Eclipse.

Dans cette section nous présentons Eclipse, puis nous définissons la notion de plugin et les étapes de son développement et nous terminons par la présentation des différents plugins utilisés lors de l'implantation de notre plug-in.

4.3.1 Eclipse

Eclipse [3] est une plate-forme universelle pour des environnements de développement intégrés fondée sur une architecture générique, ouverte et extensible. Il s'agit d'une plateforme : :

– Générique puisqu'elle est indépendante de tout langage. Elle peut être utilisée pour développer des applications en Java et, par l'intermédiaire de différents plugins, d'autres langages de programmation, comme Ada, C, C++, COBOL, etc.
– Ouverte parce qu'elle est offerte sous licence Eclipse Public License (EPL). Son code source est libre et peut être redistribué.
– Extensible puisqu'il s'agit d'un Framework offrant la possibilité de construire et d'intégrer des plugins de toute nature.

Eclipse est donc un environnement de développement intégré IDE (Integrated Development Environment) conçu dans le but de fournir une plateforme portable et modulaire. Eclipse peut donc être vu comme une collection de points d'extension, destinée à recueillir des modules supplémentaires appelés plugins.

2. eXtented Markup Language, www.w3.org/XML
3. www.eclipse.org

4.3.2 Plugin Eclipse

Un plugin est un ou plusieurs fichiers jar présentant un bloc fonctionnel (jar) en dehors du noyau. Il permet d'étendre la plateforme d'Eclipse et de fournir aux utilisateurs la possibilité d'intégrer de nouvelles fonctionnalités afin de répondre à leurs besoins. En effet, l'architecture d'Eclipse est basée sur le concept de plugins puisque la plateforme est formée d'un noyau (Runtime) et tout le reste est développé sous la forme de plugins. Pour créer un plugin et l'intégrer dans Eclipse, on passe généralement par cinq étapes :

1. Création d'un projet plugin : il s'agit de créer un nouveau projet « *Plugin Project* ». Les répertoires et les fichiers nécessaires sont automatiquement créés.
2. Configuration du fichier plugin.xml : cette phase consiste à définir les points d'extensions ainsi que les bibliothèques utilisées dans le développement du plugin.
3. Développement du code : dans cette étape l'utilisateur développe le code fonctionnel de son plugin permettant d'atteindre ses objectifs.
4. Exportation du plugin : une fois le plugin développé, nous pouvons l'exporter sous la forme d'un archive jar nécessaire pour une ultérieure intégration dans Eclipse.
5. Intégration du plugin : après l'exportation du plugin, nous l'intégrons dans Eclipse en décompressant l'archive dans le répertoire « plugins » d'Eclipse.

4.3.3 Développement d'un plugin GMF

La création d'un éditeur graphique sous Eclipse exige l'utilisation des frameworks EMF(*Eclipse Modelling Framework*), GEF(*Graphical Editing Framework*) et GMF(*Graphical Modeling Framework*). GMF permet la réalisation et la génération d'un éditeur graphique. Il utilise EMF permettant de créer le méta-modèle et GEF permettant de gérer la côté graphique.

◇ **Eclipse Modeling Framework (EMF)**

EMF est un Framework Java permettant, grâce à la génération de code, de définir et mettre en œuvre des modèles de données structurés. Un modèle de données structuré est simplement un ensemble de classes reliées traduisant les données qu'on souhaite traiter dans l'application à réaliser. EMF est constitué de deux frameworks essentiels : le framework `Core` permettant la génération de code basique, et le framework *Runtime* permettant la génération des implantations des classes Java pour un modèle donné. Il contient également un support par défaut pour le stockage persistant et la récupération des modèles dans des documents XML ou XMI. De plus, EMF offre un langage de description de modèles (méta-modèle) en format « `Ecore` ».

◇ **Graphical Editing Framework (GEF)**

GEF est un Framework open source dédié à offrir un environnement graphique riche et cohérent pour les applications d'édition sur la plate-forme Eclipse. Il fournit ainsi une architecture solide pour la création d'éditeurs visuels de modèles de données arbitraires. Ces éditeurs offrent des fonctionnalités d'édition faciles pour des données dans des domaines spécifiques. L'efficacité de GEF réside dans sa construction modulaire adaptée à l'utilisation des patrons de conception (*design patterns*). GEF se

comporte donc comme « `observer pattern` » en cas de changement de l'état d'un objet, dés qu'il détecte un changement il exerce l'action appropriée. GEF est scindé en deux parties.

- Un ensemble d'outils de dessin permettant de faciliter le dessin des différentes formes géométriques *org.eclipse.draw2d).*
- Un Framework MVC (Modèle-Vue-Contrôleur) interactif, construit au-dessus du plugin `Draw2d`. Le Framework MVC est un modèle de conception architecturale qui divise une application en parties distinctes qui communiquent entre eux d'une manière spécifique. Ceci est dans le but de séparer le modèle de données (modèle généré à l'aide d'EMF), l'interface graphique (Vue contenant les figures correspondantes aux éléments du modèle) et la logique métier (Contrôleur permettant l'édition de la représentation graphique à travers les classes « `xxxEditPart` »).

Par conséquent, le plugin GEF est destiné à ajouter des outils de sélection et de création et un outil palette, etc. De plus, il utilise EMF d'une manière transparente pour faire la mapping entre un modèle de données en format XMI, une classe Java et la représentation graphique.

◇ **Graphical Modeling Framework (GMF)**

GMF est un outil de construction d'éditeurs graphiques développé pour la plateforme Eclipse. GMF permet la génération des composants ce qui permet le développement rapide des éditeurs graphiques basés sur EMF et GEF. Le Framework fournit les fonctionnalités de base prévu pour un éditeur graphique générique comme les menus pour les commandes, une zone pour visualiser les figures, un outil de dessin palette, etc. Il prend peu de fichiers de configuration en entrée et génère un plug-in qui fonctionne sur Eclipse comme un éditeur graphique de qualité professionnelle. Dans l'éditeur, les diagrammes, basés sur un langage spécifique au domaine, peuvent être conçus et le résultat peut être sérialisé en une représentation XMI.

Dans notre cas, le langage spécifique au domaine conceptuel est AADL, qui se compose de modèles sémantiques comme les composants, leurs interactions (les connexions, les ports) et les hiérarchies entre les composants. Ces modèles sémantiques exigent les représentations graphiques qui constituent le schéma de sortie de notre éditeur graphique. Ainsi, ces modèles sémantiques constituent le pivot du processus de développement de l'éditeur graphique avec GMF. Nous avons créé, dans le cadre d'un projet de fin d'études, un éditeur graphique utilisant GMF, appelé « AADL Graphical Editor ». Il permet aux utilisateurs de concevoir graphiquement des architectures d'applications avec AADL.

En fait, GMF rend le développement d'un éditeur graphique sous forme de plugin Eclipse rapide. En effet la génération automatique de code à partir du modèle est très utile et fait gagner beaucoup de temps au concepteur. Elle permet aussi de créer une application stable.

Toutefois, le développement d'un plugin GMF suit les conventions de MVC ce qui rend le fonctionnement de ce système assez complexe. En outre, GMF présente un outil efficace pour réaliser des applications simples, mais lorsque nous voulons intégrer des fonctionnalités qui sortent des astuces ordinaires, nous heurtons rapidement quelques difficultés. De plus, nous perdons complètement le contrôle sur le code généré. En d'autres termes, il est difficile d'aller modifier les classes générées

au risque de créer des effets de bord non souhaités.

4.3.4 Création de l'éditeur « AADL Graphical Editor »

Pour la création de notre éditeur, nous avons adopté le plugin GMF qui présente actuellement un framework clé pour le développement rapide des éditeurs de modélisation graphique. Il fournit deux composantes :

- Une composante de génération (*Generative Component*) qui permet au développeur de générer automatiquement le code source Java à partir d'un ensemble de fichiers de configuration. Ce code peut être par la suite enrichi par le développeur pour répondre à tous les détails fins des exigences de l'utilisateur.
- Une composante d'exécution (*Runtime Component*) qui fournit des APIs utilisées par le code Java généré à travers la composante de génération au cours de son exécution. Elle fournit d'autres fonctions sophistiquées telles que la persistance des diagrammes.

◇ **Etude théorique**

Pour arriver à créer un éditeur graphique utilisant le plugin GMF d'Eclipse, nous avons suivi un ensemble d'instructions. Ce processus est composé de trois étapes comme l'indique la figure 4.2 :

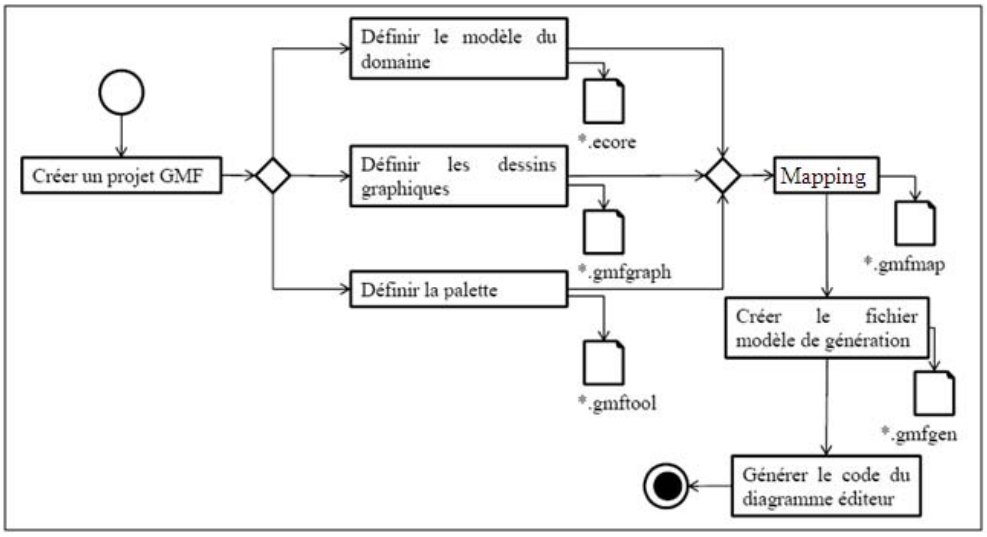

Figure 4.2 – Processus de développement d'un plugin GMF

1. La création d'un nouveau projet GMF.
2. La seconde étape est scindée en trois phases :
   (a) Définition du modèle du domaine par la création du fichier de configuration (`*.ecore`) définissant le méta-modèle de la grammaire définit par la langue du domaine spécifique.
   (b) Définition des dessins graphiques par l'intermédiaire du fichier (`*.gmfgraph`). Ce fichier contient la définition des formes graphiques associées aux différentes entités du méta-modèle. La définition graphique

nous permet de décrire les formes, leurs couleurs, leurs types (nœuds ou connexions), etc.

(c) Définition de la palette à travers le fichier de configuration (`*.gmftool`). Ce fichier permet de définir les outils de la palette permettant à l'utilisateur de concevoir un modèle dans la zone de dessin.

3. Le mapping présentant l'épine dorsale du processus de développement du plugin GMF. Il s'agit de définir le fichier (`*.gmfmap`) permettant d'établir la correspondance entre une méta-classe du méta-modèle du modèle de domaine, une représentation graphique et un élément de la palette. Tous les fichiers de configuration ainsi créés stockent les données à un niveau élevé non convenable à une génération directe du code. Ainsi un modèle générateur (`*.genmodel`) est créé, qui combine les données provenant des quatre fichiers déjà créés et remplit des détails liés à la génération comme nom des classes ou des packages.

En suivant ces différentes étapes, nous arrivons à générer le code de notre éditeur qui est annoté par « `@generated` ». Toute modification du code généré ou ajout d'un fragment de code nécessite l'annotation « `@generated NOT` »pour conserver les modifications en cas de régénération du code.

Les détails liés aux étapes d'implantation de notre plugin GMF sont décrits dans la section suivante.

◇ **Réalisation**

Pour la réalisation de notre éditeur, nous avons suivi les étapes de réalisation d'un projet GMF décrites dans la section précédente.

Après avoir créé un projet que nous l'appelons « AADLEditor », nous suivons le reste du processus étape par étape.

1. **Définition du modèle du domaine**

   La conception de notre éditeur graphique consiste principalement à la conception du méta-modèle pour le modèle d'instance d'AADL (*instance model*). Un modèle du domaine « `AADL.ecore` » est créé pour représenter le méta-modèle qui définit la manière dont le modèle construit à l'aide de l'éditeur est stocké en mémoire. Notre éditeur graphique pour AADL s'appuie sur un méta-modèle pour avoir des bases solides et rigoureuses. Il s'agit d'un modèle définissant les éléments du modèle d'instance d'AADL. Les règles du standard AADL se décrivent à l'aide du méta-modèle. Nous avons développé notre méta-modèle en utilisant EMF avec la notation `Ecore`, c'est une notation de méta-modélisation appelé méta Ecore qui est largement utilisé ailleurs. Notre méta-modèle est représenté comme un ensemble de classes avec l'addition des propriétés EMF spécifiques. Ces propriétés EMF soutiennent la génération automatique des méthodes de la manipulation des modèles d'instance pour AADL. De plus, elles permettent le stockage persistant et la récupération de ces modèles dans des documents XMI.

2. **Définition des dessins graphiques**

   Dans cette étape, nous nous intéressons à la définition des représentations graphiques de chaque élément de notre éditeur. Nous avons défini deux fichiers « `Components.gmfgraph` » pour les composants et « `Features.gmfgraph` » pour les éléments d'interface. La figure 4.3

présente ces deux fichiers. Chacun de ces fichiers montre le contenu de la définition des représentations graphiques sous la forme d'un arbre.

· *Figure Gallery Default* : Ce type d'élément (cerclé en rouge) contient des sous-nœuds décrivant les propriétés graphiques (formes géométriques : rectangle, ellipse, polygone, couleur, largeur, longueur, etc.) de chaque entité de l'éditeur.

· *Node* : Cet élément rassemble les différentes propriétés graphiques, associées à un élément dans le diagramme, dans un nœud (`Node system, Node process, Node thread, Node dataPort, Node EventPort`, etc.) qui sera utilisé par la suite dans le mapping.

· *Connection* : Ce type d'élément présente soit une connexion établie entre les éléments d'interface ou une séquence d'appels de sous-programmes (Connection `connection`, Connection `CallSequence`).

· *Compartment* : Ce type constitue une boîte permettant à un nœud de contenir d'autres nœuds. Par exemple, le composant system doit avoir un compartiment (CompartmentSystem_compartment) pour pouvoir contenir des sous-composants.

· *Diagram Label* : Ce type d'élément permet d'attribuer des étiquettes présentant les noms des entités.

3. **Définition de la palette**

   Les outils utilisés pour créer les nœuds et les connexions dans la zone de dessin de l'éditeur sont définis dans le fichier de définition d'outils. Pour cela nous avons créé le fichier `AADLTool.gmftool`. Ce fichier est présenté dans la figure 4.4.

   Comme le montre la figure, nous avons classé les outils en quatre groupes :

   · *Tool Group Components* : ce groupe contient tout les outils permettant de créer des composants et leurs sous-composants.

   · *Tool Group Calls* : ce groupe contient deux outils (« `Creation tool Subprogram` » et « `Creation tool Call Sequence` ») permettant de créer des sous-programmes et de définir des séquences d'appels entre les sous-programmes créés.

   · *Tool Group Features* : ce groupe est constitué des outils permettant la création des différentes catégories d'éléments d'interface. Pour chaque catégorie nous avons défini toutes ses directions possibles (`in, out, inout`).

   · *Tool Group Connections* : il regroupe les outils permettant de créer les différents types de connexions : « `Creation tool Data Connection` », « `Creation tool Event Connection` », etc.

4. **Mapping**

   Une fois le modèle du domaine (`AADL.ecore`), la définition graphique (`Components.gmfgraph` et `Features.gmfgraph`) et la définition de la palette (`AADLTool.gmftool`) sont définis, l'étape suivante consiste au mapping entre ces fichiers. Il s'agit d'établir la correspondance entre une méta-classe du méta-modèle, une représentation graphique et un élément de la palette. Nous associons donc à chaque élément de notre éditeur une méta-classe du méta-modèle, la définition graphique correspondante ainsi que l'outil convenable de la palette. Si l'élément contient un compartiment, nous définissons aussi ses fils.

Figure 4.3 – Définition des représentations graphiques des éléments de l'éditeur

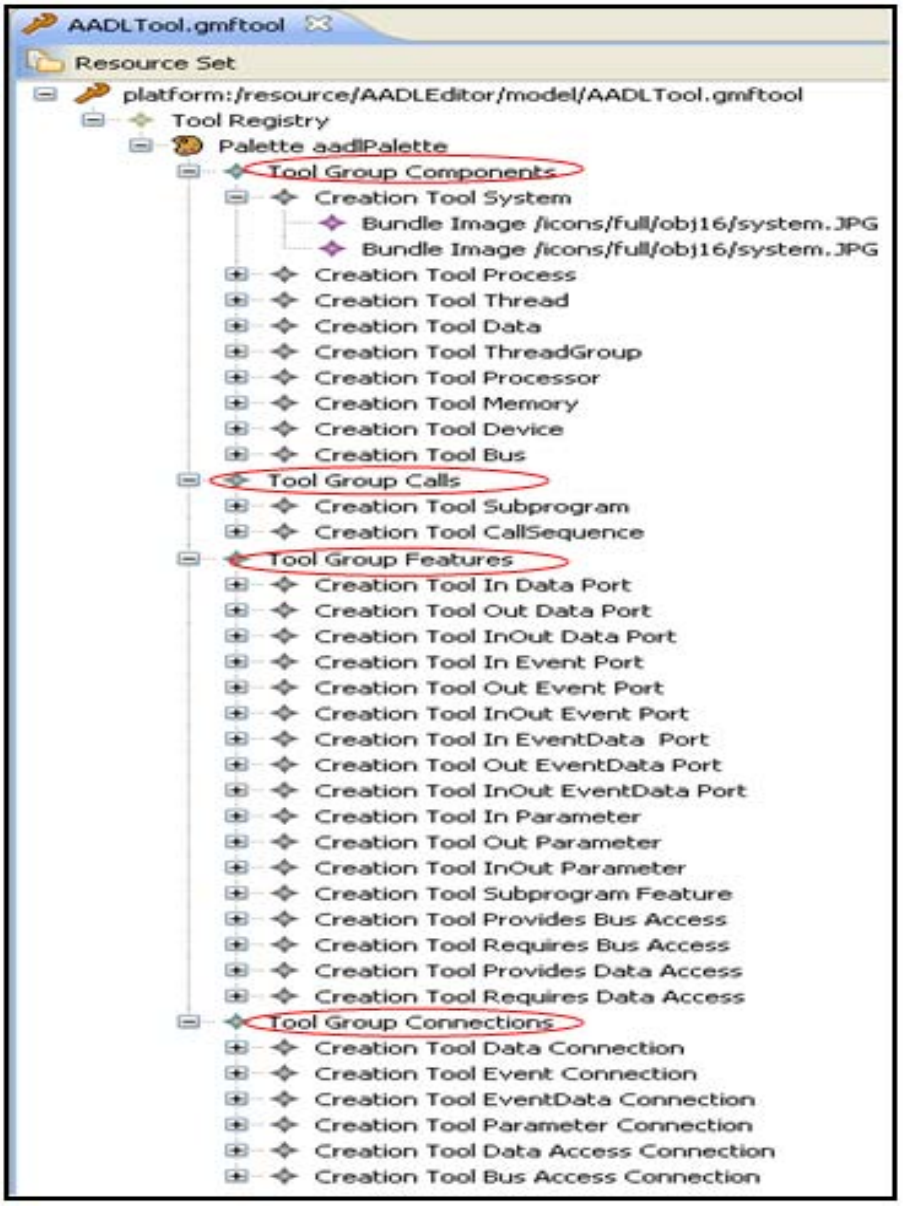

Figure 4.4 – Définition de la palette

Dans le mapping, nous pouvons définir des contraintes grâce au langage OCL (*Object Constraint Language*). Nous avons utilisé les contraintes OCL pour définir des mappings multiples pour la même méta-classe pour identifier chaque mapping d'une manière unique. Par exemple, les ports de données (`Data Port`) peuvent être en entrée, en sortie ou en entrée/sortie, mais nous avons une seule méta-classe « `DataPort` » ayant un attribut « `direction` » de type énumération (`IN, OUT, INOUT`). Donc à l'aide des contraintes OCL nous avons pu définir les types de port de donnée séparément. Il est de même pour les autres éléments d'interface.

5. **Création du fichier modèle de génération**
   Le fichier modèle de génération « `AADLMap.gmfgen` » est généré à partir

du fichier de mapping défini précédemment. Il contient tous les informations nécessaires pour la génération du code de l'éditeur.

6. **Génération du code de l'éditeur**
   A partir du fichier modèle de génération, le code de notre éditeur sera généré. Le plugin du diagramme éditeur « `AADLEditor.diagram` » sera créé contenant des classes java distribuées dans des paquetages. Ainsi, l'application peut être exécutée comme un plugin dans la plateforme Eclipse. L'interface graphique de notre éditeur généré se compose de cinq régions comme le montre la figure 4.5.

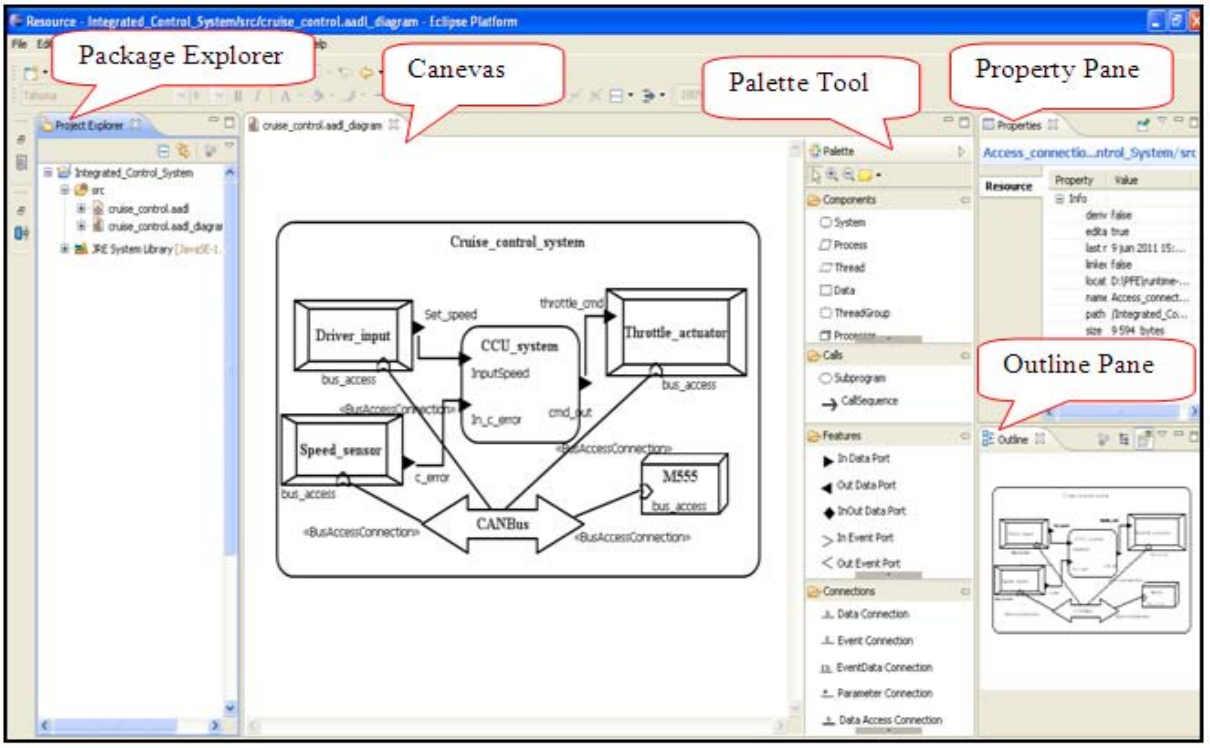

Figure 4.5 – Interface graphique de notre éditeur

· *Canevas* : c'est la zone où l'utilisateur conçoit les modèles à l'aide des outils de la palette. Un clic droit sur la toile produit un menu contextuel qui, entre autres caractéristiques, permet au concepteur d'enregistrer le diagramme dans différents formats d'image (`JPG, PNG, BMP`, etc.) via le menu « `File` → `Save As ImageFile` ».

· *Explorateur de package (Package Explorer)* : cette région contient les projets créés dans lesquels nous créons nos diagrammes.

· *Palette (Palette Tool)* : elle est divisée en quatre groupes `Components`, `Calls`, `Features` et `Connections`.

· *Aperçu des diagrammes (Outline Pane)* : il fournit un aperçu, zoom en avant, du diagramme dans la toile de sorte qu'un utilisateur peut directement pointer vers une partie du diagramme dans le volet d'aperçu des diagrammes et de travailler sur cette partie dans la toile.

· *Propriétés (Property Pane)* : la plupart des éléments de l'éditeur possèdent un ensemble de propriétés accessibles via la fenêtre des propriétés.

7. **Utilisation de l'éditeur**
   Dans cette section nous décrivons les étapes de conception d'un modèle AADL avec notre éditeur graphique.
   – Création d'un nouveau diagramme : au début, le concepteur crée un nouveau projet où il va concevoir et garder le modèle de son application. Ensuite, pour créer un nouveau diagramme, le concepteur ouvre dans le menu contextuel l'item Exemple. Une boîte de dialogue s'affiche présentant une liste de noms de diagrammes disponibles, le nom de notre diagramme `AADLDiagram` apparaît à la tête de cette liste. Le concepteur sélectionne `AADL Diagram` puis il choisit le nom de son modèle qui doit avoir l'extension « `*.aadl_diagram` » ainsi que le nom du fichier XMI associé au modèle ayant l'extension « `*.aadl` ».
   La figure 4.6 présente les étapes de création d'un nouveau diagramme.
   – Après la création d'un nouveau diagramme, l'interface de notre éditeur graphique s'affiche. L'utilisateur peut donc commencer la conception de son modèle. Tout d'abord, il commence par le dessin du système comme élément racine puis enrichit ce modèle avec des sous composants, des éléments d'interfaces, des appels à des sous programmes ou encore par des connexions entre les différents éléments constituant ce modèle.

Finalement, après l'enregistrement du modèle proposé, le concepteur peut le manipuler en accédant à la représentation graphique via l'éditeur ou en accédant au fichier XMI correspondant.

4.4 Mise en œuvre du protocole

Pour assurer une adaptation dynamique des systèmes embarqués à base de composants, nous avons eu recours à la modélisation @Runtime. Cette technique assure la mise à jour du modèle suite aux changements effectués au niveau de l'exécution et inversement. Avec ce lien de causalité entre le modèle et l'exécution, se rajoute la notion de système réflexif dont deux aspects sont reliés afin d'agir sur l'exécution.

Dans le but de mettre en évidence l'implantation de notre protocole, nous avons choisi d'utiliser le générateur RTSJ permettant la programmation orientée objet temps réel pour gérer l'envoi et la réception instantanés de messages. Nous avons utilisé également l'API `Reflexion` de Java permettant de faciliter l'adaptation du système au cours de son exécution.

Dans la suite de cette section, nous décrivons dans un premier lieu, le générateur RTSJ et l'API Reflexion. Dans un deuxième lieu, nous nous intéressons à l'implantation propre du protocole.

4.4.1 Générateur RTSJ

Le générateur RTSJ introduit le concept de l'orienté objet en utilisant le langage Java Temps-réel. Il permet de traduire les différents composants AADL en RTSJ.

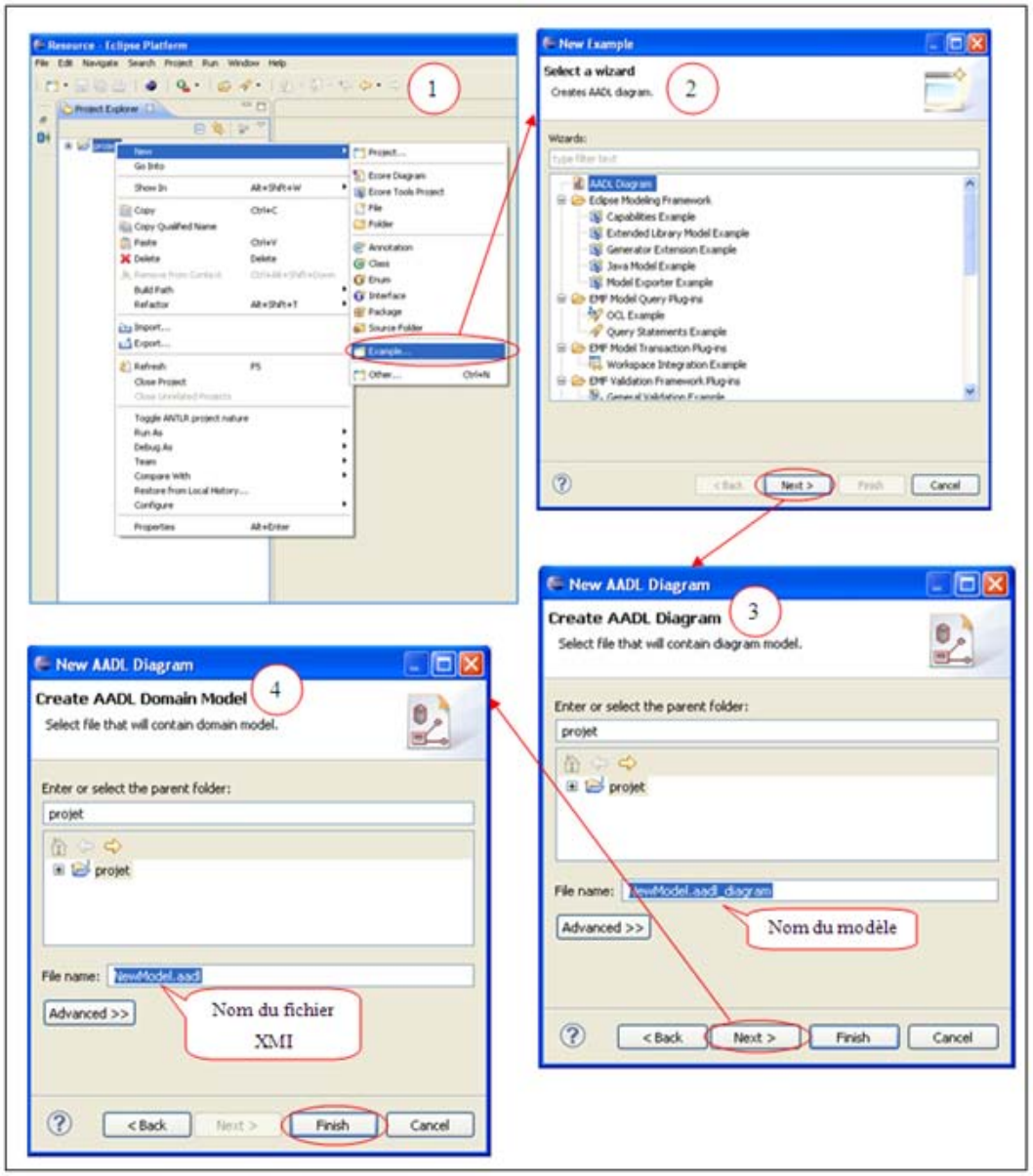

Figure 4.6 – Utilisation de notre éditeur

Ce mapping entre le langage de programmation RTSJ et le langage de description d'architecture AADL se base principalement sur la génération d'un ensemble de fichiers RTSJ [Aut09]. Parmi ces fichiers nous trouvons :*

· `Main.java` : ce fichier est utilisé pour l'initialisation de l'application.

· `Subprograms.java` : pour chaque nœud de l'application, une classe `SubPrograms` est générée contenant les différentes méthodes statiques qui se contentent d'appeler l'implantation de ce sous-programme fournie par l'utilisateur.

· `GeneratedTypes.java` : les types générés rassemble tous les types de données définis dans le modèle AADL. Pour chaque type de données du modèle AADL, une classe de type public static est générée avec le nom du type de donnée dans le modèle. La classe crée implante l'objet `GeneratedType`.

· `Activity.java` : ce fichier est utilisé pour déclarer et initialiser tous les threads

du nœud ainsi que la déclaration des ports. Le comportement des threads est implanté dans ce fichier.

· `Deployment.java` : ce fichier rassemble toutes les informations décrivant le nœud actuel et son interaction avec les autres nœuds de l'application.

4.4.2 La réflexion

Ce concept est utilisé afin de mettre à jour l'exécution suite à un changement au niveau du modèle. Tout ajout ou suppression d'un composant ou d'une connexion dans le modèle peut être concrétisé sur l'application en cours d'exécution en utilisant cette technique.

Définition

La réflexion est définie par *Brian Smith* comme suit : « *La réflexion est la capacité d'une entité à s'auto-représenter et plus généralement à se manipuler elle-même* » [Smi90].

La réflexion caractérise donc la propriété d'un système capable de raisonner et d'agir sur lui-même. Ainsi, un système réflexif est capable de contrôler son propre fonctionnement et d'adapter ses structures internes en fonction des nouveaux besoins internes et externes même pendant son exécution. Il existe deux techniques de base utilisées par les systèmes réflexifs qui sont :

1. **L'introspection** :
   Définit par Smith comme suit :« *Un système réfléchissant observant sa propre exécution* » [Smi90]. C'est donc la capacité d'un programme à examiner son propre état et à découvrir ces caractéristiques et sa composition. Autrement dit, obtenir des informations sur un type de données chargé en mémoire.
2. **L'intercession** :
   Définit par Smith par : « U*n système réfléchissant peut changer son exécution* » [Smi90]. Cette technique est plus utile que la première comme elle montre la capacité d'un programme à modifier son propre état de manière dynamique au moment d'exécution. L'intercession permet à un programme d'évoluer automatiquement en fonction des besoins.

Utilisation de la réflexion

La réflexion est un style de programmation dynamique qui apparaît dans plusieurs langages de programmation essentiellement les langages orientés objets comme java. L'architecture réflexive est implantée dans ces langages par le concept des métas donnés.

Définition des métas données : Ce sont des binaires (`bytecode`) crée à la compilation. Ils contiennent de nombreuses informations concernant tous les éléments d'un programme instancié en mémoire (comme les classes, les champs, les méthodes,etc) permettant de faire abstraction du contenu du code métier en cours d'exécution.

Principaux concepts utilisés dans l'API Réflexion

La structure d'un objet en java est caractérisée par des classes qui contiennent des constructeurs, des méthodes et des champs.

Le développeur peut également accéder à ces caractéristiques dans ses programmes et les manipuler grâce à des fonctionnalités spécifiques définit dans l'API `java.lang.reflect`. Ainsi cette librairie permet :

- L'accès à la structure des objets : le principe d'introspection est mis en place par plusieurs classes et méthodes prédéfinies dans cette API permettant de récupérer l'instance d'une classe, ses éléments (attributs, méthodes, etc) et ses caractéristiques (types des champs, type de la classe instanciée, etc).
- L'interaction avec des objets par accès et modification dynamique : le principe d'intercession est mis en œuvre par la présentation des méthodes spécifiques permettant d'avoir au moment de l'exécution la définition de n'importe quel objet et agir sur lui par invocation de ces méthodes et même par modification des valeurs de ses attributs.

4.4.3 Implantation du protocole

Comme nous l'avons déjà introduit dans le chapitre 3, notre protocole défini dans le but de satisfaire nos besoins, est implanté dans le cadre d'un projet de fin d'études. Afin d'assurer l'adaptation dynamique dans les deux sens, nous avons implanté ce protocole en tenant compte des modifications effectuées sur les deux niveaux conceptuel et exécutif.

Dans la suite de cette section, nous allons décrire cette implantation dans les deux sens.

Sens 1 : de l'exécution vers le modèle

L'implantation du protocole dans ce sens est basée sur l'interception par des aspects. Ce concept peut être exploité pendant le contrôle au niveau de l'exécution pour que tout changement soit répercuté sur le modèle. Partant de son modèle conceptuel, une application nécessite un suivi de son évolution pour des besoins de contrôle de la fiabilité du système généré ainsi que le bon fonctionnement de ces différents composants. On parle d'un mécanisme de traçabilité instantanée qui reflète l'état d'exécution. C'est le mécanisme d'interception par aspects qui est assuré grâce à la programmation orientée aspect que nous avons définie dans le premier chapitre.

La définition d'un aspect revient à la définition du pointcut et de l'advice.

- Le pointcut contient les méthodes à intercepter.
- Dans le code de l'advice de chaque pointcut, il y a :
  1. L'extraction des paramètres de la méthode pour identifier les paramètres du changement à travers le message.
  2. L'appel de la méthode adéquate selon l'interception pour créer et transférer le message XML.

Pour ce faire, nous avons implanté la classe « `AspectInterceptionExecution.aj` » contenant le code d'aspect des méthodes

de POLYORB à intercepter et l'extraction des paramètres des différentes méthodes. Nous avons également implanté la classe « `CreationEnvoiMessages.java` » permettant la création et le transfert des messages décrivant convenablement les changements au niveau de l'exécution.

Dans la suite de cette section, pour mieux expliquer l'implantation du protocole, nous nous focalisons sur la création (plus précisément l'activation) d'un nouveau thread périodique.

Listing 4.1 – Aspect d'interception d'un thread périodique

```
1  aspect AspectInterceptionExecution{
2      ...
3      pointcut active_Periodic_thread() :call(* fr..*.job(..)) ;
4      before():active_Periodic_thread() {
5      String handler_name=thisJoinPoint.getTarget().getClass().getSimpleName();
6      String thread_name=handler_name.substring(0, handler_name.indexOf("TaskHandler"));
7      Debug.debugMessage(" ACTIVE perodic Thread " + thread_name);
8      CreationEnvoiMessages.MessagesThread_XML(thread_name,"PingerTHImpl","add");}
9      ...
10   }
```

Le listing 4.1 présente un exemple d'aspect appliqué pour les threads périodiques. Ce code d'aspect définit le pointcut nommé `active_Periodic_thread`. Ce dernier intercepte toutes les exécutions des méthodes `job` de n'importe quelle classe (*.job). Le code d'aspect (advice) (lignes 5-8) indique qu'avant l'exécution des méthodes `job` interceptées, il faudra récupérer les paramètres décrivant la modification effectuée (le nom du thread, `thread_name`) et construire la trace correspondante.

La méthode `MessagesThread_XML` de la classe `CreationEnvoiMessages` permet de construire la trace décrivant l'ajout ou la suppression d'un thread. Elle admet comme paramètres le nom du thread, son type (`classifier`) et l'action d'ajout (`add`) ou de suppression (`delete`) comme le montre le listing 4.2.

Listing 4.2 – Méthode responsable du lancement de la création de la trace

```
1  public static synchronized void MessagesThread_XML(String nom,String classifier,String action)
2  {
3      MyRealtimeClass rtt = new MyRealtimeClass(nom,classifier,action);
4      PriorityParameters priority = new
5          PriorityParameters(PriorityScheduler.instance().getMinPriority());
6      rtt.setSchedulingParameters( priority );
7      rtt.start();
8  }
```

A chaque changement d'un thread, nous avons défini un thread temps réel `rtt` de type `MyRealtimeClass` qui va lancer dans sa méthode run le traitement de création et d'envoi de la trace. Cette classe, qui étend la classe `RealtimeThread`, manipule des traces sous la forme de messages XML conformément au schéma XML déjà décrit dans le chapitre 3. Le listing 4.3 montre l'ensemble des instructions à suivre pour créer et envoyer la trace convenable décrivant l'ajout ou la suppression d'un thread.

La trace est construite sous format XML en se basant su l'API JDOM qui facilite la création, la lecture ou la modification d'un fichier XML. Pour la construire, nous commençons par la balise racine contenant l'action d'ajout ou de suppression (lignes 4-6), puis nous insérons un élément `thread` avec ses différents attributs (nom, classifier)(lignes 7-12). Le message XML est par la suite bien formaté (lignes 13-14) et envoyé dans un flux de sortie du socket(lignes 15-16).

Listing 4.3 – Méthode responsable de la création et de l'envoi de la trace

```
static class MyRealtimeClass extends RealtimeThread {
    public synchronized void run() {
      try{
              Element racine = new Element(action);
              org.jdom.Document document = new Document (racine);
              racine.setAttribute("schemaLocation", SCHEMALOCATION, XSI_NS);
              Element thread = new Element("thread");
              racine.addContent(thread);
              Attribute name = new Attribute("name",nom);
              thread.setAttribute(name);
              Attribute classifieur = new Attribute("classifier",classifier);
              thread.setAttribute(classifieur);
              XMLOutputter sortie = new XMLOutputter(Format.getPrettyFormat());
              message_parent= sortie.outputString(document);
              synchronized (Connection.sock)
              {  Connection.sockOut.println(message_parent); }
      }catch (Exception e)
      { System.out.print(e); }
    }
}
```

En effet, pour assurer la communication entre les différents nœuds du système, nous avons défini tout un processus pour l'établissement de connexion. Nous avons implanté la classe `Connection` responsable de l'établissement de la connexion entre deux nœuds de l'application. Le contenu de cette classe est détaillé dans le listing 4.4.

Listing 4.4 – Établissement de la connexion entre l'exécution et le modèle

```
public class Connection{
     static Socket sock;
     static BufferedReader sockIn;
     static  PrintWriter sockOut;
     public static void connect(){
        try{
            boolean b=true;
            sock=new Socket();
            sock.setReuseAddress(true);
            while(b){
                  sock.connect(new InetSocketAddress("192.168.1.64",1425));
                  if (sock==null )b=true ;
                  else b=false;
            }
            sockIn=new BufferedReader(new InputStreamReader(sock.getInputStream()));
            sockOut= new PrintWriter( sock.getOutputStream(), true);
            System.out.println( sockIn.readLine());
        }catch (UnknownHostException e)
        {
            System.err.println("network problem");
            System.exit(1);
        }catch (IOException e)
        {
            System.err.println("Connetion impossible:inaccessible model port");}
        }
}
```

Au niveau du modèle, nous assurons la réception du message envoyé par l'intermédiaire de la classe `Model_Traitement.java` exécuté au niveau du modèle. Cette classe, comme l'indique le listing 4.5, permet d'extraire le message XML, d'effectuer un test de validité pour vérifier que le message est reçu correctement et qu'il est conforme au schéma XML prédéfini, et d'agir sur le fichier XMI représentant le modèle AADL.

Listing 4.5 – Extrait du fichier Model_Traitement.java décrivant la réception de messages

```
1  public void run()
2  {
3      File(System.getProperty("user.dir")+"/messages_recu/"+"fichier_modelegraphique.xml");
4      boolean validite;
5      String recu;
6      FileOutputStream fin;
7      try{
8          while(true ){
9              synchronized (indice)
10             {
11                 recu= depuisSocketexecution.readLine();
12                 if (recu==null)
13                 {
14                     System.out.println("fin de reception");
15                     stop();
16                     break;
17                 }
18                 if (recu.length()!=0)
19                 {
20                     nom_fichier="f"+compteur+".xml";
21                     File fichier=newFile(System.getProperty("user.dir")
22                           +"/messages_recu/"+ nom_fichier);
23                     fin=new FileOutputStream(fichier);
24                     PrintStream bw=new PrintStream(fin);
25                     System.out.println(" Reception of new message ");
26                     while(recu.length()!=0)
27                     {
28                           System.out.println(recu);
29                           bw.println(recu);
30                           recu= depuisSocketexecution.readLine();
31                     }
32                     System.out.println("Check of validity of the message:");
33                     validite=Verif_Validite("messageSchema.xsd",nom_fichier);
34                     Document doc_msg=Parser (nom_fichier);
35                     Document doc_modele=Parser("fichier_modelegraphique.xml") ;
36                     Modification_xmlModel(doc_msg,doc_modele);
37                     fichier.delete();
38                     compteur++;
39                 }
40             }
41         }
42     }catch(IOException e){}
43 }
```

Comme nous l'avons déjà mentionné, nous établissons une connexion entre le modèle et l'exécution basée sur les sockets afin de circuler ces messages au sein du système distribué. Il s'agit de lire le message reçu et de le transformer dans un fichier XML. Si le message est non nul, nous passons à la réception et la vérification de la validité du message reçu (lignes 18-32). Après la validation du message, nous analysons le fichier XML construit à partir du message et le fichier XMI représentant le modèle (lignes 33-35). Finalement, nous supprimons le fichier XML construit afin d'économiser les ressources mémoires (ligne 36).

Listing 4.6 – Extrait du fichier Model_Traitement.java décrivant la vérification de la validité des messages

```
1  public static boolean Verif_Validite(String fichSchema,  String fichXML)
2  {
3      try
4      {
5          SchemaFactory factory= SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
```

```
6           Schema schema;
7           try{
8               schema  = factory.newSchema(new StreamSource(new
9                   File(System.getProperty("user.dir")+"/messages_recu/"+fichSchema)));
10          }catch(SAXException f){
11              System.out.println("Schéma incorrect: "+fichSchema + " :"+ f.getMessage());
12              throw new Exception("Erreur sur le schéma");
13          }
14          Validator validator = schema.newValidator();
15          validator.validate(new StreamSource(new File(System.getProperty("user.dir")
16                  +"/messages_recu/"+fichXML)) );
17          System.out.println("Le message XML est valide par rapport au schema "+fichSchema);
18          return true;
19      }catch (SAXException e){
20          System.out.println("Le message XML invalide : "+e.getMessage());
21          return false;
22      }catch (IOException e){
23          System.out.println("IO : "+e.getMessage());
24          return false;
25      }catch(Exception e){
26          System.out.println("Erreur : "+e.getMessage());
27          return false;
28      }
29  }
```

Le listing 4.6 décrit la méthode de validation des messages. Nous commençons par construire un outil de validation paramétré par le schéma. L'usine en question (ligne 5) est paramétrée par une chaîne de caractères qui n'est autre que l'URL du namespace de XMLS : « http ://www.w3.org/2001/XMLSchema ». C'est ce paramétrage qui indique que l'on travaille avec des XML-schémas. Nous vérifions en premier lieu s'il existe une erreur dans le fichier schéma (lignes 6-13). En second lieu, nous nous intéressons aux erreurs dues à la non-conformité du fichier XML au schéma (lignes 14-17).

Dans le cas où nous nous assurons que la réception est correcte, nous passons à l'adaptation. Il s'agit de modifier le fichier XMI représentant notre modèle conceptuel. Selon le message reçu, nous ajoutons ou supprimons les balises décrivant les composants ou les connexions ajoutés ou supprimés.

Listing 4.7 – Extrait du fichier Model_Traitement.java décrivant l'action d'ajout d'une connexion

```
1   Modification_xmlModel(Document docmsg,Document docmodele){
2       Element racine=docmsg.getRootElement();
3       if(racine.getName().equals("add"))
4       {
5           while(j.hasNext())
6           {
7               if (composant.equals("connexion"))
8               {
9                   if (process.getChild("connections")==null)
10                      process.addContent(new Element("connections"));
11                  Element Connexion = new Element("eventDataConnection");
12                  Attribute name = new
13                          Attribute("name",courant.getAttributeValue("name"));
14                  Attribute source_port = new Attribute("src",courant.getAttributeValue("sourcePort")
15                  Attribute dest_port = new
16                          Attribute("dst",courant.getAttributeValue("destPort"));
17                  Attribute dest_thread = new
18                          Attribute("dstContext",courant.getAttributeValue("compDest"));
19                  Attribute source_thread = new
20                          Attribute("srcContext",courant.getAttributeValue("compSource"));
21                  Connexion.setAttribute(name);
22                  Connexion.setAttribute(source_port);
```

```
23                  Connexion.setAttribute(dest_port);
24                  Connexion.setAttribute(source_thread);
25                  Connexion.setAttribute(dest_thread);
26                  process.getChild("connections").addContent(Connexion);
27              }
28          }
29      }
30  }
```

Prenons l'exemple d'ajout de connexion de type `EventData`. Comme le montre le listing 4.7, après avoir détecté le type d'action (add)(lignes 2-3) et le type du composant (connexion)(ligne 7), nous construisons une balise `Connections` sous l'élément parent si elle n'existe pas (ligne 9-10). Puis nous créons un élément contenant les différents attributs de la connexion extraits du message reçu (lignes 11-21). Finalement, nous insérons l'élément `connexion` construit sous la balise `connections`(ligne 22). Ainsi, le modèle est mis à jour après l'ajout de la connexion sous forme de balise dans le fichier XMI. Le modèle sera rafraîchi automatiquement et la connexion apparaît alors dans le modèle graphique.

Sens 2 : du modèle vers l'exécution

Le principe d'adaptation utilisé dans ce sens se base pour la détection des changements sur des bibliothèques de fonctions appelées hooks. Ces hooks sont greffés dans le code de l'éditeur graphique afin de personnaliser son fonctionnement dans le but de capturer toute modification instantanément. Aprés la détection, tout changement sera alors décrit dans une trace générée sous format XML. Le même principe de génération de trace et le même schéma XML définissant le format des messages sont utilisés dans les deux sens.

Au niveau de la machine contenant le modèle, nous avons implanté une classe responsable de l'envoi et la réception des messages appelée `CreationEnvoiChangementModel.java`.

Par exemple, pour la création d'un message décrivant l'ajout ou la suppresion d'une connexion, nous avons implanté la méthode `MessagesConnexion_XML`. Comme le montre le listing 4.8, le principe est le même que le premier sens. Il s'agit d'extraire les paramètres du changement et de les décrire sous la forme d'un message XML conformément au schéma XML prédéfini. La structure du message est constituée d'une balise décrivant l'action et une sous balise décrivant le composant avec ses différents attributs. Une description détaillée de cette structure est présentée dans le chapitre 3.

Listing 4.8 – Extrait du fichier CreationEnvoiChangementModel.java décrivant un changement effectué sur une connexion

```
1  public static synchronized void MessagesConnexion_XML(String nom,String source_name,
2  String destination_name,String thread_source_name,String thread_dest_name,String action){
3       String message_parent;
4       Element racine = new Element(action);
5       org.jdom.Document document = new Document (racine);
6       racine.setAttribute("schemaLocation", SCHEMALOCATION, XSI_NS);
7       Element connexion = new Element("connexion");
8       racine.addContent(connexion);
9       Attribute name = new Attribute("name",nom);
10      Attribute source = new Attribute("sourcePort",source_name);
11      Attribute destination = new Attribute("destPort",destination_name);
```

```
12      Attribute th_destination = new Attribute("compDest",thread_dest_name);
13      Attribute th_src = new Attribute("compSource",thread_source_name);
14      connexion.setAttribute(name);
15      connexion.setAttribute(source);
16      connexion.setAttribute(destination);
17      connexion.setAttribute(th_src);
18      connexion.setAttribute(th_destination);
19      XMLOutputter sortie = new XMLOutputter(Format.getPrettyFormat());
20      message_parent= sortie.outputString(document);
21      try{
22          Model_Traitement.versSocketexecution.println(message_parent);
23      }catch (Exception e){
24          System.out.print(e);
25      }
26  }
```

L'envoi et la réception des messages dans ce sens sont assuré de la même manière que l'autre sens. Pour mettre à jour l'exécution suite à la réception et l'analyse du message reçu, un thread est lancé dans la classe `Activity` générée dés l'initialisation de l'application qui va servir pour adapter l'exécution selon l'état du modèle. Pour ce faire, nous avons défini la classe `thread_modif_execution.java` placée côté exécution dans laquelle il y aura extraction des paramètres du changement et appel à la méthode convenable pour la mise à jour de l'exécution. Ceci est fait après la réception des messages et leurs validation.

La classe `ChangementExecution.java` est responsable d'agir sur l'exécution selon la demande d'adaptation. Le listing 4.9 illustre l'invocation de la méthode de la classe `ChangementExecution.java` adéquate permettant d'agir sur l'exécution selon la demande d'adaptation.

Listing 4.9 – Extrait du fichier thread_modif_execution.java décrivant le choix de l'action convenbale sur l'exécution

```
1   while (i.hasNext()) {
2       Element courant = (Element) i.next();
3       if (courant.getName().equals("thread"))
4       {
5           String  thread_name =courant.getAttributeValue("name");
6           if( racine.getName().equals("add"))
7               ChangementExecution.Add_Thread(thread_name);
8           else
9               ChangementExecution.Delete_Thread(thread_name);
10      }
11      if (courant.getName().equals("connexion"))
12          ChangementExecution.Message_Connexion(doc_msg);
13  }
```

Par exemple, si la modification au niveau du modèle consiste à la suppression d'un thread, la méthode `Delete_Thread` de la classe `ChangementExecution.java` sera alors appelée afin de le désactiver également au niveau de l'exécution.

Listing 4.10 – Extrait du fichier ChangementExecution.java décrivant la procédure de désactivation d'un thread au niveau de l'exécution

```
1   public static void Delete_Thread(String name){
2       try{
3           Class class_principle = Class.forName("Activity");
4           String task= name+"Task";
5           Field thread_task = class_principle.getField(task);
6           Object instance_thread=thread_task.get(class_principle);
7           String type_thread=instance_thread.getClass().getSimpleName();
```

```
8           if (type_thread.equals("PeriodicTask") )
9           {
10              PeriodicTask periodic_instance= (PeriodicTask)instance_thread;
11              periodic_instance.suspend();
12              System.out.println("The thread "+task +"is desactivated");
13          }
14          if (type_thread.equals("SporadicTask") )
15          {
16              System.out.println("Suspend Sporadic");
17              SporadicTask sporadic_instance= (SporadicTask)instance_thread;
18              sporadic_instance.suspend();
19              System.out.println("The thread "+task +" is desactivated");
20          }
21      }catch(ClassNotFoundException s){
22          System.out.println("s.toString");
23      }catch(NoSuchFieldException z){
24          System.out.println("z.toString");
25      }catch(IllegalAccessException r){
26          System.out.println("r.toString");
27      }
28  }
```

Le listing 4.10 présente un extrait de la classe `ChangementExecution.java` qui met en œuvre la désactivation d'un thread. Nous commençons par l'extraction du nom du thread généré d'après l'implantation générée dans la classe `Activity` (lignes 3-6). Puis selon le type du thread (périodique ou sporadique), nous effectuons le traitement correspondant afin de le désactiver (lignes 7-20).

Ces différentes fonctionnalités offertes par ce protocole seront détaillées dans la suite dans le chapitre 5 dans le cadre d'une étude de cas d'un Guichet Bancaire Automatique (GAB).

4.5 Conclusion

Dans ce chapitre, nous avons présenté en premier lieu les différents outils et langages de programmation que nous avons utilisés pour la mise en œuvre de notre approche. En second lieu, nous avons entamé la réalisation de notre propre éditeur graphique « AADL Graphical Editor » sous la forme d'un plugin Eclipse permettant la modélisation d'un système avec AADL. Finalement, nous avons détaillé la réalisation de notre protocole.

Dans le chapitre suivant, nous validons notre approche par une étude de cas d'un guichet bancaire automatique.

Chapitre 5

Validation de notre approche

5.1 Introduction

Dans le chapitre précédent, nous avons détaillé les différentes étapes suivies pour la mise en œuvre de notre approche. Nous nous nous sommes intéressés non seulement à la réalisation de l'éditeur graphique mais aussi à la réalisation du protocole.

Dans ce chapitre, nous décrivons avec détails l'étude de cas d'un guichet bancaire automatique afin de valider notre approche. Tout d'abord nous décrivons le système. Ensuite nous détaillons son architecture et nous présentons son modèle AADL conçu avec notre éditeur. Finalement, nous présentons la preuve des différentes fonctionnalités du protocole en se basant sur les différents composants et connexions de ce cas d'étude.

5.2 Etude de cas

Pour mettre en œuvre le processus d'adaptation pour une application à base de composants proposé dans le chapitre 3, nous avons choisi l'étude de cas d'un guichet bancaire automatique (*Automated Teller Machine*) pour les systèmes bancaires. L'élaboration de ce cas d'étude est dans le but de valider notre éditeur graphique en l'utilisant pour concevoir son modèle avec AADL. De plus, dans le but de mettre en œuvre les différentes fonctionnalités du protocole, nous appliquons ses concepts sur l'exécution de l'un des processus de ce système et son modèle AADL.

5.2.1 Description

Le guichet bancaire automatique (GAB) est un système automatisé permettant aux clients d'effectuer différentes transactions bancaires telles que la consultation des soldes, les retraits et les dépôts d'argent, etc. Pour ce faire, sur la plupart des guichets bancaires automatiques, un client doit insérer une carte qui sert à l'identifier suivi de la saisie d'un code confidentiel. Cette carte en plastique contenant une bande magnétique ou une puce, possède un identifiant unique (`NumCard`) et quelques informations de sécurité à savoir la date d'expiration ou le code de vérification de la carte `CVVC` (*Card Verification Value Code*).

5.2.2 Architecture du système GAB

Le système est composé d'un ensemble de processus nécessaires pour son fonctionnement et qui sont déployés de façon distribuée : chaque processus est en exécution sur une machine différente. En fait, ce système est présenté par trois processus interconnectés entre eux : le processus `Customer`, le processus `Account` et le processus `AccountData` comme l'indique la figure 5.1.

Cette figure est enregistrée après la modélisation du système GAB en utilisant notre propre éditeur « AADL Graphical Editor ». Le processus `Customer` permet la gestion d'authentification. Lui-même, il est composé de trois threads : `Pinger` (périodique), `Validation` (sporadique) et `Gui` (sporadique). Lors d'insertion de la carte par le client, le thread `Pinger` détecte son numéro et l'envoie au thread `Validation`. Ce dernier est destiné à vérifier la validité de la carte insérée en terme de date d'expiration via un sous programme `CheckValidity`. Dans le cas ou la carte est expirée, le système la rejette en informant le client avec un message explicatif.

Toutefois, si elle est encore valide, le client est invité à saisir son code confidentiel via le thread `Gui` qui présente l'interface graphique assurant la communication entre le client et le système. C'est à travers cette interface que l'utilisateur fournit les informations nécessaires pour accéder à son compte (code) ou effectuer un retrait (montant à retirer) par exemple. Une fois le code reçu de la part de `Gui`, le thread `Validation` vérifie si le code est correct ou non via un sous programme `CheckCard`. Si le code est erroné, `Validation` informera le thread `Gui`, via le port `RestoreCode_out_V`, dans le but de demander au client de le resaisir. Le client admet trois chances pour la saisie de son code. L'échec de la troisième tentative d'authentification entraine le rejet de la carte.

Dans le cas de validité du code, le processus `Customer` se connecte alors au processus `AccountData` qui consulte toute la base des données des clients pour fournir l'identifiant de session au processus `Customer` via le port `OK_OUT_V`. Lorsque le thread `Gui` reçoit cette information, il invite le client à sélectionner l'opération qu'il souhaite effectuer ainsi que le montant de retrait ou de dépôt. Ces informations sont transmises vers le processus `Account` pour pouvoir accomplir l'opération voulue. Finalement, l'exécution de l'opération est assurée par le thread `Account` et un message est affiché au client pour l'informer de l'échec ou du succès de cette opération.

5.2.3 Mise en œuvre du protocole

Dans cette section, nous nous intéressons à la mise en œuvre de notre protocole tout en se référant à l'étude de cas du GAB décrit ci-dessus. Pour mettre en œuvre les différentes fonctionnalités du protocole, nous présentons dans ce qui suit la phase d'établissement de connexion ainsi que l'implantation du protocole pour les des deux sens.

◇ **Phase d'établissement de connexion**

Nous supposons que notre application du GAB est répartie sur plusieurs machines. C'est pour cette raison que nous nous intéressons à l'établissement d'une telle connexion. Il s'agit de la connexion entre la machine sur laquelle tourne le processus `Customer` (machine d'exécution) décrit précédemment et celle contenant

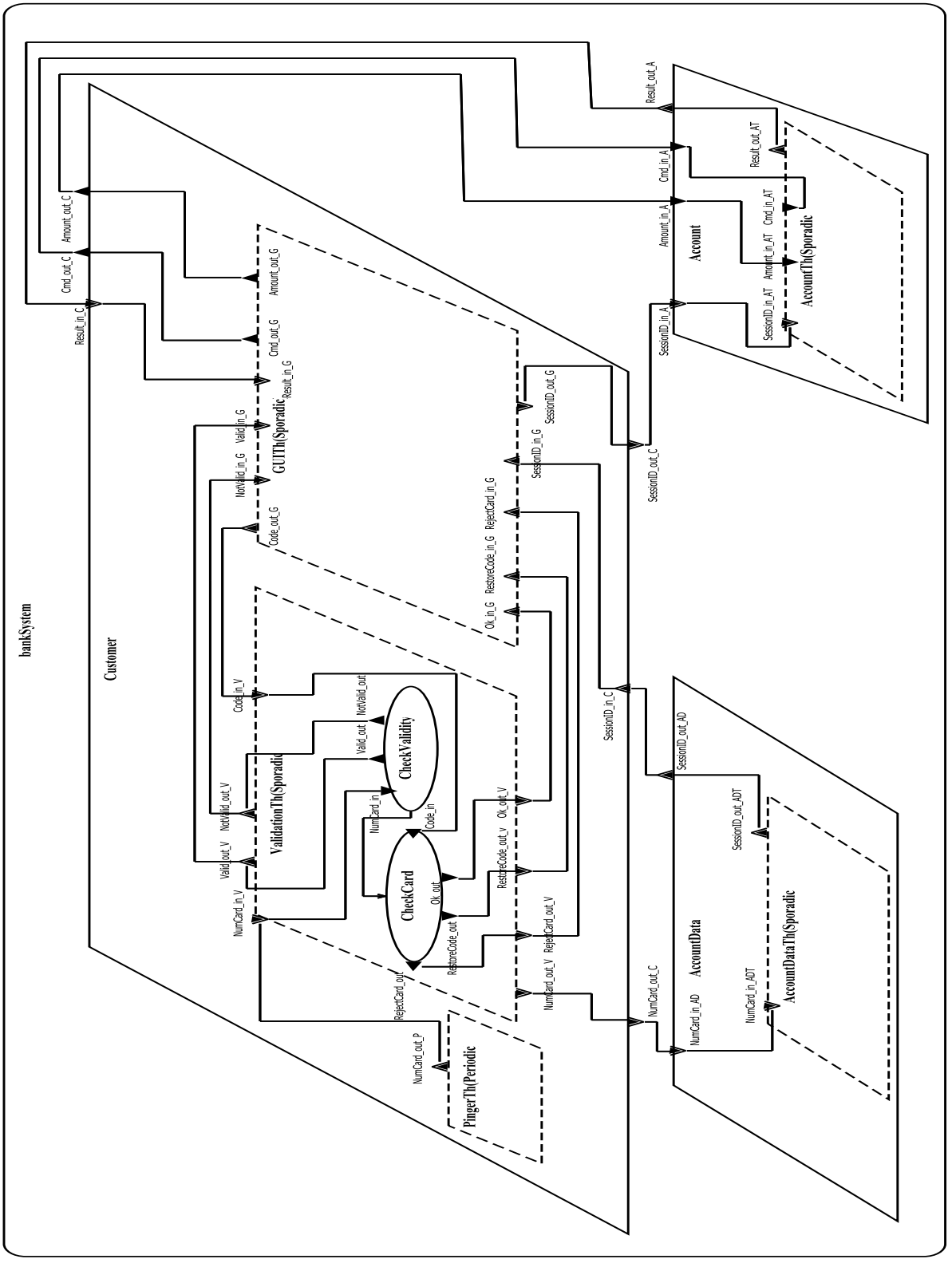

Figure 5.1 – Modélisation du GAB avec AADL

son modèle (machine du modèle) architectural décrit avec AADL présenté dans la figure 5.2.

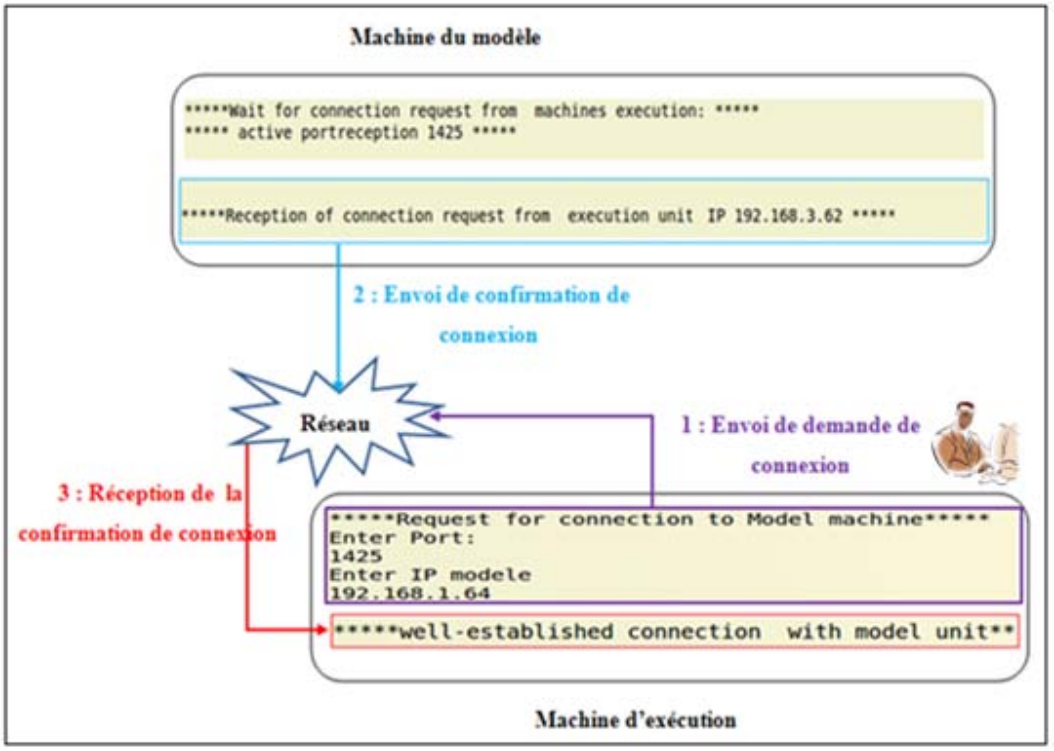

Figure 5.2 – Etablissement de connexion entre le modèle et l'exécution

Comme le montre la figure 5.2, l'établissement de connexion se fait en trois étapes :

1. Dés le lancement de l'exécution, l'utilisateur est invité à saisir les informations nécessaires pour lancer la connexion (adresse IP et port).
2. Au niveau du modèle, la machine est en écoute des tentatives de connexions. Quand elle reçoit une requête de connexion de la part de la machine d'exécution, un message est affiché indiquant la réception.
3. L'utilisateur côté exécution est par la suite informé de la réussite d'établissement de connexion.

◇ **Implantation du protocole sens1 : de l'exécution vers le modèle**

Dans ce sens, le protocole est destiné à intercepter tous les changements subis par les entités (Thread, sous programme et connexion) figurant au niveau de l'exécution à travers le mécanisme des aspects. Après avoir détecté une modification subie par l'une de ces entités, ce protocole a la charge de mettre à jour le modèle distant en se basant sur les traces sauvegardées.

– **Trace d'interception au cours de l'exécution**

Dans ce qui suit, nous présentons des exemples d'interception par aspect effectué sur des différents composants actifs dans le processus `Customer`. Nous

montrons également des messages crées correspondants aux différents changements et récupérés au sein du modèle AADL.

· *Activation d'un thread périodique*

Le thread `Pinger` du processus `Customer` est un thread périodique. Le lancement de ce thread est dû à l'appel de la méthode `job` qui caractérise un tel type de thread et contient son code fonctionnel.

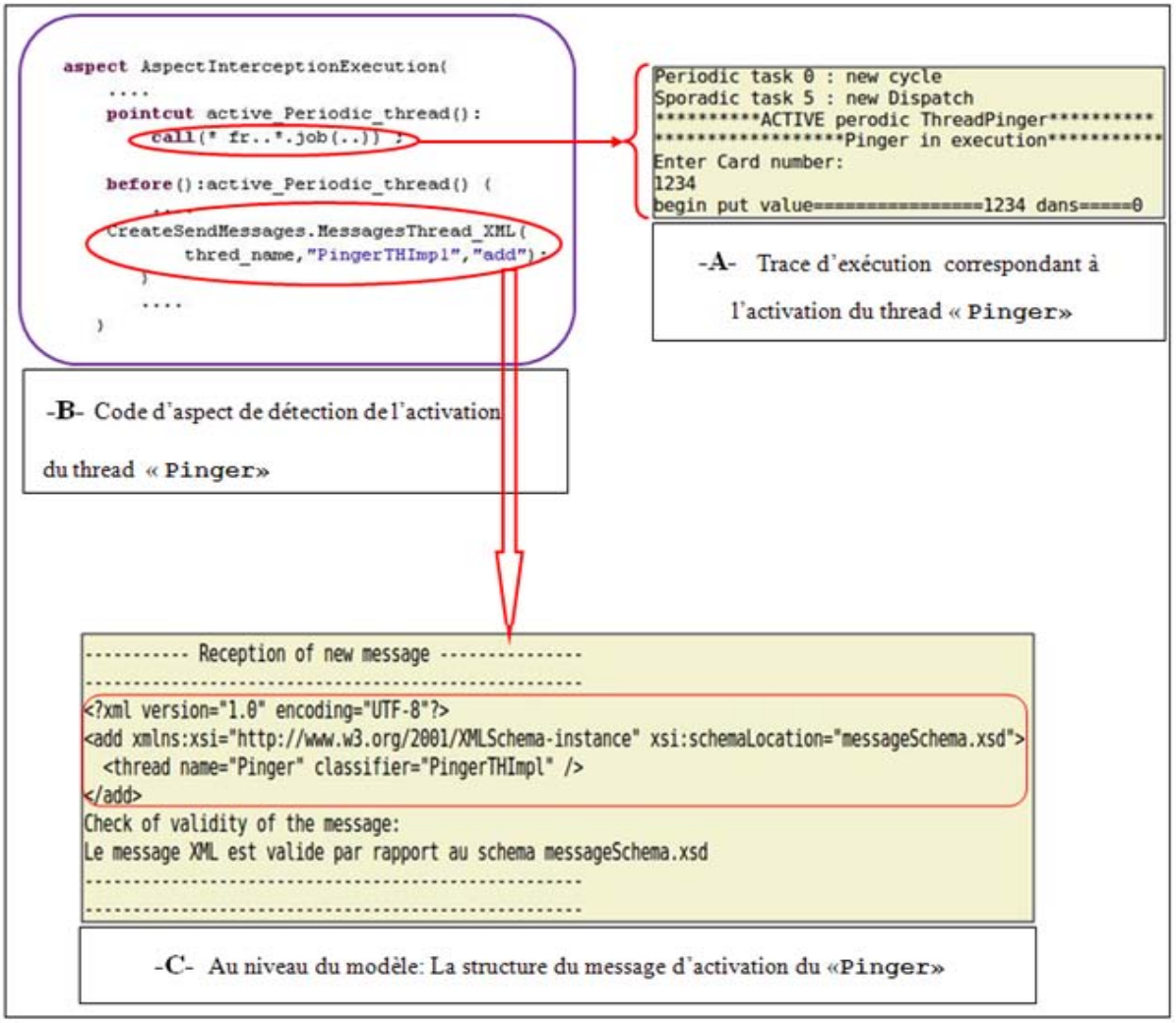

Figure 5.3 – Interception de l'activation d'un thread périodique

Comme le montre la figure 5.3, l'idée consiste donc à déterminer l'instant d'appel à cette méthode pour déclencher la création du message XML correspond : ceci est réalisé en ayant recours à la programmation orientée aspect comme mentionnée dans le code des aspects (B). Au niveau de code de l'advice de cet aspect, il y aura le traitement nécessaire pour aboutir au message d'activation du thread `Pinger` récupéré au niveau de la machine du modèle (C).

· *Désactivation d'un thread sporadique*

Nous illustrons dans la figure 5.4 la démarche de traçabilité (A) donnant lieu à la désactivation d'un thread sporadique. Nous prenons l'exemple du thread

`Gui` dont le principe est d'être inactif tout au long d'absence d'information au niveau de l'un de ses ports d'entrées. La méthode `WaitForEvents` est une méthode prédéfinit dans le PolyORB-HI indiquant que ce thread est en attente et qu'il est inactif. Cette méthode est interceptée (B) pour aboutir au message de désactivation (C).

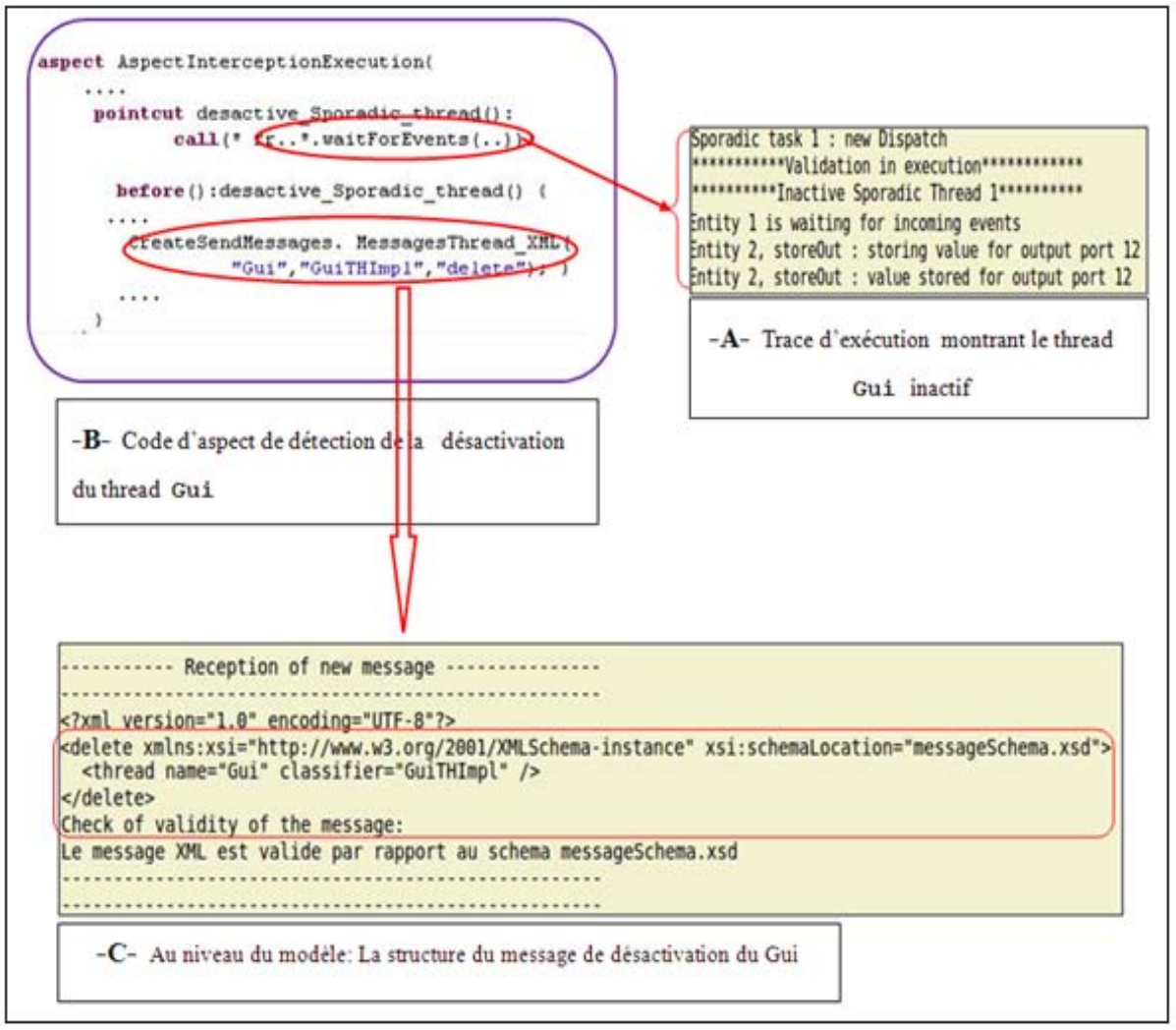

Figure 5.4 – Interception de la désactivation d'un thread sporadique

· *Interception d'envoi de message*
Dans le but de pouvoir suivre l'échange de messages entre les différents threads du processus `Customer`, nous contrôlons les différentes connexions existantes qui les relient afin de tracer ces échanges. Nous présentons dans la figure 5.5 un exemple de détection d'envoi d'un message du thread `Validation` vers le thread `Gui` pour transférer l'information de validation du code saisi par le client. Dans le code d'aspect indiqué (B) nous trouvons la méthode `sendOutput`, elle est responsable de l'envoi des messages échangés entre les ports des threads donc son appel donne la trace d'exécution (A). Le message (C) est celui qui va être envoyé au modèle comme indication de cette détection.

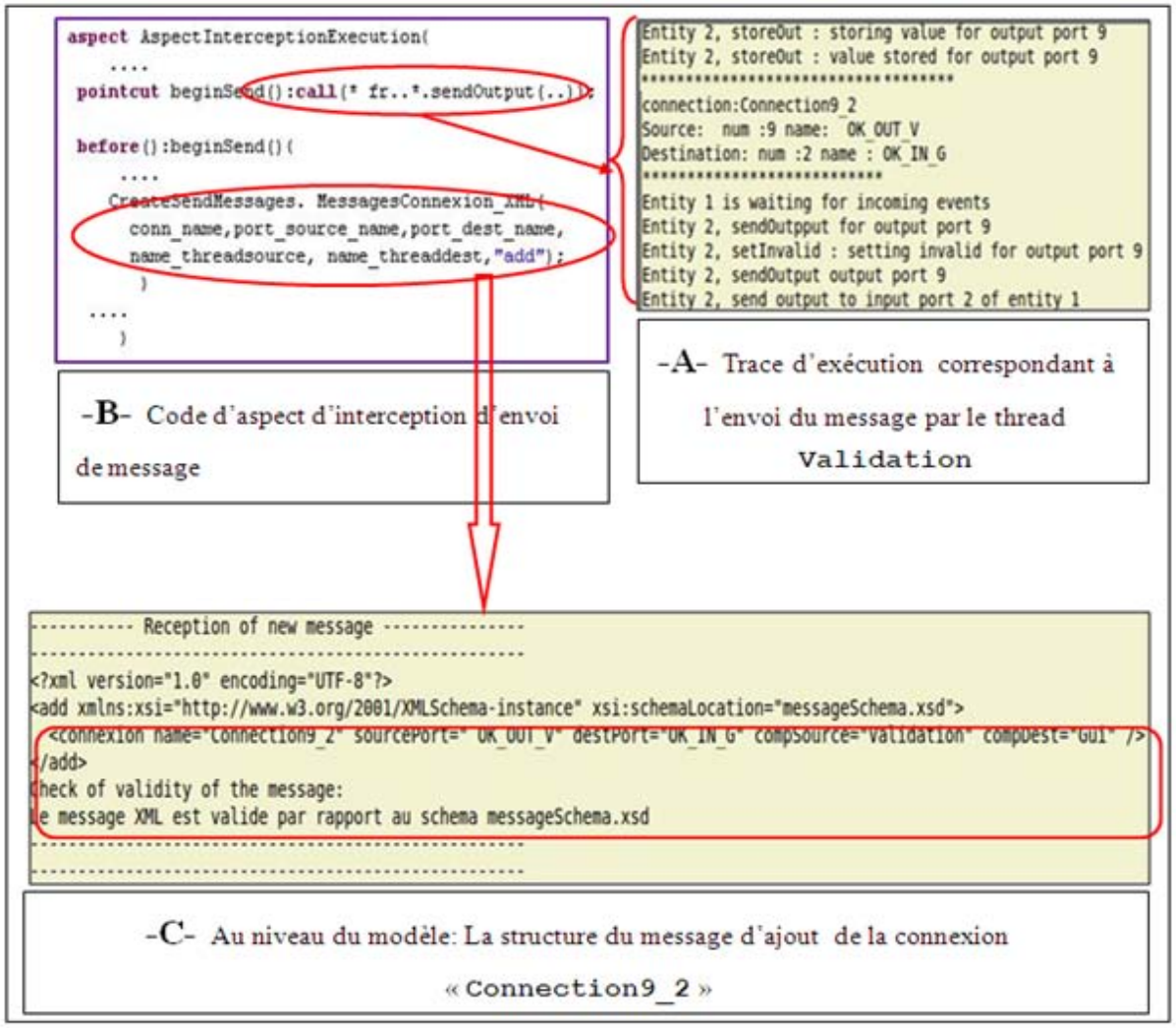

Figure 5.5 – Interception de l'envoi d'un message

Pour l'interception de la réception d'un message, il s'agit d'intercepter la méthode `storeIn` du PolyORB-HI. Cette méthode récupère l'information au niveau d'un port d'entrée.

· *Interception d'activation/désactivation d'une connexion*

L'établissement d'une connexion entre des composants AADL se fait par la création d'une socket de communication via la méthode connect de la classe `TransportLowLevelSockets` du PolyORB-HI. Cette méthode sera la cible d'interception pour transmettre au modèle un message XML d'activation d'une nouvelle connexion entre deux composants dont la structure est similaire à celle de la figure 5.5 (C).

Par contre, pour la désactivation d'une connexion, on intercepte la méthode prédéfinit `close` qui permet de fermer la socket de réception correspondante à la connexion déjà crée et par conséquent la désactiver.

· *Interception d'un appel à un sous-programme*

`CheckValidity` est un sous-programme appelé par le thread `Validation` pour

vérifier la validité de la carte du client. Nous détectons le moment d'appel à ce sous- programme par le code d'aspect de la figure 5.6 (B) pour informer le modèle par un message d'activation de cet appel présenté dans la figure (C).

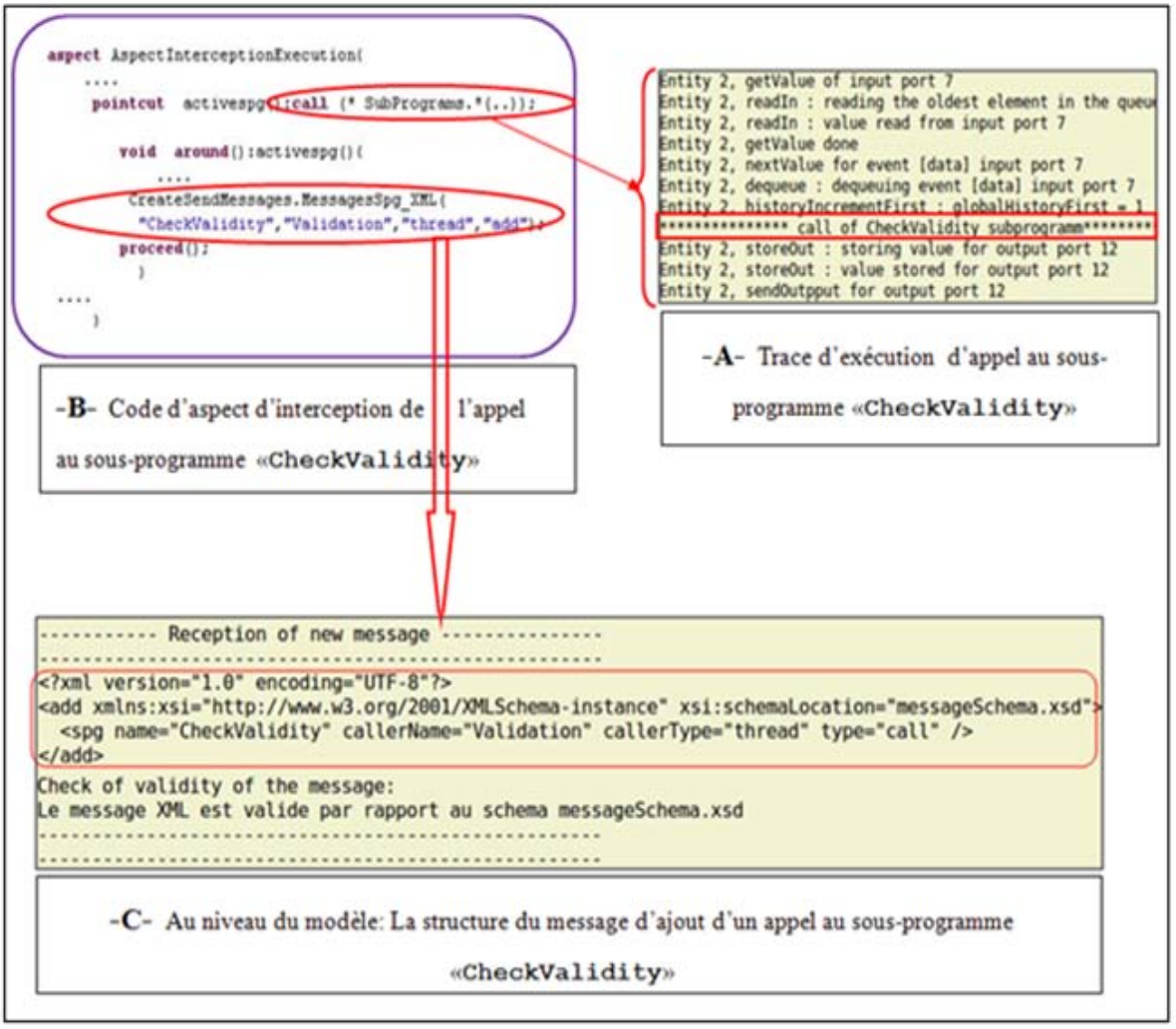

Figure 5.6 – Interception d'un appel à un sous programme

– **Mise à jour de la représentation XMI du modèle AADL**
Comme nous l'avons proposé dans le chapitre 3, après avoir reçu les messages décrivant les changements envoyés de la part de l'exécution, nous mettons à jour le fichier XMI représentant le modèle GAB. Nous illustrons dans la figure 5.7 la modification effectuée sur ce fichier afin de le mettre à jour suite à :
· L'ajout d'une nouvelle connexion entre le port source `VALID_OUT_V` et le port destination `VALID_IN_G` reliant les deux threads `Validation` et `Gui`.
· L'activation d'une instance de thread `Gui` au niveau de l'exécution.
Comme le montre la figure, la mise à jour du modèle consiste uniquement à l'insertion de nouvelles balises XMI décrivant les nouvelles entités ajoutées (Thread et connexion) en restant fidèle à la structure de base de ce fichier.

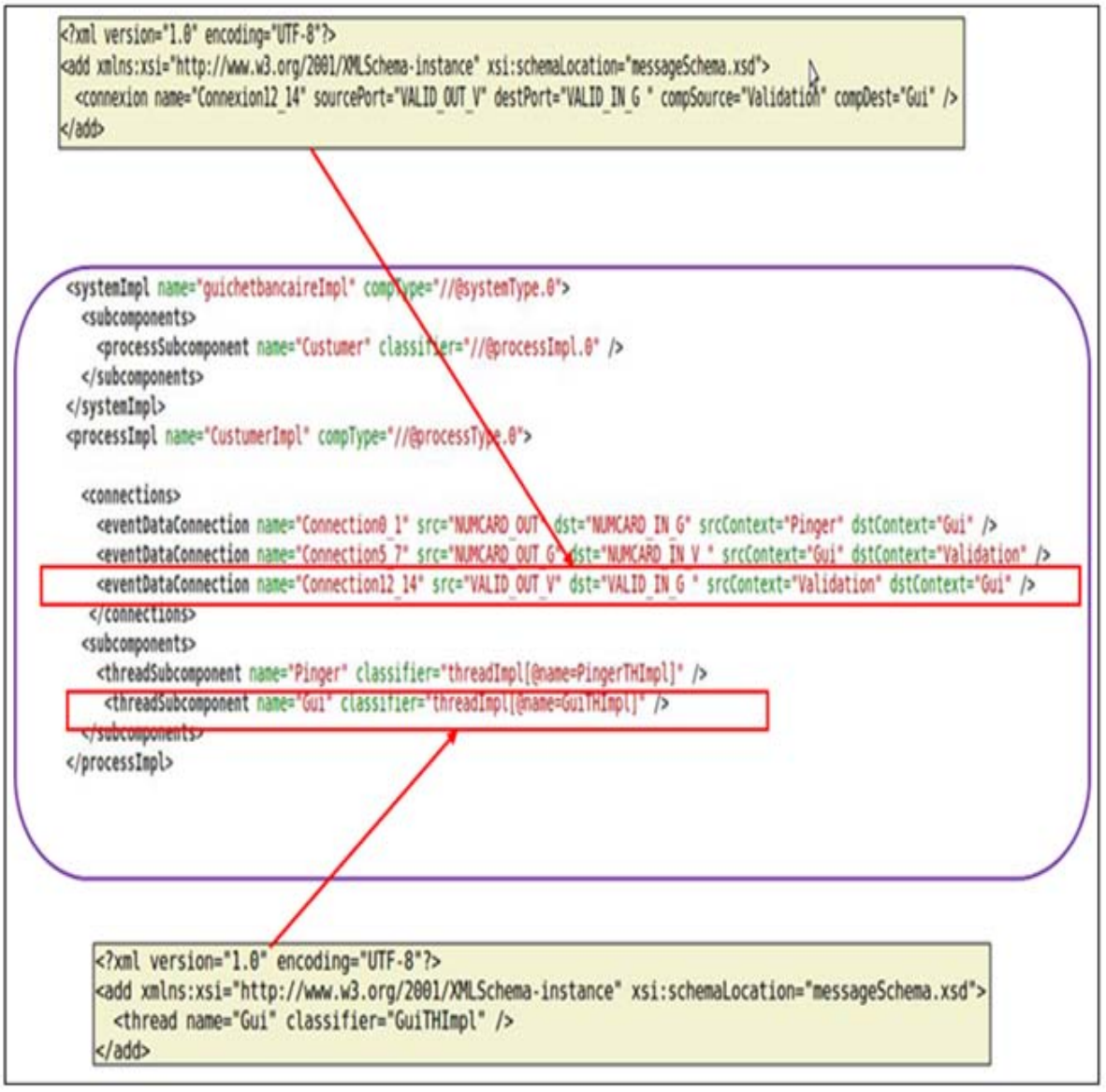

Figure 5.7 – Mise à jour du modèle suite aux changements au niveau de l'exécution

◇ **Implantation du protocole sens2 : du modèle vers l'exécution**

Pour l'adaptation du système en cours d'exécution, la capture des modifications effectuées sur le modèle revient aux méthodes hooks qui sont greffés dans le code de notre éditeur comme nous l'avons introduit dans notre contribution. Ces méthodes permettent la création et le transfert des traces décrivant les changements sous la forme de message XML vers l'exécution du système. A ce niveau, notre protocole est censé de recevoir ces messages, les analyser et vérifier la validité et enfin d'agir sur l'exécution. Nous allons, dans la suite, prendre l'exemple d'activation d'un thread et la désactivation d'une connexion.

· **Activation d'un thread**

Nous présentons dans la figure 5.8 l'activation du thread Account au niveau de

l'exécution suite à la réception d'un message d'ajout d'une instance de ce thread de la part du modèle.

Figure 5.8 – Mise à jour de l'exécution suite à l'activation du thread Account

Dans ce qui suit, nous présentons des exemples d'interception par aspect effectué sur de différents composants actifs dans le processus `Customer`. Nous montrons également des messages crées correspondants aux différentes interceptions et récupérés au sein de la machine du modèle AADL.

· **Désactivation d'une connexion**

Nous montrons dans la figure 5.9 l'adaptation de l'exécution suite à la désactivation d'une connexion au niveau du modèle.

◇ **Vérification de la validité des messages échangés**

Un test de validité des messages échangés est effectué au niveau de chaque réception soit au niveau du modèle soit au niveau de l'exécution. Cette vérification est nécessaire pour détecter les erreurs de transmission qui peuvent survenir. Nous présentons des exemples de test de la conformité des messages à une structure du

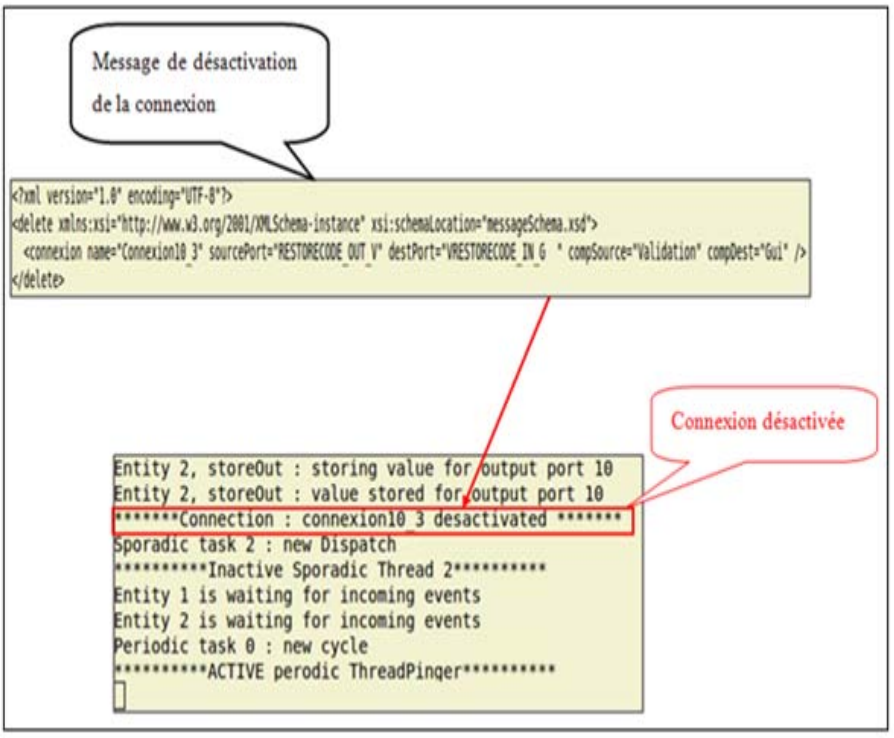

Figure 5.9 – Mise à jour de l'exécution suite à la suppression d'une connexion

schéma XML déjà défini.

· *Réception d'un message valide*

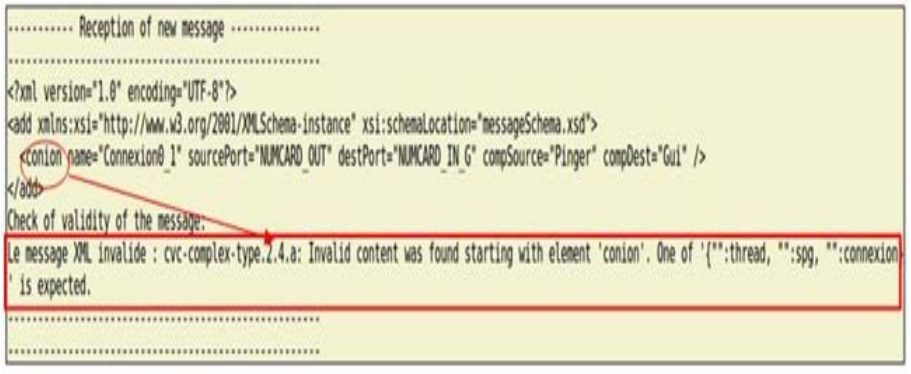

Figure 5.10 – Réception d'un message valide

La figure 5.10 présente un exemple d'un message valide reçu.

· *Réception d'un message non valide*

La figure 5.11 présente un exemple d'un message valide reçu. Il est à noter que le schéma XML est disponible au niveau de chaque nœud du système distribué pour unifier la structure des messages échangés et s'assurer de leurs validités.

Figure 5.11 – Réception d'un message non valide

5.3 Conclusion

Dans ce chapitre, nous avons détaillé l'étude de cas d'un guichet bancaire automatique afin de valider notre approche.

Nous avons commencé par décrire le système. Ensuite nous avons détaillé son architecture ainsi que son modèle son modèle AADL décrit avec notre propre éditeur. Finalement, nous avons illustré les différentes fonctionnalités du protocole en l'appliquant sur ce système.

Conclusions et perspectives

En conclusion, nous avons proposé, dans le cadre de notre projet de mastère, une approche de modélisation @Runtime des systèmes à base de composants.

Notre approche couvre tout le cycle de développement logiciel. Nous partons de la phase de conception en modélisant notre système avec AADL comme langage de description d'architecture. Nous avons développé un éditeur graphique sous la forme d'un plugin Eclipse permettant cette modélisation avec un certain niveau d'abstraction. Cet éditeur offre ainsi une vision globale du système qui facilite sa gestion au cours de l'exécution. Ce modèle décrit avec AADL peut être également enrichi avec le langage AO4AADL dans le but de séparer les différentes préoccupations techniques de celles fonctionnelles. Ce langage permet aussi de choisir certaines entités du modèle dans le cas d'une surveillance partielle. Par la suite, un processus de génération du code fonctionnel et du code des aspects est mis en place pour avoir finalement un système prêt à être exécuté. Notre contribution se focalise sur les deux phases de surveillance et d'adaptation des deux niveaux conceptuel et exécutif au cours de l'exécution. Pour cela, nous avons défini un protocole de communication qui traite l'échange des messages au sein du système distribué et assure son adaptation dynamique dans les deux sens. Ainsi, notre approche assure la correcte correspondance entre le modèle et l'exécution du système à tout moment.

Notre approche se distingue par rapport aux contributions existantes par le fait qu'elle utilise un seul formalisme (AADL) pour le modèle de conception et celui d'exécution. Ainsi elle minimise le coût de transformation nécessaire pour la navigation entre différents formalismes. En outre, ce formalisme unifié offre la possibilité d'étendre notre modèle en utilisant des annexes AO4AADL. Ceci permet de séparer les exigences fonctionnelles et transversales dés le niveau architectural. De plus, notre approche est favorisée puisqu'elle offre une adaptation dynamique des systèmes embarqués à base de composants et même des systèmes répartis dans les deux sens.

Certes, la solution que nous avons proposée répond à tous les objectifs que nous avons fixés au début de ce mastère. Néanmoins elle présente quelques limites notamment en ce qui concerne la génération et le transfert des traces. Dans le cas d'erreur de transmission, nous perdons la correspondance bidirectionnelle entre le modèle et l'exécution du système. Ainsi on peut par exemple supprimer un composant qui n'existait plus suite à la perte de l'opération qui l'ajoute. Une solution optimale consiste à revenir à la dernière situation stable. De plus, nous avons opté pour une plateforme dont le code fonctionnel est généré avec RTSJ et le code des aspects est généré en AspectJ, alors que notre approche offre une liberté de choix de la plateforme d'exécution parmi Ada, C ou Java. Ceci est à cause de l'absence des générateurs AspectAda et AspectC pour AO4AADL.

Ainsi, afin d'enrichir nos travaux, nous visons, à court terme, enrichir notre éditeur graphique par l'intégration des aspects AO4AADL. D'une part, et d'autre part greffer les différentes hooks permettant de détecter les changements au niveau du modèle.

A moyen terme, on pourra également développer les générateurs d'aspects AspectAda et AspectC pour permettre aux utilisateurs d'en profiter pour choisir la plateforme convenable à leurs besoins.

Finalement, on pourra à long terme, traiter le problème de la cohérence lors de l'adaptation au cours de l'exécution par assurer par exemple, une sauvegarde de l'information et/ou un blocage des composants actifs lors d'une adaptation.

Bibliographie

[Ada83] *Reference manual for the ada programming language*, February 1983, ANSI/MIL-STD 1815A. Also published by Springer-Verlag as LNCS 155.

[Aut09] Thomas Autret, *Génération de code real-time java pour systèmes temps-réel*, Master's thesis, Université Pierre & Marie Curie, ParisVI, France, September 2009.

[BBF09] Gordon Blair, Nelly Bencomo, and Robert B. France, *Models@ run.time*, Computer **42** (2009), 22–27.

[CCMC96] Paul C. Clements, Paul C. Clements, Thomas R. Miller, and Lt Col, *Coming attractions in software architecture*, 1996.

[FECA05] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit (eds.), *Aspect-oriented software development*, Addison-Wesley, Boston, 2005.

[FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak, *The Architecture Analysis & Design Language (AADL) : An Introduction*, Tech. report, Software Engineering Institute, 2006.

[HZ07] J. Hugues and B. Zalila, *PolyORB High Integrity User's Guide*, Tech. report, École Nationale Supérieure des Télécommunications, jan 2007.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, *An overview of aspectj*, Proceedings of the 15th European Conference on Object-Oriented Programming (London, UK, UK), ECOOP '01, Springer-Verlag, 2001, pp. 327–353.

[KIL+97] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar, *Aspect-oriented programming*, 1997.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin, *Aspect-oriented programming*, ECOOP, SpringerVerlag, 1997.

[KOS06] Philippe Kruchten, Henk Obbink, and Judith Stafford, *The past, present, and future for software architecture*, IEEE Softw. **23** (2006), no. 2, 22–30.

[LKZJ10] Sihem Loukil, Slim Kallel, Bechir Zalila, and Mohamed Jmaiel, *Toward an Aspect Oriented ADL for Embedded Systems*, Software Architecture (Muhammad Babar and Ian Gorton, eds.), Lecture Notes in Computer Science. 4th European Conference, ECSA 2010 Copenhagen, Denmark,

August 23-26, 2010 Proceedings, vol. 6285, Springer Berlin / Heidelberg, Copenhagen - Denmark, August 2010, pp. 489–492.

[Lou10] Sihem Loukil, *Extension d'un langage de description d'architecture pour la programmation orientée aspect*, Master's thesis, École Nationale d'Ingénieurs de Sfax, 2010.

[LZPH09] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues, *OCARINA : An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications*, Reliable Software Technologies - Ada-Europe 2009 (Fabrice Kordon and Yvon Kermarrec, eds.), Lecture Notes in Computer Science, vol. 5570, Springer Berlin / Heidelberg, Brest, France, jun 2009, pp. 237–250.

[Mao09] Shahar Maoz, *Using model-based traces as runtime models*, Computer **42** (2009), 28–36.

[MB08] Ahmed HARBOUCHE et Abdellah KOUIDER EL OUAHED Madiha BOUARA, *La démarche mda (model driven architecture)*, INFØDays'2008 (Chlef, Algérie), 2008.

[MBJ08] Brice Morin, Olivier Barais, and Jean-Marc Jézéquel, *K@rt : An aspect-oriented and model-oriented framework for dynamic software product lines*, in "Proceedings of the 3rd International Workshop on Models@Runtime, at MoDELS 08, 2008.

[MFB+08] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon Blair, *An aspect-oriented and model-driven approach for managing dynamic variability*, In Proceedings of ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 08) (Toulouse, France), October 2008.

[MSF+10] Naouel Moha, Sagar Sen, Cyril Faucher, Olivier Barais, and Jean-Marc Jézéquel, *Evaluation of kermeta for solving graph-based problems*, International Journal on Software Tools for Technology Transfer (STTT) **12** (2010), no. 3–4, 273–285 (EN).

[MT00] Nenad Medvidovic and Richard N. Taylor, *A classification and comparison framework for software architecture description languages*, IEEE Trans. Softw. Eng. **26** (2000), no. 1, 70–93.

[NPMH02] A. Navasa, M. A. Pérez, J. M. Murillo, and J. Hernández, *Aspect oriented software architecture : a structural perspective*, In Proceedings of the Aspect-Oriented Software Development, 2002, The, 2002.

[OPDR08] Audrey Occello, Anne-Marie Pinna-Déry, and Michel Riveill, *A Runtime Model for Monitoring Software Adaptation Safety and its Concretisation as a Service*, Models@runtime(MRT08) (Toulouse, France), , Nelly Bencomo, Gordon Blair, Robert France, Freddy Munoz, Cedric Jeanneret (eds.), September 2008, pp. 67–76.

[PW92] Dewayne E. Perry and Alexander L. Wolf, *Foundations for the study of software architecture*, SIGSOFT Softw. Eng. Notes **17** (1992), no. 4, 40–52.

[SAE04] SAE, *Architecture Analysis & Design Language (AS5506)*, September 2004, available at `http://www.sae.org`.

[SAE09] ———, *Architecture Analysis & Design Language (AS5506)*, Janvier 2009, available at `http://www.sae.org`.

[Smi90] Brin C Smith, *What do you mean, meta ?*, Workshop on Reflection and Metalevel Architechtures in OO Programming, ECOOP/OOPSLA'90, 1990.

[SSC09] Sebastien Saudrais, Athanasios Staikopoulos, and Siobhan Clarke, *Using specification models for runtime adaptations*, International Workshop on Models Run Time on MODELS 09, 2009.

[VZH06] T. Vergnaud, B. Zalila, and J. Hugues, *Ocarina : a Compiler for the AADL*, Tech. report, École Nationale Supérieure des Télécommunications, jun 2006.

[WK03] Jos Warmer and Anneke Kleppe, *The object constraint language : Getting your models ready for mda*, 2 ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[WP09] Stefan Winkler and Jens Pilgrim, *A survey of traceability in requirements engineering and model-driven development*, Software Systems Modeling **9** (2009), no. 4, 529–565.

[Zal08] Bechir Zalila, *Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture*, Ph.D. thesis, École Nationale Supérieure des Télécommunications, nov 2008.